FIFO queues are all you need for cache eviction

Juncheng Yang[*],   Yazhuo Zhang[§],   Ziyue Qiu[*],   Yao Yue[†],   K. V. Rashmi[*]

[*]Carnegie Mellon University   [§]Emory University   [†]Pelikan Foundation

Abstract

As a cache eviction algorithm, FIFO has a lot of attractive properties, such as simplicity, speed, scalability, and flash-friendliness. The most prominent criticism of FIFO is its low efficiency (high miss ratio).

In this work, we demonstrate a simple, scalable FIFO-based algorithm with three static queues (S3-FIFO). Evaluated on 6594 cache traces from 14 datasets, we show that S3-FIFO has lower miss ratios than state-of-the-art algorithms across traces. Moreover, S3-FIFO's efficiency is robust — it has the lowest mean miss ratio on 10 of the 14 datasets. FIFO queues enable S3-FIFO to achieve good scalability with 6× higher throughput compared to optimized LRU at 16 threads.

Our insight is that most objects in skewed workloads will only be accessed once in a short window, so it is critical to evict them early (also called quick demotion). The key of S3-FIFO is a small FIFO queue that filters out most objects from entering the main cache, which provides a guaranteed demotion speed and high demotion precision.

CCS Concepts: • **Information systems → Information storage systems**; • **Computer systems organization**;

1 Introduction

Software caches, such as Memcached [102] and Linux page cache [10], are widely deployed today to speed up data access and avoid repeated computation. A cache should be (1) *efficient*: it should provide a low miss ratio allowing most requests to be fulfilled by the cache with short latencies; (2) *performant*: serving data from the cache should perform minimal operations with a high throughput; and (3) *scalable*: the number of cache hits it can serve per second grows with the number of CPU cores. The heart of a cache is the eviction

algorithm, which dictates a cache's efficiency, throughput, and scalability.

Many works have looked into the design of efficient eviction algorithms [18, 35, 51, 74, 77, 79, 100, 110, 124, 166, 169]. Because LRU is believed to be more efficient than FIFO, these advanced algorithms are often LRU-based, using different techniques and metrics on top of one or more LRU queues. However, LRU suffers from two problems: (1) it requires two pointers per object, which is a significant storage overhead for workloads consisting of small objects; and (2) it is not scalable because each cache hit requires promoting the requested object to the head of the queue guarded by *locking*.

With the shrinking latency between the cache and the backend, and the rapid growth of CPU cores per socket, the cache's throughput and scalability become critical. An increasing number of works have studied this in the past few years [57, 60, 115, 143, 154, 158]. The solution is often to trade efficiency for throughput and scalability by using simple FIFO-queue-based eviction algorithms. For example, MemC3 [57], Tricache [60] use CLOCK, and Segcache [158] uses FIFO-merge. Compared to LRU, FIFO is simpler and more scalable, with the drawback of it being less efficient.

This work explores the opportunity of building a simple, scalable, yet efficient eviction algorithm with only FIFO queues. Object popularity in the cache workloads is often skewed and follows Power-law (e.g., Zipf) distribution [15, 29, 30, 157]. Our insight is that for any Zipf request sequence, the fraction of objects appearing once (called one-hit wonders) is much higher in a sub-sequence than in the full trace. Because a cache of size C only observes a short sequence of C objects before evictions, most objects will be one-hit wonders (no request after insertion) when evicted, even though they may have more requests throughout the full trace. We confirm this observation on 6594 production traces. The median one-hit-wonder ratio of all traces, when considering the entire trace, is 26%. However, when focusing on sequences that comprise 10% of the unique objects in each trace, the median one-hit-wonder ratio skyrockets to 72%.

We leverage this workload property and design S3-FIFO, a simple, scalable eviction algorithm with three static (fixed-size) FIFO queues. S3-FIFO uses a small probationary FIFO queue to filter out one-hit wonders from entering the main FIFO queue so that cache space can be used for more valuable objects (called *early eviction* or *quick demotion [155]*). Objects evicted from the small FIFO queue either enter the main or ghost FIFO queue, depending on whether it has been accessed. The main FIFO queue reinserts some popular objects during evictions. Many previous works have explored

similar ideas to quickly demote some objects [54, 79, 100], especially for scan and streaming workload patterns and in hierarchical caches. However, to the best of our knowledge, this is the first work demonstrating the importance of quick demotion for cache workloads even when there are no scan and streaming patterns. Moreover, this work designs the first FIFO-queue-only algorithm that is more efficient than state-of-the-art algorithms.

S3-FIFO is not only simple but also efficient. We compare S3-FIFO with 12 eviction algorithms on a large data collection of 6594 production traces from 14 sources. The traces overall contain 856 billion requests collected between 2007 and 2023, and cover block, key-value, and object caches. While advanced algorithms may excel at a few particular workloads, our evaluation shows that S3-FIFO achieves better efficiency (lower miss ratios) across traces at all percentiles than state-of-the-art algorithms. Moreover, S3-FIFO's efficiency is robust. Using a cache size of 10% of objects in the trace, S3-FIFO is the most efficient algorithm on 10 out of the 14 datasets and among the top three most efficient algorithms on 13 datasets. As a comparison, the next best algorithm (LIRS [77]) obtains the highest efficiency on only 2 datasets.

S3-FIFO is also more scalable because FIFO queues enable lock-free implementations. We implemented a prototype in Cachelib and show that S3-FIFO achieves more than 6× higher throughput than the highly-optimized LRU implementation on 16 cores. Compared to advanced eviction algorithms such as 2Q and TinyLFU, the throughput gap is further enlarged.

The fact that filtering objects with a small FIFO queue enables better than state-of-the-art efficiency has an implication for flash cache deployments. If the small FIFO queue is in DRAM and the main FIFO queue is on flash, then most objects evicted from DRAM do not need to be written to the flash. This reduces both flash writes and miss ratio. We compare this FIFO filter with a probabilistic filter and a machine-learning-model-based filter from Flashield [55]. The FIFO filter has the lowest miss ratio and the least flash writes evaluated on two open-source CDN traces. Moreover, in contrast to the ML model that requires a large DRAM cache (10% of total cache size) to track object access information for making good decisions, the small FIFO filter excels even when the DRAM cache is only 0.1% of the total cache size.

This work makes the following contributions.

- We show that for cache workloads with skewed popularity, most objects are one-hit wonders at eviction. Therefore, quick demotion is critical for cache efficiency.
- Leveraging this observation, we designed and implemented S3-FIFO, the first FIFO-queue-only eviction algorithm with better than state-of-the-art efficiency.
- We evaluated S3-FIFO and compared with 12 state-of-the-art eviction algorithms on 6594 traces and show that S3-FIFO is more efficient, and its efficiency is also more robust.

- Our prototype in Cachelib shows that FIFO queues enable S3-FIFO to be scalable with 6× higher throughput than an optimized LRU implementation.

2 Background

Software caches are ubiquitously deployed today, e.g., inside end-user devices [81, 92], at the edge of the Internet [16, 19, 25, 32, 58, 59, 103, 109, 122, 128, 150, 151, 156], and across system stacks in a data center [52, 56, 60, 97, 108, 111, 116, 118, 125, 144, 160, 161]. While the data stored in different types of caches have different names, e.g., block, page, object, and asset, we use the term "objects" for ease of discussion.

2.1 Metrics of a cache

The heart of a cache is the eviction algorithm, which decides the objects to store in the limited space.

Efficiency. A more *efficient* (sometimes called more "effective") eviction algorithm retains more useful objects in the cache and provides a lower *miss ratio*, which measures the fraction of requests that must be fetched from the backend. While request miss ratio is the most common efficiency metric, some cache deployments aiming to reduce bandwidth usage, e.g., proxy caches, also evaluate *byte miss ratio*: the fraction of bytes that need to be fetched from the origin.

Throughput. A cache's throughput measures the number of requests it can serve per second (QPS). Having higher throughput reduces the number of CPU cores required to serve a workload.

Scalability. Modern CPUs have a large number of cores. For example, AMD EPYC 9654P has 192 cores [13]. A cache's scalability measures how its throughput increases with the number of CPU cores. Ideally, a cache's throughput would scale linearly with the number of CPU cores. However, in many eviction algorithms, read operations necessitate metadata updates under locking. Therefore, they cannot fully harness the computation power of modern CPUs.

Flash writes. While DRAM is the most commonly used storage medium for caching, many systems today also use flash for its higher density, lower price, and lower power consumption. Flash lifetime becomes a critical metric when using flash for caching because flash only supports a limited number of writes [12, 27, 98, 129]. Moreover, small random writes on flash cause device-level write amplification, which not only reduces the flash lifetime but also increases read and write tail latency [63, 64, 88, 152]. To achieve a more manageable flash lifetime, most production flash cache systems, e.g., Apache Trafficserver [14], Memcached Extstore [101], Cachelib large object cache [24], and Google Colossus flash cache [159], use FIFO or FIFO-reinsertion. Besides the flash eviction algorithm, many systems also employ admission algorithms, e.g., bloom filter or machine-learning-based algorithms, to select "good" data to write to flash [36, 55].

Simplicity and generality. A cache eviction algorithm's complexity and generality are two additional factors that play a critical role in its adoption. While the complexity is often inversely correlate correlated with throughput and scalability, a simple design can offer benefits beyond just improved performance metrics, such as fewer bugs and reduced maintenance overhead. Linux Kernel developers stated that "Predicting which pages will be accessed in the near future is a tricky task. The kernel not only often gets it wrong, but it also wastes a lot of CPU time to make the incorrect choice" [9].

In addition to simplicity, generality is also crucial for similar reasons. If the same data structure and eviction algorithm can be used for different types of caches and storage mediums, it can help reduce the development and maintenance overheads. A similar argument can also be found in previous work from Meta [24].

2.2 Prevalence of LRU-based cache

Cache workloads exhibit temporal locality: recently accessed data are more likely to be re-accessed. Therefore, Least-Recently-Used (LRU) is more efficient than FIFO and is widely used in DRAM caches [24, 28, 102, 133]. Moreover, advanced eviction algorithms designed to improve efficiency are mostly built upon LRU. For example, ARC [100], SLRU [80], 2Q [79], EELRU [124], LIRS [77], TinyLFU [54], LeCaR [132], and CACHEUS [119] all use one or more LRU queues to order objects.

Albeit efficient, LRU and LRU-based algorithms have three problems. First, LRU is often implemented using a doubly-linked list, requiring two pointers per object, which becomes a large overhead when the object is small. As a result, Twitter and Meta have designed specialized compact caches for workloads having small objects [24, 48, 158].

Second, LRU promotes objects to the head of the queue (called promotion) upon each cache hit, which performs at least six random memory accesses protected by a lock, significantly limiting the cache's scalability [60, 112]. For example, the RocksDB developers "confess" that the LRU caches in RocksDB are the scalability bottleneck [50]. Therefore, a new cache using CLOCK [45] eviction has been implemented to address this problem in 2022 [117].

Third, LRU is not flash-friendly. The object eviction order in LRU is different from the insertion order, which leads to random writes on flash, and reduces flash lifetime.

3 Motivation

While the last few decades of eviction algorithm study are centered around LRU, we believe modern eviction algorithms should be designed with FIFO queues instead of LRU queues. FIFO can be implemented using a ring buffer without per-object pointer metadata, and it does not promote an object upon each cache hit, thus removing the scalability bottleneck.
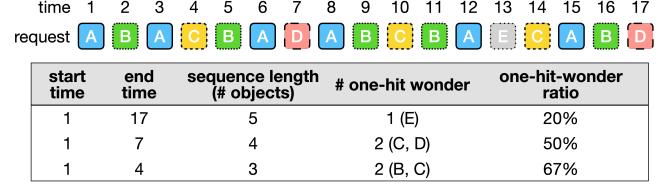

start time	end time	sequence length (# objects)	# one-hit wonder	one-hit-wonder ratio
1	17	5	1 (E)	20%
1	7	4	2 (C, D)	50%
1	4	3	2 (B, C)	67%

Figure 1. A shorter sequence has a higher one-hit-wonder ratio.

However, FIFO falls behind LRU and state-of-the-art eviction algorithms in efficiency.

What does FIFO need? The primary limitation of FIFO is its inability to retain frequently accessed objects, so the most straightforward improvement is to insert these objects back. FIFO-Reinsertion [1] is an algorithm that keeps track of object access and reinserts accessed objects during eviction. Compared to LRU, FIFO-Reinsertion incurs a lower overhead on a cache hit, requiring no operation or just an atomic set for the first request to an object. However, reinsertion alone is insufficient, and FIFO-Reinsertion still lags behind state-of-the-art eviction algorithms on efficiency (§5.2).

Our insight is that a cache experiences more one-hit wonders (objects having no access after insertion) than what common full trace analyses suggest [96, 141], highlighting the importance of swiftly removing most new objects. Specifically, we observe a median one-hit-wonder ratio of 26% across 6594 production traces. However, for a random request sequence containing 10% of unique objects in the trace, 72% of the objects have only one request in the sequence.

3.1 More one-hit wonders than expected

The term "one-hit-wonder ratio" measures the fraction of objects that are requested only once in a trace. It is commonly used in content delivery networks (CDNs) due to large one-hit-wonder ratios [19, 96].

Although the one-hit-wonder ratio varies between different types of cache workloads, we find that shorter request sequences (consisting of fewer unique objects) often have higher one-hit-wonder ratios. In the subsequent analysis, *we measure sequence length using the number of unique objects.*

Fig. 1 illustrates this observation using a toy example. The request sequence comprises seventeen requests for five objects, out of which one object (E) is accessed once. Thus, the one-hit-wonder ratio for the sequence is 20%. Considering a shorter sequence from the 1st to the 7th request, two (C, D) of the four unique objects are requested only once, which leads to a one-hit-wonder of 50%. Similarly, the one-hit-wonder ratio of a shorter sequence from the 1st to 4th request is 67%. More formally, we make the following observation.

Observation. *Assume that the object popularity of a request sequence follows the Zipf distribution with the least popular*

---

[1]FIFO-Reinsertion, Second chance, and CLOCK are different implementations of the same algorithm.

(a) Zipf trace, linear-scale **(b)** Zipf trace, log-scale **(c)** Production traces, linear-scale **(d)** Production traces, log-scale
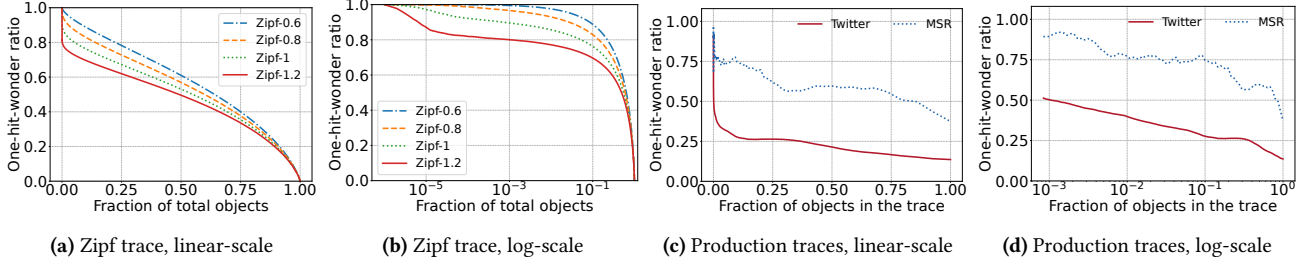
Figure 2. Left two: the one-hit-wonder ratio decreases with sequence length (as a fraction of the unique objects in the full sequence) for synthetic Zipf traces. Different curves show different skewness α. We plot both linear and log-scale X-axis for ease of reading. Right two: Production traces show similar observations. Note that the X-axis shows the fraction of objects in the trace, much smaller than the number of possible objects in the backend. Therefore, the production curves capture the left region of the Zipf curves.

object having one request, and there are M unique objects in total. Then the one-hit-wonder ratio of the complete sequence is $\frac{1}{M}$. For any sub-sequence ending with a one-hit wonder, if the sub-sequence contains C unique objects, the expected one-hit-wonder ratio $\mathcal{F}(x = C)$ monotonically decreases with the sequence length x measured in the number of objects.

The intuition is that most objects are unpopular (rank larger than $C + 1$ in Zipf distribution for a cache of size C) and have an expected number of requests between 0 and 1. If they show up in the sub-sequence, it is very likely that they will not get another request within the sub-sequence.

This setting can be viewed as a variant of the coupon collector problem where we have M unique coupons in total, and the probability of collecting coupon i follows the Zipf distribution. We would like to know the number of coupons we have collected only once when we have C unique coupons.

We use Monte Carlo simulations to find how $\mathcal{F}(x)$ changes with the sequence length x (measured in the number of objects). We first generate Zipf request traces of different skewness α under independent reference model [38], then take random sub-sequences and measure the one-hit-wonder ratios. We repeat 100 times and report the mean. The results are plotted in Fig. 2a and Fig. 2b. We show both linear and log-scale X axis for clarity. The one-hit-wonder ratio decreases with increasing sequence length. Between different curves, more skewed workloads exhibit lower one-hit-wonder ratios at the same sequence length because unpopular objects have a lower probability of appearing in more skewed workloads.

We have also performed the same measurement on production traces. Fig. 2c and Fig. 2d show a block trace (MSR hm_0) and a web trace from Twitter (cluster 52). The curves look different from the Zipf curves at first glance. This is because the production traces are not long enough to capture all objects in the backend systems, and it is not possible to know the total number of objects that can be requested. As a result, the X-axis shows the fraction of objects in the trace. Therefore, the production curves only capture the left region of the synthetic curves, and we observe that they match the synthetic curves. For example, when comparing Fig. 2a and
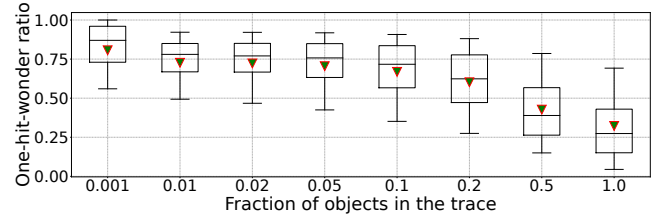

Figure 3. The one-hit-wonder ratio across 6594 traces (Table 1). The whiskers show P10 and P90, and the triangle shows the mean.

Fig. 2c, we see curves in both figures have steep drops at the beginning before slowing down. Moreover, the Twitter trace is known to be more skewed [157], and it shows a larger drop than the MSR trace, which matches the observation on the Zipf traces. Compared to the one-hit-wonder ratio of the full trace at 13% (Twitter) and 38% (MSR), a random sub-sequence containing 10% objects shows a one-hit-wonder ratio of 26% on the Twitter trace and 75% on the MSR trace. The increase is more significant when the sequence length is further reduced.

We further evaluated 6594 production traces (more details in Table 1). Fig. 3 shows the one-hit-wonder ratios of all traces at different sequence lengths. Compared to the full traces with a median one-hit-wonder ratio of 26%, sequences containing 50% of the objects in the trace show a median one-hit-wonder ratio of 38%. Moreover, sequences with 10% and 1% of the objects exhibit one-hit-wonder ratios of 72% and 78%, respectively.

Because the cache size is always much smaller than the trace footprint (the number of objects in the trace), evictions start after encountering a short sequence of requests. This observation suggests that if the cache size is set as 10% or 1% of the trace footprint, approximately 72% and 78% of the objects would not be reused before eviction.

We further corroborate the observation with cache simulations. Fig. 4 shows the distribution of object frequency at eviction. Our trace analysis (Fig. 2d) shows that the Twitter trace has a 26% one-hit-wonder ratio for sequences of
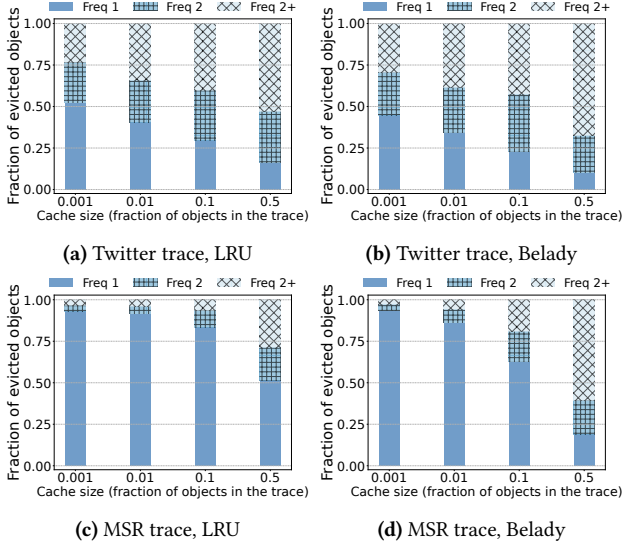
(a) Twitter trace, LRU

(b) Twitter trace, Belady

(c) MSR trace, LRU

(d) MSR trace, Belady

Figure 4. The frequency of objects at eviction.

Insert: if not in ghost, insert to small ①a, else insert to main ①b
Evict (small): if not visited, insert to ghost ②a, else insert to main ②b
Evict (main): if visited, insert back ③a, else evict ③b

Figure 5. An illustration of S3-FIFO.

10% trace length. The simulation shows a similar result: 26% and 24% of the objects evicted by LRU and Belady are not requested after insertion at a cache size of 10% of the trace footprint. Similarly, the MSR trace exhibits a higher one-hit-wonder ratio of 75% for sequences of 10% trace length (Fig. 2d), and Fig. 4 shows that 82% and 68% of the objects evicted by LRU and Belady have no reuse. *This suggests that these one-hit wonders are often good eviction candidates, and one may not need highly sophisticated eviction algorithms.*

3.2 The need for quick demotion

Based on the observation, a cache should filter out these one-hit wonders because they occupy space without providing benefits. It is a common practice to employ Bloom Filters to reject one-hit wonders from entering the cache in CDNs [96, 141]. However, Bloom Filter rejects objects too fast with a lack of precision since it rejects all objects which have not been seen before. It causes the second requests to all objects to be cache misses, which leads to mediocre efficiency (§5.2).

Filtering out one-hit wonders bears some resemblance to designing scan-resistant cache eviction algorithms, as objects requested during a scan are often one-hit wonders. Researchers have developed a variety of algorithms for storage workloads that can avoid cache pollution and thrashing caused by scanning requests, e.g., ARC [100], LRU-K [110], 2Q [79], EELRU [124], LIRS [77], LeCaR [132], CACHEUS [119], and LHD [21]. However, existing algorithms cannot guarantee the minimum and maximum time one-hit wonders stay in the cache before being removed. We find these algorithms sometimes evict too fast or too slowly, and their complexities make it difficult to reason about the behavior (§6.1).
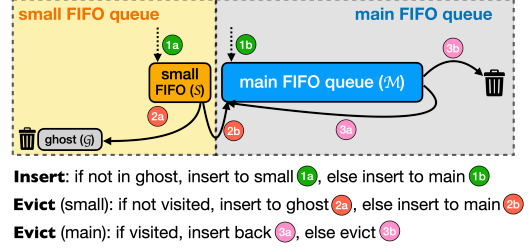
This raises the question: can we simply use a small probationary FIFO queue to guarantee that one-hit wonders are removed after a fixed number of objects are inserted?

4 Design and implementation

As mentioned in §2.1, a cache eviction algorithm needs to be simple and scalable besides being efficient. This section presents S3-FIFO, a **s**imple and **s**calable eviction algorithm that consists of only **s**tatic FIFO queues.

We start by defining the LRU queue and FIFO queue. An *LRU queue* updates object ordering during cache hits by promoting the requested object to the head of the queue. A *FIFO queue* does not update ordering during cache hits, and objects are evicted in the insertion order. However, evicted objects may be reinserted into the queue to preserve hot objects. As mentioned in §2.2, most eviction algorithms are built with LRU queue, and only a few algorithms, e.g., FIFO-Reinsertion, use FIFO queue because conventional wisdom suggests LRU queue can provide a lower miss ratio.

4.1 S3-FIFO design

S3-FIFO uses three FIFO queues: a small FIFO queue ($\mathcal{S}$), a main FIFO queue ($\mathcal{M}$), and a ghost FIFO queue ($\mathcal{G}$). We choose $\mathcal{S}$ to use 10% of the cache space based on experiments with 10 traces and find that 10% generalizes well. $\mathcal{M}$ then uses 90% of the cache space. The ghost queue $\mathcal{G}$ stores the same number of ghost entries (no data) as $\mathcal{M}$.

Cache read. S3-FIFO uses two bits per object to track object access status [155] similar to a capped counter with frequency up to 3. Cache hits in S3-FIFO increment the counter by one atomically. Note that most requests for popular objects require no update.

Cache write. New objects are inserted into $\mathcal{S}$ if not in $\mathcal{G}$. Otherwise, it is inserted into $\mathcal{M}$. When $\mathcal{S}$ is full, the object at the tail is either moved to $\mathcal{M}$ if it is accessed more than once or $\mathcal{G}$ if not. And its access bits are cleared during the move. When $\mathcal{G}$ is full, it evicts objects in FIFO order. $\mathcal{M}$ uses an algorithm similar to FIFO-Reinsertion but tracks access information using two bits. Objects that have been accessed at least once are reinserted with one bit set to 0 (similar to decreasing frequency by 1). We illustrate the algorithm in Fig. 5 and the pseudo-code in Algo. 1.

Algorithm 1 S3-FIFO algorithm

Input: The requested object x, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$

 1: **function** READ(x)
 2:     **if** x in S or x in M **then**   ▷ Cache Hit
 3:         x.freq ← min(x.freq + 1, 3)   ▷ Frequency is capped to 3
 4:     **else**   ▷ Cache Miss
 5:         insert(x)
 6:         x.freq ← 0

 7: **function** INSERT(x)
 8:     **if** cache is full **then**
 9:         evictS()
10:     **if** x in $\mathcal{G}$ **then**
11:         insert x to head of $\mathcal{M}$
12:     **else**
13:         insert x to head of $\mathcal{S}$

14: **function** EVICTS
15:     evicted ← false
16:     **while** not evicted and $\mathcal{S}$.size > 0 **do**
17:         t ← tail of $\mathcal{S}$
18:         **if** t.freq > 1 **then**
19:             insert t to $\mathcal{M}$
20:             **if** $\mathcal{M}$ is full **then**
21:                 evictM()
22:         **else**
23:             insert t to $\mathcal{G}$
24:             evicted ← true
25:         remove t from $\mathcal{S}$

26: **function** EVICTM
27:     evicted ← false
28:     **while** not evicted and $\mathcal{M}$.size > 0 **do**
29:         t ← tail of $\mathcal{M}$
30:         **if** t.freq > 0 **then**
31:             Insert t to head of $\mathcal{M}$
32:             t.freq ← t.freq-1
33:         **else**
34:             remove t from $\mathcal{S}$
35:             evicted ← true

Handle different access patterns. One important pattern we identified in §3.1 is the large one-hit-wonder ratio a cache experiences due to the limited cache space. The small FIFO queue $\mathcal{S}$ can quickly evict these one-hit wonders so they do not occupy the cache for a long time. This allows S3-FIFO to save the precious cache space for more valuable objects. Besides one-hit wonders caused by unpopular objects in skewed cache workloads, many block cache workloads have scan and loop access patterns. Like one-hit wonders, blocks accessed during scans are quickly removed to avoid cache pollution and thrashing. However, blocks not part of a scan but mixed in the scan are also moved to $\mathcal{G}$ in this process. Nevertheless, when these "good" blocks are requested again in the near future, they will be inserted into $\mathcal{M}$ and stay for a longer time.

4.2 Implementation

The FIFO queues can be implemented either using linked lists or ring buffers. Linked-list-based implementation can be added to existing LRU-based caches more easily. However, it has three drawbacks. First, it uses two pointers per object. On workloads with tiny objects [99, 158], this poses a huge storage overhead. Second, traversing through the queue requires random memory accesses. Third, eviction and insertion in linked-list-based implementation require expensive atomic operations: `compare-and-set`, which reduces the scalability.

In contrast, a ring-buffer-based implementation has less overhead and is more scalable but may not be compatible with existing LRU-based caching systems. When using a ring buffer to implement S3-FIFO, the ring buffer maintains the FIFO order, with each slot storing the object or a pointer. Eviction requires bumping the tail pointer in the ring buffer. Although more scalable with lower storage overhead, a ring-buffer-based implementation wastes space when the workload contains many deletion operations because the space of deleted objects cannot be reused until eviction. We empirically observed that deletions often arrive soon after insertions in many workloads, which makes S3-FIFO more suitable than traditional ring-buffer-based algorithms, e.g., CLOCK. This is because the space of deleted objects can be reused sooner if the deleted object is in the small FIFO queue.

Although we present S3-FIFO with three FIFO queues, it can also be implemented with one or two FIFO queue(s). Because objects evicted from $\mathcal{S}$ may enter $\mathcal{M}$, they can be implemented using one queue with a pointer pointed at the 10% mark. However, combining $\mathcal{S}$ and $\mathcal{M}$ reduces scalability because removing objects from the middle of the queue requires locking.

The ghost FIFO queue $\mathcal{G}$ can be implemented as part of the indexing structure. For example, we can store the fingerprint and eviction time of ghost entries in a bucket-based hash table [33, 37, 93, 158]. The fingerprint stores a hash of the object using 4 bytes, and the eviction time is a timestamp measured in the number of objects inserted into $\mathcal{G}$. Let $S_{\mathcal{G}}$ denote the size of the ghost queue. If $\mathcal{G}$ has N insertions, all the entries inserted before $N - S_{\mathcal{G}}$ are no longer part of $\mathcal{G}$ because of the FIFO queue.

The ghost entries stay in the hash table until they are no longer in the ghost queue. When an entry is evicted from the ghost queue, it is not immediately removed from the hash table. Instead, the hash table entry is removed during hash collision — when the slot is needed to store other entries.

4.3 Overhead analysis

4.3.1 Computation S3-FIFO performs an atomic write upon the first and second request to an object without locking. There is no operation after the second request. Because

Table 1. Datasets used in this work, the ones with no citation are proprietary datasets. For old datasets, we exclude traces with less than 1 million requests. The trace length used in measuring the one-hit-wonder ratio is measured in the fraction of objects in the trace.

Trace collections	Approx time	Cache type	time span (days)	# Traces	# Request (million)	Request (TB)	# Object (million)	Object (TB)	One-hit-wonder ratio full trace	10%	1%
MSR [104, 105]	2007	Block	7	13	410	10	74	3	0.56	0.74	0.86
FIU [83]	2008-11	Block	9-28	9	514	1.7	20	0.057	0.28	0.91	0.91
Cloudphysics [136]	2015	Block	7	106	2,114	82	492	22	0.40	0.71	0.80
CDN 1	2018	Object	7	219	3,728	3640	298	258	0.42	0.58	0.70
Tencent Photo [167, 168]	2018	Object	8	2	5,650	141	1,038	24	0.55	0.66	0.74
WikiMedia CDN [140]	2019	Object	7	3	2,863	200	56	13	0.46	0.60	0.80
Systor [84, 85]	2017	Block	26	6	3,694	88	421	15	0.37	0.80	0.94
Tencent CBS [163, 164]	2020	Block	8	4030	33,690	1091	551	66	0.25	0.73	0.77
Alibaba [2, 89, 139]	2020	Block	30	652	19,676	664	1702	117	0.36	0.68	0.81
Twitter [157]	2020	KV	7	54	195,441	106	10,650	6	0.19	0.32	0.42
Social Network 1	2020	KV	7	219	549,784	392	42,898	9	0.17	0.28	0.37
CDN 2	2021	Object	7	1273	37,460	4,925	2,652	1,581	0.49	0.58	0.64
Meta KV [11]	2022	KV	1	5	1,644	958	82	76	0.51	0.53	0.61
Meta CDN [11]	2023	Object	7	3	231	8,800	76	1,563	0.61	0.76	0.81

most requests are for popular objects (more than two requests), S3-FIFO thus performs negligible metadata updates on cache hits. Cache miss requires evicting an object from S or M. Evicting from S requires inserting the tail object into M or G. And evicting from M may involve reinserting the tail object back to M. However, if an object is not accessed, it requires no reinsertion. Therefore, the number of reinsertions is much smaller than the cache hits in practice. Moreover, removing the tail object and inserting an object to the head of a queue can be implemented lock-free using atomic operations.

4.3.2 Storage The ghost queue G stores the same number of objects (without data) as the main queue. Assuming the mean object size is 4 KB, and an object id uses 4 bytes, then G uses 0.09% of the cache size. Each cached object uses two bits to track access, consuming less than 0.01% of the cache size. Moreover, the two bits can often be piggybacked on unused bits in object metadata. If the FIFO queues are implemented using ring buffers, S3-FIFO can remove the two LRU pointers, saving 16 bytes per object or 0.4% of the cache size.

5 Evaluation

In this section, we evaluate S3-FIFO to answer the following questions.
- How does S3-FIFO's efficiency compare with the state-of-the-art eviction algorithms?
- Is S3-FIFO more scalable compared to state-of-the-art?
- Can lessons learned from S3-FIFO help flash cache design?

5.1 Evaluation setup

5.1.1 Traces We evaluated S3-FIFO using a large collection of 6594 production traces from 14 datasets, including 11 open-source and 3 proprietary datasets. These traces span from 2007 to 2023 and cover key-value, block, and object CDN caches. In total, the datasets contain 856 billion requests to 61 billion objects, 21,088 TB traffic for total 3,753 TB of data. Because many large-scale distributed caching systems are multi-tenanted and the traces represent workloads served by more than one server, we split four datasets (CDN 1, CDN 2, Tencent CBS, and Alibaba) with tenant information into per-tenant traces for an in-depth study of the workloads. More details of the datasets can be found in Table 1.

5.1.2 Simulator We implemented S3-FIFO and the state-of-the-art eviction algorithms (described in §5.2) in libCacheSim [6]. We referenced and verified the results with multiple open-source simulator implementations [1, 3–5, 7, 8]. For all state-of-the-art algorithms, we used the parameters described in the original papers. LibCacheSim is designed and tuned for high-throughput cache simulations and can process up to 20 million requests on a single CPU core.

We have also implemented a distributed fault-tolerant computation platform that allows us to run thousands of simulations in parallel. The platform's design does not affect simulation accuracy and is out of the scope of this work. We describe it in a separate blog post [2].

This distributed computation platform and the Cloudlab testbed [53] enable us to evaluate different algorithms and cache sizes on our large datasets (Table 1). The simulation processed the datasets in close to 100 passes using different algorithms, cache sizes, and parameters. We estimated that over 80,000 billion requests were processed using a million CPU core · hours.

Unless otherwise mentioned, we ignore object size in the simulator because most production systems use slab storage for memory management, for which evictions are performed within the same slab class (objects of similar sizes). However, we remark that supporting object size is non-trivial for systems that do not use slab-based memory management. Moreover, we do not consider the metadata size in different
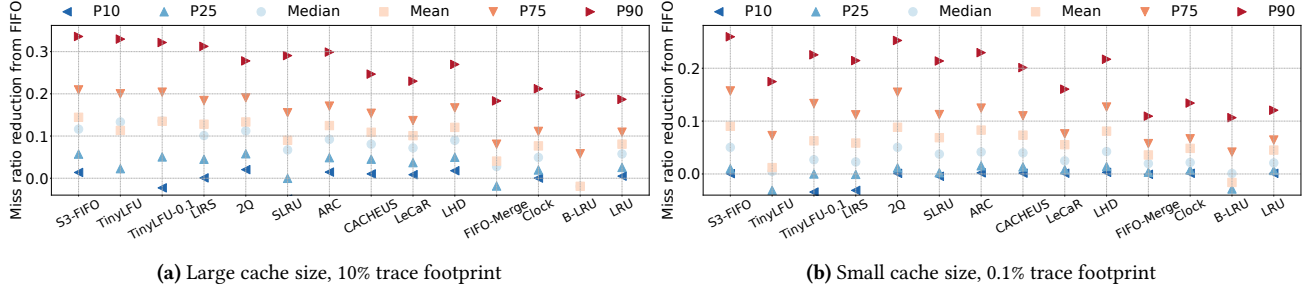
(a) Large cache size, 10% trace footprint

(b) Small cache size, 0.1% trace footprint

Figure 6. Each algorithm's miss ratio reduction (from FIFO) at different percentiles across all traces. A larger reduction is better.

algorithms, although S3-FIFO requires fewer metadata. We evaluated the algorithms at multiple different cache sizes, and we present one large size using 10% of the trace footprint (number of objects in the trace) and one small size at 0.1% of the trace footprint. At 0.1% trace footprint, the cache size may be too small for some traces, so we ignore a trace if the cache size is smaller than 1000 objects. For byte miss ratio evaluation, we considered object size and used the trace footprint in bytes instead of objects.

Because the large number of traces used in the evaluation have a very wide range of miss ratios, we choose to present the miss ratio reduction compared to FIFO: $\frac{MR_{fifo}-MR_{algo}}{MR_{fifo}}$ where MR stands for miss ratio. If an algorithm has a miss ratio higher than FIFO, we calculate FIFO's miss ratio reduction compared to the algorithm and take the negative value: $-\frac{MR_{algo}-MR_{fifo}}{MR_{algo}}$, which bounds the value between -1 and 1. This avoids the impact of outliers on the mean value.

5.1.3 Prototype We have implemented S3-FIFO in Cachelib [47]. Cachelib uses slab memory management, which pre-allocates all memory during initialization and is highly optimized for LRU-based eviction algorithms. Its extensive usage of metaprogramming and many LRU-based optimizations (e.g., compressed pointers) tightly couple different components. Therefore, we implemented $\mathcal{S}$ and $\mathcal{M}$ using linked lists and $\mathcal{G}$ using a hash table. We implemented a trace replay tool that replays traces in a closed loop for benchmarking. Because the backend often decides the latency and throughput of cache misses, we focus on the cache hit performance and on-demand fill cache misses using pre-generated data. We compared S3-FIFO with three algorithms implemented by Cachelib developers: LRU, a variant of 2Q, and TinyLFU. Cachelib developers have devoted huge efforts to improving the throughput and scalability of the three algorithms with techniques such as lock combining, delayed LRU promotion, try-lock-based promotion, and compressed pointers. Besides Cachelib, we also evaluated Segcache, the state-of-the-art scalable key-value cache using open-source code [158].

5.1.4 Open source We have open-sourced the code and data with more information at the end of the paper.

5.1.5 Evaluation setup We performed all evaluations on Cloudlab [53]. The simulations used multiple types of nodes from the Clemson site, depending on node availability. The prototype evaluation used c6420 nodes from the Clemson site. We turned off turbo boost, pinned one thread to one core, and used `numactl` to allocate all memory pages on the same NUMA node.

5.2 Efficiency (miss ratio)

5.2.1 Miss ratio The primary criticism of the FIFO-based eviction algorithms is their efficiency, the most important metric for a cache. We compare S3-FIFO with state-of-the-art eviction algorithms designed in the past few decades. The algorithms used in the comparison are either deployed in production or commonly used in other papers. We use all efficiency results from simulation because it allows us to (1) study different types of cache workloads, e.g., block, key-value, and object, (2) focus on and isolate the impact of the eviction algorithm, and (3) requires fewer computation resources to scale up to evaluate the huge datasets. Fig. 6 shows the (request) miss ratio reduction (compared to FIFO) of different algorithms across traces. At the large cache size, S3-FIFO has the largest reductions across almost all percentiles than other algorithms. For example, S3-FIFO reduces miss ratios by more than 32% on 10% of the traces (P90) with a mean of 14% on the large cache size.

TinyLFU [54] is the closest competitor. TinyLFU uses a 1% LRU window to filter out unpopular objects and stores most objects in a SLRU cache. TinyLFU's good performance corroborates our observation that quick demotion is critical for efficiency. However, TinyLFU does not work well for all traces, with miss ratios being lower than FIFO on almost 20% of the traces (the P10 point is below -0.05 and not shown in the figure). This phenomenon is more pronounced when the cache size is small, where TinyLFU is worse than FIFO on close to 50% of the traces.

There are two reasons why TinyLFU falls short. First, the 1% window LRU is too small, evicting objects too fast. Therefore, increasing the window size to 10% of the cache size (TinyLFU-0.1) significantly improves the efficiency at the tail (bottom of the figure). However, increasing the window size reduces its improvement on the best-performing traces

(Fig. 6a). Second, when the cache is full, TinyLFU compares the least recently used entry from the window LRU and main SLRU, then evicts the less frequently used one. This allows TinyLFU to be more adaptive to different workloads. However, if the tail object in the SLRU happens to have a very high frequency, it may lead to the eviction of an excessive number of new and potentially useful objects.

LIRS [77] uses LRU stack (reuse) distance as the metric to choose eviction candidates. Because one-hit wonders do not have reuse distance, LIRS utilizes a 1% queue to hold them. This small queue performs quick demotion and is the secret source of LIRS's high efficiency. Similar to TinyLFU, the queue is too small, and it falls short on some cache workloads. However, compared to TinyLFU, fewer traces show higher-than-FIFO miss ratios because the inter-recency metric in LIRS is more robust than the frequency in TinyLFU. In particular, TinyLFU cannot distinguish between many objects with the same low frequency (e.g., 2), but these objects will have different inter-recency values. The downside is that LIRS requires a more complex implementation than TinyLFU.

2Q [79] has the most similar design to S3-FIFO. It uses 25% cache space for a FIFO queue, the rest for an LRU queue, and also has a ghost queue. Besides the difference in queue size and type, objects evicted from the small queue are *not* inserted into the LRU queue. Having a large probationary queue and not moving accessed objects into the LRU queue are the primary reasons why 2Q is not as good as S3-FIFO. Moreover, the LRU queue does not provide observable benefits compared to the FIFO queue (with reinsertion) in S3-FIFO.

SLRU [67, 80] uses four equal-sized LRU queues. Objects are first inserted into the lowest-level LRU queue and promoted to higher-level queues upon cache hits. An inserted object is evicted if not reused in the lowest LRU queue, which performs quick demotion and allows SLRU to show good efficiency. However, unlike other schemes, SLRU does not use a ghost queue, making it not scan-tolerant because popular objects mixed in the scan cannot be distinguished. Therefore, we observe that SLRU performs poorly on many block cache workloads (not shown).

ARC uses four LRU queues: two for data and two for ghost entries. The two data queues are used to separate recent and frequent objects. Cache hits on objects in the recency queue promote the objects to the frequency queue. Objects evicted from the two data queues enter the corresponding ghost queue. The sizes of queues are adaptively adjusted based on hits on the ghost queues. When the recency queue is small, newly inserted objects are quickly evicted, enabling ARC's high efficiency. However, ARC is less efficient than S3-FIFO because the adaptive algorithm is not sufficient. We discuss more in §6.2.

Recent algorithms, including CACHEUS [119], LeCaR [132], LHD [21], and FIFO-Merge [158], are also evaluated. However, we find these algorithms are often less competitive than the traditional ones. In particular, FIFO-merge was designed for log-structured storage and key-value cache workloads without scan resistance. Therefore, similar to SLRU, it performs better on web cache workloads but much worse on block cache workloads.

Common algorithms, such as B-LRU (Bloom Filter LRU), CLOCK, and LRU, are weaker than the ones discussed. CLOCK and LRU do not allow quick demotion, so their miss ratio reductions are small. B-LRU rejects all one-hit wonders at the cost of the second request for all objects being cache misses. Because of these misses, B-LRU is worse than LRU in most cases. Because an object's second request often arrives *soon* after the first request (temporal locality), the small FIFO queue in S3-FIFO allows these requests to be served as cache hits.

Adversarial workloads for S3-FIFO. We studied the limited number of traces on which S3-FIFO performed poorly and identified one pattern. Most objects in these traces are accessed only twice, and the second request falls out of the small FIFO queue S, which causes the second request to these objects to be cache misses. We remark that these workloads are adversarial for most algorithms that partition the cache space, e.g., TinyLFU, LIRS, 2Q, and CACHEUS. Because the partition for newly inserted objects is smaller than the cache size, it is possible that the second request is a cache hit in LRU and FIFO, but not in these advanced algorithms.

This request pattern resembles a scan because most objects are not requested very soon after the first request. However, it is not a typical scan because any object may show this pattern, and the objects showing this pattern may not be requested consecutively. In our large datasets, we find that the second request often arrives within one minute in these workloads. Therefore, the second request being a miss is a problem only when the cache size is very small, e.g., 1000s of objects. Moreover, using an adaptive algorithm to adjust the queue size can often mitigate the problem, and we discuss more in §6.2.

5.2.2 Miss ratio per dataset We have shown the results across all 6594 traces. However, the number of traces from each dataset differs, and the result could be affected by the dominating dataset. Fig. 7 shows the mean miss ratio reduction on each dataset using selected algorithms. We observe that S3-FIFO often outperforms all other algorithms by a large margin. Moreover, it is the best algorithm on 10 out of the 14 datasets using a large cache size and 7 out of the datasets using a small cache size. As a comparison, no other algorithm is the best on more than 3 datasets.

Besides being the best on most datasets, S3-FIFO is also more robust than other algorithms — S3-FIFO is among the top three most efficient algorithms on 13 of the 14 datasets at the large cache size. As a comparison, TinyLFU and LIRS are among the top algorithms on some datasets, but on other datasets, they are among the worst algorithms. While it is
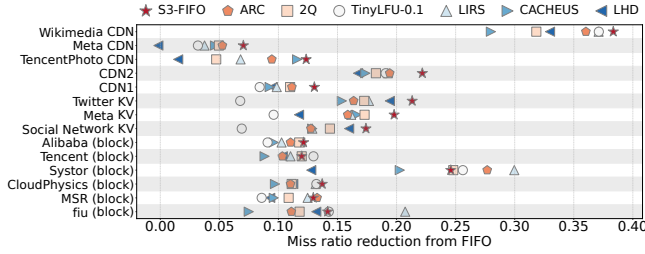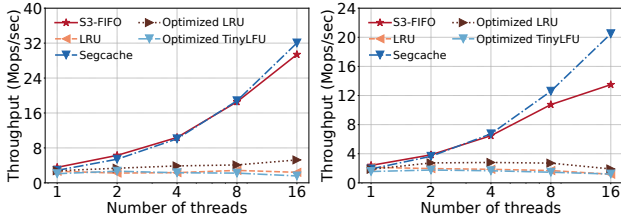
(a) Large cache size
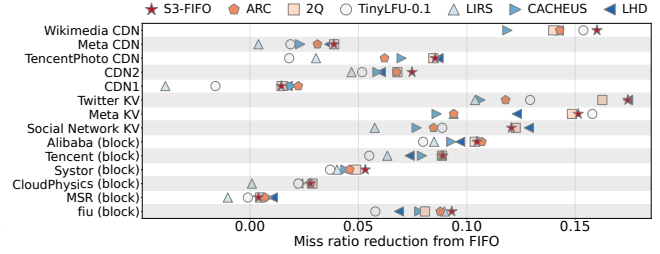
(b) Small cache size

Figure 7. The mean miss ratio reduction on each dataset. TinyLFU on the TencentPhoto dataset at the large size is -0.11 and not shown.

(a) Large cache, LRU miss ratio 0.02 (b) Small cache, LRU miss ratio 0.21

Figure 8. Throughput scaling with CPU cores on synthetic Zipf ($\alpha = 1.0$) trace.

hard to explain why S3-FIFO is more robust, we conjecture that simplicity contributes to its robustness. In conclusion, we find that quick demotion is a key factor for an efficient eviction algorithm. By leveraging this observation, S3-FIFO, a simple algorithm with only FIFO queues, can outperform state-of-the-art.

5.2.3 Byte miss ratio While (request) miss ratio is important for most cache deployments, CDNs also widely use byte miss ratio to measure bandwidth reduction. We evaluated the same set of eviction algorithms on byte miss ratio. We used the object sizes from each trace and set the cache size to 10% and 0.1% of trace footprint in bytes. The results (not shown due to space limit) are not significantly different from the miss ratio in Fig. 6. Compared to other algorithms, S3-FIFO presents larger byte miss ratio reductions at almost all percentiles. We have also compared S3-FIFO with LRB [126] on ten traces (LRB took too long to run on the full dataset), including the Wikimedia traces used in LRB's evaluation. We observe that S3-FIFO and LRB have similar efficiency, although S3-FIFO is much simpler than LRB.

5.3 Performance (throughput)

S3-FIFO consists of only FIFO queues without locking on either read or write. As a comparison, LRU-based eviction algorithms, such as LRU, 2Q, and TinyLFU, require locking on both cache hits and cache misses. We implemented S3-FIFO in Cachelib to compare the throughput of different algorithms. Because prototype experiments run much longer and cannot be run in parallel, we only evaluated using a

synthetic Zipf trace similar to previous work [57]. Moreover, we verified that the miss ratio results from the prototype are consistent with the simulator using a few randomly selected traces. The Zipf workload contains $100 \cdot n_{thread}$ million requests for n_{thread} million 4 KB objects. Fig. 8 shows that compared to (strict) LRU, the optimized LRU has both higher throughput and better scalability. However, it cannot scale beyond two cores. Compared to LRU, TinyLFU needs to check and update the count-min sketch on cache hits and move objects between the window LRU and the main SLRU on cache misses. Therefore, we observe a lower throughput than LRU due to the extra operations. The optimized 2Q in Cachelib has a similar result (not shown).

Compared to LRU-based eviction algorithms, S3-FIFO performs fewer operations during cache hits, with a higher throughput on a single thread. Moreover, because of the lock-free implementation, its throughput scales with the number of CPU cores. Under both small and large cache sizes, S3-FIFO runs more than 6× faster than the optimized LRU in Cachelib with 16 threads.

Segcache [158] is the state-of-the-art key-value cache using log-structured storage with the FIFO-Merge eviction algorithm. It uses macro management and FIFO-based eviction to achieve close-to-linear scalability. The macro management enables Segcache to perform much fewer synchronization — Segcache needs atomic updates only when a segment-chain is changed, which is 100-1000× less frequent than cache misses. However, Segcache is slower than S3-FIFO on a single thread because the merge-based eviction needs to copy data. Moreover, the eviction algorithm cannot perform quick demotion and is not scan-resistant. We have shown in Fig. 6 that FIFO-Merge can only achieve efficiency similar to LRU.

5.4 Flash-friendliness

In many flash cache deployments, the flash stores all the cached objects, and DRAM is used for hot objects (and index) [14, 25]. However, writing all data to the flash reduces its lifetime.

The surprising finding that using a small FIFO queue to perform quick demotion can achieve the state-of-the-art miss ratio has an implication for flash cache design. Because most
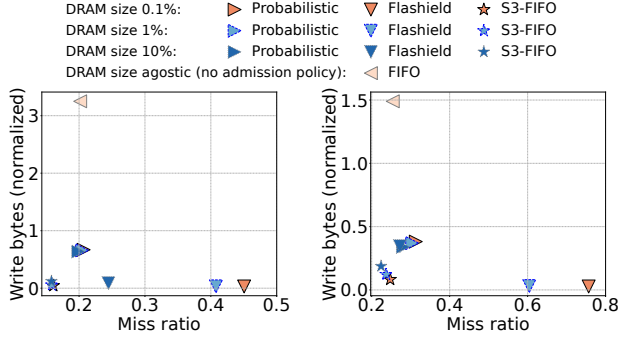
Figure 9. The write bytes and miss ratio of no admission control and using different admission algorithms. Both metrics are better when they are lower. Write bytes are normalized to the number of bytes in the trace. Left: Wikimedia CDN trace, right: Tencent Photo CDN trace.

objects evicted from the $\mathcal{S}$ are not worthwhile to be kept in $\mathcal{M}$, we can place $\mathcal{S}$ in DRAM and $\mathcal{M}$ on flash. Objects evicted from DRAM are *not* written to the flash. Only objects requested in $\mathcal{S}$ and $\mathcal{G}$ are written to the flash. This setup reduces *both* flash writes and miss ratio.

Because CDN caches are often deployed using flash, we compare the miss ratio and write bytes using open-source CDN traces from Wikimedia [140] and Tencent Photo CDN [168]. We compare with three schemes. FIFO does not use an admission control and writes everything to the flash. Probabilistic admission uses an LRU DRAM cache and a 20% probability to admit DRAM-evicted objects into the flash cache randomly. Flashield uses a machine learning model (SVM) to predict which objects are worthwhile writing to the flash. S3-FIFO uses a small FIFO queue in DRAM (0.1%, 1%, 10%), and objects requested at least twice in the DRAM are admitted onto flash. Because the flash cache eviction algorithm is orthogonal to the admission policy, we used FIFO [14, 24, 102] in all experiments (including in S3-FIFO). We have also evaluated other flash-friendly algorithms [159] and observed similar results. We set the cache size to 10% of the trace footprint in bytes. We further normalize the write bytes to the number of unique bytes in the trace.

Fig. 9 shows that compared to no admission control (FIFO), an admission policy can significantly reduce the number of write bytes. However, both probabilistic admission and Flashield trade-off the miss ratio for the reduced write bytes. In contrast, using a small FIFO queue for admission is surprisingly effective at reducing both write bytes and miss ratios. Unlike probabilistic admission, which has almost no dependency on the DRAM size, S3-FIFO and Flashield make admission decisions based on access in DRAM. With a large DRAM (10% of flash cache size), Flashield achieves close to S3-FIFO miss ratio with slightly more writes. However, when the DRAM size is small, objects do not accumulate enough access for the machine-learning model to predict accurately. Meta engineers have also made a similar observation [24].
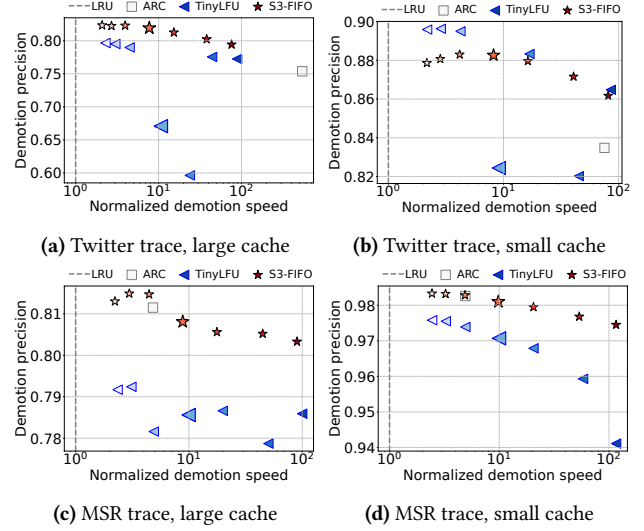

(a) Twitter trace, large cache

(b) Twitter trace, small cache

(c) MSR trace, large cache

(d) MSR trace, small cache

Figure 10. The normalized mean quick demotion speed and precision of different algorithms. TinyLFU and S3-FIFO use different $\mathcal{S}$ sizes (1%, 2%, 5%, 10%, 20%, 30%, and 40% of cache size) and have multiple points with lighter colors representing larger $\mathcal{S}$. The marker of 10% small queue size is highlighted with a larger size.

Table 2. Miss ratio when using different $\mathcal{S}$ sizes (as a fraction of cache size). Increasing $\mathcal{S}$ sizes lead to slower but more accurate quick demotion. Thus miss ratio for S3-FIFO first decreases, then increases with $\mathcal{S}$ size. But TinyLFU sometimes shows anomalies. The table should be read together with Fig. 10.

$\mathcal{S}$ size	0.40	0.30	0.20	0.10	0.20	0.02	0.01
Twitter trace, large cache, ARC miss ratio *0.0483*, LRU miss ratio 0.0488 (Fig. 10a)							
TinyLFU	0.0451	0.0445	0.0441	*0.0530*	*0.0586*	0.0437	0.0437
S3-FIFO	0.0455	0.0442	0.0432	0.0424	0.0422	0.0422	0.0423
Twitter trace, small cache, ARC miss ratio *0.1941*, LRU miss ratio 0.2005 (Fig. 10b)							
TinyLFU	0.1744	0.1718	0.1697	*0.1766*	0.1688	*0.1775*	0.1722
S3-FIFO	0.1846	0.1802	0.1765	0.1743	0.1740	0.1752	0.1768
MSR trace, large cache, ARC miss ratio 0.2891, LRU miss ratio 0.3188 (Fig. 10c)							
TinyLFU	0.2990	0.2949	0.2936	0.2900	0.2893	0.2904	0.2895
S3-FIFO	0.2989	0.2936	0.2896	0.2891	0.2884	0.2887	0.2889
MSR trace, small cache, ARC miss ratio 0.4899, LRU miss ratio 0.5263 (Fig. 10d)							
TinyLFU	0.4952	0.4923	0.4903	0.4907	0.4922	0.4993	0.5120
S3-FIFO	0.4940	0.4903	0.4890	0.4910	0.4926	0.4953	0.4970

6 Discussion

6.1 Why is S3-FIFO effective?

The key to S3-FIFO's efficiency is the small probationary FIFO queue $\mathcal{S}$ that filters out one-hit wonders. Removing low-value items is not new. Admission algorithms, e.g., Bloom Filter, Adaptsize [25], are designed for a similar purpose. However, they reject objects too early and show low efficiency for most cache workloads. Besides admission algorithms, many cache eviction algorithms designed to be scan-resistant, e.g., ARC and 2Q, share a similar idea. They separate new and frequent objects into two queues (denote using $\mathcal{S}$ and $\mathcal{M}$)

so that popular objects are not affected by scan requests. This work shows that a small static FIFO queue, one of the simplest designs to filter out low-value objects, works better than many more advanced alternatives.

But why? We take a closer look at demotion speed and precision using the same trace from §3 to get a deeper understanding. The *normalized quick demotion speed* measures how long objects stay in $\mathcal{S}$ before they are evicted or moved to $\mathcal{M}$. We use the LRU eviction age as a baseline and calculate the speed as $\frac{\text{LRU eviction age}}{\text{time in } \mathcal{S}}$. We use logical time measured in request count. The *quick demotion precision* measures how many objects evicted from $\mathcal{S}$ are not reused soon. Using an idea similar to previous work [126], if the number of requests till an object's next reuse is larger than $\frac{\text{cache size}}{\text{miss ratio}}$, then we say the quick demotion results in a correct early eviction.

An algorithm with both faster and more precise quick demotion exhibits a lower miss ratio. Fig. 10 shows that ARC, TinyLFU, and S3-FIFO can quickly demote new objects and have lower miss ratios compared to LRU (Table 2). **ARC** uses an adaptive algorithm to decide the size of $\mathcal{S}$. We find that the algorithm can identify the correct direction to adjust the size, but the size it finds is often too large or too small. For example, Fig. 10a shows that ARC chooses a very small $\mathcal{S}$ on the Twitter trace, causing most new objects to be evicted too quickly with low precision. This happens because of two trace properties. First, objects in the Twitter trace often have many requests; Second, new objects are constantly generated. Therefore, objects evicted from $\mathcal{M}$ are requested very soon, causing $\mathcal{S}$ to shrink to a very small size (around 0.01% of cache size). Meanwhile, constantly generated new (and popular) objects in $\mathcal{S}$ face more competition and often have to suffer a miss before being inserted in $\mathcal{M}$, which causes low precision and a high miss ratio (Table 2). On the MSR trace, ARC has a reasonable speed with relatively high precision, which correlates with its low miss ratio. **TinyLFU and S3-FIFO** have a predictable quick demotion speed — reducing the size of $\mathcal{S}$ always increases the demotion speed. When using the same $\mathcal{S}$ size, TinyLFU demotes slightly faster than S3-FIFO because it uses LRU, which keeps some old but recently-accessed objects, squeezing the available space for newly-inserted objects.

Besides, S3-FIFO often shows higher precision than TinyLFU at a similar quick demotion speed, which explains why S3-FIFO has a lower miss ratio. TinyLFU compares the eviction candidates from $\mathcal{S}$ and $\mathcal{M}$, then evicts the less-frequently-used candidate. When the eviction candidate from $\mathcal{M}$ has a high frequency, it causes many worth-to-keep objects from $\mathcal{S}$ to be evicted. This causes not only a low precision but also unpredictable precision and miss ratio cliffs. For example, the precision shows a large dip at 5% and 10% in Fig. 10a, corresponding to a sudden increase in the miss ratio (Table 2).

Although S3-FIFO does not use advanced techniques, it achieves a robust and predictable quick demotion speed and precision. As $\mathcal{S}$ size increases, the speed decreases monotonically (moving towards the left in the figure), and the precision also increases until it reaches a peak. When $\mathcal{S}$ is very small, popular objects do not have enough time to accumulate a hit before being evicted, so the precision is low. Increasing $\mathcal{S}$ size leads to higher precision. When $\mathcal{S}$ is very large, many unpopular objects are requested in $\mathcal{S}$ and moved to $\mathcal{M}$, leading to reduced precision as well. Table 2 shows that at similar quick demotion speed, higher precision always leads to lower miss ratios.

In summary, S3-FIFO *guarantees* that newly inserted unpopular objects are evicted in a predictably short time. The quick demotion is often more precise and robust compared to existing approaches. This combination allows S3-FIFO to obtain better than state-of-the-art miss ratios.

6.2 How about adaptive eviction algorithms?

6.2.1 Is queue size sensitive? We chose $\mathcal{S}$ to use 10% of the cache size based on results from ten traces and found that it generalizes well across the 6594 traces. Fig. 11 shows how the miss ratios change with $\mathcal{S}$ size. We observe that a smaller $\mathcal{S}$ leads to larger miss ratio reductions, confirming the importance of quick demotion. For example, when the cache size is large, the best-performing traces (P90) have the largest reduction when $\mathcal{S}$ uses 1% of the cache size. However, a smaller $\mathcal{S}$ also causes more traces to have miss ratios higher than FIFO. This aligns with the observation in §6.1 where we see smaller $\mathcal{S}$ leads to faster quick demotion, but the precision decreases after the peak. Overall, the predictability between efficiency and $\mathcal{S}$ size makes it easy to choose the $\mathcal{S}$ size. And the efficiency does not change much for most traces if $\mathcal{S}$ size is between 5% and 20% of the cache size.

6.2.2 Making queue size adaptive! We designed and implemented an algorithm that adaptively changes the FIFO queue sizes, which we call S3-FIFO-d, S3-FIFO with dynamic queue sizes. S3-FIFO-d maintains a balance between marginal hits on the evicted objects from $\mathcal{S}$ and $\mathcal{M}$. It uses two small ghost queues to track objects evicted from $\mathcal{S}$ and $\mathcal{M}$. Each ghost queue is sized to store 5% of the cached objects (without data). Each time the two ghost queues have more than 100 hits, and one has 2× more hits than the other, S3-FIFO-d moves 0.1% of cache space to the queue whose evicted objects receive more hits. By balancing the marginal hits on the evicted objects, S3-FIFO minimizes the gradient of hits on the evicted objects. If $\mathcal{S}$ is too small, its evicted objects will receive many hits causing an expansion of $\mathcal{S}$. Vice versa. Besides the algorithm described above, we also experimented with another adaptive algorithm similar to ARC, which increases queue size by one upon a hit on the ghost. However, we find this algorithm less robust than S3-FIFO-d.
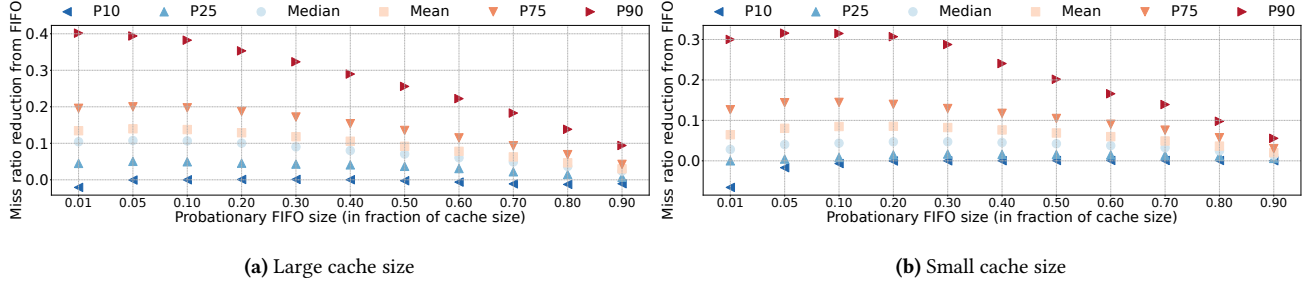
(a) Large cache size

(b) Small cache size

Figure 11. Miss ratio reduction percentiles using different sizes for the small FIFO. Left: large cache size, right: small cache size.

We compare S3-FIFO-D and S3-FIFO (not shown) and find that S3-FIFO is better than S3-FIFO-D on most traces except the 2% traces at the tail, on which using 10% cache size for $\mathcal{S}$ is far from optimal. In other words, the adaptive algorithm is only useful when the workload is adversarial (which is rare). We tried to tune the parameters in the adaptive algorithm. However, tuning for a few traces is easy, but obtaining good results across traces is very challenging [3].

6.2.3 Where do adaptive algorithms fail? The parameter tuning problem is not unique to S3-FIFO-D. Most, if not all, adaptive algorithms have many parameters. For example, queue resizing requires several parameters, e.g., the frequency of resizing, the amount of space moved each time, the lower bound of queue sizes, and the threshold for trigger resizing. This is not unique for S3-FIFO-D, but also for algorithms such as ARC, whose parameters are less obvious. For example, ARC moves one slot upon a hit on the ghost. But the question remains why one slot instead of half or two? And is it better to handle the hit at the head and tail of the ghost queue differently?

Besides the many hard-to-tune parameters, adaptive algorithms adapt based on observation of the past. However, the past may not predict the future. We find that small perturbations in the workload often cause the adaptive algorithm to overreact. It is unclear how to balance between under-reaction and overreaction without introducing more parameters. Moreover, some adaptive algorithms, including S3-FIFO-D, implicitly assume that the miss ratio curve is convex because following the gradient direction leads to the global optimum. However, the miss ratio curves of scan-heavy workloads are often not convex [23, 135].

Although we have shown that S3-FIFO is not sensitive to $\mathcal{S}$ size, and the queue size is easier to choose than tuning an adaptive algorithm. We believe adaptations are still important, but how to adapt remains to be explored. For systems that need to find the best parameter, downsized simulations using spatial sampling can be used [135, 136].

6.3 LRU or FIFO?

S3-FIFO only uses FIFO queues, but do LRU queues provide better efficiency? We experimented with different queue-type combinations by replacing both the small FIFO queue and the main FIFO queue with LRU queues. And we have also experimented with moving objects from $\mathcal{S}$ to $\mathcal{M}$ upon cache hits and during evictions. Due to space limits, the results are not shown, but we observe that LRU queues *do not* improve efficiency. In particular, using two LRU queues, such as in ARC, is worse than S3-FIFO most of the time. In conclusion, with quick demotion, the queue type does not matter.

7 Related Work

We have discussed many related works throughout the §2 and §5.2. We discuss the rest in this section.

Efficiency-oriented cache design. Besides the eviction algorithms we compared with, many other algorithms are designed to improve the cache efficiency [18, 26, 41, 70, 82, 87, 97, 131, 166]. S3-FIFO differs from existing algorithms in the following way. First, S3-FIFO uses only FIFO queues and does not require promotion on cache hits. Second, S3-FIFO explicitly guarantees the time one-hit wonders stay in the cache before testing popularity. Third, this work shows why a very small probationary cache is needed and uses a smaller probationary queue than most previous works. Quickly removing one-hit wonders is similar to removing scan/streaming/sequential/looping requests that motivated many previous works [21, 79, 100, 119]. Moreover, similar ideas have also been applied to removing low-priority blocks from lower cache layers in a cache hierarchy [106, 142, 147]. Our previous work also discussed two general techniques for cache efficiency and scalability — lazy promotion and quick demotion [155]. S3-FIFO is an example of applying the two techniques on FIFO queues to achieve high efficiency and good scalability. Sieve is another eviction algorithm that is simpler than LRU, but provides state-of-the-art efficiency for web workloads [162]. Compared to Sieve, which is not scan-resistant, S3-FIFO is suitable for general cache workloads. Moreover, Sieve can be used to replace the large FIFO queue in S3-FIFO to further improve efficiency. Besides eviction algorithms, several other works improved cache efficiency

---

[3]We believe algorithm design should not be tuned on the traces used for evaluation (test dataset), but rather on a validation dataset.

via removing cliffs in miss ratio curves [22], space partitioning [42–44, 66], prefetching [49, 62, 91, 127, 134, 153], exploiting spatial locality [75], compression [72, 145], leveraging application hints [71, 90, 94, 113, 120, 147], cooperative caching [46, 76, 78, 149], read-write separation [20], reducing metadata, and removing expired objects [158]. These works complement the eviction algorithm designs. However, we remark that integrating multiple designs, e.g., eviction and prefetching are non-trivial and requires additional exploration [17, 31, 34, 62, 127].

Scalability-oriented cache design. Segmented FIFO was designed to achieve low overhead; however, it has no ghost queue and does not perform quick demotion. Therefore, its efficiency is lower than LRU [130]. Segcache [158] improves a cache's throughput and scalability by eliminating promotions and locking. Segcache uses log-structured storage to improve scalability. However, Segcache is more efficient for workloads using TTLs. MemC3 [57] and Tricache [60] use CLOCK for scalable data access. However, CLOCK has lower efficiency than S3-FIFO because it cannot quickly remove one-hit wonders. FrozenHot [115] freezes part of the cached data to provide scalable data access and does not improve the eviction algorithm's scalability. Besides improving an eviction algorithm, sharding is commonly used to improve scalability. Sharding partitions the key space, and each CPU core serves a slice of the keys. However, cache workloads often follow Zipfian popularity, so sharding leads to load imbalance [58, 65, 68, 95, 116] and limits the whole system's throughput. Besides improving the cache eviction algorithm's scalability, several other works have improved other parts in a key-value cache/store [93]. Compared to these works, S3-FIFO focuses on the eviction algorithm.

Flash endurance. Endurance is a well-known problem for caching on flash. Many works have designed flash-friendly cache eviction algorithms, such as RIPQ [129], SpatialClock [81], and offline algorithms [40]. FlashTier [121], DIDACache [123], Pannier [86] studied the flash cache design beyond eviction algorithms to improve flash cache performance and endurance. Flash cache admission control (also called selective caching in some works) has been explored in LARC [69], WEC [36], S-RAC [107] and SieveStore [114], which use window-based or ghost-based frequency threshold to selectively cache objects on flash. Such designs are similar to using counting Bloom Filter LRU. However, they do not explicitly consider *the role of DRAM to cache new (and unpopular) objects*, which is particularly important as we have shown that B-LRU cannot achieve the optimal efficiency (§5). Flashield [55] and ML-QP [165] track object access in the DRAM cache and use a machine-learning model to decide admission. However, Flashield requires too much DRAM to work. Besides, several works used social features to predict object access patterns [137, 138], which are only applicable in social network cache workloads. While early eviction and selective caching, and selective placement can help with

flash endurance, they are also widely used in cache hierarchy to achieve exclusive caching and address the lack of locality. Different algorithms [39, 73, 148, 169], interfaces and systems [61, 142, 146, 147] have been designed to improve the efficiency of hierarchical caches.

8 Conclusion

We demonstrate that a cache often experiences a higher one-hit-wonder ratio than common full trace analysis. Our study on 6594 traces reveals that quickly removing one-hit wonders (quick demotion) is the secret weapon of many advanced algorithms. Motivated by this, we design S3-FIFO, a **s**imple and **s**calable cache eviction algorithm composed of only **s**tatic FIFO queues. Our evaluation shows that S3-FIFO achieves better and more robust efficiency than state-of-the-art algorithms. Meanwhile, it is more scalable than LRU-based algorithms.

Availability

The code and data used in this work are open-sourced at https://github.com/TheSys-lab/sosp23-s3fifo. This includes the simulator, the prototype, and our fault-tolerant distributed computation platform.

We have also developed cache libraries using S3-FIFO for different programming languages. More information is available at https://s3fifo.com, which provides documentation and tracks the adoptions of S3-FIFO in production systems.

References

[1] Adaptive replacement cache implementation in python. https://gist.github.com/pior/da3b6268c40fa30c222f. Accessed: 2023-07-28.

[2] Alibaba block-trace. https://github.com/alibaba/block-traces. Accessed: 2023-01-12.

[3] Cache implementation in python. https://github.com/trauzti/cache. Accessed: 2023-07-28.

[4] Cacheus implementation. https://github.com/sylab/cacheus. Accessed: 2023-07-28.

[5] Caffeine java cache package. https://github.com/ben-manes/caffeine. Accessed: 2023-07-28.

[6] libcachesim: a high-performance cache simulator. https://github.com/1a1a11a/libCacheSim. Accessed: 2023-07-28.

[7] Lirs implementation and discussion. https://github.com/facebook/CacheLib/discussions/99. Accessed: 2023-07-28.

[8] Lrb implementation. https://github.com/sunnyszy/lrb. Accessed: 2023-07-28.

[9] The multi-generational lru. https://lwn.net/Articles/851184. Accessed: 2023-07-28.

[10] Page frame reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html. Accessed: 2023-02-06.

[11] Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval. Accessed: 2023-02-12.

[12] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, USA, 2008. USENIX Association.

[13] AMD. Epyc™ 9654p. https://www.amd.com/en/products/cpu/amd-epyc-9654p. Accessed: 2023-02-06.

[14] Apache. Apache traffic server. https://trafficserver.apache.org/. Accessed: 2023-02-06.

[15] 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, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Sung Hoon Baek and Kyu Ho Park. Prefetching with adaptive cache culling for striped disk arrays. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 363–376, USA, June 2008. USENIX Association.

[18] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies*, FAST'04, 2004.

[19] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-Based Caching for Content Delivery. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, pages 45–46, Urbana-Champaign Illinois USA, June 2017. ACM.

[20] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. AWOL: an adaptive write optimizations layer. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 1–14, USA, February 2008. USENIX Association.

[21] 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, 2018.

[22] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pages 64–75, Burlingame, CA, USA, February 2015. IEEE.

[23] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[24] 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.

[25] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adapt-Size: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX symposium on networked systems design and implementation*, NSDI'17, pages 483–498, 2017.

[26] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[27] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX conference on File and stroage technologies*, FAST'10, pages 115–128, 2010.

[28] Bradfitz. group cache. https://github.com/golang/groupcache. Accessed: 2023-02-06.

[29] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134 vol.1, New York, NY, USA, 1999. IEEE.

[30] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of Zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[31] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 157–168, New York, NY, USA, June 2005. Association for Computing Machinery.

[32] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 47–60, New York, NY, USA, August 2002. Association for Computing Machinery.

[33] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, pages 353–362, Washington District of Columbia USA, September 2019. ACM.

[34] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[35] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, USITS'97, Monterey, CA, December 1997. USENIX Association.

[36] Yunpeng Chai, Zhihui Du, Xiao Qin, and David A. Bader. WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data. *IEEE Transactions on Computers*, 64(11):3304–3316, November 2015. Conference Name: IEEE Transactions on Computers.

[37] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, Houston TX USA, May 2018. ACM.

[38] H. Che, Z. Wang, and Y. Tung. Analysis and design of hierarchical Web caching systems. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1416–1424, Anchorage, AK, USA, 2001. IEEE.

[39] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference, General Track*, ATC'03, pages 269–281, 2003.

[40] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *2016 USENIX Annual Technical Conference*

(USENIX ATC 16), ATC, pages 379–392, 2016.

[41] Jongmoo Choi, Sam H Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement scheme. In *USENIX annual technical conference*, ATC'99, pages 239–252, 1999.

[42] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX workshop on hot topics in cloud computing*, HotCloud'15, 2015.

[43] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX symposium on networked systems design and implementation*, NSDI'16, pages 379–392, 2016.

[44] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX annual technical conference*, ATC'17, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[45] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[46] Michael D Dahlin, Randolph Y Wang, Thomas E Anderson, and David A Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI'94, pages 19–es, 1994.

[47] Meta developers. Cachelib - pluggable caching engine to build and scale high performance cache services. https://cachelib.org/. Accessed: 2023-04-06.

[48] Pelikan developers. Pelikan. https://github.com/pelikan-io/pelikan. Accessed: 2023-02-06.

[49] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[50] Siying Dong. Reducing lock contention in rocksdb. https://rocksdb.org/blog/2014/05/14/lock.html. Accessed: 2023-02-06.

[51] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[52] Rémi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gaël Thomas. NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 186–198, June 2021. arXiv:2105.10397 [cs].

[53] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[54] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, December 2017.

[55] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019. USENIX Association.

[56] 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. USENIX Association, July 2020.

[57] 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, 2013.

[58] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, SOCC'11, pages 1–12, Cascais, Portugal, 2011. ACM Press.

[59] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. PA-Cache: Evolving Learning-Based Popularity-Aware Content Caching in Edge Networks, December 2020.

[60] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 395–411, Carlsbad, CA, July 2022. USENIX Association.

[61] Binny S Gill. On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions. In *FAST*, volume 8 of *FAST'08*, pages 1–17, 2008.

[62] Binny S Gill and Luis Angel D Bathen. AMP: Adaptive multi-stream prefetching in a shared cache. In *FAST*, volume 7, pages 185–198, 2007.

[63] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 168–183, New York, NY, USA, October 2017. Association for Computing Machinery.

[64] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2020.

[65] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, Santa Clara California, October 2013. ACM.

[66] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference*, ATC'15, pages 57–69, 2015.

[67] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, November 2013. Association for Computing Machinery.

[68] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets'14, pages 1–7, Los Angeles CA USA, October 2014. ACM.

[69] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Transactions on Storage*, 12(2):8:1–8:24, February 2016.

[70] Andhi Janapsatya, Aleksandar Ignjatovic, Jorgen Peddersen, and Sri Parameswaran. Dueling CLOCK: Adaptive cache replacement policy based on the CLOCK algorithm. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 920–925, Dresden, March 2010. IEEE.

[71] R. Jauhari, Michael J. Carey, and Miron Livny. Priority-hints: an algorithm for priority-based buffer management. In *Proceedings of the sixteenth international conference on Very large databases*, pages

708–721, San Francisco, CA, USA, September 1990. Morgan Kaufmann Publishers Inc.

[72] Yichen Jia, Zili Shao, and Feng Chen. SlimCache: Exploiting Data Compression Opportunities in Flash-Based Key-Value Caching. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 209–222, Milwaukee, WI, September 2018. IEEE.

[73] S. Jiang and X. Zhang. ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 168–177, March 2004. ISSN: 1063-6927.

[74] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'05, page 35, USA, April 2005. USENIX Association.

[75] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4 of *FAST'05*, pages 8–8, 2005.

[76] Song Jiang, F. Petrini, Xiaoning Ding, and Xiaodong Zhang. A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 42–42, July 2006. ISSN: 1063-6927.

[77] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002.

[78] Wenjie Jiang, Rui Zhang-Shen, Jennifer Rexford, and Mung Chiang. Cooperative content distribution and traffic engineering in an ISP network. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 239–250, New York, NY, USA, June 2009. Association for Computing Machinery.

[79] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994. Morgan Kaufmann Publishers Inc.

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

[81] Hyojun Kim, Moonkyung Ryu, and Umakishore Ramachandran. What is a good buffer cache replacement scheme for mobile flash storage? *ACM SIGMETRICS Performance Evaluation Review*, 40(1):235–246, 2012.

[82] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *4th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'00, USA, 2000. USENIX Association.

[83] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[84] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Systor '17 traces (SNIA IOTTA trace 5102). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2016.

[85] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pages 1–11, New York, NY, USA, May 2017. Association for Computing Machinery.

[86] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and Analysis of a Container-Based Flash Cache for Compound Objects. *ACM Transactions on Storage*, 13(3):1–34, October 2017.

[87] Cong Li. CLOCK-pro+: improving CLOCK-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 1–7, New York, NY, USA, May 2019. Association for Computing Machinery.

[88] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 591–605, New York, NY, USA, March 2020. Association for Computing Machinery.

[89] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.

[90] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. {Second-Tier} Cache Management Using Write Hints. 2005.

[91] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: mining block correlations in storage systems. In *Proceedings of the 3rd USENIX conference on File and storage technologies*, FAST'04, page 13, USA, March 2004. USENIX Association.

[92] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime. In *20th USENIX Conference on File and Storage Technologies*, FAST'22, pages 445–459, 2022.

[93] 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.

[94] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: CLient-Informed Caching for Storage Servers. In *FAST*, pages 297–310, 2009.

[95] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX conference on file and storage technologies*, FAST'19, pages 143–157, Boston, MA, February 2019. USENIX Association.

[96] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, July 2015.

[97] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, April 2022. ISSN: 2378-203X.

[98] 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, pages 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[99] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of

caching billions of tiny objects on flash. In *ACM Transactions on Storage*, volume 18 of *TOS'22*, August 2022.

[100] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003.

[101] Memcached. Extstore. https://github.com/memcached/memcached/wiki/Extstore. Accessed: 2023-02-06.

[102] Memcached. Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2023-02-06.

[103] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video CDNs: building strong lines of defense. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, EuroSys'14, pages 1–13, Amsterdam, The Netherlands, 2014. ACM Press.

[104] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007.

[105] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[106] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. *ACM SIGMOD Record*, 20(2):387–396, April 1991.

[107] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-RAC: SSD Friendly Caching for Data Center Workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–12, Haifa Israel, June 2016. ACM.

[108] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 385–398, 2013.

[109] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, August 2010.

[110] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, June 1993.

[111] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'19, pages 193–205, Santa Cruz CA USA, November 2019. ACM.

[112] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 813–827, 2020.

[113] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 79–95, New York, NY, USA, December 1995. Association for Computing Machinery.

[114] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 163–174, New York, NY, USA, June 2010. Association for Computing Machinery.

[115] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 557–573, New York, NY, USA, May 2023. Association for Computing Machinery.

[116] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache:load-balanced,low-latency cluster caching with online erasure coding. In *12th USENIX symposium on operating systems design and implementation*, OSDI'16, pages 401–417, 2016.

[117] RocksDB. Lock-free clock cache. https://github.com/facebook/rocksdb/issues/10306. Accessed: 2023-02-06.

[118] Liana V. Rodriguez, Alexis Gonzalez, Pratik Poudel, Raju Rangaswami, and Jason Liu. Unifying the data center caching layer: Feasible? Profitable? In *Proceedings of the 13th ACM workshop on hot topics in storage and file systems*, HotStorage '21, pages 50–57, New York, NY, USA, 2021. Association for Computing Machinery.

[119] 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.

[120] Prasenjit Sarkar and John Hartman. Efficient Cooperative Caching Using Hints. 1996.

[121] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, page 267, Bern, Switzerland, 2012. ACM Press.

[122] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 465–478, New York, NY, USA, July 2020. Association for Computing Machinery.

[123] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: An Integration of Device and Application for Flash-based Key-value Caching. *ACM Transactions on Storage*, 14(3):26:1–26:32, October 2018.

[124] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.

[125] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage. pages 839–853, 2021.

[126] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020.

[127] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 377–390, USA, June 2008. USENIX Association.

[128] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT'17, pages 55–67, Incheon Republic of Korea, November 2017. ACM.

[129] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, 2015.

[130] Rollins Turner and Henry Levy. Segmented FIFO page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 10(3):48–51, September 1981.

[131] Cristian Ungureanu, Biplob Debnath, Stephen Rago, and Akshat Aranya. TBF: A memory-efficient replacement policy for flash-based caches. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1117–1128, April 2013. ISSN: 1063-6382.

[132] 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 *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018. USENIX Association.

[133] Vimeo. Galaxy cache. hhttps://github.com/vimeo/galaxycache. Accessed: 2023-02-06.

[134] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 33–43, New York, NY, USA, June 1998. Association for Computing Machinery.

[135] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX annual technical conference*, ATC'17, pages 487–498, Santa Clara, CA, July 2017. USENIX Association.

[136] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX conference on file and storage technologies*, FAST'15, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[137] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP'18, pages 1–10, Eugene OR USA, August 2018. ACM.

[138] Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. Cache What You Need to Cache: Reducing Write Traffic in Cloud Cache via "One-Time-Access-Exclusion" Policy. *ACM Transactions on Storage*, 16(3):1–24, August 2020.

[139] Qiuping Wang, Jinhong Li, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022.

[140] wikimedia. Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2023-02-06.

[141] WikiMedia. Better handling for one-hit-wonder objects. https://phabricator.wikimedia.org/T144187. Accessed: 2023-02-06.

[142] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference*, ATC'02, pages 161–175, 2002.

[143] 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.

[144] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching. pages 1–16, 2022.

[145] 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, pages 1–15, London United Kingdom, April 2016. ACM.

[146] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. MC2: Multiple Clients on a Multilevel Cache. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 722–730, June 2008. ISSN: 1063-6927.

[147] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Transactions on Computer Systems*, 29(2):5:1–5:51, May 2011.

[148] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All Replacement for a Multilevel Cache. 2007.

[149] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, May 2013. ISSN: 2160-1968.

[150] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, Seattle WA USA, October 2020. ACM.

[151] Gang Yan and Jian Li. Towards Latency Awareness for Content Delivery Network Caching. ATC'22, pages 789–804, 2022.

[152] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage*, 13(3):1–26, October 2017.

[153] 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*, SoCC '17, pages 66–79, New York, NY, USA, September 2017. Association for Computing Machinery.

[154] 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)*, FAST'23, pages 115–134, 2023.

[155] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, pages 70–79, New York, NY, USA, June 2023. Association for Computing Machinery.

[156] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. 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, Renton, WA, April 2022. USENIX Association.

[157] Juncheng Yang, Yao Yue, and K. V. Rashmi. 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. USENIX Association, November 2020.

[158] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518. USENIX Association, April 2021.

[159] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

[160] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4 of *SIGMETRICS'20*, pages 1–27, May 2020.

[161] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the Cache Warm? Manufacturing a Rule of Thumb. 2020.

[162] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K .V. Rashmi. Sieve: an efficient turn-key cache eviction algorithm simpler than lru. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, 2024.

[163] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace set 27917). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, October 2018.

[164] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2023 U2ENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798. USENIX Association, July 2023.

[165] Yu Zhang, Ke Zhou, Ping Huang, Hua Wang, Jianying Hu, Yangtao Wang, Yongguang Ji, and Bin Cheng. A Machine Learning Based Write Policy for SSD Cache in Cloud Block Storage. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, DATE'20, pages 1279–1282, Grenoble, France, March 2020. IEEE.

[166] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM.

[167] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27476). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

[168] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 284–294, New York, NY, USA, 2018. Association for Computing Machinery.

[169] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'01, pages 91–104, USA, 2001. USENIX Association.