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;

ACM Reference Format:

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, 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 6x 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 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 6x 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 complexity is often inversely 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]. Generality is crucial for similar reasons. If the same data structure and eviction algorithm can be used for different types of caches, 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.

<table>
<thead>
<tr>
<th>request</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>start time</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>17</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>17</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>17</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>end time</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>18</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>18</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>18</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>sequence length (# objects)</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td># one-hit-wonder</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>one-hit-wonder ratio</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

**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 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 suggested [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 ratio of 50%. Similarly, the one-hit-wonder ratio of a shorter sequence from the 1st to the 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 object access rate.
The intuition is that most objects are unpopular (rank higher than \( C + 1 \)) in Zipf distribution for a cache of size \( C \), and there are \( M \) unique objects in total, so that 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 \( F(x = C) \) monotonically decreases with the sequence length \( x \) measured in the number of objects.

The intuition is that most objects are unpopular (rank higher than \( C + 1 \)) in Zipf distribution for a cache of size \( C \), and there are \( M \) unique objects in total, so that 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 \( F(x = C) \) monotonically decreases with the sequence length \( x \) measured in the number of objects.

The intuition is that most objects are unpopular (rank higher than \( C + 1 \)) in Zipf distribution for a cache of size \( C \), and there are \( M \) unique objects in total, so that 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 \( F(x = C) \) monotonically decreases with the sequence length \( x \) measured in the number of objects.

The intuition is that most objects are unpopular (rank higher than \( C + 1 \)) in Zipf distribution for a cache of size \( C \), and there are \( M \) unique objects in total, so that 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 \( F(x = C) \) monotonically decreases with the sequence length \( x \) measured in the number of objects.
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, a Bloom Filter rejects objects too fast with a lack of precision since it rejects all objects that have not been seen before. It causes the second requests to all objects to be cache misses, which often leads to mediocri 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 probabilatory 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 simple and scalable eviction algorithm that consists of only static 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 (S), a main FIFO queue (M), and a ghost FIFO queue (G). We choose S to use 10% of the cache space based on experiments with 10 traces and find that 10% generalizes well. M then uses 90% of the cache space. The ghost queue G stores the same number of ghost entries (no data) as 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 atomically increment the counter by one. Note that most requests for popular objects require no update.

**Cache write.** New objects are inserted into S if not in G. Otherwise, it is inserted into M. When S is full, the object at the tail is either moved to M if it is accessed more than once or G if not. And its access bits are cleared during the move. When G is full, it evicts objects in FIFO order. 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 \( S \), main FIFO queue \( M \),
ghost FIFO queue \( G \)

1: function \texttt{read}(x)
2: \quad if \( x \) in \( S \) or \( x \) in \( M \) then \quad \triangleright \text{Cache Hit}
3: \quad \quad \text{\( x \).freq} \leftarrow \min(\text{\( x \).freq + 1, 3 \}) \quad \triangleright \text{Frequency is capped to 3}
4: \quad else
5: \quad \quad \text{insert}\( (x) \)
6: \quad \quad \text{\( x \).freq} \leftarrow 0
7: function \texttt{insert}(x)
8: \quad \text{while cache is full do}
9: \quad \quad \text{evict()}
10: \quad \quad if \( x \) in \( G \) then
11: \quad \quad \quad \text{insert} \( x \) to head of \( M \)
12: \quad \quad else
13: \quad \quad \quad \text{insert} \( x \) to head of \( S \)
14: function \texttt{evict}
15: \quad if \( S \).size \geq 0.1 \cdot \text{cache size} then
16: \quad \quad \text{evict}(S)
17: \quad else
18: \quad \quad \text{evict}(M)
19: function \texttt{evictS}
20: \quad \text{evicted} \leftarrow \text{false}
21: \quad \text{while not evicted and \( S \).size > 0 do}
22: \quad \quad \text{\( t \)} \leftarrow \text{tail of} \( S \)
23: \quad \quad \quad if \( t \).freq > 1 then
24: \quad \quad \quad \quad \text{insert} \( t \) to \( M \)
25: \quad \quad \quad \quad if \( M \) is full then
26: \quad \quad \quad \quad \quad \text{evict}(M)
27: \quad \quad \quad \quad else
28: \quad \quad \quad \quad \quad \text{insert} \( t \) to \( G \)
29: \quad \quad \quad \quad evicted \leftarrow \text{true}
30: \quad \quad \text{remove} \( t \) from \( S \)
31: function \texttt{evictM}
32: \quad \text{evicted} \leftarrow \text{false}
33: \quad \text{while not evicted and \( M \).size > 0 do}
34: \quad \quad \text{\( t \)} \leftarrow \text{tail of} \( M \)
35: \quad \quad \quad if \( t \).freq > 0 then
36: \quad \quad \quad \quad \text{Insert} \( t \) to head of \( M \)
37: \quad \quad \quad \quad \text{\( t \).freq} \leftarrow \text{\( t \).freq} - 1
38: \quad \quad \quad else
39: \quad \quad \quad \quad \text{remove} \( t \) from \( M \)
40: \quad \quad \quad evicted \leftarrow \text{true

Handling 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 \( S \) 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 \( G \) in this process. Nevertheless, when these “good” blocks are requested again in the near future, they will be inserted into \( M \) 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.

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

The ghost FIFO queue \( G \) can be implemented as part of the indexing structure. For example, we can store object fingerprint and insertion time of ghost entries in a bucket-based hash table \([33, 37, 93, 158]\). The fingerprint is a 4-byte hash of the object ID. The insertion time is a virtual timestamp, counting the number of objects inserted into \( G \) thus far. Let \( S_G \) denote the size of the ghost queue. If the current time is \( N \) (i.e., there were \( N \) insertions into \( G \)), then all the entries whose timestamp is lower than \( N - S_G \) are no longer in \( G \). A ghost entry is removed from the hash table when the object is requested or during hash collision — when the slot is needed to store another entry.

4.3 Overhead analysis

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 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.
Traces. We evaluated Table 1. Datasets used in this work, the ones with no citation are proprietary datasets. For old datasets, we exclude traces with less than 12007 to 2023 and cover key-value, block, and object CDN. More details of the datasets can be found in Table 1.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Approx</th>
<th>Cache</th>
<th>time span (days)</th>
<th># Traces</th>
<th># Request (million)</th>
<th>Request (TB)</th>
<th># Object (million)</th>
<th>Object (TB)</th>
<th>One-hit-wonder ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR [104, 105]</td>
<td>2007</td>
<td>Block</td>
<td>7</td>
<td>13</td>
<td>410</td>
<td>10</td>
<td>74</td>
<td>3</td>
<td>0.56</td>
</tr>
<tr>
<td>FIU [83]</td>
<td>2008-11</td>
<td>Block</td>
<td>9-28</td>
<td>9</td>
<td>514</td>
<td>1.7</td>
<td>20</td>
<td>0.057</td>
<td>0.28</td>
</tr>
<tr>
<td>Cloudphysics [136]</td>
<td>2015</td>
<td>Block</td>
<td>7</td>
<td>106</td>
<td>2,114</td>
<td>82</td>
<td>492</td>
<td>0.4</td>
<td>0.40</td>
</tr>
<tr>
<td>CDN 1</td>
<td>2018</td>
<td>Object</td>
<td>7</td>
<td>219</td>
<td>3,728</td>
<td>3690</td>
<td>249</td>
<td>0.42</td>
<td>0.58</td>
</tr>
<tr>
<td>Tencent Photo [167, 168]</td>
<td>2018</td>
<td>Object</td>
<td>8</td>
<td>2</td>
<td>3,650</td>
<td>141</td>
<td>1,038</td>
<td>0.55</td>
<td>0.66</td>
</tr>
<tr>
<td>Wikimedia CDN [140]</td>
<td>2019</td>
<td>Object</td>
<td>7</td>
<td>3</td>
<td>2,863</td>
<td>200</td>
<td>56</td>
<td>0.46</td>
<td>0.60</td>
</tr>
<tr>
<td>Systor [84, 85]</td>
<td>2017</td>
<td>Block</td>
<td>26</td>
<td>6</td>
<td>3,694</td>
<td>88</td>
<td>421</td>
<td>0.37</td>
<td>0.80</td>
</tr>
<tr>
<td>Tencent CBS [163, 164]</td>
<td>2020</td>
<td>Block</td>
<td>8</td>
<td>4030</td>
<td>33,690</td>
<td>1091</td>
<td>551</td>
<td>0.25</td>
<td>0.73</td>
</tr>
<tr>
<td>Alibaba [2, 89, 139]</td>
<td>2020</td>
<td>Block</td>
<td>30</td>
<td>652</td>
<td>19,676</td>
<td>664</td>
<td>1702</td>
<td>0.36</td>
<td>0.68</td>
</tr>
<tr>
<td>Twitter [157]</td>
<td>2020</td>
<td>KV</td>
<td>7</td>
<td>54</td>
<td>195,441</td>
<td>106</td>
<td>10,650</td>
<td>0.19</td>
<td>0.32</td>
</tr>
<tr>
<td>Social Network 1</td>
<td>2020</td>
<td>KV</td>
<td>7</td>
<td>219</td>
<td>549,784</td>
<td>392</td>
<td>42,898</td>
<td>0.17</td>
<td>0.28</td>
</tr>
<tr>
<td>CDN 2</td>
<td>2021</td>
<td>Object</td>
<td>7</td>
<td>1273</td>
<td>37,460</td>
<td>4,925</td>
<td>2,652</td>
<td>0.49</td>
<td>0.58</td>
</tr>
<tr>
<td>Meta KV [11]</td>
<td>2022</td>
<td>KV</td>
<td>1</td>
<td>5</td>
<td>1,644</td>
<td>958</td>
<td>82</td>
<td>0.51</td>
<td>0.53</td>
</tr>
<tr>
<td>Meta CDN [11]</td>
<td>2023</td>
<td>Object</td>
<td>7</td>
<td>3</td>
<td>231</td>
<td>8,800</td>
<td>76</td>
<td>0.61</td>
<td>0.76</td>
</tr>
</tbody>
</table>

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 algorithms, although S3-FIFO often requires fewer metadata than other algorithms. 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

---

1https://blog.jasony.me/random/tool/2023/08/01/distributed-computation
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: \( \frac{MR_{\text{algo}} - MR_{\text{FIFO}}}{MR_{\text{FIFO}}} \)

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: \( \frac{MR_{\text{FIFO}} - MR_{\text{algo}}}{MR_{\text{FIFO}}} \), which bounds the value between -1 and 1. This avoids the impact of outliers on the mean value.

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 \( S \) and \( M \) using linked lists and \( G \) 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 object value. 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].

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

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 6420 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)

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 uses a 1% window to filter out unpopular objects and stores most objects in a SLRU cache. 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. Because 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 with more details 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., 100s of objects. Moreover, using an adaptive algorithm to adjust the queue size can often mitigate the problem, and we discuss more in §6.2.

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 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.

Byte miss ratio. While (request) miss ratio is important for most cache deployments, CDNs also widely use byte...
Figure 7. The mean miss ratio reduction of different algorithms on each dataset. TinyLFU on the TencentPhoto dataset at the large size is -0.11 and not shown.

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

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-$n_{\text{thread}}$ million requests for $n_{\text{thread}}$ 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, the lock-free implementation enables the throughput to scale with the number of CPU cores. Under both small and large cache sizes, S3-FIFO runs more than 6x 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 less synchronization — Segcache needs atomic updates only when a segment-chain is changed, which is 100-1000x 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, Segcache does not have a comparable efficiency as S3-FIFO as we have shown in Fig. 6.

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 objects evicted from the $S$ are not worthwhile to be kept in $M$, we can place $S$ in DRAM and $M$ on flash. Objects evicted from DRAM are not written to the flash. Only objects requested in $S$ and $G$ 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 and ghost queue in DRAM (0.1%, 1%, 10%) to filter objects, 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, 101] in all experiments (including in S3-FIFO). We have also evaluated other flash-friendly algorithms, such as FIFO-Reinsertion [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].

<table>
<thead>
<tr>
<th>S</th>
<th>Miss ratio</th>
<th>Write bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1%</td>
<td>0.4952</td>
<td>0.2990</td>
</tr>
<tr>
<td>1%</td>
<td>0.4923</td>
<td>0.2949</td>
</tr>
<tr>
<td>10%</td>
<td>0.4903</td>
<td>0.2936</td>
</tr>
</tbody>
</table>

Table 2. Miss ratio when using different S sizes (as a fraction of cache size). Increasing S sizes leads to slower but more accurate quick demotion. Thus miss ratio for S3-FIFO first decreases, then increases with S size. But TinyLFU sometimes shows anomalies. The table should be read together with Fig. 10. The font color matches the color in Fig. 10, and the italics show the miss ratio anomaly of TinyLFU.
ARC and 2Q, share a similar idea. They separate new and frequent objects into two queues (denote using $S$ and $M$) 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 $S$ before they are evicted or moved to $M$. We use the LRU eviction age as a baseline and calculate the speed as $\frac{LRU\text{ eviction age}}{time\text{ in }S}$. We use logical time measured in request count. The quick demotion precision measures how many objects evicted from $S$ 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 the cache size, 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 $S$. 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 $S$ 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 $M$ are requested very soon, causing $S$ to shrink to a very small size (around 0.01% of cache size). Meanwhile, constantly generated new (and popular) objects in $S$ face more competition and often have to suffer a miss before being inserted in $M$, 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 $S$ always increases the demotion speed. When using the same $S$ 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 $S$ and $M$, then evicts the less-frequently-used candidate. When the eviction candidate from $M$ has a high frequency, it causes many worth-to-keep objects from $S$ 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 $S$ size increases, the speed decreases monotonically (moving towards the left in the figure), and the precision also increases until it reaches a peak. When $S$ is very small, popular objects do not have enough time to accumulate a hit before being evicted, so the precision is low. Increasing $S$ size leads to higher precision. When $S$ is very large, many unpopular objects are requested in $S$ and moved to $M$, 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?

**Is queue size sensitive?** We chose $S$ 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 $S$ size. We observe that a smaller $S$ 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 $S$ uses 1% of the cache size. However, a smaller $S$ also causes more traces to have miss ratios higher than FIFO. This aligns with the observation in §6.1 where we see smaller $S$ leads to faster quick demotion, but the precision decreases after the peak. Overall, the predictability between efficiency and $S$ size makes it easy to choose the $S$ size. And the efficiency does not change much for most traces if $S$ size is between 5% and 20% of the cache size.

**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 $S$ and $M$. It uses two small ghost queues to track objects evicted from $S$ and $M$. 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 $2 \times$ 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 $S$ is too small, its evicted objects will receive many hits causing an expansion of $S$. 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.
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.

**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-react and overreact with out 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 techniques to improve cache efficiency and scalability — lazy promotion and quick demotion [155]. S3-FIFO is an example of applying the two techniques on FIFO queues to design simple, efficient, and scalable cache eviction algorithms. SIEVE is another eviction algorithm focusing on simplicity, efficiency, and scalability. However, SIEVE is not scan-resistant and only works on web workloads [162].

Besides eviction algorithms, several other works improved cache efficiency via removing cliffs in miss ratio curves [22], space partitioning [42–44, 66], prefetching [49, 62, 91, 127, 142].

---

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

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], DIDA Cache [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. This is particularly important as we have shown that B-LRU cannot achieve the optimal efficiency [85]. Flashfield [55] and ML-QP [165] track object access in the DRAM cache and use a machine-learning model to decide admission. However, Flashfield 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, selective caching, and selective placement can help with flash endurance, they are also widely used in hierarchical caches 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 simple and scalable cache eviction algorithm composed of only static 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.

**Acknowledgments**

We thank the anonymous reviewers for their valuable feedback and our shepherd Gala Yadgar for her constructive suggestions. We also would like to thank the people and organizations that have open-sourced and shared production traces. We thank Cloudlab [53] for the infrastructure support for running experiments. This work is funded in part by a Meta Fellowship, and NSF grants CNS 1901410 and CNS 1956271. We also thank the members of the PDL Consortium for their interest, insights, feedback, and support.

**References**


Badrish Chandramouli, Guna Prasada, Donald Kossoss, 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.


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 144
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.


