SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches

Yazhuo Zhang, *Emory University;* Juncheng Yang, *Carnegie Mellon University;*
Yao Yue, *Pelikan Foundation;* Ymir Vigfusson, *Emory University and Keystrike;*
K.V. Rashmi, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches

Yazhuo Zhang*
Emory University

Juncheng Yang†
Carnegie Mellon University

Yao Yue
Pelikan Foundation

Ymir Vigfusson
Emory University & Keystrike

K. V. Rashmi
Carnegie Mellon University

Abstract

Caching is an indispensable technique for low-cost and fast data serving. The eviction algorithm, at the heart of a cache, has been primarily designed to maximize efficiency—reducing the cache miss ratio. Many eviction algorithms have been designed in the past decades. However, they all trade off throughput, simplicity, or both for higher efficiency. Such a compromise often hinders adoption in production systems.

This work presents SIEVE, *an algorithm that is simpler than LRU and provides better than state-of-the-art efficiency and scalability* for web cache workloads. We implemented SIEVE in five production cache libraries, requiring fewer than 20 lines of code changes on average. Our evaluation on 1559 cache traces from 7 sources shows that SIEVE achieves up to 63.2% lower miss ratio than ARC. Moreover, SIEVE has a lower miss ratio than 9 state-of-the-art algorithms on more than 45% of the 1559 traces, while the next best algorithm only has a lower miss ratio on 15%. SIEVE's simplicity comes with superior scalability as cache hits require no locking. Our prototype achieves twice the throughput of an optimized 16-thread LRU implementation. SIEVE is more than an eviction algorithm; it can be used as a cache primitive to build advanced eviction algorithms just like FIFO and LRU.

1 Introduction

Web caches, such as Content Delivery Networks (CDNs) and key-values caches, are widely deployed in today's digital landscape to reduce user request latency [14, 21, 22, 33, 69, 73, 76, 100], network bandwidth [54, 55, 79, 95], and repeated computation [28, 89, 97, 98]. As a critical component of modern infrastructure, these caches often have a large footprint. For example, Netflix used 18,000 servers for caching over 14 PB of application data in 2021 [68]; while Twitter reportedly had 100s of clusters using 100s of TB of DRAM and 100,000s of CPU cores for in-memory caching in 2020 [96].

At the heart of a cache is the eviction algorithm, which plays a crucial role in managing limited cache space. Such algorithms are efficient when they can retain more valuable objects in the cache to achieve a lower miss ratio—the fraction of requested objects that must be fetched from the backend. The quest for high efficiency has spurred a long repertoire of clever algorithms, but most, if not all, trade off simplicity in exchange for efficiency gains. For example, ARC [67], SLRU [55], 2Q [60], and MQ [106] manage multiple least-recently-used (LRU) queues to achieve better efficiency. LHD [16], CACHEUS [75], LRB [79], and GL-Cache [93] use machine learning techniques that further increase system and lookup complexity. Furthermore, many of these algorithms require explicit or implicit parameter tuning to achieve good efficiency on a target workload.

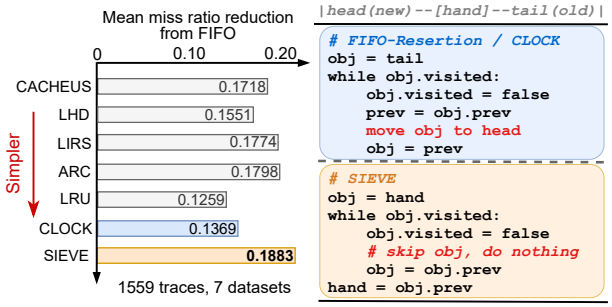


Figure 1: SIEVE is simple and efficient. The code snippet shows how FIFO-Reinsertion and SIEVE find eviction candidates. Minor code changes convert FIFO-Reinsertion to SIEVE, unleashing lower miss ratios than state-of-the-art algorithms.

The conventional wisdom among systems operators is that *simple is beautiful*: simplicity is a key appealing feature for an algorithm to be deployed in production since it commonly correlates with effectiveness, maintainability, scalability, and low overhead. To illustrate, note that most caching systems or libraries in use today, such as ATS [2], Varnish [11], Nginx [7], Redis [9], and groupcache [25], use only FIFO and LRU policies.

We have stumbled upon an easy improvement (Fig. 1) to a decades-old algorithm (FIFO-Reinsertion) that materially improves its efficiency across a wide range of web cache workloads. Instead of moving the to-be-evicted object that

*Equal contribution.
†Corresponding author: Juncheng Yang, sieve@cachemon.com.

has been accessed to the head of queue, SIEVE keeps it in its original position. It should be noted that both SIEVE and FIFO-Reinsertion insert new objects at the head of the queue. The new algorithm is called SIEVE [1]: a simple and efficient turn-key cache eviction policy. We implemented SIEVE in five production cache libraries, which required fewer than 20 lines of change on average, underscoring the ease of real-world deployment.

Despite a simple design, SIEVE can quickly remove unpopular objects from the cache, achieving comparatively high efficiency compared to the state-of-the-art algorithms. By experimentally evaluating SIEVE on 1559 traces from five public and two proprietary datasets, we show that SIEVE achieves similar or higher efficiency than 9 state-of-the-art algorithms across traces. Compared to ARC [67], SIEVE reduces miss ratio by up to 63.2% with a mean of 1.5% [2]. As a comparison, ARC reduces LRU's miss ratio by up to 33.7% with a mean of 6.7%. Moreover, compared to the best of all algorithms, SIEVE has lower miss ratio on over 45% of the 1559 traces. In comparison, the runner-up algorithm, TwoQ, only outperforms other algorithms on 15% of the traces.

SIEVE's design eliminates the need for locking during cache hits, resulting in a boost in multi-threaded throughput. Our prototype implementation in Cachelib [37] demonstrates that SIEVE achieves twice the throughput of an optimized LRU implementation when operating with 16 threads.

Through empirical evidence and analysis, we illustrate that SIEVE's efficiency stems from sifting out unpopular objects over time. SIEVE transcends a single standalone algorithm — it can also be embedded within other cache policies to design more advanced algorithms. We demonstrate the idea by replacing the LRU components in ARC, TwoQ, and LeCaR with SIEVE. The SIEVE-supported algorithms significantly outperform the original LRU-based algorithms. For example, ARC-SIEVE reduces ARC's miss ratio by up to 62.5% with a mean reduction of 3.7% across the 1559 traces.

Our work makes the following contributions.

- We present the design for SIEVE: an easy, fast, and surprisingly efficient cache eviction algorithm for web caches.
- We demonstrate SIEVE's simplicity by implementing it in five production cache libraries by changing less than 20 lines of code on average.
- Using 1559 traces from 7 datasets, we show that SIEVE outperforms all state-of-the-art eviction algorithms on more than 45% of the traces.
- We illustrate SIEVE's scalability using our Cachelib-based implementation, which achieves 17% and 125% higher throughput than optimized LRU at 1 and 16 threads.
- We show how SIEVE, as a turn-key cache primitive, opens new opportunities for designing advanced eviction algorithms, e.g., replacing the LRU in ARC, TwoQ, and LeCaR with SIEVE.

[1] SIEVE sifts out unpopular objects from cache over time (§5).

[2] Due to a large number of traces, the mean miss ratio looks small.

2 Background and Related Work

2.1 Web caches

Web caches are essential components of modern Internet infrastructure, playing a crucial role in reducing data access latency and network bandwidth. Key-value caches, e.g., Memcached [5], Pelikan [8] and Cachelib [37], are widely used in modern web services such as Twitter [97] and Meta [20] to reduce service latency. CDN caches are deployed close to users to reduce data access latency and high WAN bandwidth cost [14, 91, 95, 101].

Cache metrics. Caches are measured along two primary axes: efficiency and throughput performance. Cache efficiency measures how well the cache can store and serve the required data. A cache miss occurs when the requested data is not found in the cache, requiring access to the backend storage to retrieve the data. Common cache efficiency metrics include (1) object miss ratio: the fraction of requests that are cache misses; (2) byte miss ratio: the fraction of bytes that are cache misses. A lower miss ratio indicates higher cache efficiency, as more requests are served directly from the cache, reducing backend load, access latency, and bandwidth costs.

Throughput performance, on the other hand, is as important as efficiency because the goal of a cache is to serve data quickly and help scale the application. Beyond throughput, scalability is also increasingly important [72, 98] as modern CPUs often surpass 100 cores. Scalability measures throughput growth with the number of threads accessing the cache. A more scalable cache can better harness the many cores in a modern CPU.

Access patterns. Web cache workloads typically follow Power-law (generalized Zipfian) distributions [20, 26, 27, 34, 49, 52, 55, 81, 82, 97], where a small subset of objects account for a large proportion of requests. In detail, the i^{th} popular object has a relative frequency of $1/i^{\alpha}$, where α is a parameter that decides the skewness of the workload. Previous works find different α values from 0.6 to 0.8 [26], 0.56 [49], 0.71–0.76 [51], 0.55–0.9 [20], and 0.6–1.5 [97]. The reasons for the large range of α include (1) the different types of workloads, such as web proxy and in-memory key-value cache workloads; (2) the layer of the cache, noting that many proxy/CDN caches are secondary or tertiary cache layers [55]; and (3) the popularity of the service, such as the most popular objects receiving greater volume of requests in more popular (widely-used) web applications. Moreover, web caches often serve constantly growing datasets — new content and objects are created every second.

In contrast, the backend of enterprise storage caches or single-node caches, such as the page cache, often has a fixed size, not regularly observing new objects. Further, many storage cache workloads often have scan and loop patterns [75], in which a range of block addresses are sequentially requested in a short time. Such patterns are rare in web cache workloads according to our observation on 1559 traces from 7 datasets.

2.2 Cache eviction policies

The cache eviction algorithm, which decides which objects to store in the limited cache space, governs the performance and efficiency of a cache. The field of cache eviction algorithms has a rich literature [12, 17–19, 23, 29, 32, 35, 36, 39, 41, 44–46, 53, 58, 62, 63, 71, 74, 78, 83, 86, 88, 90, 102].

Increasing complexity. Most works on cache eviction algorithms focused on improving efficiency, such as LRU-k [70], TwoQ [60], SLRU [61], GDSF [29], EELRU [77], LRFU [39], LIRS [59], ARC [67], MQ [105], CAR [15], CLOCK-pro [58], TinyLFU [42, 43], LHD [16], LeCaR [84], LRB [79], CACHEUS [75], GLCache [93], and HALP [80]. Over the years, new cache eviction algorithms have gradually convoluted. Algorithms from the 1990s use two or more static LRU queues or use different recency metrics; algorithms from the 2000s employ size-adaptive LRU queues or use more complicated recency/frequency metrics, and algorithms from the 2010s and 2020s start to use machine learning to select eviction candidates. Each decade brought greater complexity to cache eviction algorithms. Nevertheless, as we show in §4, while the new algorithms excel on a few specific traces, they do not show a significant improvement (and some are even worse) compared to the traditional ones on a large number of workloads. The combination of limited improvement and high complexity explains why these algorithms have not been used in production systems.

The trouble with complexity. Multiple problems come with increasing complexity. First, complex cache eviction algorithms are difficult to debug due to their intricate logic. For example, we find two open-source cache simulators used in previous works have two different bugs in the LIRS [59] implementation. Second, complexity may affect efficiency in surprising ways. For example, previous work reports that both LIRS and ARC exhibit Belady's anomaly [50, 85]: miss ratio increases with the cache size for some workloads. It's worth noting that FIFO, although simple, also suffers from this anomaly. Third, complexity often negatively correlates with throughput performance. A more intricate algorithm performs more computation with potentially longer critical sections, reducing both throughput and scalability. Furthermore, many of these algorithms need to store more per-object metadata, which reduces the effective cache size that can be used for caching data. For example, the per-object metadata required by CACHEUS is 3.3× larger than that of LRU. Fourth, complex algorithms often have parameters that can be difficult to tune. For example, all the machine-learning-based algorithms include many parameters about learning. Although some algorithms do not have explicit parameters, e.g., LIRS, previous work shows that the implicit ghost queue size can impact the efficiency [85].

Trade-offs in using simple eviction algorithms. Besides works focusing on improving cache efficiency, several other works have improved cache throughput and scalability. For example, MemC3 [47] uses Cuckoo hashing and CLOCK eviction to improve Memcached's throughput and scalability; MICA [64] uses log-structured storage, data partitioning, and a lossy hash table to improve key-value cache throughput and scalability. Segcache [98] uses segment-structured storage with a FIFO-based eviction algorithm and leverages macro management to improve scalability. Frozenhot [72] improves cache scalability by freezing hot objects in the cache to avoid locking. However, it's crucial to note that while these approaches excel in throughput and scalability, they often compromise on cache efficiency due to the use of simpler, weaker eviction algorithms such as CLOCK[3] and FIFO.

2.3 Lazy promotion and quick demotion

Promotion and demotion are two cache internal operations used to maintain the logical ordering between objects[4]. Recent work [94] shows that "lazy promotion" and "quick demotion" are two important properties of efficient cache eviction algorithms.

Lazy promotion refers to the strategy of promoting cached objects only at eviction time. It aims to retain popular objects with minimal effort. An example of lazy promotion is adding reinsertion to FIFO. In contrast, FIFO has no promotion, and LRU performs eager promotion – moving objects to the head of the queue on every cache hit. Lazy promotion can improve (1) throughput due to less computation and (2) efficiency due to more information about an object at eviction.

Quick demotion removes most objects quickly after they are inserted. Many previous works have discussed this idea in the context of evicting pages from a scan [16, 60, 67, 70, 75, 77]. Recent work also shows that not only storage workloads but web cache workloads also benefit from quick demotion [94] because object popularity follows a power-law distribution, and many objects are unpopular.

To the best of our knowledge, our proposed cache eviction algorithm, which we call SIEVE, is the simplest one that effectively achieves both lazy promotion and quick demotion.

3 Design and Implementation

3.1 SIEVE Design

In this section, we introduce SIEVE, a cache eviction algorithm that achieves both simplicity and efficiency.

Data structure. SIEVE requires only one FIFO queue and one pointer called "hand". The queue maintains the insertion order between objects. Each object in the queue uses one bit to track the visited/non-visited status. The hand points to the next eviction candidate in the cache and moves from the tail to the head. Note that, unlike existing algorithms, e.g., LRU, FIFO, and CLOCK, in which the eviction candidate is always the tail object, the eviction candidate in SIEVE is an object

[3] CLOCK was recently shown to be more efficient than LRU [94].

[4] Note that the terms "promotion" and "demotion" are also commonly used in the context of cache hierarchy. In this case, promotion refers to the process of moving data to a faster device, while demotion involves moving the data to a slower device [65, 87].

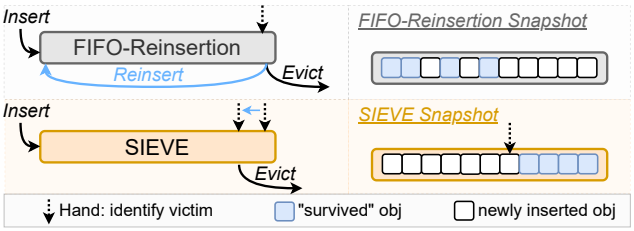


Figure 2: An illustration of SIEVE. Note that FIFO-Reinsertion and CLOCK are different implementations of the same algorithm. We use FIFO-Reinsertion in the illustration but will use CLOCK in the rest of the text because it is more commonly used and is shorter.

somewhere in the queue.

SIEVE operations. A cache hit in SIEVE changes the visited bit of the accessed object to 1. For a popular object whose visited bit is already 1, SIEVE does not need to perform any operation. During a cache miss, SIEVE examines the object pointed by the hand. If it has been visited, the visited bit is reset, and the hand moves to the next position (the retained object stays in the original position of the queue). It continues this process until it encounters an object with the visited bit being 0, and it evicts the object. After the eviction, the hand points to the next position (the previous object in the queue). While an evicted object is in the middle of the queue most of the time, a new object is always inserted into the head of the queue. In other words, the new objects and the retained objects are not mixed together.

At first glance, SIEVE is similar to CLOCK/Second Chance/FIFO-Reinsertion [5]. Each algorithm maintains a single queue in which each object is associated with a visited bit to track its access status. Visited objects are retained (also called "survived") during an eviction. Notably, new objects are inserted at the head of the queue in both SIEVE and FIFO-Reinsertion. However, the hand in SIEVE moves from the tail to the head over time, whereas the hand in FIFO-Reinsertion stays at the tail. The key difference is where a retained object is kept. SIEVE keeps it in the old position, while FIFO-Reinsertion inserts it at the head, together with newly inserted objects, as depicted in Fig. 2.

We detail the algorithm in Alg. 1. Line 1 checks whether there is a hit, and if so, then line 2 sets the visited bit to one. In the case of a cache miss (Line 3), Lines 5-12 identify the object to be evicted.

Lazy promotion and quick demotion. Despite a simple design, SIEVE effectively incorporates both lazy promotion and quick demotion. As described in §2.3, an object is only promoted at the eviction time in lazy promotion. SIEVE operates in a similar manner. However, rather than promoting the object to the head of the queue, SIEVE keeps the object at its original location. The "survived" objects are generally more popular than the evicted ones, thus, they are likely to be accessed again in the future. By gathering the "survived" objects, the hand in SIEVE can quickly move from the tail to the area near the head, where most objects are newly inserted. These newly inserted objects are quickly examined by the hand of SIEVE after they are admitted into the cache, thus achieving quick demotion. This eviction mechanism makes SIEVE achieve both lazy promotion and quick demotion without adding too much overhead.

The key ingredient of SIEVE is the moving hand, which functions like an adaptive filter that removes unpopular objects from the cache. This mechanism enables SIEVE to strike a balance between finding new popular objects and keeping old popular objects. We discuss more in §5.

[5]Note that Second Chance, CLOCK, and FIFO-Reinsertion are different implementations of the same eviction algorithm.

Algorithm 1 SIEVE

```
Input: The request x, doubly-linked queue T, cache size C, hand p
1:  if x is in T then                                   ▷ Cache Hit
2:      x.visited ← 1
3:  else                                                ▷ Cache Miss
4:      if |T| = C then                                 ▷ Cache Full
5:          o ← p
6:          if o is NULL then
7:              o ← tail of T
8:          while o.visited = 1 do
9:              o.visited ← 0
10:             o ← o.prev
11:             if o is NULL then
12:                 o ← tail of T
13:         p ← o.prev
14:         Discard o in T                              ▷ Eviction
15:     Insert x in the head of T.
16:     x.visited ← 0                                   ▷ Insertion
```

3.2 Implementation

Simulation. We implemented SIEVE in libCacheSim [92]. LibCacheSim is a high-performance cache simulator designed for running cache simulations and analyzing cache traces. It supports many state-of-the-art eviction algorithms, including ARC [67], LIRS [59], CACHEUS [75], LeCaR [84], TwoQ [60], LHD [16], Hyperbolic [24], FIFO-Reinsertion/CLOCK [35], B-LRU (Bloom Filter LRU), LRU, LFU, and FIFO. For all state-of-the-art algorithms, we used the configurations from the original papers.

Prototype. Because of SIEVE's simplicity, it can be implemented on top of a FIFO, LRU, or CLOCK cache in just a few lines by adding, initializing, and tracking the "hand" pointer. The object pointed to by the hand is either evicted or retained, depending on whether it has been accessed.

We implemented SIEVE caching in five different open-source cache libraries: Cachelib [20], groupcache [25], mnemonist [6], lru-dict [3], and lru-rs [4]. These represent the most popular cache libraries of five different programming languages: C++, Golang, JavaScript, Python, and Rust. All five of these production cache libraries implement LRU as the eviction algorithm of choice. Aside from mnemonist, which uses arrays, they all use doubly-linked-list-based implementations of LRU. Adapting these LRU implementations to use SIEVE was a low effort, as mentioned earlier.

Table 1: Datasets used in this work. CDN 1 and 2 are proprietary, and all others are publicly available.

trace collections	approx time	# traces	cache type	# request (million)	# object (million)
CDN 1	2021	1273	object	37,460	2,652
CDN 2	2018	219	object	3,728	298
Tencent Photo [103]	2018	2	object	5,650	1,038
Wiki CDN [1]	2019	3	object	2,863	56
Twitter KV [97]	2020	54	KV	195,441	10,650
Meta KV [10]	2022	5	KV	1,644	82
Meta CDN [10]	2023	3	object	231	76

4 Evaluation

In this section, we evaluate SIEVE to answer the following questions.

- Does SIEVE have higher efficiency than state-of-the-art cache eviction algorithms?
- Can SIEVE improve a cache's throughput and scalability?
- Is SIEVE simpler than other algorithms?

4.1 Experimental setup

Workloads. Our experiments use open-source traces from Twitter [97], Meta [10], Wikimedia [1], TencentPhoto [103, 104], and two proprietary CDN datasets. We list the dataset information in Table 1. It consists of 1559 traces that together contain 247,017 million requests to 14,852 million objects. Notably, our research is centered around web traces. We replayed the traces in the simulator and the prototypes as a closed system with instant on-demand fill.

Metrics. Miss ratio serves as a key performance indicator when evaluating the efficiency of a cache system. However, when analyzing different traces (even within the same dataset), the miss ratios can vary significantly, making direct comparisons and visualizations infeasible, as shown in Fig. 3. Therefore, we calculate the miss ratio reduction relative to a baseline method (FIFO in this work): $\frac{mr_{FIFO}-mr_{algo}}{mr_{FIFO}}$ where mr stands for miss ratio. If an algorithm's miss ratio is higher than FIFO, we use $\frac{mr_{FIFO}-mr_{algo}}{mr_{algo}}$. This metric has a range between -1 and 1.

We measure throughput in millions of operations per second (Mops) to quantify a cache's performance. To evaluate scalability, we vary the number of trace replay threads from 1 to 16 and measure the throughput.

Testbed. Our evaluations were conducted on Cloudlab [40] and focused on two key aspects: simulation-based efficiency and prototype-based throughput and simplicity.

We used libCacheSim [92], a high-performance cache simulator, to evaluate the efficiency of different cache algorithms. These simulations ran on various node types at either the Clemson or Utah sites, subject to availability.

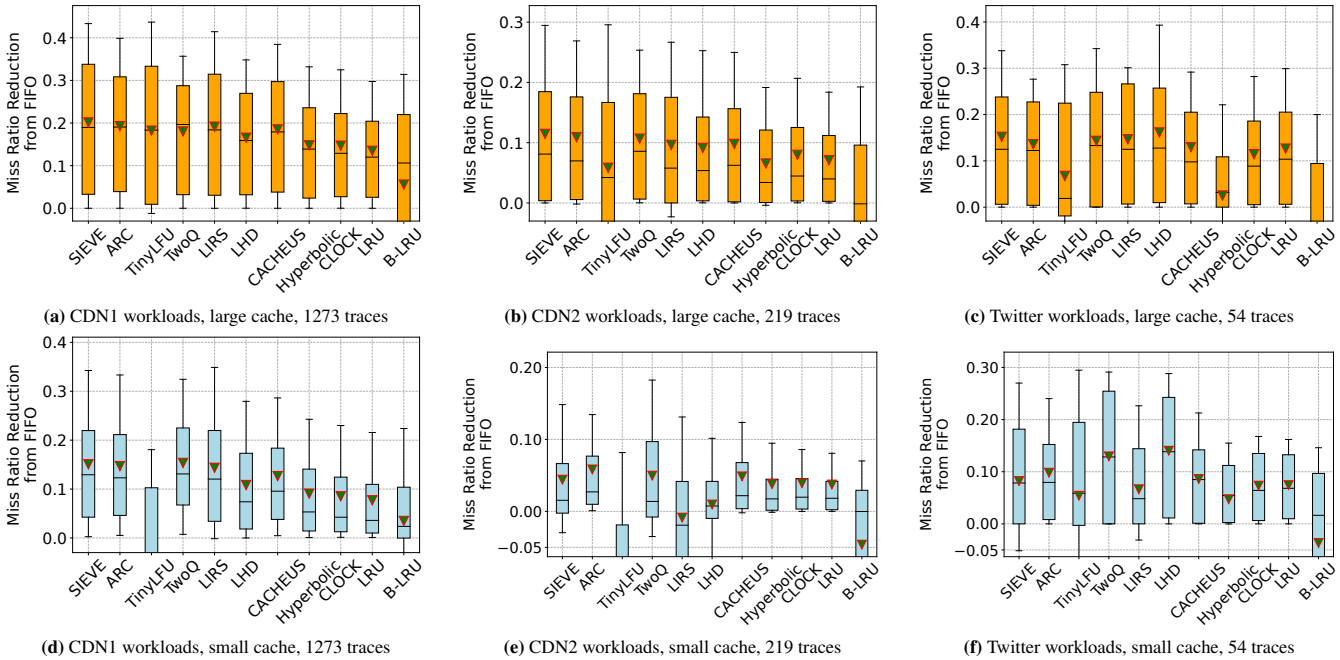
We evaluate the throughput and simplicity using prototypes, as described in §3.2. The prototype evaluations were conducted on the c6420 node from the Clemson site. This node type has a dual-socket Intel Gold 6142 running at 2.6 GHz and is equipped with 384 GB DDR4 DRAM. We turned off turbo boost and pinned threads to CPU cores in one NUMA node in our evaluations. We validated the efficiency results from the simulator and prototype using 60 randomly selected traces and found the same conclusion.

4.2 Efficiency results

In this section, we compare the efficiency of different eviction algorithms. Because many caches today use slab-based space management, in which evictions happen on objects of similar sizes, we do not consider object size in this section. The cache sizes are determined as a percentage of the number of objects in a trace. We evaluate eight cache sizes using 1559 traces from the 7 datasets and present two representative cache sizes at 0.1% and 10% of the trace footprint (the number of unique objects in the trace).

Three large datasets CDN1, CDN2 and Twitter. Fig. 3 shows the miss ratio reduction (from FIFO) of different algorithms across traces. The whiskers on the boxplots are defined using p10 and p90, allowing us to disregard extreme data and concentrate on the typical cases. At the large cache size, SIEVE demonstrates the most significant reductions across nearly all percentiles. For example, SIEVE reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21% on the CDN1 dataset using the large cache size (Fig. 3a). As a comparison, all other algorithms have smaller reductions on this dataset. For example, CLOCK/FIFO-Reinsertion, which is conceptually similar to SIEVE, can only reduce FIFO's miss ratio by 15% on average. Compared to advanced algorithms, e.g., ARC, SIEVE reduces ARC miss ratio by up to 63.2% with a mean of 1.5%. We remark that a 1.5% mean miss ratio reduction on the huge number of traces is significant. For example, ARC only reduces LRU's miss ratio by 6.3% on average (not shown). A similar observation can be made on the CDN2 dataset. Although LHD is the best algorithm on the Twitter dataset, SIEVE scores second and outperforms most other state-of-the-art algorithms.

When the cache is very small, TwoQ and LHD sometimes outperform SIEVE. This is because TwoQ and LHD can quickly remove newly-inserted low-value objects similar to SIEVE. The primary reason for SIEVE's relatively poor performance is that new objects cannot demonstrate their popularity before being evicted when the cache size is very small. A similar problem also happens with ARC and LIRS. ARC's adaptive algorithm sometimes shrinks the recency queue to very small and yields a high miss ratio. LIRS, which uses a 1% queue for new objects, suffers the most when the cache size is small, as we see its miss ratio on some traces higher than FIFO. In contrast, TwoQ does not suffer from the small cache sizes because it reserves a fixed 25% of the cache space for new objects, preventing overly aggressive demotion. However, we remark that the production miss ratios reported in previous works [13, 55, 97, 98] are close to the miss ratios we observe at the large cache size.



(a) CDN1 workloads, large cache, 1273 traces

(b) CDN2 workloads, large cache, 219 traces

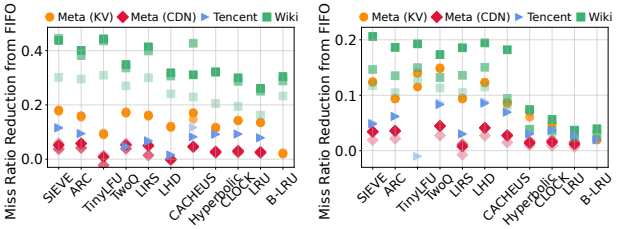
(c) Twitter workloads, large cache, 54 traces

(d) CDN1 workloads, small cache, 1273 traces

(e) CDN2 workloads, small cache, 219 traces

(f) Twitter workloads, small cache, 54 traces

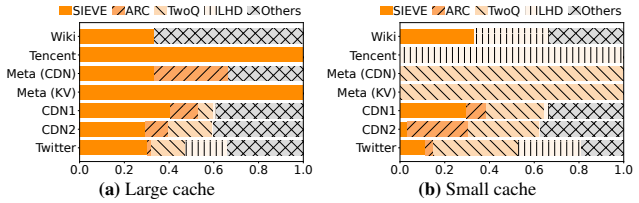
Figure 3: The box shows the miss ratio reduction from FIFO over all traces in the dataset. The box shows P25 and P75, the whiskers show P10 and P90, and the triangle shows the mean. The large cache uses 10% of the trace footprint, and the small cache uses 0.1% of the trace footprint. SIEVE achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.



(a) Large cache

(b) Small cache

Figure 4: Miss ratio reduction on Meta (KV + CDN), Wiki CDN, and Tencent Photo CDN datasets. The different opacity of the same color indicates multiple traces from the dataset. Some negative results are not shown.



(a) Large cache

(b) Small cache

Figure 5: Best-performing algorithms on each dataset. Table 1 shows the number of traces per dataset.

The secret behind SIEVE's efficiency is the ability to quickly remove newly-inserted unpopular objects (quick demotion), the ability to sift out old unpopular objects, and the balance between new and old objects. We discuss more in §5.

Four small datasets: Meta KV, Meta CDN, Wiki, and TencentPhoto. Because each dataset contains fewer than ten traces, we use scatter plots to compare the algorithms. Fig. 4 demonstrates that SIEVE outperforms all other algorithms on all four datasets at the large cache size. When the cache size is small, the observation is similar to that made in Fig. 3. SIEVE is the best algorithm on the Wiki dataset. TwoQ and LHD are the best on Meta and TencentPhoto datasets. Although not the best, SIEVE remains highly competitive.

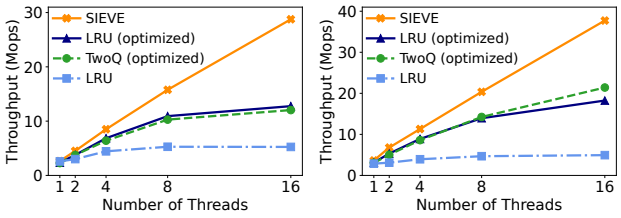
Best-performing algorithm per dataset. We have demonstrated that SIEVE provides larger miss ratio reductions across traces than state-of-the-art algorithms. For a more quantitative comparison, Fig. 5 shows the fraction of traces each algorithm performs the best.

With a large cache size, SIEVE outperforms all other algorithms on the Tencent Photo, Wiki, and Meta KV datasets. On the CDN1 and CDN2 datasets, SIEVE is the best algorithm on 48% and 38% of the 1273 and 219 traces. On the Twitter dataset, although SIEVE is the best on only 30% of the traces, it is important to note that no other algorithms are the best on more than 18% of the traces. When using the small cache size, SIEVE, TwoQ is the best algorithm winning on the two Meta datasets. On the other datasets, SIEVE and LHD have similar shares being the best-performing algorithms. The reason for the observation is similar to that previously explained.

4.3 Throughput performance

Besides efficiency, throughput is the other important metric for caching systems. Although we have implemented SIEVE in five different libraries, we focus on Cachelib's results. Because all other libraries implement strict LRU and do not consider object sizes, evaluations yield the same miss ratio as our simulation. Moreover, strict LRU is not scalable, as we show next.

Fig. 6 shows how throughput grows with the number of



(a) Meta KV trace **(b)** Twitter trace

Figure 6: Throughput scaling with CPU cores on two KV-cache workloads.

Table 2: Lines of code requires modification to add SIEVE to a production cache library.

Cache library	Language	Lines
groupcache [25]	Golang	21
mnemonist [6]	Javascript	12
lru-rs [4]	Rust	16
lru-dict [3]	Python + C	21

trace replay threads using two production traces from Meta and Twitter. To better emulate real-world deployments in which the working set size (dataset size) grows with the hardware specs (#cores and DRAM sizes), we scale the cache size and working set size together with the number of threads. To scale the working set size, each thread plays the same trace with the object id transformed into a new space. For example, the benchmark sends 4× more requests to 4× larger cache size at 4 threads compared to the single-thread experiment. We set the cache size to be $4 \times n_{thread}$ GB for both traces, which gives miss ratios of 7% (Meta) and 2% (Twitter). We remark that the miss ratio is close to previous reports [13, 98].

The LRU and TwoQ in Cachelib use extensive optimizations to improve the scalability. For example, objects that were promoted to the head of the queue in the last 60 seconds are not promoted again, which reduces lock contention without compromising the miss ratio. Cachelib further adds a lock combining technique to elide expensive coherence and synchronization operations to boost throughput [38]. As a result of the optimizations, both LRU and TwoQ show impressive scalability results compared to the unoptimized LRU: the throughput is 6× higher at 16 threads than using a single thread on the Twitter trace. As a comparison, unoptimized LRU's throughput plateaus at 4 threads.

Compared to these LRU-based algorithms, SIEVE does not require "promotion" at each cache hit. Therefore, it is faster and more scalable. At a single thread, SIEVE is 16% (17%) faster than the optimized LRU (TwoQ) and on both traces. At 16 threads, SIEVE shows more than 2× higher throughput than the optimized LRU and TwoQ on the Meta trace.

4.4 Simplicity

Prototype implementations. SIEVE not only achieves better efficiency, higher throughput, and better scalability, but it is also very simple. We chose the most popular cache libraries/systems from five different languages: C++, Go, JavaScript, Python, and Rust, and replaced the LRU with SIEVE.

Table 3: Lines of code (excluding comments and empty lines) and per-object metadata size required to implement each algorithm in our simulator. We assume that frequency counter and timestamps use 4 bytes and pointers use 8 bytes.

Algorithm	cache hit	eviction	insertion	metadata size
FIFO	1	4	3	16B
LRU	5	4	3	16B
ARC	64	108	20	17B
LIRS	96	120	64	17B
LHD	192	81	64	13B
LeCaR	72	76	20	40B
CACHEUS	168	140	150	54B
TwoQ	28	16	8	17B
Hyberbolic	4	20	4	16B
CLOCK	4	9	3	17B
SIEVE	4	9	3	17B

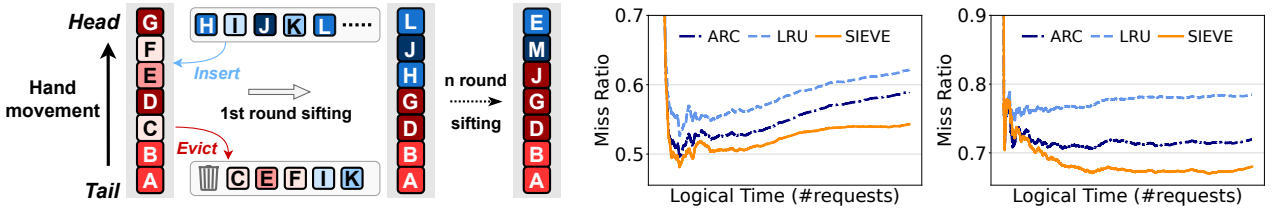
Although different libraries/systems have different implementations of LRU, e.g., most use doubly-linked-list, and some use arrays, we find that implementing SIEVE is very easy. Table 2 shows the number of lines (not including the tests) needed to replace LRU — all implementations require no more than 21 lines of code changes [6].

Advanced algorithms in simulator. Most of the complex algorithms we evaluated in §4.2 are not implemented in production systems. Therefore, we compare the lines of code needed to implement cache hit, insert, and evict in our simulator. Although we implemented our own linked list and hash table data structures in C for our simulator, we do not include the code lines related to list and hash table operations, i.e., appending to the list head or inserting to the hash table requires one line.

Table 3 shows that FIFO requires the fewest number of lines to implement. On top of FIFO, implementing LRU adds a few lines to promote an object upon cache hits. CLOCK and SIEVE require close to 10 lines to implement the eviction function because both need to find the first object that has not been visited. However, we remark that SIEVE is simpler than LRU and CLOCK because SIEVE does not require moving objects to the head of the queue in either hit or miss (evict). Besides these, all other algorithms require one to two orders more lines of code to implement the three functions.

Per-object metadata. In addition to the implementation complexity, we also quantified the per-object metadata needed to implement each algorithm. FIFO does not require any metadata when implemented using a ring buffer. However, such an implementation does not support overwrite or delete. So common FIFO implementation also uses a doubly-linked list with 16 bytes of per-object metadata similar to LRU. CLOCK and SIEVE are similar, both requiring 1-bit to track object access status. When implemented using a doubly linked list,

[6] While most LRU implementations are straightforward to adapt for SIEVE, CacheLib is an exception. Cachelib is highly optimized for LRU-based algorithms. Many optimizations are not needed for SIEVE, making it impractical to quantify code modifications for integration with SIEVE. Therefore, it is not included in Table 2.



(a) Density of colors indicates inherent object popularity (blue: newly inserted objects; red: old objects in each round), and the letters represent object IDs. The first queue captures the state at the start of the first round, and the second queue captures the state at the end of the first round.

(b) Trace 1, full trace (one week)

(c) Trace 2, first two days of a week-long trace

Figure 7: Left: illustration of the sifting process. Right: Miss ratio over time for two traces. The gaps between SIEVE's miss ratio and others enlarge over time.

they use 17 bytes per-object metadata. Compared to SIEVE, advanced algorithms often require more per-object metadata. Many key-value cache workloads have objects as small as 10s of bytes [66, 97], and large metadata wastes the precious cache space.

ZERO parameter. *Besides being easy to implement and having less metadata,* SIEVE *also has no parameters*. Except for FIFO, LRU, CLOCK, and Hyperbolic, all other algorithms have explicit or implicit parameters, e.g., the sizes of queues in LIRS, the learning rate in LeCaR and CACHEUS, and the decay rate and age granularity in LHD. Note that although ARC has no explicit parameters, its adaptive algorithm uses implicit parameters in deciding when and how much space to move between the queues. As a comparison, SIEVE has no parameter and requires no tuning.

5 Distilling SIEVE's Effectiveness

Our empirical evaluation shows that SIEVE is simultaneously simple, fast, scalable, and efficient. In a well-trodden field like cache eviction, SIEVE's competitive performance was a genuine surprise to us as well. We next report our analysis that seeks to understand the secrets behind its efficiency.

5.1 Visualizing the sifting process

The workhorse of SIEVE is the "hand" that functions as a sieve: it sifts through the cache to filter out unpopular objects and retain the popular ones. We illustrate this process in Fig. 7a, where each column (queue) represents a snapshot of the cached objects over time from left to right. As the hand moves from the tail (the oldest object) to the head (the newest object), objects that have not been visited are evicted – the same sweeping mechanism that underlies CLOCK [30, 35]. For example, after the first round of sifting, objects at least as popular as A remain in the cache while others are evicted. The newly admitted objects are placed at the head of the queue — much like the CLOCK policy, but a departure from CLOCK, which does in-place replacements to emulate LRU. During the subsequent rounds of sifting, if objects that survived previous rounds remain popular, they will stay in the cache. In such a case, since most old objects are not evicted, the eviction hand quickly moves past the old popular objects to the queue positions close to the head. This allows newly inserted objects to be quickly assessed and evicted, putting greater eviction pressure on unpopular items (such as "one-hit wonders") than LRU and its variations [67]. As previous work has shown [16, 55, 94], quick demotion is crucial for achieving high cache efficiency.

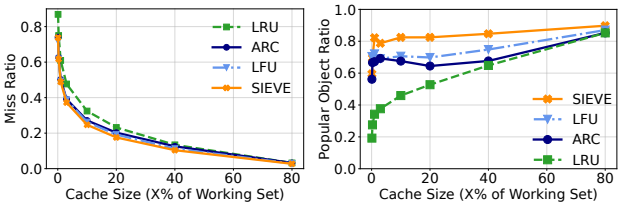
Fig. 7b and Fig. 7c show the cumulative miss ratio over time of different algorithms on two representative production traces. After the cache is warmed up, the miss ratio gaps between SIEVE and other algorithms widen over time, supporting the interpretation that SIEVE indeed sifts out unpopular objects and retains popular ones. A similar observation can be seen in Fig. 10a.

5.2 Analyzing the sifting process

We now analyze the popularity retention mechanism in SIEVE. To clarify the exposition, suppose the SIEVE cache can fit C equally sized objects. Since SIEVE always inserts new objects at the head, and objects that are retained remain in their original positions within the queue, the algorithm implicitly partitions the cache between new and old objects. This partition is dynamic, allowing SIEVE to strike a balance between exploration (finding new popular objects) and exploitation (enjoying hits on old popular objects).

SIEVE performs sifting by moving the hand from the tail to the head, evicting unpopular objects along the way, which we call one round of sifting. We use r to denote the number of rounds. We first enumerate the queue positions p from the tail ($p = 0$) to the head ($p = C - 1$). We then further denote that an object at position p in round r is *examined* (during eviction) or *inserted* at time T_p^r. Note that T effectively defines a logical timer for the examined objects: whenever an object is examined, T increases by 1, regardless of whether the examined object is evicted or retained. In addition, T changes *once* each round for an old object (retained from previous rounds).

For an old object x at position p, we define the "inter-examination time" $I_e(p^r) = T_p^r - T_{p'}^{r-1}$ where p' was the position of x in round $r-1$. Clearly, $p' \geq p$. For a new object inserted in the current round, the inter-examination time is defined as the time between its examination and insertion. We further define an old object x's "inter-arrival time" $I_a(x^r)$ as the time, measured again in the number of objects examined, between the first request to the x in round r and the last re-



(a) Miss ratio over size **(b)** Popular object ratio over size

Figure 8: Miss ratio and popular object ratio on a Zipfian dataset ($\alpha = 1.0$).

quest to x in round $r-1$. For a new object, the inter-arrival time is the time between its insertion and the second request. If an old object is not requested in the last round or a new object does not have a second request, its inter-arrival time is infinite.

In round r, consider two consecutive retained objects x_1 and x_2 at position p_1 and $p_2 = p_1 + 1$. The inter-examination times are $I_e(p_1^r) = T_{p_1}^r - T_{p_1'}^{r-1}$ and $I_e(p_2^r) = T_{p_2}^r - T_{p_2'}^{r-1}$, respectively. The transition yields two invariants:

$$\begin{aligned} T_{p_2}^r - T_{p_1}^r &= 1 \\ T_{p_2'}^{r-1} - T_{p_1'}^{r-1} &\geq 1 \end{aligned}$$

The first equation follows from x_1 and x_2 being consecutively retained objects; the second inequality expresses that other evictions may have taken place between x_1 and x_2 in the previous round. Together, these imply that $I_e(p_1^r) \geq I_e(p_2^r)$. The result generalizes further: for any two retained old objects in the queue, the object closer to the head has a smaller inter-examination time.

Moreover, if an object is retained, its inter-arrival time must be no greater than its inter-examination time. Therefore, for any retained object x at position p_x, its inter-arrival time $I_a(x^r)$ must be smaller than the tail object's inter-examination time:

$$I_a(x^r) \leq I_e(p_x^r) \leq I_e(p_0^r) \tag{1}$$

Using the commonly assumed independent reference model [31, 48, 56, 57] with a Poisson arrival, we can expect any retained object to be more popular than some dynamic threshold set by the tail object's inter-examination time $I_e(p_0^r)$. Since evicting an object keeps the hand pointer at its original position (relative to the tail), the more objects are evicted during a round, the longer the inter-examination time. As a result, SIEVE effectively adapts the popularity threshold so that more objects are retained in the next round.

Following our sifting process metaphor, the mesh size in SIEVE is determined by the tail object's inter-examination time $I_e(p_0^r)$, which is dynamically adjusted based on object popularity change. If too few objects are retained in one round (mesh size too small), then we will have an increased tail inter-examination time $I_e(p_0^r)$ (a larger mesh size) in the next round.

5.3 Deeper study with synthetic workloads

Production trace workloads are often too complex and dynamic to analyze. One consistent finding from past workload characterization work, however, is that object popularity in web cache workloads invariably follows a heavy-tailed power-law (generalized Zipfian) distribution [27, 97]. Therefore, we opted for synthetic power-law workloads for our study. It allows us to easily modify workload features to better understand their impact on performance. Using these synthetic workloads, we further scrutinize SIEVE's effectiveness.

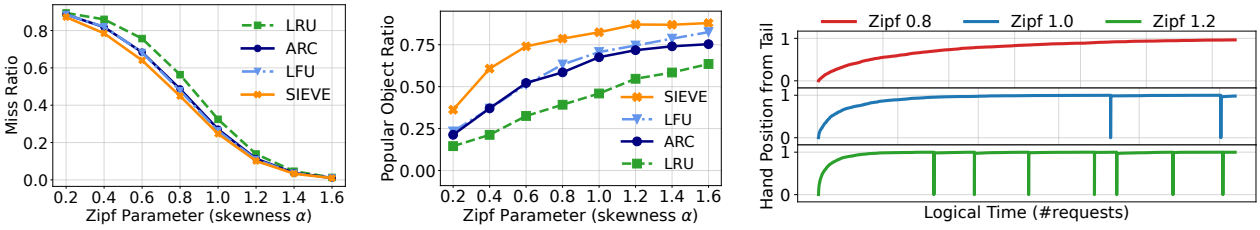
Miss ratio over size. Fig. 8a displays the miss ratio of LRU, LFU, ARC, and SIEVE at different cache sizes. Notably, LFU, ARC, and SIEVE all exhibit lower miss ratios than LRU, demonstrating their efficiency. Despite being considered optimal for synthetic power-law workloads, LFU performs similarly to ARC and is visibly worse than SIEVE. This is because objects with medium popularity, such as objects with ranks around the cache size C, are only requested once before their eviction. LFU cannot distinguish the true popularity of these objects and misses out on an opportunity for better performance. As a comparison, both ARC and SIEVE can quickly remove new and potentially unpopular objects, which allows cached objects to enjoy more time in the cache to demonstrate their popularity. Between the two algorithms, SIEVE further extends the tenure of these objects in the cache because when the hand sweeps through the newly inserted objects, the objects closer to the head must have strictly shorter inter-arrival times (expected to be more popular) to survive.

Popular object ratio over size. To capture how different algorithms manage popular objects, we define a metric called "popular object ratio". Under the assumption of a static and known popularity distribution, the optimal caching policy retains the most popular content within the cache at all times. Given a cache size C and a workload following a power-law distribution, the popular objects are the C most frequent objects in the workload, denoted by H. The popular ratio of objects in the cache at time t is calculated by $I_t = \frac{|H \cap A_t|}{C}$ where A_t denotes the cache contents at time t.

Fig. 8b shows the popular object ratio at different cache sizes. LRU evicts objects based on recency, which only weakly correlates with popularity. In this scenario, LRU stores the least number of popular objects. LFU stores slightly more "popular objects" than ARC. SIEVE, however, successfully filters out unpopular objects from the cache.

Varying the popularity skew. Fig. 8 shows a distribution with Zipfian skewness $\alpha = 1$. We further studied how different concentration of popularity affects SIEVE's effectiveness. Due to space restrictions, we focus on results with large cache sizes for the remainder of this subsection. Results using the small cache size are either similar or do not reveal interesting patterns.

Fig. 9a and Fig. 9b demonstrate the impact of varying skew on miss and popular object ratios. As skew increases, making popular objects more prominent, it becomes easier to identify



(a) Miss ratio **(b)** Popular object ratio **(c)** Hand movement in SIEVE

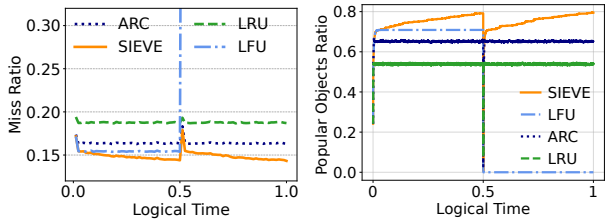
Figure 9: Left two: miss ratio and popular object ratio on Zipfian workloads with different α. Right: hand position in the cache over time in Zipfian workloads.

and cache the popular objects, increasing the popular object ratio and reducing the miss ratio for all tested algorithms. Among ARC, LFU, and SIEVE, we observe that SIEVE always shows a higher popular ratio with a lower miss ratio across skewness, indicating the efficiency of SIEVE is not limited to very skewed workloads.

Fig. 9c illustrates the hand position in the SIEVE cache over time, advancing towards the head with each retained object and pausing during evictions. Therefore, the more objects are retained, the faster the movement. We observe that the hand moves more slowly in the first round than in the later rounds because that is when many unpopular objects are evicted. In subsequent rounds, the hand lingers at positions close to the head for most of the time because SIEVE keeps a new object at position p only if it is more popular (shorter inter-arrival time) than the object at position $p-1$. In other words, SIEVE performs quick demotion [87].

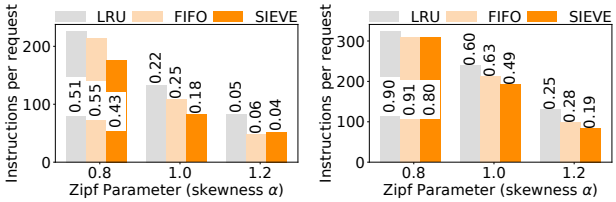
In more skewed workloads, the hand moves quickly due to early arrival and higher request volumes for popular objects, allowing SIEVE to cache most popular objects by the end of the first round. Consequently, the hand rapidly transitions from tail to head with fewer evictions and spends less time near the head, as new objects are more likely to be retained, hastening its progress. Nevertheless, the time of each round varies depending on the frequency of encountering potentially popular objects, highlighting SIEVE's adaptability to workload shifts. When new popular objects appear, the hand accelerates, replacing existing cached objects with the newcomers by giving less time to set their visited bit.

SIEVE is adaptive. To visualize SIEVE's adaptivity via the sifting process, we created a new workload by joining two Zipfian ($\alpha = 1.0$) workloads that request different populations of objects. Fig. 10 shows the interval miss ratio (per 100,000 requests) over time on this conjoined workload. The changeover happens at the 50% midway time mark. We observe that the interval miss ratio of LFU skyrockets to nearly 100% (beyond figure bounds) since new objects cannot replace the old objects. Relative to LRU and ARC, SIEVE's miss ratio spike is larger because it takes time for the hand to move back to the tail before it can evict old objects. However, SIEVE's spike is invisible when the cache size is small (not shown). With respect to the interval miss ratio spike, we observe the popular object ratio of all algorithms (the curves overlap) dropping to



(a) Interval miss ratio **(b)** Popular object ratio over time

Figure 10: Interval miss ratio and popular object ratio over time on a workload constructed by connecting two different Zipfian workloads ($\alpha = 1$).



(a) Large cache **(b)** Small cache

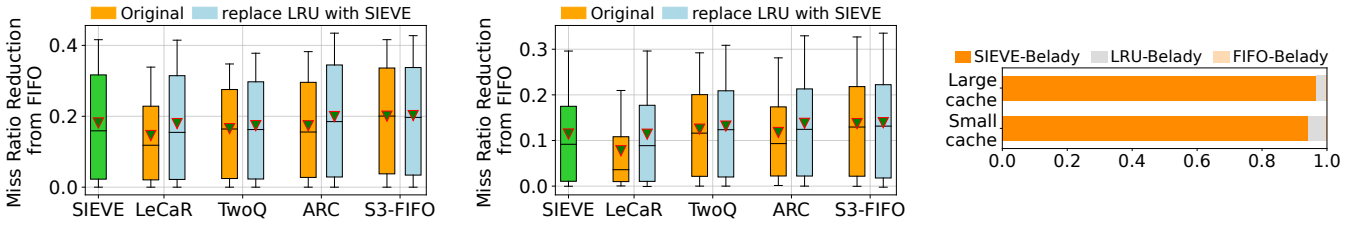
Figure 11: Average number of instructions per request when running LRU, FIFO, and SIEVE caches. The top number denotes the miss ratio.

0 when the workload changes at the midway point. Whereas LFU never recovers from the drop, the popular object ratios in all other algorithms quickly recover to large proportions. Finally, the figures corroborate our interpretation of the sifting process: SIEVE's miss ratio drops over time, while the fraction of popular objects increases over time.

6 SIEVE as a Turn-key Cache Primitive

6.1 Cache primitives

Beyond being a cache eviction algorithm, SIEVE can serve as a cache primitive for designing more advanced eviction policies. To study the range of such policies, we categorize existing cache eviction algorithm designs into four main approaches. **(1)** We can design simple and easy-to-understand eviction algorithms, such as FIFO queues, LRU queues, LFU queues, and Random eviction. We call these simple algorithms *cache primitives*. SIEVE falls under this category. **(2)** We can improve the cache primitives. For example, FIFO-Reinsertion is designed by adding reinsertion to FIFO; LRU-K [70] is designed by changing the recency metric in LRU. **(3)** We can compose multiple cache primitives with objects moved between them. For example, ARC, SLRU, and MQ use multiple LRU queues. **(4)** We can run multiple cache primitives and



(a) Large cache

(b) Small cache

(c) Best-performing algorithms across traces.

Figure 12: Impact of replacing LRU with SIEVE in advanced algorithms (**a**,**b**). The potential of FIFO, LRU, and SIEVE when endowed with foresight (**c**).

craft a decision-maker to select eviction candidates suggested by the primitives. For example, LeCaR [84] uses reinforcement learning to choose between the eviction candidates from LRU and LFU; HALP [80] uses machine learning (MLP) to choose one object from the eight objects at the LRU tail.

Having an efficient cache primitive not only provides an effective and simple eviction algorithm but also enables other approaches to design more efficient algorithms. The ideal cache primitive is simultaneously (1) simple, (2) efficient, and (3) fast — in terms of high throughput. For example, FIFO and LRU meet these requirements and are frequently used to construct more advanced algorithms. However, they are less efficient than complex algorithms.

While we have shown that SIEVE is simple, efficient, and fast in §4, to further understand SIEVE as a cache primitive, we compare the number of instructions needed to run FIFO, LRU, and SIEVE caches. We remark that the number of instructions may not necessarily correlate with latency or throughput but rather a rough metric of CPU resource usage. We used `perf stat` to measure the number of instructions for serving power-law workloads (100 million requests, 1 million objects) in our simulator. We then deduct the simulator overhead by measuring a no-op cache, which performs nothing on cache hits and misses.

Fig. 11 shows that SIEVE generally executes fewer instructions per request than FIFO and LRU, a difference accentuated in skewed workloads and larger cache sizes. Compared to LRU, SIEVE requires fewer instructions since SIEVE needs only to check and possibly update a Boolean field on cache hits, which is much simpler than moving an object to the head of the queue. Besides LRU, SIEVE also requires fewer instructions than FIFO because of the difference in miss ratios. Because SIEVE has a lower miss ratio than FIFO, fewer objects need to be inserted due to cache misses, leading to fewer instructions per request on average. The only exception is when SIEVE and FIFO have similar miss ratios, in which case, FIFO executes fewer instructions than SIEVE. Overall, SIEVE requires up to 40% and 24% fewer instructions than LRU and FIFO, respectively.

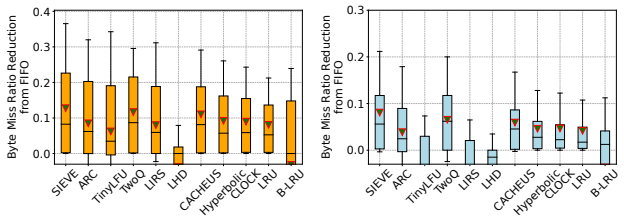
6.2 Turn-key cache eviction with SIEVE

As a cache primitive, SIEVE can facilitate the design of more advanced eviction algorithms. To understand the benefits of using a better cache primitive, we replaced the LRU in LeCaR, TwoQ, and ARC with SIEVE. Note that for ARC, we only replace the LRU for frequent objects.

We evaluate these algorithms on all traces and show the miss ratio reduction(from FIFO) in Fig. 12a and Fig. 12b. Compared to SIEVE, LeCaR has much lower efficiency; however, when replacing the LRU in LeCaR with SIEVE, it significantly reduces LeCaR's miss ratio by 4.5% on average. TwoQ and ARC achieve efficiency close to SIEVE; however, when replacing the LRU with SIEVE, the efficiency of both algorithms gets boosted. For example, ARC-SIEVE achieves the best efficiency among all compared algorithms at both small and large cache sizes. It reduces ARC's miss ratio by 3.7% on average and up to 62.5% on the large cache size (recall that ARC reduces LRU's miss ratio by 6.3% on average). ARC-SIEVE also reduces SIEVE's miss ratio by an average of 2.4% and up to 40.6%.

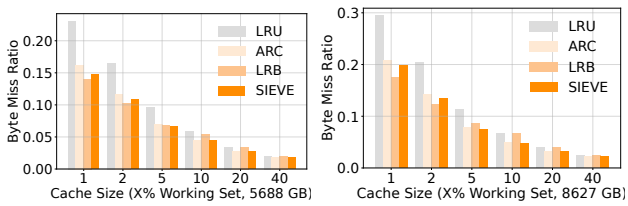
To understand the potential in suggesting eviction candidates, we evaluated the efficiency of FIFO, LRU, and SIEVE, granting them access to future request data. Each eviction candidate is either evicted or reinserted, depending on whether the object will be requested soon. We assume that an object will be requested soon if the logical time (number of requests) till the object's next access is no more than $\frac{C}{mr}$, where C is the cache size and mr is the miss ratio. This mimics the case that we have a perfect decision-maker choosing between the eviction candidates suggested by multiple simple eviction algorithms. Fig. 12c shows that when supplied with this additional information, SIEVE achieves the lowest miss ratio on 97% and 94% of the 1559 traces at the large and small cache size, respectively.

These results highlight the potential of SIEVE as a powerful cache primitive for designing advanced cache eviction algorithms. Leveraging lazy promotion and quick demotion, SIEVE not only performs well on its own but also bolsters the performance of more complex algorithms.



(a) Large cache **(b)** Small cache

Figure 13: Byte miss ratio across all CDN traces.



(a) Wiki2018 trace **(b)** Wiki2019 trace

Figure 14: Byte miss ratios at different cache sizes on two Wiki CDN traces used in LRB evaluation.

7 Discussion

7.1 Byte miss ratio

To gauge SIEVE's efficiency in reducing network bandwidth usage in CDNs, we analyzed its byte miss ratio by considering object sizes. We chose the cache size at 10% and 0.1% of the trace footprint in bytes. Fig. 13a and Fig. 13b show that SIEVE presents larger byte miss ratio reductions at *ALL* percentiles than state-of-the-art algorithms at both cache sizes, showcasing its high efficiency in CDN caches.

We further compared SIEVE with LRB [79], the state-of-the-art machine-learning-based cache eviction algorithm optimized for byte miss ratio. Due to LRB's long run time, we only evaluated LRB on the two open-source Wiki traces provided by the authors. Fig. 14a and Fig. 14b show that LRB performs better at small cache sizes (1% and 2%), while SIEVE excels at larger cache sizes. We conjecture that at a small cache size, the ideal objects to cache are popular objects with many requests, which LRB can more easily identify because they have more features (most of LRB's features are about the time between accesses to an object). When the cache size is large, most objects in the cache have few requests. Without enough features, a learned model can provide little benefits [94, 99]. In summary, compared to complex machine-learning-based algorithms, SIEVE still has competitive efficiency.

7.2 SIEVE is not scan-resistant

Besides web cache workloads, we evaluated SIEVE on some block cache workloads. However, we find that SIEVE sometimes shows a miss ratio higher than LRU. The primary reason for this discrepancy is that SIEVE is not scan-resistant. In block cache workloads, which frequently feature scans, popular objects often intermingle with objects from scans. Consequently, both types of objects are rapidly evicted after insertion. Since SIEVE does not use a ghost cache — a shadow cache that keeps track of recently evicted items to make smarter future eviction decisions — it cannot recognize the popular objects when they are requested again. This problem is less severe on the large cache size, but when the cache size is small, we observe that having a ghost is critical to be scan-resistant. We conjecture that not being scan-resistant is probably the reason why SIEVE remained undiscovered over the decades of caching research, which has been mostly focused on page and block accesses.

7.3 TTL-friendliness

Time-to-live (TTL) is a common feature in web caching [97, 98]. It specifies the duration during which an object can be used. After the TTL has elapsed, the object expires and can no longer be served to the user, even if it may still be cached. Most existing eviction algorithms today do not consider object expiration and require a separate procedure, e.g., scanning the cache, to remove expired objects. Similar to FIFO, SIEVE maintains objects in insertion order, which allows objects in TTL-partitioned caches, e.g., Segcache [98], to be sorted by expiration time. This provides a convenient method for discovering and removing expired objects.

8 Conclusion

We design SIEVE, a simple, efficient, fast, and scalable cache eviction algorithm for web caches that leverages "lazy promotion" and "quick demotion". The high efficiency in SIEVE comes from gradually sifting out the unpopular objects. SIEVE *is the first and the simplest cache primitive that supports both lazy promotion and quick demotion*. This serves as the foundation for SIEVE's high efficiency and high performance. Evaluated on 1559 traces from 7 datasets, we show that SIEVE outperforms complex state-of-the-art algorithms on over 45% of the traces. We implemented SIEVE in five open-source production libraries using less than 20 lines on average.

Availability

The code and data used in this work are open-sourced at `https://github.com/cacheMon/NSDI24-SIEVE`. This repository includes both the simulator and prototypes.

Additionally, we have engineered cache libraries based on SIEVE for various programming languages. More information is available at `https://sievecache.com`.

Acknowledgments

We thank the anonymous reviewers and our shepherd Kay Ousterhout for constructive suggestions. We are grateful to the individuals and organizations that have generously open-sourced and shared production traces. We thank Cloudlab [40] for the infrastructure support for running experiments. We also appreciate the members of SimBioSys and PDL Consortium for their interest, insights, feedback, and support.

References

[1] Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2023-04-27.

[2] Apache traffic server. https://trafficserver.apache.org/. Accessed: 2023-04-27.

[3] lru-dict. https://github.com/amitdev/lru-dict. Accessed: 2023-04-27.

[4] lru-rs. https://github.com/jeromefroe/lru-rs. Accessed: 2023-04-27.

[5] Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2023-04-27.

[6] mnemonist. https://github.com/Yomguithereal/mnemonist. Accessed: 2023-04-27.

[7] Nginx. https://nginx.org/. Accessed: 2023-04-27.

[8] pelikan. https://github.com/pelikan-io/pelikan. Accessed: 2023-04-27.

[9] Redis. http://redis.io/. Accessed: 2023-04-27.

[10] Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval. Accessed: 2023-04-27.

[11] Varnish cache. https://varnish-cache.org/. Accessed: 2023-04-27.

[12] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. ACME: Adaptive Caching Using Multiple Experts. In *WDAS*, volume 2, pages 143–158, 2002.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture*, HPCA'17, pages 109–120, Austin, TX, February 2017. IEEE.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[20] 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.

[21] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX symposium on operating systems design and implementation*, OSDI'18, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[22] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: 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.

[23] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018.

[24] 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.

[25] bradfitz. group cache. https://github.com/golang/groupcache. Accessed: 2023-04-27.

[26] 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.

[27] 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.

[28] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM'18, pages 84–95, Philadelphia PA USA, June 2018. ACM.

[29] 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.

[30] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, December 1981. Association for Computing Machinery.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[37] Meta developers. Cachelib. https://cachelib.org. Accessed: 2023-04-27.

[38] Meta developers. Distributed mutex. https://github.com/facebook/folly/blob/2c00d14adb9b632936f3abfbf741373871cd64a6/folly/synchronization/DistributedMutex.h. Accessed: 2023-04-27.

[39] 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.

[40] 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.

[41] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, Rennes France, November 2018. ACM.

[42] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage*, 18(3), August 2022.

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

[44] 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.

[45] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of machine learning and systems*, volume 1 of *mlsys'20*, pages 40–52, 2019.

[46] 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.

[47] 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.

[48] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[49] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, 2007.

[50] Xiaoming Gu and Chen Ding. On the theory and potential of lru-mru collaborative cache management. *SIGPLAN Not.*, 46(11):43–54, jun 2011.

[51] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 283–294, 2008.

[52] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM 2014-IEEE conference on computer communications*, pages 460–468. IEEE, 2014.

[53] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX annual technical conference*, ATC'16, pages 351–364, Denver, CO, June 2016. USENIX Association.

[54] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '22, pages 72–90, New York, NY, USA, November 2022. Association for Computing Machinery.

[55] 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.

[56] Stratis Ioannidis, Laurent Massoulie, and Augustin Chaintreau. Distributed caching over heterogeneous mobile networks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'10, pages 311–322, 2010.

[57] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 113–124, Antibes Juan-les-Pins France, June 2016. ACM.

[58] 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.

[59] 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.

[60] 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.

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

[62] Cong Li. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 59–64, New York, NY, USA, June 2018. Association for Computing Machinery.

[63] Conglong Li and Alan L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1–15, Bordeaux France, April 2015. ACM.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] Sailesh Mukil. Cache warming: Leveraging ebs for moving petabytes of data. https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3. Accessed: 2023-04-27.

[69] 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.

[70] 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.

[71] Sejin Park and Chanik Park. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[72] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern software. In *Twenty-third EuroSys Conference*, EuroSys'23, New York, NY, USA, 2023. Association for Computing Machinery.

[73] 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.

[74] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems*, SIGMETRICS'90, pages 134–142, New York, NY, USA, 1990. Association for Computing Machinery.

[75] 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.

[76] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021. ACM.

[77] 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.

[78] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[79] 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.

[80] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. Halp: Heuristic aided learned preference eviction policy for youtube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 1149–1163, Boston, MA, April 2023. USENIX Association.

[81] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 41–54, 2004.

[82] 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*, pages 55–67, 2017.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] Nan Wu and Pengcheng Li. Phoebe: Reuse-Aware Online Caching with Reinforcement Learning for Emerging Storage Models, November 2020.

[89] 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.

[90] 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.

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

[92] Juncheng Yang. libcachesim: a high-performance library for building cache simulators. https://libcachesim.com/. Accessed: 2023-04-27.

[93] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. FAST'23, pages 115–134, 2023.

[94] 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 *The 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, 2023.

[95] 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.

[96] Juncheng Yang, Yao Yue, and K. V. Rashmi. Slides of a large scale analysis of hundreds of in-memory cache clusters at twitter. https://www.usenix.org/sites/default/files/conference/protected-files/osdi20_slides_yang.pdf. Accessed: 2023-04-27.

[97] 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.

[98] 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.

[99] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K.V. Rashmi. Fifo queues are all you need for cache eviction. In *Symposium on Operating Systems Principles (SOSP'23)*, 2023.

[100] 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.

[101] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, FAST'13, pages 59–72, USA, February 2013. USENIX Association.

[102] 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.

[103] 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.

[104] 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.

[105] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, June 2004.

[106] 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.