Demystifying and Improving Lazy Promotion in Cache Eviction

Qinghan Chen
Carnegie Mellon University
Pittsburgh, PA
qinghanc@andrew.cmu.edu

Muhammad Haekal Muhyidin Al-Araby
Sepuluh Nopember Institute of Technology
Surabaya, Indonesia
5024221030@student.its.ac.id

Ziyue Qiu
Carnegie Mellon University
Pittsburgh, PA
ziyueqiu@andrew.cmu.edu

Zhuofan Chen
Carnegie Mellon University
Pittsburgh, PA
zhuofanc@andrew.cmu.edu

Rashmi Vinayak
Carnegie Mellon University
Pittsburgh, PA
rvinayak@cs.cmu.edu

Juncheng Yang
Harvard University
Cambridge, MA
juncheng@seas.harvard.edu

ABSTRACT

Cache eviction algorithms play a critical role in the performance of modern data systems, yet their scalability is often limited by the high computational overhead associated with object promotions. Lazy Promotion techniques have emerged as relaxations of traditional Least-Recently-Used (LRU) methods, designed to alleviate lock contention and increase throughput. This work uses production traces from real-world systems to benchmark five Lazy Promotion strategies: Probabilistic-LRU, Batch-LRU, Delay-LRU, FIFO-reinsertion, and Random-LRU. We evaluate these techniques across miss ratio, scalability, promotion count, and a novel metric called promotion efficiency, which measures the number of hits per promotion. Our results reveal that Delay-LRU and FIFO-reinsertion significantly improve promotion efficiency, whereas Batch-LRU and Probabilistic-LRU struggle to reduce promotions without significantly increasing miss ratio. We further explore the impact of lazy promotion in advanced algorithms such as ARC and 2Q and make a similar observation. Moreover, we uncover substantial optimization potential, showing that most cache promotions are unnecessary when equipped with oracle knowledge. To further reduce promotions in LRU, we propose two novel enhancements—Delayed FIFO-reinsertion (D-FR) and Age-Guided Eviction (AGE)—that reduce promotions by 20—60% while achieving a similar or lower miss ratio.

PVLDB Reference Format:
Qinghan Chen, Muhammad Haekal Muhyidin Al-Araby, Ziyue Qiu, Zhuofan Chen, Rashmi Vinayak, and Juncheng Yang. Demystifying and Improving Lazy Promotion in Cache Eviction. PVLDB, 19(4): 549 - 562, 2025.
doi:10.14778/3785297.3785299

PVLDB Artifact Availability:
The source code and data are available at https://github.com/cacheMon/Lazy-Promotions.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785299

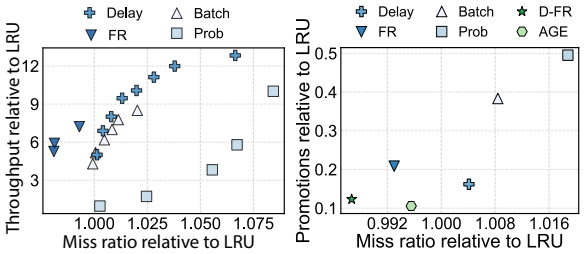


Figure 1: Left: overview of Lazy Promotion techniques. delay and FIFO-reinsertion (FR) are more effective than Batch-LRU and Probabilistic-LRU on reducing promotions and improving scalability (throughput at 16 threads) without significantly increasing the miss ratio. **Right**: We introduce two simple techniques, D-FR and AGE, to reduce promotions without increasing miss ratio.

1 INTRODUCTION

Caching plays an important role in data systems. A cache stores a small portion of popular objects on a fast storage device, allowing requests to be served very quickly from the cache. Caching is now integral to nearly every layer of modern computer systems. For example, databases rely on the buffer pool to load data quickly [28, 37]. In MySQL, the InnoDB buffer pool [31] acts as a cache, storing frequently accessed rows and index pages in memory. This reduces disk I/O, enabling faster data retrieval and improving query performance. Moreover, an operating system relies on the page cache to access data swiftly [2], and a content delivery network uses caching at the edge of the Internet to deliver images and videos to end users cheaply and quickly [39].

The efficiency of a cache is measured by *miss ratio*—the proportion of data requests that must be served by the backend. Reducing miss ratio allows more requests to be served from the cache and enables a faster and more responsive system. Besides efficiency, the other important metric of a cache is *throughput*, which measures the hits a cache can serve every second. A faster cache enables the operator to utilize fewer computational resources, thereby enhancing the utility of caching. Lastly, modern computer servers often have tens to hundreds of cores per socket [6, 7, 15, 53], making *scalability*—the capability to scale throughput with the number of cores, a critical requirement.

At the core of caching is the cache eviction algorithm, which determines the objects stored in the cache and the order in which they are evicted. A good eviction algorithm maximizes the likelihood of cache hits, thus minimizing backend accesses and reducing data access latency. Least-Recently-Used (LRU) is the most popular eviction algorithm used in many production systems [3, 18, 56]. LRU uses a doubly-linked list to maintain objects in the order of their last access time, positioning the most recently used item at the head and the least recently used at the tail. When an object is accessed, it is *promoted* to the head of the list. During eviction, the object at the tail is removed. Because operations on items in the middle of a linked list cannot be performed atomically, all the operations in LRU require locking. This significantly limits the throughput of an LRU cache when using multiple CPU cores, as it must wait to acquire the lock. Although FIFO is more scalable than LRU, it has not been widely adopted because LRU is widely considered more efficient than FIFO, achieving a lower miss ratio [10, 17].

As scalability has increasingly become a concern for modern cache systems, production system engineers have developed different relaxations of LRU promotions to mitigate the scalability bottleneck. For example, Meta Cachelib [11] delays the promotion of a recently promoted object to reduce lock contention; Meta HHVM [45] employs a probabilistic LRU using try-lock — promote an object to the head of the linked list only upon a successful lock operation; Twitter Segcache [66] and Google Cliquemap [52] both use a batched eviction; and RocksDB [21] and PostgreSQL [47] use FIFO-reinsertion to reduce lock contention. Because these techniques primarily improve scalability by reducing the number of promotions and lock contention, similar to previous work [63], we refer to them as Lazy Promotion techniques.

Despite the broad deployment of Lazy Promotion techniques in production systems, there has not been a good understanding in the effectiveness of these techniques. It is often believed that Lazy Promotion makes a tradeoff between cache efficiency and scalability. As a result, these relaxations are sometimes called weak LRU [16, 29] because they sacrifice miss ratio to improve throughput and scalability.

We implement and benchmark Lazy Promotion techniques at scale to understand their effectiveness in improving scalability, reducing promotions, and maintaining miss ratio. We evaluate five different Lazy Promotion techniques from real-world systems: Probabilistic-LRU (§3.1), Batch-LRU (§3.2), Delay-LRU (§3.3), FIFO-reinsertion (§3.4), and Random-LRU (§3.5) using traces from 9 different companies (Table 2), including Alibaba [36, 60], Cloudphysics [58], Meta [4], Microsoft [40], Tencent[75], Twitter [64], Wikipedia [61], and two content delivery networks. In total, we use 6357 traces of 346 billion requests for 2,818 TB of data.

We propose a new metric, *promotion efficiency*, to evaluate the effectiveness of different Lazy Promotion techniques. It measures the (average) number of hits a promotion leads to. A Lazy Promotion technique with high promotion efficiency indicates that its promotions are more valuable. LRU has a mean promotion efficiency of 0.037—each promotion only leads to 0.037 hits. Among the Lazy Promotion techniques, we find that Delay-LRU and FIFO-reinsertion are most effective. When the miss ratio is within 1% of LRU's performance, Delay-LRU and FIFO-reinsertion have promotion efficiency of as high as 0.4 and 0.3 hits per promotion, respectively. Batch-LRU is worse, achieving less than 0.1 hits per promotion. Probabilistic-LRU performs the worst because it randomly reduces promotion without distinguishing between popular and unpopular objects.

Many advanced eviction algorithms consist of one or more LRU queues. Our findings are not limited to LRU. We add Lazy Promotion to ARC [37] and 2Q [30]. We find that with Delay-LRU and FIFO-reinsertion can effectively reduce the number of promotions in ARC and 2Q, meanwhile Probabilistic-LRU and Batch-LRU are less effective, often leading to significant increases in miss ratio.

Using Belady's MIN [10] as a reference for optimal cache eviction, we find that most cache promotions in LRU are unnecessary, with only 10% needed on average. This finding underscores the potential for optimization. Moreover, we find that promotion (reinsertion) on eviction (as in FIFO-reinsertion) is strictly better than promotion on cache hits. If we equip FIFO-reinsertion with future knowledge, it can reduce promotions by over 90% on average across all 6,357 traces while also slightly reducing miss ratio.

Leveraging our findings, we introduce two simple Lazy Promotion techniques, Delayed FIFO-reinsertion (D-FR) and Age-Guided Eviction (AGE), to further reduce the number of promotions and improve promotion efficiency. D-FR leverages delay in FIFO-reinsertion to reduce promotions, while AGE utilizes recency information to filter out unnecessary reinsertions in FIFO-reinsertion. Our evaluations on production traces show that these techniques achieve a lower or similar miss ratio while further reducing promotions by 20% – 60% and improving the promotion efficiency on average by more than 80%.

We make the following contributions in this paper:

- We implemented and benchmarked the effectiveness of five different Lazy Promotion techniques from real-world systems and integrated them into advanced algorithms, including ARC and 2Q, using 6357 production traces.
- We introduced a new metric, promotion efficiency, to evaluate how well algorithms minimize promotions while keeping a low miss ratio. Our results show that Delay-LRU and FIFO-reinsertion improve promotion efficiency.
- We discovered that most promotions are unnecessary, suggesting a huge potential to further reduce promotions.
- We present two simple techniques, D-FR and AGE, to reduce the unnecessary promotions while maintaining a similar or lower miss ratio. Evaluation on production traces shows that they can reduce promotions by 20–60% without impacting miss ratio.

2 BACKGROUND AND MOTIVATION

2.1 Software cache and eviction algorithm

Caching plays a vital role in modern data systems, enabling efficient data access across various scenarios. In compute-storage disaggregated architectures, caching mitigates the high latency of network-based data transfers [51, 59, 69]. Similarly, database systems use caching to store frequently accessed data in memory, reducing disk I/O and improving query response times.

At the core of effective caching lies the eviction algorithm, which determines which objects remain in a cache when it reaches capacity. Two of the most popular algorithms, Least Recently Used (LRU) and First-In-First-Out (FIFO), represent two extremes in managing

Table 1: Descriptions of different Lazy Promotion techniques.

Algorithm name	Algorithm parameter	Description	Deployed systems
Delay-LRU	*delay ratio*	Skip recent promotions	Meta Cachelib [18]
Probabilistic-LRU	*prob*	Skip promotions by chance	Meta HHVM [45]
Batch-LRU	*batch ratio*	Batch promotions between fixed intervals	CliqueMap [52], Ristretto [19]
FIFO-reinsertion	frequency bits	Defers promotions to eviction	RocksDB [21]
Random-LRU	*sample size*	Sample eviction candidates and evict based on last access time	Redis [46]

cache hits, with respect to two critical metrics: *efficiency* (measured by the miss ratio) and *scalability* (measured by throughput, or the rate of requests served per second).

LRU and its variants are the most widely used algorithms in production [2, 3, 18, 38, 41, 43, 56]. LRU retains the most recently accessed objects, based on the principle that recently accessed pages are likely to be reused in the near future [10, 17]. LRU usually achieves high efficiency but poor scalability due to its need to *promotion*: moving the accessed object to the head in a doubly linked list , a process that must be protected by a global lock to ensure consistency [49, 63]. This lock contention restricts throughput growth as the number of CPU cores increases, making LRU poorly suited for highly parallel environments, such as databases.

FIFO evicts the oldest objects without considering access patterns. It excels in scalability without promotions or locking during cache hits. However, its simplicity often results in poor efficiency, as it fails to retain frequently accessed data. The two algorithms represent opposite ends of the spectrum: LRU promotes every object upon a hit, while FIFO promotes none.

2.2 Lazy promotion techniques

To reduce the computation per cache hit and make LRU more scalable, Lazy Promotion techniques have emerged in real-world systems as relaxations of the LRU algorithm [63]. These techniques focus on reducing the frequency of promotions during cache hits while maintaining low miss ratios. In this subsection, we describe these Lazy Promotion techniques, highlighting their underlying intuitions and design principles with a summary in Table 1.

Probabilistic-LRU. A straightforward solution to reducing the number of promotions is to avoid them when they become a bottleneck. HHVM [45], developed by Meta, employs probabilistic promotion using try-lock. Promotions are randomly skipped based on lock availability: an object is promoted only when a thread successfully acquires a lock using try-lock. If the lock is unavailable, the promotion is bypassed. As a result, the likelihood of promotion is directly tied to the thread's ability to acquire the lock. Under high contention, most promotions are bypassed to avoid waiting for the lock, improving the scalability.

Batch-LRU. Because each promotion requires an expensive lock operation, another solution to improve scalability is to increase the critical section size and reduce the number of locking operations. For example, BP-wrapper [20] batches the LRU promotions to improve scalability. Similarly, Google's CliqueMap [52] is a distributed caching system that also uses Batch-LRU for eviction. In distributed deployments, the cache resides on the server, and clients use one-sided Remote Direct Memory Access (RDMA) to retrieve data directly without communicating with the server. As a result, the server remains unaware of the access pattern and cannot decide which objects to evict when writing new objects. To address this problem, clients *periodically* share requested objects with the cache server using RPC. This allows the server to perform batched promotions. This batching strategy reduces the promotion overhead in distributed caches and the scalability bottleneck in local caches.

Delay-LRU. Most of the cache requests are for popular objects, which do not require frequent promotion to stay in the cache. Therefore, we do not need to promote an object if it was promoted recently. Moreover, it takes a long time for a promoted object to traverse through the cache before being evicted. As long as the next promotion happens before eviction, it is sufficient. Leveraging this insight, Meta CacheLib [11] uses Delay-LRU. It employs a tunable parameter that controls the *delay ratio*, ensuring that promotions are triggered only after a specified duration has elapsed since the last promotion. Increasing the *delay ratio* reduces the number of promotions, improving scalability; however, doing so also takes the risk that useful objects might be evicted, increasing the miss ratio.

FIFO-reinsertion. A traditional solution to reduce promotion overhead is to delay promotion until eviction time. The corresponding algorithm has different names based on the underlying implementation, e.g., CLOCK, FIFO-Reinsertion, or Second Chance. In the following text, we use FIFO-reinsertion to refer to this algorithm. Because FIFO-reinsertion does not need to move the requested object to the head upon each request, it can be implemented using a ring buffer or atomic updates on the linked list. This not only improves throughput but also scalability. For example, the new eviction algorithm for the block cache in RocksDB [21] and PostgreSQL [47] uses FIFO-reinsertion to improve scalability

Random-LRU. Although a linked list is the most common data structure for an LRU cache implementation, it brings the unavoidable scalability problem. An alternative approach to implement an approximate LRU can leverage random sampling at eviction. For example, Redis [46] employs a Random-LRU policy that randomly samples k (e.g., 5) objects from the cache and evicts the least recently used one among them. This approach addresses the scalability issue of LRU by reducing the need for locking during cache hits. Random-LRU does not need to maintain a fully ordered list. Instead, each object tracks the last access time, updating which does not require locking, thus improving the thread scalability. Although Random-LRU is not a Lazy Promotion technique technically because it does not perform any promotion, we include it in our study because it is also an approximate LRU that improves scalability.

3 HOW DO EXISTING LAZY PROMOTION TECHNIQUES PERFORM?

This section presents our benchmarks and measurements of different Lazy Promotion techniques.

Table 2: Datasets used in this work. For old datasets, we exclude traces with fewer than 1 million requests.

Trace collections	Approx time	Cache type	time span (days)	# Traces	# Request (million)	Request (TB)	# Object (million)	Object (TB)
MSR [40]	2007	Block	7	14	410	9.8	73	3.0
FIU [33]	2008–11	Block	9–28	9	513	1.6	19	0.056
Cloudphysics [58]	2015	Block	7	106	2,115	80.7	492	21.9
CDN 1	2018	Object	7	301	3,429	3,227	237	207
Tencent Photo [75]	2018	Object	8	2	5,649	147	1,038	23.7
WikiMedia CDN [61]	2019	Object	7	4	12,400	195	249	13.1
Tencent CBS [71]	2020	Block	8	4,048	33,268	1,143	2,113	108
Alibaba [1, 36, 60]	2020	Block	30	609	20,038	647	1,676	112
Twitter [64]	2020	KV	7	51	233,965	103	17,342	4.9
CDN 2	2021	Object	7	1,205	20,470	2,079	1,232	732
Meta KV [4]	2022	KV	1	5	13,314	8.2	371	0.43
Meta CDN [4]	2023	Object	7	3	231	8,594	76	1,527

Dataset and testbed. We used a diverse set of real-world traces from sources including Alibaba[25], CloudPhysics[58], FIU[33], Meta, Tencent[70, 74], Wikipedia[61], and two private CDN service companies for our measurements. These traces were collected between 2007 and 2023, covering key-value, block, and object caches. Altogether, the datasets encompass 6357 traces with *346 billion requests* for *25 billion objects*, with *16,625 TB of traffic* for a total of *2,818 TB of data*. More details can be found in Table 2.

Miss Ratio Measurement. We implemented each Lazy Promotion technique on top of libCacheSim [5] and replayed the traces in our dataset to measure miss ratio. We evaluate each technique using three cache sizes: 0.1%, 1%, and 10% of the working set size (the total number of objects in the trace). Because the diverse traces exhibit different access patterns, the miss ratios at the same cache size, e.g., 1% of the working set size, can range from less than 1% to more than 80%. We use the LRU miss ratio as a baseline, and calculate the relative miss ratio as

$$miss_A / miss_{LRU}$$

for a technique A being studied. We present the results using boxplots, where the box represents the 25th and 75th percentiles, and the whiskers represent the 10th and 90th percentiles. We used a cluster of c8220 nodes on Cloudlab for miss ratio measurements [1].

Scalability Measurement. We implemented concurrent versions of each Lazy Promotion technique. Unlike miss ratio measurements, which can be performed in parallel, the throughput/scalability measurements cannot be performed in parallel due to interference. Moreover, running all 6,357 real-world traces for every Lazy Promotion technique and parameter configuration would require over a year to complete. Therefore, we performed the scalability measurement using a synthetic Zipfian trace consisting of 10 million requests for 1 million unique objects, with a skewness parameter of 1.0. The Zipfian parameter was selected to capture the cache access patterns observed in real-world workloads [8, 11, 13, 65]. Because miss ratio can affect throughput, we chose different cache sizes for different Lazy Promotion techniques so that they all achieve the same miss ratio. We experimented with both 10% and 1% miss ratios; however, we only present the 1% miss ratio results due to space constraints. For the ease of visualization, we normalized the throughput of each Lazy Promotion technique relative to the LRU baseline by calculating the ratio

$$throughput_A / throughput_{LRU}$$

for a Lazy Promotion technique A. All measurements were conducted using r650 servers from Cloudlab, which are equipped with dual Intel Xeon Platinum 8360Y 36-core processors and 256GB of DDR4 DRAM. We disabled hyperthreading and turbo-boost, and limited our benchmark to use only one NUMA domain to ensure consistency across different benchmarks. Moreover, we repeated each experiment five times using randomly generated Zipfian traces and reported the mean results.

Promotion Measurement. While the throughput under different numbers of threads shows scalability, the throughput results depend on hardware and implementations. Therefore, we also measured the *number of promotions* as a deterministic cost metric to capture the benefit of Lazy Promotion . Recall that each promotion requires locking, which limits the scalability. Fewer promotions lead to fewer lock operations and better scalability. This metric is also computed relatively to LRU, as

$$\#\ promos_A / \#\ promos_{LRU}$$

for a Lazy Promotion technique A.

Promotion Efficiency Measurement. i.e.,

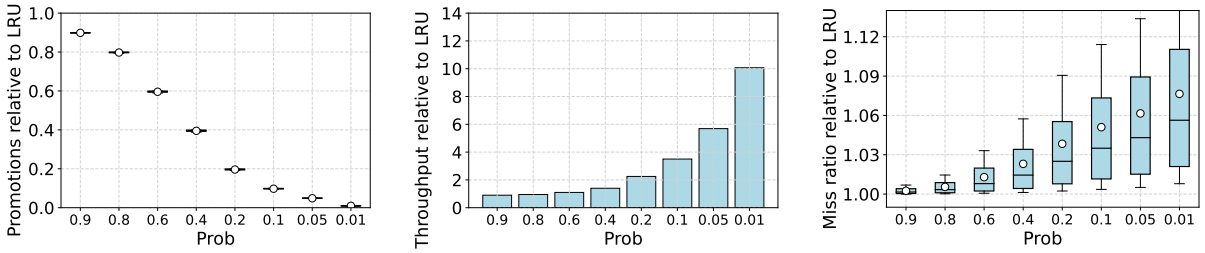
$$(\#miss_{FIFO} - \#miss_A) / \#promos_A$$

We use FIFO as a baseline in this metric because it performs no promotion and is the extreme of a Lazy Promotion technique. A high promotion efficiency indicates that the promotions in the technique are more valuable.

Although we performed measurements using three different cache sizes —0.1%, 1%, and 10% of the working set size —we found that the observations across different cache sizes are similar. Therefore, we only present the results at the 1% cache size for space reasons. We will include the results of different cache sizes in our open-source repository.

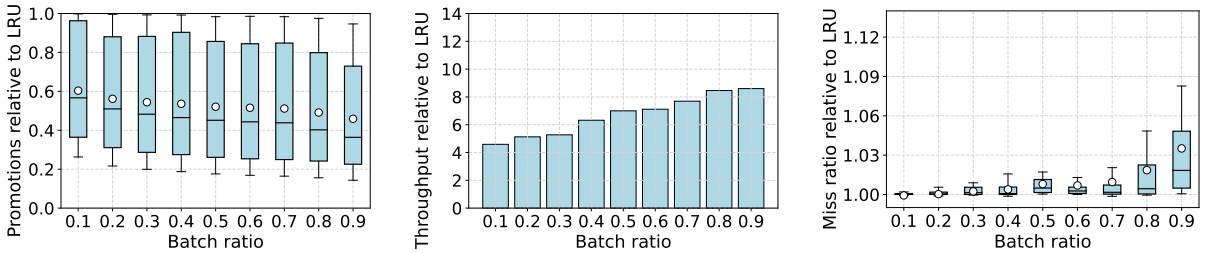
3.1 Probabilistic-LRU

[1] As the miss ratio is deterministic and independent of the underlying hardware, the specific hardware used does not affect the outcome of these measurements.



(a) Relative number of promotions **(b) Throughput Imprv (16 threads)** **(c) Relative miss ratio**

Figure 2: Probabilistic-LRU skips promotions randomly based on a given parameter *prob*. It reduces the number of promotions proportionally to the *prob*. However, the throughput does not increase significantly until the probability is reduced to very low. Meanwhile, Probabilistic-LRU increases miss ratio even when the Probabilistic-LRU is low.



(a) Relative number of promotions **(b) Throughput Imprv (16 threads)** **(c) Relative miss ratio**

Figure 3: Batch-LRU promotes objects in batches to reduce the number of promotions. A larger batch (*batch ratio*) leads to fewer promotions and higher throughput, but also a higher miss ratio.

```
p = random(0,1)
if p < prob:
    promote(obj)
```

Listing 1: promotion during cache hit in Probabilistic-LRU

The root cause of LRU's poor scalability is the large number of promotions and corresponding locking operations. Therefore, Probabilistic-LRU skips promotions randomly based on a given probability *prob*, which ranges from 0 to 1. When *prob* is 1, Probabilistic-LRU becomes LRU. When *prob* is 0, Probabilistic-LRU becomes FIFO [2].

By randomly skipping promotions, Probabilistic-LRU is very effective at reducing the number of promotions. Figure 2a shows that the number of promotions decreases proportionally with the decrease of *prob*. For example, a *prob* of 0.1 can reduce the number of promotions by 90%.

Although Probabilistic-LRU can significantly reduce the number of promotions, we observe that the throughput does not increase significantly when the probability is larger than 0.4 (Figure 2b). Because Probabilistic-LRU skips promotions randomly without distinguishing popular and unpopular objects, popular objects still have many promotions when using Probabilistic-LRU technique. Moreover, the promotions of the same popular object are often performed on different CPU cores, which incurs a non-trivial number of CPU cache invalidation and coherence traffic. We conjecture that this overhead from CPU cache management limits the throughput of Probabilistic-LRU. As a result, only when *prob* is very small, e.g., 0.05 or 0.01, can we observe a significant throughput increase.

When *prob* is very small, many medium-popularity objects cannot be promoted in a timely manner, which leads to a large increase in miss ratio (fig. 2c). At a *prob* of 0.05, the miss ratio on 6357 traces increases by 6% on average. Overall, Probabilistic-LRU cannot maintain a miss ratio similar to LRU. Even a large *prob* would increase the cache miss ratio non-trivially. For example, at *prob* of 0.5, the cache miss ratio increases by 2% on average.

Finding. Although Probabilistic-LRU is very effective in reducing the number of promotions, the throughput only increases when the *prob* is very small. However, at a very small *prob*, the miss ratio increases significantly.

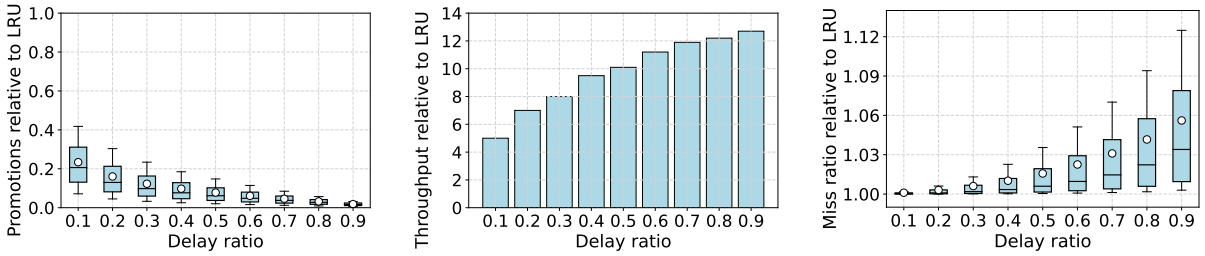
3.2 Batch-LRU

```
batch_time = batch_ratio * cache_size
batch.insert(obj)
if time - last_prom_time >= batch_time:
    for item in batch:
        if item in cache:
            promote(item)
    batch.clear()
```

Listing 2: promotion during cache hit in Batch-LRU

Because the locking operation at each promotion is computationally expensive [20] and leads to contention, Batch-LRU performs

[2] Although we describe *prob* as a constant, Probabilistic-LRU implementations in production systems are often more complex, and *prob* may change based on contention [34] as described in Section 2.2.



(a) Relative number of promotions **(b) Throughput Imprv (16 threads)** **(c) Relative miss ratio**

Figure 4: Delay-LRU reduces cache promotions by skipping recently-promoted objects. When the *delay ratio* is 20% of the cache size, Delay-LRU increases LRU's miss ratio by no more than 0.1% while reducing the number of promotions by over 82% and increasing throughput by 7×.

promotions in batches to reduce the overhead and contention. Instead of immediately moving the requested object to the head of the linked list, Batch-LRU tracks the requested object IDs in a buffer and periodically promotes them. For 8-byte object id, a batch of 128 consumes only 1 KB, resulting in negligible memory overhead.

The efficiency and performance of Batch-LRU depend on one parameter: *batch ratio*, which decides how often Batch-LRU performs a batched promotion. For the sake of consistency in this paper, we measure the time between promotions using insertion time, which increments by one upon each insertion into the cache. Therefore, a *batch ratio* of 0.1 means that there are 0.1×*cache size* insertions between each batch promotion. The details can be found in Listing 2.

Batching can effectively reduce the number of promotions because multiple promotions of a popular object are combined into a single promotion; however, the reduction depends heavily on the workload. Figure 3a shows that the variance in promotion reduction is highly variable. Recall that each box shows the 25th and 75th percentiles of the 6357 traces. The long box indicates that some traces enjoy a huge promotion reduction, while others do not observe much benefit. For example, at a small *batch ratio* of 0.1, the number of promotions on the mean and median trace can drop to 60% of that in LRU, while the worst trace does not benefit from batched promotion at all. We find that *the effectiveness of Batch-LRU depends on the skewness of the workload: a more skewed workload tends to exhibit larger benefits* because more requests are for a few popular objects. Therefore, more requests in a batch are coalesced into a single request.

Although Batch-LRU shows a large variance in promotion reduction, it is effective in improving throughput when the number of promotions can be reduced. Figure 3b shows that even at a small *batch ratio* of 0.1, the throughput increases by more than 4× at 16 threads. Further increasing the *batch ratio* leads to continued throughput increase. Compared to Probabilistic-LRU, Batch-LRU provides better scalability because Batch-LRU reduces more promotions from popular objects while Probabilistic-LRU reduces promotion for popular and unpopular objects with equal probability.

Besides being an effective scalability improvement technique, Batch-LRU is also more effective in maintaining a low miss ratio. Figure 3c shows that when the *batch ratio* is small, e.g., no more than 0.5, the miss ratio increases for a median trace by less than 1%. The reason is that Batch-LRU prioritizes reducing the promotion of popular objects, which are less likely to cause an increase in the miss ratio. When *batch ratio* is very large, some objects in the batch may have been evicted before they are promoted, leading to an increased miss ratio.

Finding. Batch-LRU is an effective Lazy Promotion technique as it can significantly reduce the number of promotions, leading to better scalability. Meanwhile, Batch-LRU can maintain a miss ratio similar to LRU. However, the effectiveness of Batch-LRU is highly variable and heavily depends on the workload.

3.3 Delay-LRU

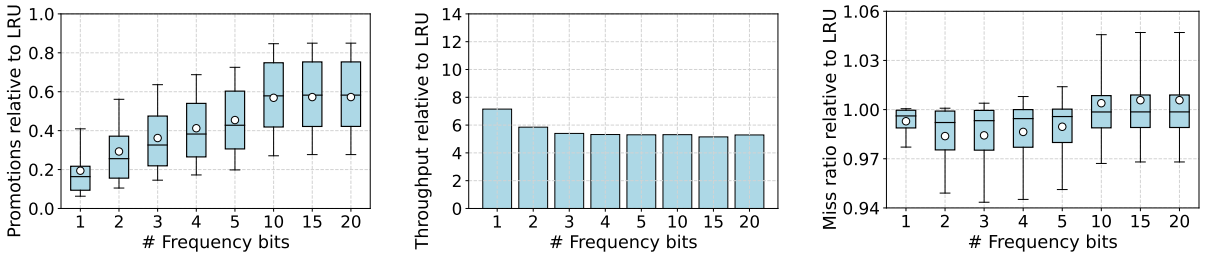
```
delay_time = delay_ratio * cache_size
if time - obj.last_prom_time > delay_time:
    promote(obj)
    obj.last_prom_time = current_time
```

Listing 3: promotion during cache hit in Delay-LRU

Upon a cache hit, Delay-LRU checks and promotes only if sufficient time has elapsed since the object's last promotion. Although this approach requires storing a timestamp for each object, a 4-byte timestamp adds negligible 0.1% overhead to typical 4 KB objects. Similar to Section 3.2, the delay time is measured using the number of insertions. *delay ratio* is the ratio between delay time and cache size, ranging from 0 to 1. A *delay ratio* of 1 would cause every object to be evicted before promotion. Upon a cache hit, we compare the last promotion time of the object and the current time to decide whether the object should be promoted.

Compared to Probabilistic-LRU and Batch-LRU, Delay-LRU is very effective at reducing the number of promotions. Figure 4a shows that when *delay ratio* is 10% of cache size, the number of promotions is reduced to 24% of that in LRU on average without visibly increasing miss ratio. This indicates that around 76% of promotions occur soon after the previous request to the same object, and Delay-LRU can avoid these unnecessary promotions.

The significant reduction in the number of promotions allows Delay-LRU to achieve notable scalability gains. At the 0.1 *delay ratio*, Delay-LRU exhibits a 5× higher throughput than LRU at 16 threads (Figure 4b). As the *delay ratio* increases, the speedup further increases and can achieve more than 12× throughput speedup at a 0.9 *delay ratio*. However, such a scalability improvement comes with a cost on efficiency—the long *delay ratio* increases the miss ratio



(a) Relative number of promotions **(b) Throughput Imprv (16 threads)** **(c) Relative miss ratio**

Figure 5: FIFO-reinsertion reduces both promotions and cache misses. The reduced promotions translate to a higher throughput. However, the scalability upper bound is lower than other Lazy Promotion techniques such as Delay-LRU.

by 6% on average on 6357 traces as shown in Figure 4c. Although a large *delay ratio* leads to a high miss ratio, when the *delay ratio* is small, the increase in the miss ratio is negligible. For example, at a *delay ratio* of 0.1, adding delayed promotion to LRU does not increase the miss ratio for 39.5% of the traces. This suggests that although promotions are helpful for LRU in achieving a low miss ratio, frequently promoting popular objects does not bring much benefit, and we can skip these promotions without hurting the miss ratio.

Both Batch-LRU and Delay-LRU prioritize reducing the promotions of popular objects; however, Delay-LRU is more effective and offers consistent benefits compared to Batch-LRU. For example, at a *delay ratio* of 0.1, Delay-LRU achieves almost the same miss ratio as LRU with a 5× higher throughput. To achieve a similar throughput, Batch-LRU needs to use a *batch ratio* of 0.2, which has a miss ratio 0.2% higher than LRU on average. Moreover, if we prefer a higher throughput, Delay-LRU can provide more than 12× LRU's throughput, while Batch-LRU can only achieve at most 8.2× LRU's throughput [3]. The reason behind Batch-LRU's lower throughput is that Batch-LRU needs to perform more work in the critical section.

Finding. Delay-LRU can effectively improve LRU's scalability with a minimal impact on efficiency. Its effectiveness remains consistent across diverse workloads.

3.4 FIFO-reinsertion

```
obj_to_evict = queue.front()
while obj_to_evict.freq > 0:
    obj_to_evict.freq -=1
    next_obj = obj_to_evict.next
    promote(obj_to_evict)
    obj_to_evict = next_obj
evict(obj_to_evict)
```

Listing 4: FIFO-reinsertion performs promotion (reinsertion) during cache eviction.

FIFO-reinsertion, as the name suggests, uses a FIFO queue to order objects, and there is no change to the ordering during cache hits. During cache evictions, popular objects at the tail are reinserted into the cache. Unlike other Lazy Promotion techniques that reduce the number of promotions at each cache hit, FIFO-reinsertion delays the "promotion" to eviction time. Therefore, we count the number of reinsertions as promotions.

Each object in FIFO-reinsertion is associated with a counter that tracks the object's access frequency. The counter is capped based on the number of bits used. For example, a frequency bits of 1 caps the maximum frequency at 1, effectively acting as a boolean variable. A frequency bits of 3 caps the frequency at 7: objects accessed more than 7 times in the cache are treated as 7. Upon a cache hit, the counter increments by 1. During evictions, objects with a counter larger than 0 are reinserted with the counter decremented by 1, and objects with a frequency counter of 0 are evicted.

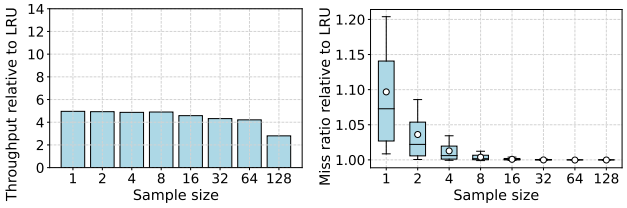
Figure 5a shows that the number of promotions (reinsertions) can significantly reduce in FIFO-reinsertion, especially when the number of frequency bits is small. When the number of frequency bits increases, the number of promotions increases until it reaches a plateau. This occurs because, after a certain threshold, increasing the frequency cap further does not affect the reinserted objects, and it is always the same set of objects that are being reinserted.

Similar to Delay-LRU and Batch-LRU, most of the promotion reduction comes from popular objects. As a result, Figure 5b shows that FIFO-reinsertion also enjoys a significant boost in throughput. However, the highest throughput FIFO-reinsertion can achieve is lower than Delay-LRU. Because Delay-LRU can reduce the number of promotions by more than 95%, it can increase throughput by more than 12×. In contrast, FIFO-reinsertion can only reduce the number of promotions by 80% at frequency bits 1.

Although FIFO-reinsertion is slightly worse than Delay-LRU in terms of scalability, it is more efficient. *It is the only Lazy Promotion technique that reduces LRU's miss ratio.* Figure 5c shows that with a 2-bit frequency counter, FIFO-reinsertion reduces LRU's miss ratio by almost 2% on average. Increasing the number of bits in the frequency counter initially reduces the miss ratio but eventually causes it to rise. This occurs because a large frequency cap causes some short-lived popular objects to become stuck in the cache. In such workloads, hot objects frequently change, but their high frequencies prevent them from being evicted quickly.

Finding. FIFO-reinsertion is most effective when we use 1-bit or 2-bit counters. It not only improves scalability, but also improves cache efficiency.

[3]Note that because *delay ratio* and *batch ratio* are different, we cannot compare the throughput and miss ratio of the same *delay ratio* and *batch ratio*. We should only compare them under the same miss ratio or throughput.



(a) Throughput improvement (b) Relative miss ratio

Figure 6: Random-LRU samples objects and evicts the least-recently-used one during eviction. Removing the global linked list allows it to be more scalable than LRU. However, it underperforms LRU in efficiency.

3.5 Random-LRU

Random-LRU samples some objects from the cache during eviction and evicts the least-recently used sample. It is not a Lazy Promotion promotion technique; however, it can also improve scalability. Therefore, we add it to the discussion.

Because Random-LRU does not maintain a global linked list and there is no locking during cache hits, it is more scalable than LRU. Figure 6a shows that Random-LRU can achieve a throughput of 5× higher than LRU. However, the improvement is smaller compared to Delay-LRU due to the overhead of random sampling and locking during eviction. This overhead increases with the number of samples. We can see that the throughput reduces to 3× that of LRU when Random-LRU samples 128 objects at each eviction.

While Random-LRU improves scalability, Figure 6b shows that it is lackluster in efficiency. Using just one sample, it becomes the random eviction algorithm without using recency information. We observe that it increases the miss ratio by 10% on average on the 6357 traces. Increasing the number of samples reduces the miss ratio. With 16 samples per eviction, we find that Random-LRU achieves a similar miss ratio to LRU. Conventional wisdom suggests that Random-LRU can achieve a miss ratio lower than LRU for workloads that exhibit loop access patterns because such a pattern causes thrashing for LRU when the loop size is larger than the cache size. Surprisingly, this observation does not appear in our evaluations. Although over 70% of the 6357 traces are block cache workloads, which tend to have loop and scan access patterns, we find LRU to be better than Random-LRU on less than 12% of the traces under different sample sizes.

Finding. Random-LRU can improve scalability without compromising the miss ratio but requires careful choice of parameters. Using 16 samples per eviction provides a good balance. Increasing the sample size reduces throughput, while reducing the sample size increases the miss ratio.

3.6 Promotion efficiency

We have shown that different Lazy Promotion techniques perform differently. Probabilistic-LRU can significantly reduce the number of promotions with the cost of increasing miss ratio; Batch-LRU heavily depends on the workload—very effective in reducing the number of hits for some workloads without increasing miss ratio; Delay-LRU performs the best in reducing the number of promotions while maintaining miss ratio; FIFO-reinsertion, compared to other Lazy Promotion techniques, not only reduces the number of promotions but also reduces miss ratio.

To further characterize the effectiveness of promotion, we propose a new metric called "promotion efficiency", which calculates the misses that each promotion reduces from FIFO on average. If a technique or algorithm can significantly reduce FIFO's miss ratio with very few promotions, then it has a high promotion efficiency. Since FIFO has no promotion, it is not included in the figure. LRU promotes upon each cache hit, most of which are unnecessary, resulting in an average promotion efficiency of 0.037.

Probabilistic-LRU has a promotion efficiency similar to LRU when *prob* is high (Figure 7a). However, when Probabilistic-LRU drops to 0.01, we observe an increase in promotion efficiency. This happens because when the probability of promotion is very low, only very popular objects may be promoted. Unlike FIFO, which does not keep popular objects in the cache, these occasional promotions of very popular objects help keep them in the cache, driving up promotion efficiency.

Figure 7b shows the promotion efficiency of Batch-LRU. We find that across different *batch ratios*, promotion efficiency is consistently low and not sensitive to *batch ratio*. This occurs because Batch-LRU's effectiveness depends on the workload and cannot significantly reduce the number of promotions for many traces.

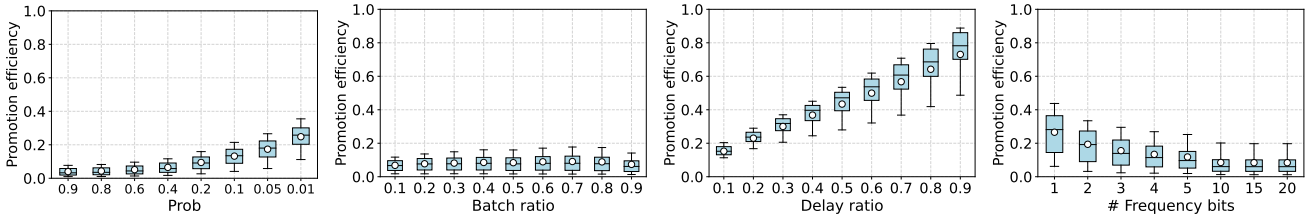
Compared to Probabilistic-LRU and Batch-LRU, Figure 7c shows that Delay-LRU can achieve remarkably high promotion efficiency. At a *delay ratio* of 0.4, each promotion in Delay-LRU can reduce 0.4 miss from FIFO. Further increasing the *delay ratio* to 0.9, Delay-LRU can achieve the highest mean promotion efficiency among all Lazy Promotion techniques at 0.72. Because Delay-LRU reduces the promotion of popular objects, which is very effective at identifying and reducing the most useless promotions.

Figure 7d shows that FIFO-reinsertion has the highest promotion efficiency when using a 1-bit counter. Using higher frequency caps only reduces the promotion efficiency. This is because using more bits to record frequency often leads to unnecessary promotions.

Among all the Lazy Promotion techniques, we find that Delay-LRU has the most efficient promotions. FIFO-reinsertion closely follows Delay-LRU when using a 1-bit counter. Batch-LRU and Probabilistic-LRU have the lowest promotion efficiency, and many promotions do not reduce cache misses.

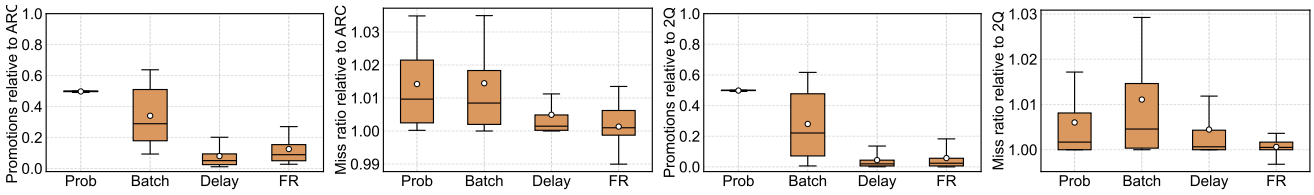
3.7 Lazy Promotion on Advanced Eviction Algorithms

Most advanced cache eviction algorithms designed in the past two decades are built on top of one or more LRU queues using different promotion metrics. For example, ARC[37], SLRU[27], 2Q[30], MQ[76], and multi-generational LRU[44] leverage multiple LRU queues to differentiate between frequently and infrequently accessed items. Others, such as LIRS[28] and LIRS2[73], retain an LRU queue but apply different metrics, e.g., stack distance, to determine how to promote an object. Because these advanced algorithms are built on top of LRU, they also suffer from limited scalability. The Lazy Promotion techniques studied in this paper can also be used to improve their scalability. We illustrate this with 2Q [30] and ARC [37].



(a) Probabilistic-LRU (b) Batch-LRU (c) Delay-LRU (d) FIFO-reinsertion

Figure 7: Promotion efficiency calculated as the number of misses reduced from FIFO over the number of promotions. Higher is better.



(a) ARC (b) ARC (c) 2Q (d) 2Q

Figure 8: Using Lazy Promotion in advanced LRU-based eviction algorithms. Similar to LRU, delay and reinsertion are more effective than batch and probabilistic promotion.

Both 2Q and ARC comprise LRU queues that transition objects from "recent" to "frequent" states to improve miss ratio (A_1 to A_m in 2Q; T_1 to T_2 in ARC). We refrain from interfering with the policy decision to enter the frequent queue. We add Lazy Promotion to the LRU queue for frequent objects in each algorithm, i.e., within A_m (2Q) and T_2 (ARC), to regulate in-queue promotions.

We have examined the impact of parameters on each Lazy Promotion technique in this section. We will use 0.5 for the rest of the sections so that we only have one box for each technique. Figure 8 shows that ARC and 2Q exhibit patterns similar to those observed on LRU: delay and reinsertion are more effective than batch and probabilistic promotion. Delay can significantly reduce promotions without significantly increasing the miss ratio, while reinsertion can reduce both the miss ratio and the number of promotions. Moreover, the effectiveness of batching promotions is workload-dependent—some workloads benefit more, while others benefit less. These results indicate that LP's benefits are not artifacts of a single policy, but rather arise from suppressing low-value promotions that are common in LRU-style designs.

4 CAN LAZY PROMOTIONS BE LAZIER WITH FUTURE INFORMATION?

Lazy Promotion techniques have shown that reducing the number of promotions in caching systems can significantly enhance scalability without sacrificing the miss ratio. In this section, we investigate whether ranking, the goal of promotion, is fundamentally necessary and whether Lazy Promotion can be even lazier if we have oracle information. More specifically, we use oracle future knowledge as an analytical tool to establish an upper bound on how few promotions are fundamentally needed. We first use the future access time obtained from offline analysis to perform early eviction and find that ranking is mostly unnecessary if we can predict whether an object will be requested before it is evicted (Section 4.1). Building on this insight, we find that using future information in FIFO-reinsertion not only reduces the number of promotions to 6%, but also reduces the miss ratio by more than 1% (Section 4.2), compared to LRU.

4.1 Object ranking is unnecessary

In this subsection, we evaluate whether ranking objects in the cache is necessary to obtain a low miss ratio.

Belady early eviction. Traditional Belady evicts the object with the longest future reuse distance whenever space is needed. We extend this idea by allowing eviction at *insertion and request time*: if the next use of an object is too far in the future, it can be discarded immediately. We refer to this as *Belady early eviction* (BEE). Because BEE only requires a binary decision from the oracle—whether this object will be used before being evicted—it is essentially easier to predict than the exact reuse time.

Oracle definition. Let $R(o)$ denote the reuse distance (number of requests until the next access) of object o, and let S denote the average eviction age measured in the number of requests. Under Oracle knowledge, early eviction is:

$$\text{evict } o \text{ at access time if } R(o) > S.$$

Approximate oracle form. In practice, the average eviction age S is not directly observable and changes with decisions on other objects. We approximate it by

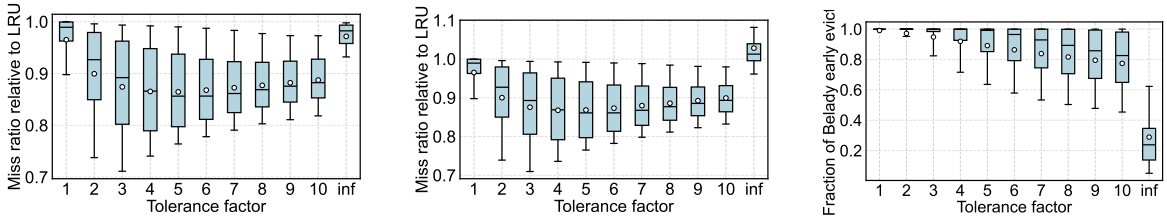
$$\widehat{S} \approx \frac{\text{cache size}}{\text{miss ratio}},$$

To tolerate estimation error, we introduce a *tolerance factor* $\alpha \geq 1$:

$$\text{evict } o \text{ if } R(o) > \alpha \cdot \widehat{S}.$$

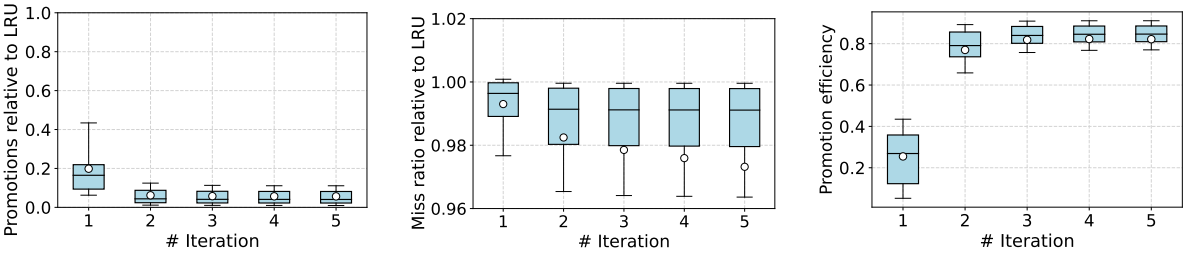
Here, $\alpha = 1$ applies the strict estimator, while $\alpha > 1$ retains objects more conservatively. When Belady early eviction cannot remove enough objects, we use Random-LRU and Random for eviction, denoted as Belady-RandomLRU and Belady-Random, respectively.

Ranking objects for eviction is not important. Figure 9a and Figure 9b show that BEE significantly reduces miss ratio compared



(a) Miss ratio of Belady-RandomLRU (b) Miss ratio of Belady-Random (c) Fraction of Belady early evictions

Figure 9: Using future access time to early evict objects (denoted as Belady early eviction) allows a cache to avoid ranking (promoting) objects. a) and b) when Belady early eviction does not evict enough objects, Random-LRU and Random are used for eviction, respectively. We find that Random-LRU and Random do not show much difference in miss ratio, suggesting that ranking objects in the cache for eviction is unnecessary. c) The reason why Belady-RandomLRU and Belady-Random show similar miss ratios is that Belady early eviction accounts for most of the cache evictions.



(a) Relative number of promotions (b) Relative miss ratio (c) Misses reduced per promotion

Figure 10: Offline FIFO-reinsertion: using future information to filter out unnecessary promotions reduces both promotions and miss ratio.

to LRU, achieving a 14% average reduction at tolerance factor 5. Moreover, Belady-RandomLRU and Belady-Random exhibit similar miss ratios across tolerance factors (except at ∞), indicating that the choice of the fallback eviction algorithm is largely unimportant in the presence of early-eviction signals. In other words, ranking cached objects for eviction—i.e., promotion—is not important. In particular, Belady-Random and Belady-RandomLRU do not rely on promotions; they primarily use future reuse distances to drive most evictions (Figure 9c) and, by proactively removing cold objects, achieve lower miss ratios than LRU without ranking.

If we employ a binary classifier to predict whether an object will be reused before being evicted. The model can be used to trigger early eviction. This differs from existing learned eviction algorithms, e.g., LRB [54], which predict each object's reuse distance using regression as an objective [4].

This observation also explains why the new eviction algorithm, such as S3-FIFO [67] and SIEVE [72], achieves state-of-the-art efficiency. They evict new objects quickly (called quick demotion) without ranking, but can still achieve a low miss ratio.

4.2 Lazier upon cache misses

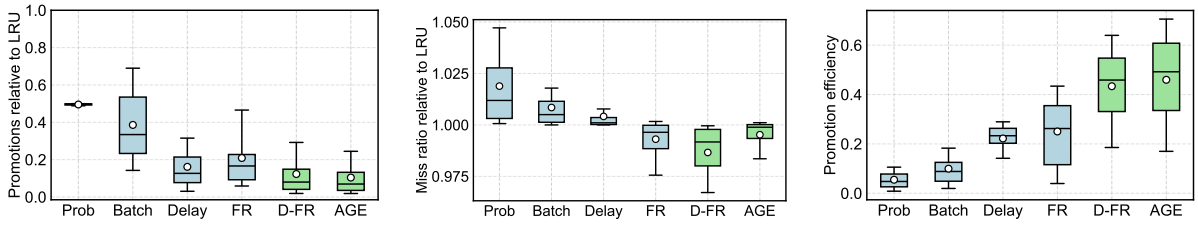
We have demonstrated that most of the promotions in the cache are unnecessary if we have future information. However, the previous section assumes that the cache can evict objects at any time and requires a binary prediction to be made at each request, which is computationally expensive. In this section, we study how future information can help FIFO-reinsertion, which limits promotion and thus binary prediction to eviction time. Specifically, we design offline FIFO-reinsertion by not promoting an object if its next access time is too far in the future. For any request sequence and cache capacity C, policies that promote objects on hits offer no additional benefit once promotion at eviction is allowed. A policy that only promotes at eviction time is guaranteed to be as good as any policy that promotes at object access time. Moreover, promotion decisions at eviction time can leverage more information.

Offline FIFO-reinsertion design. Building on FR, we design Offline FIFO-reinsertion, which runs multiple iterations through a trace. The zeroth iteration is the same as FIFO-reinsertion, however, we mark the promotions (reinsertions) that do not lead to a cache hit. In subsequent iterations, the marked promotions are not performed. Therefore, some promotions are reduced in each iteration.

Offline FR reduces both promotions and miss ratios. Figure 10a shows that offline FIFO-reinsertion can reduce promotions from LRU by more than 90%, demonstrating the huge potential. Reducing the number of promotions in offline FIFO-reinsertion does not bring the side effect of increasing miss ratio. We observe that offline FIFO-reinsertion achieves a lower miss ratio compared to FIFO-reinsertion (Figure 10b), with consistent reductions in both median and mean. As the number of iterations increases, the number of promotions and the miss ratio start to converge.

Figure 10c shows promotion efficiency of offline FIFO-reinsertion. We find that filtering out unnecessary promotions significantly improves promotion efficiency, achieving over 0.8—each promotion reduces 0.8 cache misses. This suggests that promotion at eviction is sufficient to achieve both a low number of promotions and a low miss ratio, and the future Lazy Promotion techniques can be built on top of a FIFO queue using FIFO-reinsertion.

[4]Binary classification is simpler than multi-class classification, which is often simpler than regression.



(a) Relative number of promotions (b) Relative miss ratio (c) Relative promotion efficiency

Figure 11: D-FR and AGE significantly reduce the number of promotions, without compromising the miss ratio. We use the best parameter from each Lazy Promotion technique that balances between miss ratio and promotions.

5 PRACTICAL LAZIER PROMOTION

The previous section shows that promotion and ranking objects are not important: if we have future information, we can further reduce the number of promotions without increasing the miss ratio. This section demonstrates how to achieve this goal without relying on future information.

In Section 5.1, we introduce Delayed FIFO-reinsertion (D-FR), a practical enhancement to FIFO-reinsertion inspired by Delay-LRU. D-FR reduces both promotions and miss ratios compared to FIFO-reinsertion, however, the promotion reduction is limited. In Section 5.2, we introduce Age-Guided Eviction (AGE). AGE uses recency to predict whether a promotion will be useful during eviction and discards unnecessary promotions. Compared to D-FR, AGE can reduce more promotions; however, it increases the miss ratio for some traces compared to FIFO-reinsertion.

5.1 Delayed FIFO-reinsertion (D-FR)

```
delay_time = delay_ratio * cache_size
# On access
t = current_time - obj.access_time
if t > delay_time:
    obj.freq += 1
    obj.access_time = current_time
```

Listing 5: Delayed FIFO-reinsertion

Section 3.3 shows that Delay-LRU consistently achieves the best promotion efficiency while keeping the miss ratio close to that of LRU, indicating that many promotions occur too soon after a previous one and bring little value. This suggests two principles: (1) do not reward multiple hits in a short window, and (2) if we must reward, do it at the *eviction time*, where the decision is most informed. Building on these, we design *D-FR*, which does not increment the frequency counter upon clustered hits. More specifically, D-FR tracks the last access and insertion time. If the current hit falls within a delay-time threshold, the frequency counter will not increment. Similar to Delay-LRU, D-FR adds a 4-byte timestamp. The computational overhead from conditional checks (metadata read from CPU cache) is outweighed by the reduction in promotions (metadata write), yielding a 5% throughput increase over FIFO-reinsertion in our evaluations.

Figure 11 shows that compared to FIFO-reinsertion, D-FR further reduces the number of promotions by 60% for a median trace. Meanwhile, it also reduces the miss ratio similar to offline FIFO-reinsertion. As a result, D-FR achieves a higher promotion efficiency compared to existing Lazy Promotion techniques.

The results in Figure 11 use a default frequency bit of 1 and a delay ratio of 5%. We study the sensitivity of delay time in Figure 12. We find that D-FR is not sensitive to the delay ratio, and different ratios show similar results. Choosing a larger delay ratio allows for a greater promotion reduction, albeit at the cost of slightly higher miss ratios. However, unlike Delay-LRU, which increases the miss ratio over LRU, D-FR can consistently achieve a miss ratio lower than LRU.

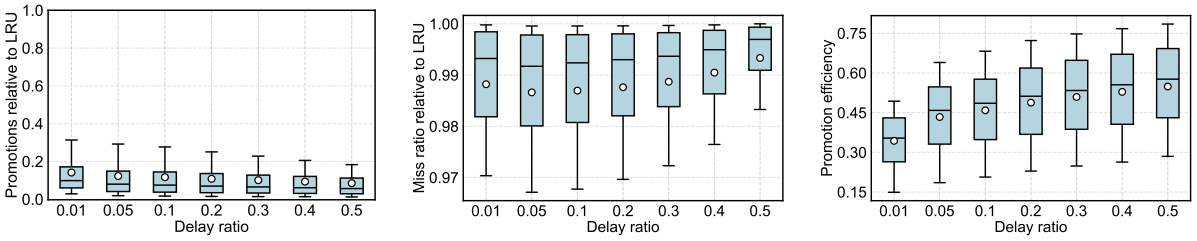
5.2 Age guided eviction (AGE)

```
# Eviction
threshold = cache_size / miss_ratio * factor
obj_to_evict = queue.front()
while obj_to_evict.freq > 0:
    obj_to_evict.freq -= 1
    t = obj_to_evict.last_access_time
    age = current_time - t
    if (age >= threshold):
        break # evict the object
    else:
        # check the next object
    ...
```

Listing 6: Age guided eviction (AGE)

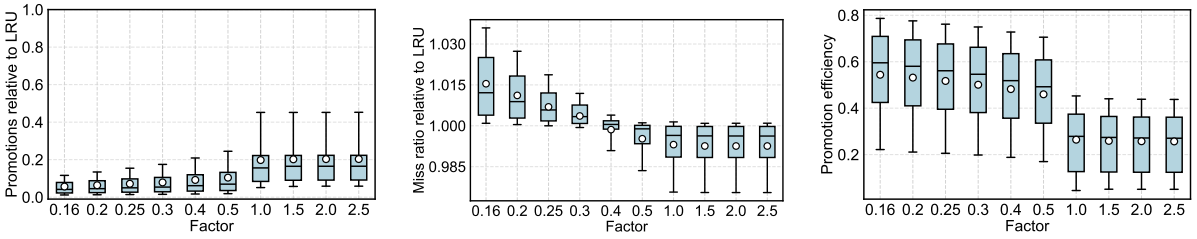
FIFO-reinsertion makes promotion decisions based on frequency without considering recency information. To further reduce unnecessary promotions in FIFO-reinsertion, we design AGE, which leverages object age as a filter. If the time since the last access to an object has been very long, the object may be unpopular, and the likelihood of it receiving a request will be low. Therefore, AGE does not promote objects whose age exceeds a threshold. It calculates the threshold similar to BEE (section 4.1) with a tolerance factor (listing 6).

Figure 11 shows that AGE has fewer promotions than D-FR with increased miss ratio. However, unlike D-FR, which reduces miss ratio over FIFO-reinsertion, AGE achieves a similar or slightly higher miss ratio compared to FIFO-reinsertion. As a result, it achieves a similar promotion efficiency to D-FR, around 48%, and both are significantly higher than FIFO-reinsertion, at approximately 24%.



(a) Relative number of promotions **(b) Relative miss ratio** **(c) Relative promotion efficiency**

Figure 12: D-FR is not sensitive to delay ratio. As delay ratio increases, more promotions are reduced at the cost of slightly higher miss ratios.



(a) Relative number of promotions **(b) Relative miss ratio** **(c) Relative promotion efficiency**

Figure 13: Age is not sensitive to the factor parameter.

We show AGE with a tolerance factor of 0.5 in Figure 11. To evaluate the sensitivity, we evaluate a wide range of factors in Figure 13. We find that it is straightforward to choose this parameter. A smaller factor reduces promotions more aggressively, while leading to a higher miss ratio; a larger factor has a smaller impact on both promotions and miss ratios. Increasing the factor over a certain threshold results in zero promotions being filtered out, yielding the same algorithm as FIFO-reinsertion.

6 RELATED WORK

We have covered the most relevant works in Section 2. This section provides additional works related to caching and workload measurements.

Cache efficiency and scalability. The concept of Lazy Promotion is first introduced in a HotOS work [63], where the authors show that eviction algorithms should use lazy promotion to improve throughput and scalability, and quick demotion to improve efficiency. However, it does not delve into different lazy promotion techniques. Many other works also improve a cache's efficiency by designing a better eviction algorithm, such as ARC [37], LIRS [28, 35, 73], TinyLFU [23, 68], 2Q [30], LRFU [22], GDSF [14], LeCaR [57], CACHEUS [50], LHD [9], LRB [54], HALP [55], GL-Cache [62], SIEVE [72]. Most of these algorithms typically have lower throughput compared to LRU due to computational overhead. There are also many works that improve the throughput and scalability of caches, such as MemC3 [24], Segcache [66], and BP-wrapper [20]. Unlike traditional system designs, this work focuses on understanding different techniques that have already been deployed in production.

Workloads and performance measurement. This work uses 6357 traces to characterize different Lazy Promotion techniques. Many previous works have characterized production systems or studied the workloads of production systems. Juncheng performed a detailed study of over 100 Memcached cache clusters at Twitter [64]; Siying conducted a comprehensive study of the RocksDB deployments at Meta [21]; Nishtala and Atikoglu studied the Memcached deployment challenges and workloads at Meta [8, 42]. Additionally, Alibaba and Tencent have conducted detailed workload studies on their elastic block storage cloud offerings [36, 70]. Moreover, while many studies on eviction-algorithm design evaluate multiple algorithms [12, 26, 32, 54, 72], they rarely examine in depth those not proposed in the paper. Ziyue shows that aggressively increasing hit ratio can hurt cache throughput due to contention on promotion, but our work demonstrates that suppressing low-value promotions can simultaneously preserve miss ratio and significantly improve scalability [48]. To the best of our knowledge, this work is the first to examine the effectiveness of the widely deployed lazy promotion techniques.

7 CONCLUSION

Although Lazy Promotion techniques are widely used in production systems, there has been very limited understanding about their effectiveness. This paper presents a comprehensive evaluation of Lazy Promotion techniques in cache evictions. We find that Delay-LRU and FIFO-reinsertion are effective at reducing promotions while maintaining miss ratios. However, promotion reductions in Probabilistic-LRU come at a cost of increased miss ratios, and Batch-LRU is heavily workload-dependent. In addition, we identify that most cache promotions are unnecessary, exposing considerable opportunities for optimization. We introduce two new simple methods, D-FR and AGE, and demonstrate that they can effectively reduce the number of promotions without compromising miss ratios.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and suggestions.

REFERENCES

[1] 2025. Alibaba block-trace. https://github.com/alibaba/block-traces. Accessed: 2025-09-30.

[2] 2025. Page frame reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html. Accessed: 2025-09-30.

[3] 2025. Redis. http://redis.io/. Accessed: 2025-09-30.

[4] 2025. Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval. Accessed: 2025-09-30.

[5] 1a1a11a. 2024. libCacheSim: A Lightweight Library for Cache Simulation. https://github.com/1a1a11a/libCacheSim. Accessed: 2024-11-25.

[6] Advanced Micro Devices. 2022. AMD EPYC 9004 Series Processors: Technical Overview. https://www.amd.com/en/products/cpu/epyc-9004-series. Accessed: 2025-09-30.

[7] Ampere Computing. 2020. Ampere Altra Max Processor: Expanding to 128 Cores for Cloud Native Workloads. https://amperecomputing.com/press/ampere-altra-family-of-cloud-native-processors-expands-to-128-cores-with-altra-max. Accessed: 2025-09-30.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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)*. Association for Computing Machinery, New York, NY, USA, 53–64. doi:10.1145/2254756.2254766

[9] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX symposium on networked systems design and implementation (NSDI'18)*. 389–403.

[10] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. doi:10.1147/sj.52.0078

[11] 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. 2020. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation (OSDI'20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[12] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference (ATC'17)*. USENIX Association, Santa Clara, CA, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

[13] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*. 239–252. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

[14] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Citeseer.

[15] George Chrysos. 2012. Intel® Xeon Phi coprocessor (codename Knights Corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*. 1–31. doi:10.1109/HOTCHIPS.2012.7476487

[16] Sergio De Agostino. 2006. Bounded size dictionary compression: Relaxing the LRU deletion heuristic. *International Journal of Foundations of Computer Science* 17, 06 (2006), 1273–1280.

[17] Peter J. Denning. 2021. Working Set Analytics. *Comput. Surveys* 53, 6 (Nov. 2021), 1–36. doi:10.1145/3399709

[18] Meta developers. 2025. Cachelib - pluggable caching engine to build and scale high performance cache services. https://cachelib.org/. Accessed: 2023-04-06.

[19] Inc. Dgraph Labs. 2024. Ristretto: A High-Performance Memory-Bounded Cache. https://github.com/dgraph-io/ristretto. Accessed: 2024-11-28.

[20] Xiaoning Ding, Song Jiang, and Xiaodong Zhang. 2009. BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free. In *2009 IEEE 25th International Conference on Data Engineering (ICDE'07)*. 369–380. doi:10.1109/ICDE.2009.96 ISSN: 2375-026X.

[21] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[22] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12 (Dec. 2001), 1352–1361. doi:10.1109/TC.2001.970573

[23] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage* 13, 4 (Dec. 2017), 1–31. doi:10.1145/3149371

[24] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX symposium on networked systems design and implementation (NSDI'13)*. 371–384.

[25] Alibaba Group. 2020. Alibaba Cloud Block Traces. https://github.com/alibaba/block-traces Accessed: 2025-09-30.

[26] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. 2022. 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)*. Association for Computing Machinery, New York, NY, USA, 72–90. doi:10.1145/3555050.3569134

[27] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*. Association for Computing Machinery, New York, NY, USA, 167–181. doi:10.1145/2517349.2522722

[28] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS'02, Vol. 30)*. 31–42. doi:10.1145/511399.511340

[29] Song Jiang and Xiaodong Zhang. 2005. Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. *IEEE Trans. Comput.* 54, 8 (2005), 939–952.

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

[31] Woon-Hak Kang, Gi-Tae Yun, Sang-Phil Lim, Dong-In Shin, Yang-Hun Park, Sang-Won Lee, and Bongki Moon. 2012. Innodb doublewrite buffer as read cache using ssds. In *10th USENIX Conference on File and Storage Technologies*.

[32] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. 2019. RL-Cache: Learning-Based Cache Admission for Content Delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML - NetAI'19 (NetAI'19)*. ACM Press, Beijing, China, 57–63. doi:10.1145/3341216.3342214

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

[34] kunga. 2024. Concurrent Shared Cache [#8447]. https://github.com/ydb-platform/ydb/issues/8447. Accessed: 2025-09-30.

[35] Cong Li. 2018. 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)*. Association for Computing Machinery, New York, NY, USA, 59–64. doi:10.1145/3211890.3211891

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

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

[38] Memcached. 2025. Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2025-02-06.

[39] Kianoosh Mokhtarian and Hans-Arno Jacobsen. 2014. Caching in video CDNs: Building strong lines of defense. In *Proceedings of the ninth European conference on computer systems*. 1–13.

[40] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage* 4, 3 (Nov. 2008), 1–23. doi:10.1145/1416944.1416949

[41] Nginx. 2025. Nginx. https://nginx.org/. Accessed: 2025-09-30.

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

[43] Oracle Corporation. 2025. MySQL 8.0 Reference Manual: InnoDB Buffer Pool. https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html. Accessed: 2025-09-30.

[44] Noritaka Osawa, Toshitsugu Yuba, and Katsuya Hakozaki. 1997. Generational replacement schemes for a www caching proxy server. In *High-Performance Computing and Networking: International Conference and Exhibition Vienna, Austria, April 28–30, 1997 Proceedings 5*. Springer, 940–949.

[45] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–165.

[46] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2019. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*. ACM, Santa Cruz CA USA, 193–205. doi:10.1145/3357223.3362729

[47] PostgreSQL Global Development Group. 2025. *PostgreSQL 17.2 Documentation*. PostgreSQL Global Development Group. https://www.postgresql.org/docs/current/ Accessed: 2025-11-20.

[48] Ziyue Qiu, Juncheng Yang, and Mor Harchol-Balter. 2024. Can Increasing the Hit Ratio Hurt Cache Throughput?. In *EAI International Conference on Performance Evaluation Methodologies and Tools*. Springer.

[49] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*. Association for Computing Machinery, New York, NY, USA, 557–573. doi:10.1145/3552326.3587446

[50] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 341–354. https://www.usenix.org/conference/fast21/presentation/rodriguez

[51] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 498–512.

[52] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*. ACM, Virtual Event USA, 93–105. doi:10.1145/3452296.3472934

[53] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. 2011. IBM POWER7 multicore server processor. *IBM Journal of Research and Development* 55, 3 (2011), 1:1–1:29. doi:10.1147/JRD.2011.2127330

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

[55] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. 2023. {HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1149–1163.

[56] Varnish Software. 2025. Varnish Cache. https://varnish-cache.org/. Accessed: 2025-09-30.

[57] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems (hotStorage'18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[58] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *13th USENIX conference on file and storage technologies (FAST'15)*. USENIX Association, Santa Clara, CA, 95–110. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[59] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data*. 37–44.

[60] Qiuping Wang, Jinhong Li, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in {Log-Structured} Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 429–444.

[61] Wikimedia. 2025. Analytics/Data Lake/Traffic/Caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2025-09-30.

[62] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST'23)*. 115–134. https://www.usenix.org/conference/fast23/presentation/yang-juncheng.

[63] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. 2023. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS'23)*. Association for Computing Machinery, New York, NY, USA, 70–79. doi:10.1145/3593856.3595887

[64] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. 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)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[65] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage* 17, 3 (Aug. 2021), 17:1–17:35. doi:10.1145/3468521

[66] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. 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)*. USENIX Association, 503–518. https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng

[67] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. Association for Computing Machinery, New York, NY, USA, 130–149. doi:10.1145/3600006.3613147

[68] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. doi:10.48550/arXiv.2401.02385 arXiv:2401.02385 [cs] version: 1.

[69] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the effect of data center resource disaggregation on production DBMSs. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[70] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2018. Tencent Block Storage Traces (SNIA IOTTA Trace Set 27917). In *SNIA IOTTA Trace Repository*, Geoff Kuenning (Ed.). Storage Networking Industry Association. http://iotta.snia.org/traces/parallel?only=27917

[71] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798. https://www.usenix.org/conference/atc20/presentation/zhang-yu

[72] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. 2024. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1229–1246. https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

[73] Chen Zhong, Xingsheng Zhao, and Song Jiang. 2021. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. ACM, Haifa Israel, 1–12. doi:10.1145/3456727.3463772

[74] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. 2016. Tencent Photo Cache Traces (SNIA IOTTA Trace Set 27476). In *SNIA IOTTA Trace Repository*, Geoff Kuenning (Ed.). Storage Networking Industry Association. http://iotta.snia.org/traces/parallel?only=27476

[75] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. 2018. Demystifying Cache Policies for Photo Stores at Scale: A Tencent Case Study. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/3205289.3205299

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