FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion

Juncheng Yang  
Carnegie Mellon University

Ziyue Qiu  
Carnegie Mellon University

Yazhuo Zhang  
Emory University

Yao Yue*  
Pelikan Foundation

K. V. Rashmi  
Carnegie Mellon University

Abstract

LRU has been the basis of cache eviction algorithms for decades, with a plethora of innovations on improving LRU’s miss ratio and throughput. While it is well-known that FIFO-based eviction algorithms provide significantly better throughput and scalability, they lag behind LRU on miss ratio, thus, cache efficiency.

We performed a large-scale simulation study using 5307 block and web cache workloads collected in the past two decades. We find that contrary to what common wisdom suggests, some FIFO-based algorithms, such as FIFO-Reinsertion (or CLOCK), are, in fact, more efficient (have a lower miss ratio) than LRU. Moreover, we find that quick demotion — evicting most new objects very quickly — is critical for cache efficiency. We show that when enhanced by quick demotion, not only can state-of-the-art algorithms be more efficient, a simple FIFO-based algorithm can outperform five complex state-of-the-art in terms of miss ratio.

ACM Reference Format:

1 Introduction

Caching is a well-known and widely deployed technique to speedup data accesses [9, 17, 23, 25, 32, 36, 39, 43, 44, 46, 53, 63, 66, 79, 80, 85], reduce repeated computation [40, 50, 64, 82] and data transfer [16, 18, 21, 41, 52, 57, 69–71, 81, 86, 89].

A core component of a cache is the eviction algorithm, which chooses the objects stored in the limited cache space. Two metrics describe the performance of an eviction algorithm: efficiency measured by the miss ratio and throughput measured by the number of requests served per second.

The study of cache eviction algorithms has a long history [14, 24, 26, 68], with a majority of the work centered around LRU (that is to evict the least-recently-used object). LRU maintains a doubly-linked list, promoting objects to the head of the list upon cache hits and evicting the object at the tail of the list when needed. Belady and others found that memory access patterns often exhibit temporal locality — “the most recently used pages were most likely to be reused in the immediate future”. Thus, LRU using recency to promote objects was found to be better than FIFO [14, 27].

Most eviction algorithms designed to achieve high efficiency start from LRU. For example, many algorithms such as ARC [56], SLRU [6, 48], 2Q [7, 30, 47, 48], MQ [92] and multi-generational LRU [5], use multiple LRU queues to separate hot and cold objects. Some algorithms, e.g., LIRS [45] and LIRS2 [90], maintain an LRU queue but use different metrics to promote objects. While other algorithms, e.g., LRFU [29], EE-LRU [67], LeCaR [72] and CACHEUS [65], augment LRU’s recency with different metrics. In addition, many recent works, e.g., Talus [13], improve LRU’s ability to handle scan and loop requests.

Besides efficiency, there have been fruitful studies on enhancing the cache’s throughput performance and thread scalability. Each cache hit in LRU promotes an object to the head of the queue, which requires updating at least six pointers guarded by locks. These overheads are not acceptable in many deployments that need high performance [10, 37, 62, 78]. Thus, performance-centric systems often use FIFO-based algorithms to avoid LRU’s overheads. For example, FIFO-Reinsertion and variants of CLOCK [24, 60, 68] have been developed, which serve as LRU approximations. It is often perceived that these algorithms trade miss ratio for better throughput and scalability [11, 38, 43, 47, 60].

Via a large-scale simulation study, we make a case for breaking away from LRU completely and instead designing eviction algorithms based on FIFO. To the best of our
knowledge, this is by far the most comprehensive eviction algorithm study. Compared to previous work [65], our datasets have 16× more traces with 58,100× more requests. And the datasets are more diverse, containing traces of block, key-value, and object caches collected over two decades.

FIFO provides many benefits compared to LRU, including fewer metadata, less computation, better scalability [37, 83] and flash friendliness [22, 55, 75, 84]. However, FIFO alone often leaves a large efficiency headroom. To bridge the gap, we introduce two broad classes of techniques — Lazy Promotion and Quick Demotion.

Lazy Promotion (LP) performs promotion only at the eviction time. An example of this technique is “reinsertion”, which puts the eviction candidate back into the cache if requested since the last insertion. Common wisdom suggests that FIFO with Lazy Promotion is an LRU approximation that is less efficient than LRU [11, 38, 43, 47, 60]. However, our large-scale empirical study on 5307 traces shows that such “weak LRU” are more efficient than LRU (§3).

Quick Demotion (QD) removes most objects quickly after they are inserted. We show that the opportunity cost of waiting for new objects to traverse through the queue(s) is too high. We demonstrate the importance of QD by adding a small probationary FIFO queue and a metadata-only ghost queue to five state-of-the-art eviction algorithms. Evaluations show that QD-enhanced ARC can reduce ARC’s miss ratio by up to 59.8%, and QD-enhanced LIRS can reduce LIRS’s miss ratio by up to 49.8%. On average, QD-enhanced algorithms reduce the miss ratio from the corresponding state-of-the-art algorithm by 2.7% on the 5307 traces. Note that the seemingly small improvement is huge due to the large number of traces. We further demonstrate a simple eviction algorithm QD-LP-FIFO by applying the aforementioned Lazy Promotion and Quick Demotion on top of FIFO. QD-LP-FIFO is simple yet efficient. Our evaluations show that QD-LP-FIFO achieves lower miss ratios than state-of-the-art eviction algorithms. For example, QD-LP-FIFO reduces the miss ratios of LIRS and LeCaR by 1.6% and 4.3% on average.

We believe that further innovations in better Lazy Promotion and Quick Demotion techniques will lead to a class of simple and efficient eviction algorithms. Moreover, we envision that future eviction algorithms can be designed like building a LEGO by adding different LP and QD techniques to a base algorithm such as FIFO.

This paper makes two main contributions:

- Contrary to the common belief that LRU approximations are less efficient, we show that FIFO with Lazy Promotion (e.g., FIFO-Reinsertion/CLOCK) achieves a lower miss ratio than LRU on a large collection of workloads.
- We demonstrate that Quick Demotion is critical for cache efficiency. A simple QD technique, e.g., a probationary FIFO, can reduce the miss ratio of state-of-the-art algorithms by up to 59.8%.

2 Why FIFO and What it needs

The benefits of FIFO over LRU have been explored in many previous works [37, 38, 82, 83]. For example, FIFO has less metadata (if any) and requires no metadata update on each cache hit, and thus is faster and more scalable than LRU. In contrast, LRU requires updating six pointers on each cache hit, which is not friendly for modern computer architecture due to random memory accesses and extensive locking. Moreover, FIFO is always the first choice when implementing a flash cache because it does not incur write amplification [15, 22, 55, 84]. Although FIFO has throughput and scalability benefits, it is common wisdom that FIFO provides lower efficiency (higher miss ratio) than LRU.

To understand the various factors that affect the miss ratio, we introduce a cache abstraction (Fig. 1). A cache can be viewed as a logically total-ordered queue with four operations: insertion, removal, promotion, and demotion. Objects in the cache can be compared and ordered based on some metric (e.g., time since the last request), and the eviction algorithm eviction the least valuable object based on the metric. Insertion and removal are user-controlled operations, where removal can either be directly invoked by the user or indirectly via the use of time-to-live (TTL). Promotion and demotion are internal operations of the cache used to maintain the logical ordering between objects.

We observe that most eviction algorithms use promotion to update the ordering between objects. For example, all the LRU-based algorithms promote objects to the head of the
FIFO-Reinsertion (LP) leads to QD

CLOCK-2

2-bit CLOCK, small size

2-bit CLOCK, large size

(a) FIFO-Reinsertion, small size
(b) FIFO-Reinsertion, large size
(c) 2-bit CLOCK, small size
(d) 2-bit CLOCK, large size
(e) LP leads to QD

Figure 2: (a,b,c,d): the fraction of the 5307 traces on which an algorithm has a lower miss ratio when comparing LRU with FIFO-Reinsertion (1-bit CLOCK) and 2-bit CLOCK. FIFO-Reinsertion and 2-bit CLOCK are more efficient than LRU, with a lower miss ratio on most traces. (e): Lazy Promotion often leads to Quick Demotion. Using FIFO-Reinsertion as an example, the newly-inserted object $G$ will be pushed down by both objects requested before (e.g., $B$, $D$) and after $G$. In contrast, only objects requested after $G$ can push $G$ down in LRU.

queue on cache hits, which we call eager promotion. Meanwhile, demotion is performed implicitly: when an object is promoted, other objects are passively demoted. We call this process passive demotion, a slow process as objects need to traverse through the cache queue before being evicted. However, we will show that instead of eager promotion and passive demotion, eviction algorithms should use Lazy Promotion ($\S$3) and Quick Demotion ($\S$4).

3 Lazy Promotion

To avoid popular objects from being evicted while not incurring much performance overhead, we propose adding Lazy Promotion on top of FIFO (called LP-FIFO), which promotes objects only when they are about to be evicted. Lazy Promotion aims to retain popular objects with minimal effort. An example is FIFO-Reinsertion: an object is reinserted at eviction time if it has been requested while in the cache.

LP-FIFO has several benefits over eager promotion (promoting on every access) used in LRU-based algorithms. First, LP-FIFO inherits FIFO’s throughput and scalability benefits because few metadata operations are needed when an object is requested. For example, FIFO-Reinsertion only needs to update a Boolean field upon the first request to a cached object without locking. Second, performing promotion at eviction time allows the cache to make better decisions by accumulating more information about the objects, e.g., how many times an object has been requested.

To understand LP-FIFO’s efficiency, we performed a large-scale simulation study on 5307 production traces from 10 data sources (Table 1), which include open-source and proprietary datasets collected between 2007 and 2020. The 10 datasets contain 814 billion (6,386 TB) requests and 55.2 billion (533 TB) objects, and cover different types of caches, including block, key-value (KV), and object caches. We further divide the traces into block and web (including Memcached and CDN). We choose small/large cache size as 0.1%/10% of the number of unique objects in the trace.

We compare the miss ratios of LRU with two LP-FIFO algorithms: FIFO-Reinsertion and 2-bit CLOCK. 2-bit CLOCK tracks object frequency up to three, and an object’s frequency decreases by one each time the CLOCK hand scans through it. Objects with zero frequency are evicted.

Common wisdom suggests that these two LP-FIFO examples are LRU approximations and will exhibit higher miss ratios than LRU \[11, 38, 43, 47, 60\]. However, we found that LP-FIFO often exhibits miss ratios lower than LRU.

Fig. 2 shows that FIFO-Reinsertion and 2-bit CLOCK are better than LRU on most traces. Specifically, FIFO-Reinsertion is better than LRU on 9 and 7 of the 10 datasets using a small and large cache size, respectively. Moreover, on half of the datasets, more than 80% of the traces in each dataset favor FIFO-Reinsertion over LRU at both sizes. On the two social network datasets, LRU is better than FIFO-Reinsertion (especially at the large cache size). This is because most objects are accessed more than once \[3\], and using one bit to track object access is insufficient. Therefore, when increasing the one bit in FIFO-Reinsertion (CLOCK) to two bits (2-bit CLOCK),

\[\text{Note that FIFO-Reinsertion, 1-bit CLOCK, and Second Chance are different implementations of the same eviction algorithm.}\]

\[\text{We suspect this impression came from the 1960s when LRU and CLOCK were designed for virtual memory page replacement. We conjecture that CLOCK may not work as well as LRU for such workloads because LRU can better adapt to sudden working set changes. According to Denning, memory access patterns show abrupt changes between phases [27]. However, we do not observe such patterns in the block and web cache workloads.}\]

\[\text{Many cache traces are collected after the first-layer cache, e.g., the CDN cache is behind browser caches, and the block traces record requests after the page cache. The two social network cache datasets used are from first-layer caches, contributing to high object access frequencies. Moreover, the high frequency could also come from the nature of being a social network or key-value cache workload.}\]
Table 2: The miss ratios of the algorithms in Fig. 3.

<table>
<thead>
<tr>
<th>Algorithm/workload</th>
<th>LRU</th>
<th>ARC</th>
<th>LHD</th>
<th>Belady</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR</td>
<td>0.5263</td>
<td>0.4899</td>
<td>0.5131</td>
<td>0.4438</td>
</tr>
<tr>
<td>Twitter</td>
<td>0.2005</td>
<td>0.1841</td>
<td>0.1756</td>
<td>0.1309</td>
</tr>
</tbody>
</table>

we observe that the number of traces favoring LP-FIFO increases to around 70%. Across all datasets, 2-bit CLOCK is better than FIFO on all datasets at the small cache size and 9 of the 10 datasets at the large cache size.

Two reasons contribute to LP-FIFO’s high efficiency. First, Lazy Promotion often leads to Quick Demotion (§4). For example, under LRU, a newly-inserted object G is pushed down the queue only by 1) new objects and 2) cached objects that are requested after G. However, besides the objects requested after G, the objects requested before G (but have not been promoted, e.g., B, D) also push G down the queue when using FIFO-Reinsertion (Fig. 2e). Second, compared to promotion at each request, object ordering in LP-FIFO is closer to the insertion order, which we conjecture is better suited for many workloads that exhibit popularity decay – old objects have a lower probability of getting a request.

While LP-FIFO surprisingly wins over LRU in miss ratio, it cannot outperform state-of-the-art algorithms. We next discuss another building block that bridges this gap.

4 Quick Demotion

Efficient eviction algorithms not only need to keep popular objects in the cache but also need to evict unpopular objects fast. In this section, we show that Quick Demotion (QD) is critical for an efficient eviction algorithm, and it enables FIFO-based algorithms to achieve state-of-the-art efficiency.

Because demotion happens passively in most eviction algorithms, an object typically traverses through the cache before being evicted. Such traversal gives each object a good chance to prove its value to be kept in the cache. However, cache workloads often follow Zipf popularity distribution [8, 15, 20, 82] with most objects being unpopular. This is further exacerbated by 1) the scan and loop access patterns in the block cache workloads [12, 65, 72], and 2) the vast existence of dynamic and short-lived data, the use of versioning in object names, and the use of short TTLs in the web cache workloads [82]. We believe the opportunity cost of new objects demonstrating their values is often too high: the object being evicted at the tail of the queue may be more valuable than the objects recently inserted.

Removing low-value objects faster is not a new idea and has been discussed under various contexts, such as removing scan pages [12, 45], correlated accesses [47], and one-hit wonders [2, 54]. These observations have inspired eviction algorithms such as 2Q [47], MQ [92], ARC [56], SLRU [42], LHD [12], and Hyperbolic [19]. However, we find that the demotion in existing algorithms is often not fast enough.

We study how different algorithms spend cache resources on objects of varying popularity. The resource consumption of an object is calculated using $C_{obj} = \sum (T_{eviction} - T_{insertion}) \times S_{obj}$ similar to the idea in the previous work [12]. Throughout this work, we assume objects to be uniform in size so that we can focus on the effect of access patterns on efficiency. Fig. 3 shows two representative traces, and Table 2 shows the corresponding miss ratios. ARC and LHD often spend fewer resources on unpopular objects than LRU and show lower miss ratios. Between ARC and LHD, ARC spends fewer resources on unpopular objects and has a notably lower miss ratio than LHD on the MSR trace. We have a similar observation on the Twitter trace as well. Moreover, among all four algorithms, Belady [14] always spends the fewest resources on unpopular objects and has significantly lower miss ratios. In summary, efficient algorithms often spend fewer resources on unpopular objects.

To further illustrate the importance of Quick Demotion, we add a simple QD technique on top of state-of-the-art eviction algorithms (Fig. 4). The QD technique consists of a small probationary FIFO queue storing cached data and a ghost FIFO queue storing metadata of objects evicted from the probationary FIFO queue. The probationary FIFO queue uses 10% of the cache space and acts as a filter for unpopular objects: objects not requested after insertion are evicted early from the FIFO queue. The main cache runs a state-of-the-art algorithm and uses 90% of the space. And the ghost FIFO stores as many entries as the main cache. Upon a cache miss, the object is written into the probationary FIFO queue unless
it is in the ghost FIFO queue, in which case, it is written into the main cache. When the probationary FIFO queue is full, if the object to evict has been accessed since insertion, it is inserted into the main cache. Otherwise, it is evicted and recorded in the ghost FIFO queue.

We add this FIFO-based QD technique to five state-of-the-art eviction algorithms, ARC [56], LIRS [45], CACHEUS [65], LeCaR [72] and LHD [12]. We used the open-source LHD implementation from the authors, implemented the others following the corresponding papers, and cross-checked with open-source implementations. We evaluated the QD-enhanced and original algorithms on the 5307 traces. Because the traces have a wide range of miss ratios, we choose to present each algorithm’s miss ratio reduction from FIFO calculated as $\frac{m_{\text{FIFO}} - m_{\text{alg}}}{m_{\text{FIFO}}}$.

Fig. 5 shows that the QD-enhanced algorithms further reduce the miss ratio of each state-of-the-art algorithm on almost all percentiles. For example, QD-ARC (QD-enhanced ARC) reduces ARC’s miss ratio by up to 59.8% with a mean reduction of 1.5% across all workloads on the two cache sizes, QD-LIRS reduces LIRS’s miss ratio by up to 49.6% with a mean of 2.2%, and QD-LeCaR reduces LeCaR’s miss ratio by up to 58.8% with a mean of 4.5%. Note that achieving a large miss ratio reduction on a large number of diverse traces is non-trivial. For example, the best state-of-the-art algorithm, ARC, can only reduce the miss ratio of LRU 6.2% on average.

The gap between the QD-enhanced algorithm and the original algorithm is wider 1) when the state-of-the-art is relatively weak, 2) when the cache size is large, and 3) on the web workloads. With a weaker state-of-the-art, the opportunity for improvement is larger, allowing QD to provide more prominent benefits. For example, QD-LeCaR reduces LeCaR’s miss ratios by 4.5% average, larger than the reductions on other state-of-the-art algorithms. When the cache size is large, unpopular objects spend more time in the cache, and Quick Demotion becomes more valuable. For example, QD-ARC and ARC have similar miss ratios on the block workloads at the small cache size. But QD-ARC reduces ARC’s miss ratio by 2.3% on average at the large cache size. However, when the cache size is too large, e.g., 80% of the number of objects in the trace, adding QD may increase the miss ratio (not shown). At last, QD provides more benefits on the web workloads than the block workloads, especially when the cache size is small. We conjecture that web workloads have more short-lived data and exhibit stronger popularity decay, which leads to a more urgent need for Quick Demotion. While Quick Demotion improves the efficiency of most state-of-the-art algorithms, for a small subset of traces, QD may increase the miss ratio when the cache size is small because the probationary FIFO is too small to capture some potentially popular objects.

Although adding the probationary FIFO improves efficiency, it further increases the complexity of the already complicated state-of-the-art algorithms. To reduce complexity, we add the same QD technique on top of the 2-bit CLOCK discussed in §3 and call it QD-LP-FIFO. QD-LP-FIFO uses two FIFO queues to cache data and a ghost FIFO queue to track evicted objects. It is not hard to see QD-LP-FIFO is simpler than all state-of-the-art algorithms — it requires at most one metadata update on a cache hit and no locking for any cache operation. Therefore, we believe it will be faster and more
scalable than all state-of-the-art algorithms. Besides enjoying all the benefits of simplicity, QD-LP-FIFO also achieves lower miss ratios than state-of-the-art algorithms (Fig. 5). For example, compared to LIRS and LeCaR, QD-LP-FIFO reduces miss ratio by 1.6% and 4.3% on average respectively across the 5307 traces. While the goal of this work is not to propose a new eviction algorithm, QD-LP-FIFO illustrates how we can build simple yet efficient eviction algorithms by adding **Quick Demotion** and **Lazy Promotion** techniques to a simple base eviction algorithm such as FIFO.

5 Discussions

**LP and QD techniques.** We have demonstrated reinsertion as an example of LP (§3) and the use of a small probationary FIFO queue as an example of QD (§4). However, these are not the only techniques. For example, reinsertion can leverage different metrics to decide whether the object should be reinserted. Besides reinsertion, several other techniques are often used to reduce promotion and improve scalability, e.g., periodic promotion [62], batched promotion [76], promoting old objects only [15], promoting with try-lock [3]. Although these techniques do not fall into our strict definition of **Lazy Promotion** (promotion on eviction), many of them effectively retain popular objects from being evicted. On the **Quick Demotion** side, besides the small probationary FIFO queue, one can leverage other techniques to define and discover unpopular objects such as Hyperbolic [19] and LHD [12]. Moreover, admission algorithms, e.g., TinyLFU [33, 34], Bloom Filter [18, 54], probabilistic [15] and ML-based [35] admission algorithms, can be viewed as a form of QD — albeit some of them are too aggressive at demotion (rejecting objects from entering the cache).

We remark that QD bears similarity with some generational garbage collection algorithms [28, 61] which separately store short-lived and long-lived data in young-gen and old-gen heaps. Therefore, ideas from garbage collection may be borrowed to strengthen cache eviction algorithms.

We believe that the design of QD-LP-FIFO opens the door to designing simple yet efficient cache eviction algorithms by innovating on LP and QD techniques. And we envision future eviction algorithms can be designed like building LEGO — adding **Lazy Promotion** and **Quick Demotion** on top of a base eviction algorithm.

**Why “X” is not better than QD-LP-FIFO.** Eviction algorithms that use multiple queues (e.g., ARC, 2Q, and 2Q variants in many production systems [4, 6, 7, 15]) share similarities with QD-LP-FIFO. However, there are two major differences between QD-LP-FIFO and previous works. First, QD-LP-FIFO only uses FIFO queues, and promotion to a different queue (e.g., main cache) only happens when an object is being evicted. Second, QD-LP-FIFO uses a tiny fixed-size FIFO queue (10% of cache size) for **Quick Demotion**, while previous works use *much larger* (e.g., 50% of cache size) or adaptive queue sizes. Ideally, the adaptive algorithms (e.g., ARC) should provide similar or lower miss ratios than **Quick Demotion**. However, our study suggests otherwise. There are a few reasons behind this. First, the adaptive algorithms’ methods to adjust queue size are not optimal. For ARC, we observe that manually limiting the queue size and slowing down the queue size adjustment often reduce miss ratios. Second, **Lazy Promotion** is resistant to request bursts and better suited for workloads with popularity decay (§3). We observe that replacing the LRU queues in ARC with FIFO-Reinsertion also reduces the miss ratio. In general, adaptive algorithms, such as ARC and CACHEUS, adapt their parameters based on a limited number of past requests, which may not predict the future well.

**Limitations.** Throughout this work, to focus on how access patterns affect cache efficiency, we ignore other factors, such as object size and TTL, which are important for web cache workloads. While the **Lazy Promotion** and **Quick Demotion** techniques we have discussed are not size-aware, designing size-aware **Lazy Promotion** and **Quick Demotion** techniques are worth pursuing in the future.

6 Conclusion

To the best of our knowledge, this is by far the most comprehensive eviction algorithm study. Contrary to the common belief, we discover that LRU-FIFO (e.g., FIFO-Reinsertion) is more efficient than LRU with lower miss ratios (in addition to its well-known benefits on throughput and scalability). Moreover, we demonstrate the importance of **Quick Demotion** for efficient caching by adding a probationary FIFO queue to five state-of-the-art eviction algorithms. The QD-enhanced algorithms can further improve the state-of-the-art algorithms’ efficiency. This study illustrates the importance of **Lazy Promotion** and **Quick Demotion** for eviction algorithms’ throughput and efficiency. And it demonstrates a new LEGO-like approach to designing future eviction algorithms.

**Acknowledgments**

We thank the anonymous reviewers for their valuable feedback. We also would like to thank the people and organizations that have open-sourced and shared production traces. We thank Cloudlab [31] for the infrastructure support for running experiments. This work is funded in part by a Meta Fellowship, and NSF grants CNS 1901410 and CNS 1956271. We also thank the members of the PDL Consortium for their interest, insights, feedback, and support.

**Availability**

The code we used is open-sourced at https://github.com/TheSys-lab/HotOS23-QD-LP.
References


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


[37] Ohad Eytan, Danny Harnik, Efi 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.


