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:
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 *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3593856.3595887

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

*work done in part at Twitter.

HotOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595887

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 LRUs" 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%.

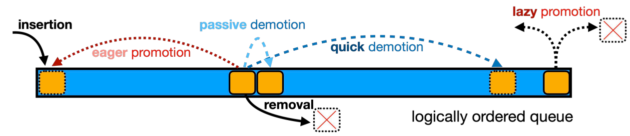


Figure 1: The cache abstraction.

Table 1: Datasets used in this work (traces with less than 1 million requests or 10,000 objects are excluded).

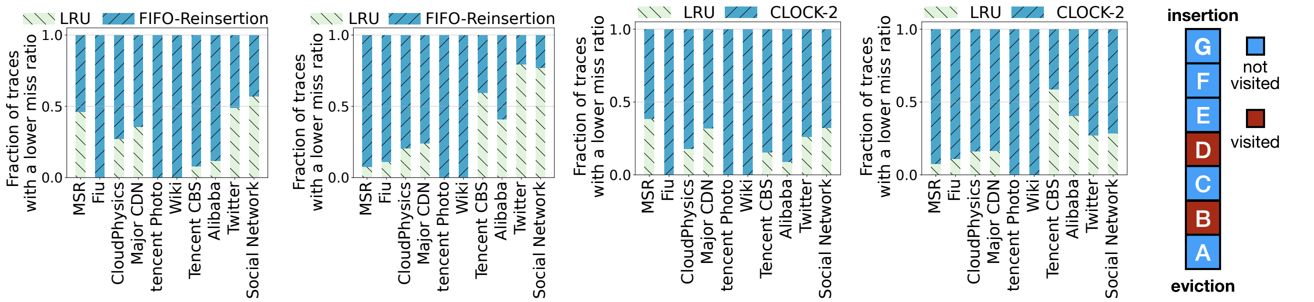
trace collections	approx time	# traces	cache type	# request (million)	# object (million)
MSR [58, 59]	2007	13	block	410	74
FIU [49]	2008	9	block	514	20
Cloudphysics[73]	2015	106	block	2,114	492
Major CDN	2018	219	object	3,728	298
Tencent Photo [91]	2018	2	object	5,650	1,038
Wiki CDN [77]	2019	3	object	2,863	56
Tencent CBS [87, 88]	2020	4030	block	33,690	551
Alibaba [1, 51, 74]	2020	652	block	19,676	1702
Twitter [82]	2020	54	KV	195,441	10,650
Social Network	2020	219	KV	549,784	42,898

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 evicts 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



(a) FIFO-Reinsertion, small **(b) FIFO-Reinsertion, large** **(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 (§3) and Quick Demotion (§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 [1]: 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 [2] [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),

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

[2]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.

[3]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.

Algorithm/workload	LRU	ARC	LHD	Belady
MSR	0.5263	0.4899	0.5131	0.4438
Twitter	0.2005	0.1841	0.1756	0.1309

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.

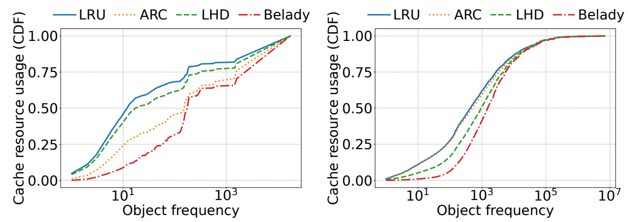
While LP-FIFO surprisingly wins over LRU in miss ratio, it cannot outperform state-of-the-art algorithms. We next discuss another building block that bridges this gap.

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],



(a) MSR trace (hm0) **(b) Twitter trace (cluster52)**

Figure 3: Cache resource consumption by objects in different algorithms. More efficient algorithms spend fewer resources on unpopular objects.

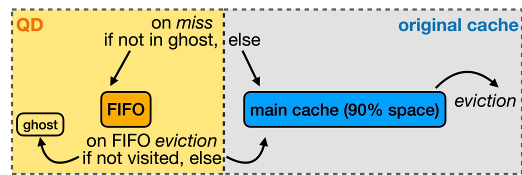
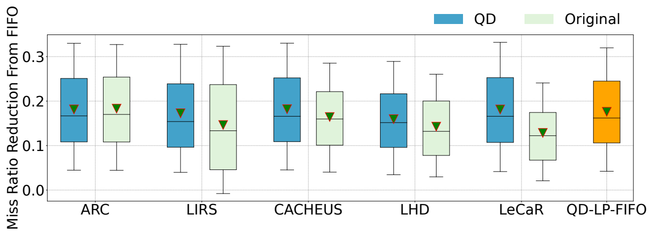


Figure 4: An example of QD: add a probationary FIFO queue to an existing cache.

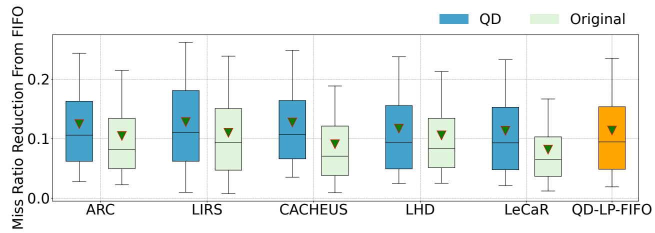
LHD [12], and Hyperbolic [19]. However, we find that the demotion in existing algorithms is often not fast enough.

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.

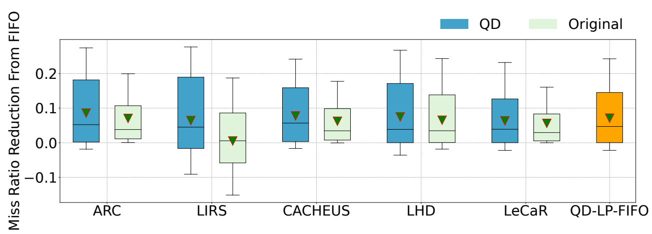
To further illustrate the importance of Quick Demotion, we add a simple QD technique on top of state-of-the-art eviction algorithms (Fig. 4). The QD technique consists of a small probationary FIFO queue storing cached data and a ghost FIFO queue storing metadata of objects evicted from the probationary 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



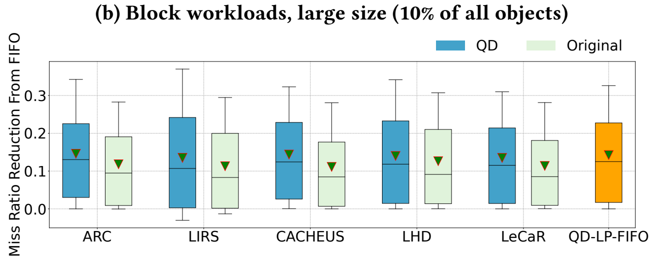
(a) Block workloads, small size (0.1% of all objects)



(b) Block workloads, large size (10% of all objects)



(c) Web workloads, small size (0.1% of all objects)



(d) Web workloads, large size (10% of all objects)

Figure 5: Evaluated on the 5307 traces, QD-enhanced algorithms outperform state-of-the-art algorithms at both small and large cache sizes. QD-LP-FIFO achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.

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 [4]. 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 $\frac{mr_{FIFO} - mr_{algo}}{mr_{FIFO}}$.

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

[4] All state-of-the-art algorithms are complex, and we found two different open-source LIRS implementations used in previous works have bugs.

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 LP-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 NFS 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

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

[2] Better handling for one-hit-wonder objects. https://phabricator.wikimedia.org/T144187. Accessed: 2021-12-06.

[3] Hhvm concurrent lru cache. https://github.com/facebook/hhvm/blob/master/hphp/util/concurrent-lru-cache.h. Accessed: 2023-01-25.

[4] Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2021-12-06.

[5] Multi-gen lru. https://docs.kernel.org/admin-guide/mm/multigen_lru.html. Accessed: 2023-01-12.

[6] Mysql 5.7 reference manual, 14.5.1 buffer pool. https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html. Accessed: 2023-01-12.

[7] Page frame reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html. Accessed: 2023-01-16.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

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

[10] Jiwoo Bang, Chungyong Kim, Sunggon Kim, Qichen Chen, Cheongjun Lee, Eun-Kyu Byun, Jaehwan Lee, and Hyeonsang Eom. Finer-LRU: A Scalable Page Management Scheme for HPC Manycore Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–576, May 2021. ISSN: 1530-2075.

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

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

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

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

[15] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768. USENIX Association, November 2020.

[16] Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, Hotnets'18, pages 134–140, Redmond WA USA, November 2018. ACM.

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

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

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

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

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

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

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

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

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

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

[27] Peter J. Denning. Working Set Analytics. *ACM Computing Surveys*, 53(6):1–36, November 2021.

[28] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, 2004.

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

[30] Dormando. Replacing the cache replacement algorithm in memcached. https://memcached.org/blog/modern-lru/. Accessed: 2023-01-12.

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

[32] Dominik Durner, Badrish Chandramouli, and Yinan Li. Crystal: a unified cache storage system for analytical databases. *Proceedings of the VLDB Endowment*, 14(11):2432–2444, July 2021.

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

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

December 2017.
[35] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019. USENIX Association.
[36] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of machine learning and systems*, volume 1 of *mlsys'20*, pages 40–52, 2019.
[37] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX workshop on hot topics in storage and file systems*, hotStorage'20. USENIX Association, July 2020.
[38] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX symposium on networked systems design and implementation*, NSDI'13, pages 371–384, 2013.
[39] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 395–411, Carlsbad, CA, July 2022. USENIX Association.
[40] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, Virtual USA, April 2021. ACM.
[41] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '22, pages 72–90, New York, NY, USA, November 2022. Association for Computing Machinery.
[42] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, November 2013. Association for Computing Machinery.
[43] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'05, page 35, USA, April 2005. USENIX Association.
[44] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4 of *FAST'05*, pages 8–8, 2005.
[45] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002.
[46] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. Association for Computing Machinery.
[47] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994. Morgan Kaufmann Publishers Inc.
[48] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994.
[49] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.
[50] Jinhyung Koo, Jinwook Bae, Minjeong Yuk, Seonggyun Oh, Jungwoo Kim, Jung-Soo Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. All-Flash Array Key-Value Cache for Large Objects. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 784–799, New York, NY, USA, May 2023. Association for Computing Machinery.
[51] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.
[52] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime. In *20th USENIX Conference on File and Storage Technologies*, FAST'22, pages 445–459, 2022.
[53] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX conference on file and storage technologies*, FAST'19, pages 143–157, Boston, MA, February 2019. USENIX Association.
[54] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, July 2015.
[55] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. In *ACM Transactions on Storage*, volume 18 of *TOS'22*, August 2022.
[56] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003.
[57] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video CDNs: building strong lines of defense. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, EuroSys'14, pages 1–13, Amsterdam, The Netherlands, 2014. ACM Press.
[58] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007.
[59] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.
[60] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 20 of *SIGMETRICS'92*, pages 35–46, June 1992.
[61] Paula Pufek, Hrvoje Grgić, and Branko Mihaljević. Analysis of garbage collection algorithms and memory management in java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1677–1682. IEEE, 2019.
[62] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern software. In *Twenty-third EuroSys*

Conference, EuroSys'23, New York, NY, USA, 2023. Association for Computing Machinery.
[63] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache:load-balanced,low-latency cluster caching with online erasure coding. In *12th USENIX symposium on operating systems design and implementation*, OSDI'16, pages 401–417, 2016.
[64] Liana V. Rodriguez, Alexis Gonzalez, Pratik Poudel, Raju Rangaswami, and Jason Liu. Unifying the data center caching layer: Feasible? Profitable? In *Proceedings of the 13th ACM workshop on hot topics in storage and file systems*, HotStorage '21, pages 50–57, New York, NY, USA, 2021. Association for Computing Machinery.
[65] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354. USENIX Association, February 2021.
[66] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021. ACM.
[67] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.
[68] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.
[69] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020.
[70] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT'17, pages 55–67, Incheon Republic of Korea, November 2017. ACM.
[71] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, 2015.
[72] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018. USENIX Association.
[73] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX conference on file and storage technologies*, FAST'15, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.
[74] Qiuping Wang, Jinhong Li, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022.
[75] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. Austere flash caching with deduplication and compression. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, ATC, pages 713–726, 2020.
[76] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached. *Intel document*, 2012.
[77] Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2020-05-06.
[78] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX conference on file and storage technologies*, FAST'21, pages 307–323. USENIX Association, February 2021.
[79] Gang Yan and Jian Li. Towards Latency Awareness for Content Delivery Network Caching. ATC'22, pages 789–804, 2022.
[80] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 66–79, New York, NY, USA, September 2017. Association for Computing Machinery.
[81] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX symposium on networked systems design and implementation*, NSDI'22, pages 1159–1177, Renton, WA, April 2022. USENIX Association.
[82] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020.
[83] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518. USENIX Association, April 2021.
[84] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.
[85] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4 of *SIGMETRICS'20*, pages 1–27, May 2020.
[86] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the Cache Warm? Manufacturing a Rule of Thumb. 2020.
[87] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace set 27917). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, October 2018.
[88] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798. USENIX Association, July 2020.
[89] Yuchao Zhang, Pengmiao Li, Zhili Zhang, Bo Bai, Gong Zhang, Wendong Wang, Bo Lian, and Ke Xu. AutoSight: Distributed Edge Caching in Short Video Network. *IEEE Network*, 34(3):194–199, May 2020.
[90] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM.
[91] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27476). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

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