# Using Lock-Free Design for Throughput-Optimized Cache Eviction

CHAO DONG and FANG WANG*, Huazhong University of Science and Technology, China
HONG JIANG, University of Texas at Arlington, USA
DAN FENG, Huazhong University of Science and Technology, China

In large-scale information systems, storage device performance continues to improve while workloads expand in size and access characteristics. This growth puts tremendous pressure on caches and storage hierarchy in terms of concurrent throughput. However, existing cache eviction policies often struggle to provide adequate concurrent throughput due to their reliance on coarse-grained locking mechanisms and complex data structures.

This paper presents a practical approach to cache eviction algorithm design, called Mobius, that optimizes the concurrent throughput of caches and reduces cache operation latency by utilizing lock-free data structures, while maintaining comparable hit ratios. Mobius includes two key designs. First, Mobius employs two lock-free FIFO queues to manage cache items, ensuring that all cache operations are executed efficiently in parallel. Second, Mobius integrates a consecutive detection mechanism that merges multiple modifications during eviction into a single operation, thereby reducing data races. Extensive evaluations using both synthetic and real-world workloads from high-concurrency clusters demonstrate that Mobius achieves a concurrent-throughput improvement ranging from 1.2× to 8.5× over state-of-the-art methods, while also maintaining lower latency and comparable cache hit ratios. The implementation of Mobius in CacheLib and RocksDB highlights its effectiveness in enhancing cache performance in practical scenarios.

CCS Concepts: • **Information systems → Information storage systems**; • **Computer systems organization**;

Additional Key Words and Phrases: Cache eviction policy, Concurrency, Throughput, Lock-free, FIFO queue

**ACM Reference Format:**
Chao Dong, Fang Wang, Hong Jiang, and Dan Feng. 2025. Using Lock-Free Design for Throughput-Optimized Cache Eviction. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 2, Article 44 (June 2025), 28 pages. https://doi.org/10.1145/3727136

*Corresponding author

Authors' Contact Information: Chao Dong, chaosdong@hust.edu.cn; Fang Wang, wangfang@mail.hust.edu.cn, Huazhong University of Science and Technology, Key Laboratory of Information Storage System (Wuhan National Laboratory for Optoelectronics), Engineering Research Center of data storage systems and Technology (School of Computer Science and Technology), Ministry of Education of China, Wuhan, Hubei, China; Hong Jiang, hong.jiang@uta.edu, University of Texas at Arlington, Department of Computer Science and Engineering, Arlington, Texas, USA; Dan Feng, dfeng@hust.edu.cn, Huazhong University of Science and Technology, Key Laboratory of Information Storage System (Wuhan National Laboratory for Optoelectronics), Engineering Research Center of data storage systems and Technology (School of Computer Science and Technology), Ministry of Education of China, Wuhan, Hubei, China.


ACM 2476-1249/2025/6-ART44
https://doi.org/10.1145/3727136

## 1 Introduction

Caches store frequently accessed data in fast but limited storage to expedite data retrieval and optimize resource utilization [26, 78]. With a history spanning over 60 years [8], the cache technology has evolved as an integral part of the computer technology, underscoring its widespread importance. Initially, caches served as bridges between fast CPUs and slow storage in processor designs [47, 51, 52]. Nowadays, the cache technology has been utilized across various domains, including file systems [16, 80], databases [72, 76], web servers [7], and middleware components [24].

Caches studied in this paper are tailored for large-scale information systems such as content delivery networks (CDNs) [27], parallel file systems [80], and databases [72]. In such environments, data often resides in large, sluggish storage devices like tapes, hard disks, and SSDs, collectively known as the backend storage. Because of the inefficiency of accessing data directly from the backend storage, caches are introduced, which are typically built upon fast storage mediums like DRAM [21], NVM [70], and flash memory [40, 57, 62]. The generalized caches for large-scale information systems include CacheLib [3], Memcached [21, 43], Redis [5, 55], and Ehcache [59, 71], which are popular in production systems.

The effectiveness of a cache system is commonly measured by two metrics: hit ratio and throughput. The hit ratio, or cache hit ratio, has traditionally been the most important metric for cache systems [2, 46, 66], representing the percentage of requests satisfied by the cache without accessing the backend storage. Since caches are much faster than the backend, cache hits can significantly accelerate data accesses [1]. The key to enhancing hit ratio is using advanced cache eviction policy. When the cache reaches its maximum capacity, the eviction policy is responsible for evicting old items to make room for new ones. Most existing cache eviction policies are based on least-recently-used (LRU) queues. The LRU-based policies manage the historical access order of cache items and evict the least recently used ones [11]. Due to the careful maintenance of access order, LRU-based policies can generally achieve relatively high hit ratios [45, 56, 58].

The other important metric is cache throughput. In large-scale information systems, multiple processes or threads typically execute tasks concurrently. Therefore, in this paper, throughput refers to concurrent throughput, which denotes the number of requests a cache system can handle simultaneously from multiple tasks within a given time frame. This metric was often underappreciated in the past. Caches typically outperformed backend storage by a much wider margin than they do in today's large-scale information systems, making cache performance less likely to become a bottleneck. However, as storage innovations have narrowed the performance gap between caches and backend storage, and as workloads continue to scale, cache throughput can no longer be overlooked [54, 74]. Similarly, cache latency, which reflects the time cost for a single cache request, has gained attention as the latency gap between caches and backend storage shrinks. As a result, the First-In-First-Out (FIFO) eviction policy has emerged as a new focus of research due to its high throughput and low latency [18, 28, 74].

Unlike the LRU-based eviction policy, which evicts items based on access order, FIFO evicts items in the order they were inserted. The simplicity of the FIFO policy helps improve throughput and reduce latency, but it is not optimal for achieving high cache hit ratios. Recent research has proposed several innovative FIFO-based eviction policies that aim to strike a better balance. While these policies achieve comparable or even better hit ratios than LRU-based policies, they come with trade-offs. For instance, S3FIFO [75] and QDLP [73] use multiple queues to improve hit ratios, but at the cost of increased memory footprint and operational latency. SIEVE [79], on the other hand, evicts items from the cache queue using exclusive locks as the concurrency control mechanism, which impedes throughput.

This paper focuses on optimizing the concurrent throughput of cache eviction while maintaining an optimal hit ratio and low cache latency. It emphasizes the importance of cache throughput and identifies locking mechanisms as a key factor limiting scalability. To address this, we introduce Mobius, a lock-free cache eviction design that builds on SIEVE with two key design innovations to substantially enhance throughput. First, Mobius employs lock-free FIFO queues as its core data structures, significantly improving cache throughput while preserving SIEVE's eviction algorithm. Second, Mobius incorporates a consecutive detection mechanism to minimize data races during eviction victim selection. These novel designs make Mobius significantly different from and superior to SIEVE.

We implement Mobius in CacheLib [3, 42] and RocksDB [12, 13, 49, 64] as the cache eviction policy, and compare its performance against state-of-the-art cache eviction policies (i.e., SIEVE[79], S3FIFO[75], CLOCK[10], OPTLRU and TinyLFU[14]) under both synthetic and real-world workloads. The evaluation results indicate that, on CacheLib, Mobius outperforms state-of-the-art policies in terms of concurrent throughput by $1.2\times - 8.5\times$, while achieving better latency and maintaining comparable cache hit ratio. On RocksDB, Mobius outperforms the original implementation in the throughput of two primary read operations: point-read and range-read. A point-read retrieves the value of a single specific key, while a range-read (also called a range scan) retrieves multiple key-value pairs within a specified key range. Mobius increases range-read throughput by 14% and point-read throughput by 4%. To the best of our knowledge, Mobius is the first caching scheme to achieve a fundamentally balanced and satisfactory performance across hit ratio, cache throughput, and cache latency.

The rest of the paper is structured as follows. We begin with the background of existing cache eviction policies (§2). Then, we delve into the motivation behind this paper (§3). Next, we discuss the design of Mobius (§4). Subsequently, we detail the entire evaluation process for Mobius (§5). Finally, we conclude the paper (§6).

## 2 Background

In this section, we provide the necessary background for our research. First, we explore why cache throughput has garnered more attention recently (§2.1). Second, we introduce several popular cache eviction policies most relevant to the proposed Mobius policy (§2.2). Third, we assess their performance in the CacheLib platform to gain important insight for our research (§2.3).

### 2.1 Cache throughput

Cache throughput was not a major concern in earlier cache systems. However, it has gained increasing tractions in recent years [19]. In 2022, developers of RocksDB observed that the block cache was limiting the performance of RocksDB due to its poor concurrency capabilities [23]. In response, the developers implemented a new cache with highly concurrent throughput in multi-threaded environments, albeit at the expense of a lower hit ratio. Testing revealed that the new cache significantly improved RocksDB's performance across various workloads. Similarly, researchers found that the existing page cache in the Linux kernel could decrease throughput in high-performance computing (HPC). Direct access to the backend storage is sometimes more efficient than using the cache due to its poor concurrent throughput [53]. Cache throughput has become increasingly important in system performance because of the following two reasons:

**Faster backend storage**. Innovations in storage media and interfaces, such as 3D-XPoint [25], 3D-NAND flash [22], and the NVMe protocol [39], have popularized all-flash storage (or solid-state storage) in academic and industrial sectors [35, 44]. Solid-state drives (SSDs) composed of flash memory chips offer high throughput and low latency. Mainstream enterprise NAND SSDs can achieve 1.1M IOPS for random reads [60], while Intel Optane SSDs have reached 1.5M IOPS [30]. As

observed in our tests (Section 2.3), the maximum throughput of CacheLib (a DRAM cache system) in its default configuration is around 3 MOPS. The throughput gap between the backend storage and cache systems has narrowed. If caches offer no significant performance advantage, their practical application may be questioned.

**Highly concurrent workloads**. The proliferation of data-intensive applications, such as large language model (LLM) [48], big data analytics [15], and real-time processing [38], imposes greater I/O demands on storage systems and caches. These applications generate highly concurrent requests for processing and analysis. For instance, in a social graph cache of Meta, there were 1.5M requests every two minutes at peak traffic times [3]. In HPC platforms, the I/O throughput generated by a single job may reach 10 TB/s and 100 MOPS, involving millions of processes [31]. At the CDN company Cloudflare, the peak throughput reaches 55 million requests per second. [67]. This surge of concurrent requests strains cache systems' throughput capabilities, necessitating robust solutions.

As high throughput has become an important design goal in the last few years, some notable solutions have been proposed to improve the scalability of cache systems. One solution is cache sharding and fine-grained locking, which is used in RocksDB block cache[74] and Segcache[74]. The cache is split into multiple shardings managed by independent cache queue, mitigating global contention between shardings. Another solution is designing novel cache eviction policies, of which we introduce a few most relevant to our Mobius study next.

## 2.2 Eviction policy

In cache systems, the eviction policy plays a pivotal role in managing the limited cache space. It determines which data item is least likely to be accessed in the future and thus evicts it when the cache reaches its full capacity. The eviction policy significantly influences hit ratios and can become a bottleneck for cache throughput. Therefore, studying eviction policies is crucial for addressing throughput issues.

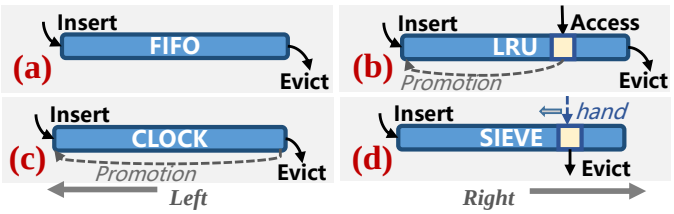



Fig. 1. Principles of FIFO, LRU, CLOCK, and SIEVE.

Most eviction policies fall into two categories: FIFO-based [32, 73, 75] and LRU-based [11, 34, 45]. The FIFO eviction policy evicts the oldest data item in the cache, while the LRU eviction policy evicts the least recently accessed data item. The FIFO policy, depicted in Figure 1a, manages data based on insertion order. New cache items are inserted on the left of the queue and oldest items are evicted from the right. Differently, the LRU policy, shown in Figure 1b, promotes an accessed data item to prolong its stay in the cache.

FIFO's simplicity can result in inefficient caching decisions. The CLOCK algorithm was introduced to retain FIFO's simplicity while approximating the LRU cache replacement policy. The principle of CLOCK is illustrated in Figure 1c. In CLOCK, each cache item is associated with a reference bit. When a cache hit occurs, the item's reference bit is set to 1. During eviction, the algorithm examines the oldest item from the right of the queue. If the item's reference bit is 1, it is granted a "second chance" to remain in the cache. The reference bit is then reset to 0, and the item is treated as a new

entry, effectively re-inserting it at the left of the queue. The CLOCK algorithm continues searching for an item with a reference bit of 0, which is selected as the eviction victim. This "second-chance" mechanism is what sets CLOCK apart from the classic FIFO policy.

Derived from CLOCK, SIEVE [79] is a novel FIFO-based policy that introduces a key modification to the eviction process. Unlike CLOCK, SIEVE retains accessed cache items in their original positions rather than reinserting them into the queue. As shown in Figure 1d, the eviction process in SIEVE is managed by a scanning hand that identifies eviction candidates. When the scanning hand encounters an item with a reference bit set to 1 (indicating that it has been accessed), SIEVE clears the reference bit and moves the hand to the next item, scanning from right to left. This process continues until an unaccessed item with a reference bit of 0 is found, which is then evicted from the queue. New items are always inserted at the left end of the queue, ensuring a clear separation between newly added and retained items. By performing evictions directly within the queue, SIEVE effectively maintains a balance between simplicity and improved cache management.

Extensive evaluations in prior research have highlighted SIEVE's superior performance. As reported in [79], SIEVE surpasses nine state-of-the-art algorithms in cache hit ratio for over 45% of the 1,559 tested traces. This strong cache performance is attributed to its ability to rapidly evict unpopular data, enabled by the continuous movement of the eviction hand. Quick eviction has been recognized as an effective strategy for handling scan workloads and widely used in cache eviction design, including LHD[2], ARC[41], LRU-K[45], and S3FIFO[75]. Hence, SIEVE efficiently handles scan requests and large-scale transient workloads, freeing up space for hot data.

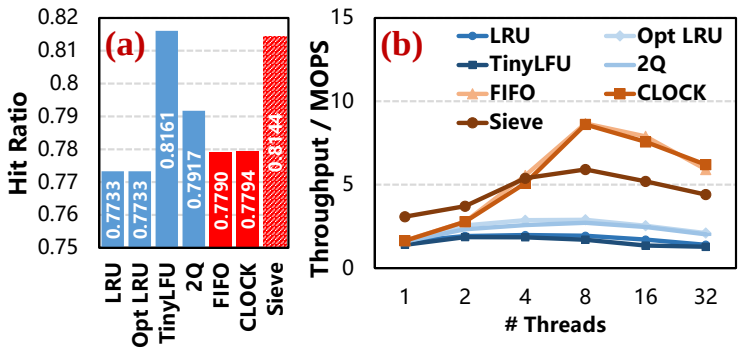



Fig. 2. Performance analysis of LRU, Optimized LRU, TinyLFU, 2Q, FIFO, CLOCK, and SIEVE in CacheLib with synthetic workloads following Zipfian distributions. (a) Hit ratio in a single thread. (b) Throughput across 1-32 threads.

The four cache eviction policies can serve as building blocks for designing advanced policies with higher cache hit ratios. LRU-K (typically with K=2) enhances LRU with a more aggressive eviction strategy. LRU-2 maintains a history of the last two accesses for each item, prioritizing the eviction of items accessed only once, followed by those with the oldest second access timestamp. This approach enables LRU-K to quickly remove cold data. TwoQ [34] employs a FIFO queue as an access history buffer to enforce stricter admission control. MQ [81] utilizes multiple LRU queues to classify items based on their access frequency. LIRS [33] combines LRU for frequently accessed items with FIFO for recently accessed but less frequently used items. Furthermore, replacing FIFO/LRU queues in these policies with CLOCK or SIEVE can yield more advanced eviction strategies. For instance, substituting LRU with SIEVE in TwoQ improves the cache hit ratio while also increasing throughput [79].

### 2.3 Performance analysis.

In this section, we report and analyze our performance evaluation of seven popular or state-of-the-art eviction policies, including four LRU-based policies (LRU [11], Optimized LRU, TinyLFU [14], 2Q [34]) and three FIFO-based policies(FIFO, CLOCK [10], and SIEVE[79]). Tests were conducted using CacheLib with a dataset comprising 1M unique objects. The cache size is 10% of the dataset size. Workloads are 100M generated reads to the dataset, following Power-law (generalized Zipfian) distributions [9] with skewness $\alpha = 1$. The Zipfian distribution represents the majority of real-world workloads [4, 63, 77].

Figure 2(a) illustrates the cache hit ratios of the seven eviction policies in a single-thread test, and Figure 2(b) shows the cache throughput as a function of the thread count from 1 to 32. The FIFO-based policies are marked in reddish colors, and the LRU-based policies are marked in blueish ones. From Figure 2(a), we observe that most FIFO-based policies exhibit a lower hit ratio in the single-thread test. Among them, SIEVE demonstrates outstanding performance, maintaining a competitive hit ratio compared to Tiny-LFU. However, as depicted in Figure 2(b), although the concurrent throughput of SIEVE is higher than LRU-based policies, it is much lower than that of CLOCK and FIFO. With the thread count of eight, SIEVE reaches a maximum throughput of 5.9 MOPS, which is only 68.6% of that of FIFO and CLOCK. This observation motivates us to investigate into the factors contributing to SIEVE's inferior throughput performance to FIFO and CLOCK.

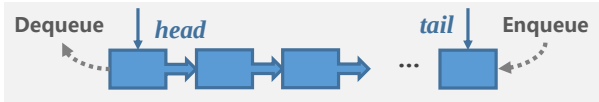
## 3 Motivation

A cache system operates through three fundamental actions: *access*, *insert*, and *evict*. When a user reads data from the cache, an *access* operation is triggered. Two scenarios are possible: if the cache hits, the data is returned directly to the user; if the cache misses, the system first *evicts* a stale cache item if the cache is full, and then *inserts* the new item into the cache. These three fundamental actions form the foundation of all cache operations. For example, a cache *remove* operation can be implemented by marking the target cache item as unreadable. Since the item can no longer be accessed, it eventually becomes an eviction candidate. Similarly, a *replace* operation combines *insert* and *remove* behaviors. These three actions are the cornerstone of cache management, directly impacting the efficiency and performance of cache eviction policies.

In LRU-based eviction policies, items are frequently moved from their original positions upon access, a process known as promotion [73]. To enable efficient promotion from any position, LRU-based policies are typically implemented using doubly linked lists [21, 34, 41]. However, promotion from a doubly linked list requires modifying the pointers of neighboring items. Concurrent promotions from the same list can lead to race conditions, compromising the consistency of the list. As a result, LRU-based policies rely on coarse-grained locks as the concurrency control mechanism, which ensures the correct executions of concurrent promotions while maintaining data consistency, integrity, and isolation. Unfortunately, coarse-grained locks significantly hinder concurrent throughput. To mitigate the overhead associated with promotion, CacheLib adopts Optimized LRU (OPTLRU) as its default eviction policy. OPTLRU limits the number of promotions within a given time frame, thereby reducing lock contention and improving throughput.

SIEVE outperforms LRU-based policies by eliminating the need for locks during data access. In SIEVE, accessing a cache item only involves changing its reference bit, without altering its position. However, the doubly linked list and locking are still required for evictions. SIEVE employs a hand to select eviction victims cyclically within the queue. Item removal of SIEVE, akin to the promotion of LRU-based policies, is inside the queue, necessitating locks to keep the consistency of the doubly linked list.

On the other hand, the FIFO and CLOCK eviction policies are simpler than their counterparts. These policies do not remove items inside the queue but only involve two primary types of data movements: *enqueue* and *dequeue*. Implementation of these operations is typically a lock-free FIFO queue based on a singly linked list [61, 68]. As depicted in Figure 3, *enqueue* operations are linking new data at the tail of the list, while *dequeue* operations remove old data from the head of the list.




**Dequeue ( ):**
**repeat:**
*oldHead* ← *head*
**until** CAS_head (*oldHead*, *oldHead.next*) is true
return *oldHead*

**Enqueue (*node*):**
**repeat:**
*oldTail* ← *tail*
**until** CAS_tail (*oldTail*, *node*) is true
*oldTail.next* ← *node*

Fig. 3. Lock-free FIFO queue implementation using a singly linked list to support *enqueue* and *dequeue* operations. *Enqueue* inserts new data at the tail of the list, while *dequeue* removes old data from the head of the list.

The lock-free implementation depicted in Figure 3 relies on single-word compare-and-swap (CAS) atomic primitives, which are atomic instructions to achieve efficient concurrency control without using locks [29]. These primitives compare the content of a memory location with a given value and atomically modifying it if they match. For example, in the instruction *CAS_tail(oldTail, node)*, *tail*, *oldTail*, and *node* are all single-word pointers (memory addresses) of cache items. This atomic instruction first compares *tail* to *oldTail*. If they are the same, set *tail* to *node* and return true. If they are different, do nothing but return false. CAS primitives are widely supported across different platforms and are significantly more efficient than exclusive locks.

To intuitively illustrate the concurrency control overhead of different cache eviction policies, we further evaluate LRU, OptLRU, SIEVE, and CLOCK in CacheLib using 16 threads. The testing method and data traces align with those in Section §2.3. We measure the time for access, insert, and evict operations to assess their overhead, which is divided into two parts: the concurrency control cost for locks or CAS primitives and the remaining cost associated with operating data structures. The results are presented in Figure 4.

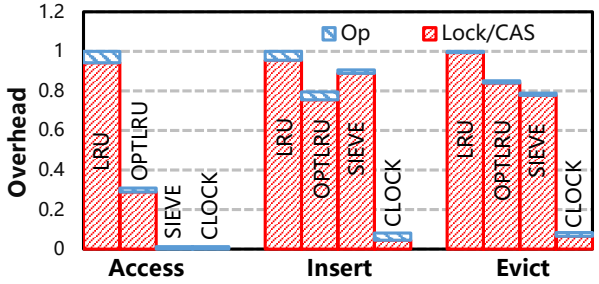



Fig. 4. Overhead of *access*, *insert*, and *evict* operations in different cache eviction policies. All results are normalized to the total overhead of LRU.

In the figure, there are three groups of bars representing the overhead of access, insert, and evict operations, respectively. Each group contains four bars corresponding to the four eviction

policies. The red sections of the bars indicate the concurrency control overhead (Lock/CAS), while the blue sections represent the remaining operational overhead (Op). LRU serves as the baseline, with its overhead set to 1 in each group. The overhead of the other policies is normalized to the total overhead of LRU. From the figure, we can draw four observations.

a) Lock contention constitutes over 90% of *LRU*'s overhead in the 16-thread tests, highlighting it as the primary scalability bottleneck for current cache eviction algorithms.

b) *OptLRU* exhibits lower access overhead than *LRU*. The result aligns with its design goal of reducing promotions during accesses and mitigating lock contentions. However, its insert and evict overhead remains high.

c) SIEVE shows far less access overhead due to its lack of cache queue updates, which enhances throughput compared to LRU-based policies. However, its insert and evict operations still incur high overhead due to locking mechanisms.

d) CLOCK exhibits notably lower overhead, particularly in concurrency control costs during insert and evict operations, being only 10% of LRU. This efficiency is attributed to its lock-free design, contributing to its superior performance.

SIEVE reduces lock contention during access but cannot alleviate it during eviction. This limitation is shared by several recent policies. For instance, the FrozenHot cache [54] periodically constructs a read-only, lock-free hash table to index hot cache items, avoiding locks during cache hits. However, FrozenHot uses an LRU-based list to manage missed cache items, which still leads to lock contention during cache misses and evictions. Furthermore, the cache hit ratio periodically decreases when the hash table is being rebuilt.

A more effective approach is to use lock-free data structures exclusively for managing cache items. LearnStore [36, 69] randomly selects cache items and pushes them into a FIFO queue as eviction candidates, avoiding lock contention during cache operations. However, this approach struggles to evict stale cache items promptly, resulting in wasted cache space and a reduced hit ratio. S3FIFO [75] employs three lock-free FIFO queues for cache management, achieving optimal hit ratio and scalable throughput. However, its complex design and use of multiple data structures increase the latency of all cache operations.

Existing research does not offer an optimal solution across all dimensions of hit ratio, cache latency, and throughput. To address this, we propose Mobius, a lock-free version of SIEVE, which retains the optimal hit ratio and latency of SIEVE while enhancing throughput. The key features of Mobius include the use of lock-free data structures with CAS primitives instead of coarse-grained locks, and the incorporation of a consecutive detection mechanism to minimize data races.

## 4 Design of Mobius

Mobius incorporates two key design features for highly concurrent throughput, making it significantly different from SIEVE on which it is built . First, Mobius leverages two separate lock-free FIFO queues to manage cache items, thus eliminating the need for locks in all cache operations (§4.1). Second, Mobius employs a consecutive detection mechanism to minimize the occurrence of *enqueue* and *dequeue* operations, thereby mitigating data races (§4.2).

### 4.1 Two lock-free FIFO queues in rotation

SIEVE resorts to using throughput-impeding locks due to the inherent complexity of removing items within the queue. To circumvent this problem, Mobius introduces a novel approach by splitting the single queue with locks used in SIEVE into two lock-free FIFO queues. One queue is responsible for storing newly inserted data, called the *active queue*. The other queue holds retained items, termed the *dormant queue*. When the cache reaches its full capacity, Mobius dequeues items from the active queue. If the dequeued item has been recently accessed, Mobius enqueues it to the dormant queue.

When the active queue is almost empty, the two queues switch roles. In doing so, the active queue receives new cache items while the dormant queue stores previously accessed items, achieving the same purpose of the original SIEVE algorithm but without any locks. As a result, and importantly, the eviction process, which consists of enqueuing and dequeuing items from the two FIFO queues, remains lock-free.

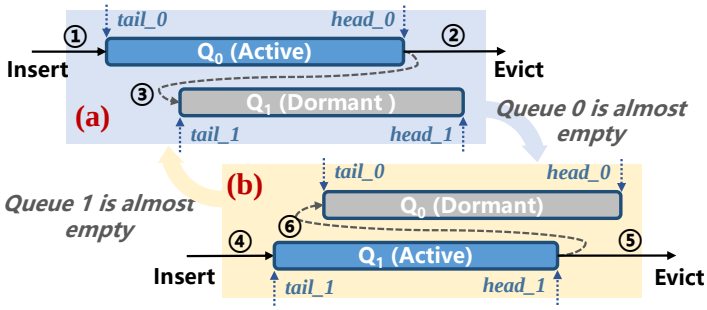



Fig. 5. Principle of Mobius's two FIFO queues. Mobius alternates cyclically between states (a) and (b).

Figure 5 illustrates the functionality of the two FIFO queues in Mobius. As shown in Figure 5a, initially, Queue 0 ($Q_0$) serves as the active queue, while Queue 1 ($Q_1$) acts as the dormant queue. New cache items are inserted into $Q_0$ (①). As the cache fills up, eviction operations are triggered. If an item dequeued from $Q_0$ has not been accessed recently (with the reference bit of 0), Mobius evicts it (②). Conversely, if the item has been accessed (with the reference bit of 1), Mobius clears its reference bit and enqueues it into $Q_1$ (③). It's worth noting that the maximum cache capacity is fixed, meaning the total size of $Q_0$ and $Q_1$ remains constant from this point onward. As cache items are moved from $Q_0$ to $Q_1$, $Q_1$ grows while $Q_0$ shrinks. Once $Q_0$ is almost empty, Mobius switches the roles of the two queues: $Q_1$ becomes the new active queue and $Q_0$ is the new dormant queue, as shown in Figure 5b. New cache items are inserted into $Q_1$ (④). Eviction candidates are dequeued from $Q_1$ (⑤) while accessed items are re-added into $Q_0$ (⑥).

Mobius employs simple and effective methods to keep data consistency in concurrent operations. All operations in Mobius rely on *enqueue* and *dequeue* actions. Races between *enqueue* and *enqueue* (or *dequeue* and *dequeue*) are protected by CAS primitives. The main challenge, however, is handling races between *enqueue* and *dequeue*. As shown in Figure 3, *enqueue* operates on the *tail* of the FIFO queue, while *dequeue* operates on the *head*. As long as the queue is not empty, the *tail* and *head* point to different memory addresses, allowing concurrent *enqueue* and *dequeue* operations to proceed without interfering with each other, ensuring thread safety. However, there are still two situations where conflicts may arise between *enqueue* and *dequeue*.

The first occurs when the queue is empty. Assuming there are two threads, denoted by $T_1$ and $T_2$, where $T_1$ launches an *enqueue* operation and $T_2$ launches a *dequeue* operation to the same empty queue. Since the queue is empty, the *enqueue* requires changing the *head* and *tail* simultaneously. An inconsistency may occur as follows: $T_1$ first updates the *head* due to the *enqueue*. Then $T_2$ clears the *head* because of the *dequeue*. Thereafter, $T_1$ updates the *tail* to finish the *enqueue*. In this case, the *head* is cleared, but the *tail* is not, resulting in an inconsistent state. To address this issue, Mobius performs *enqueue* operations by updating the *tail* first, and then modifying the *head*.

The second case arises when the queue is close to becoming empty. Consider the scenario where $T_1$ is about to *dequeue* the last item from the queue, while $T_2$ simultaneously initiates an *enqueue* operation. This creates a complex situation. The *head* and *tail* may point to the *enqueue* item and

*dequeue* item respectively, leading to an inconsistent state for the queue. To address this, Mobius optimizes the process by ensuring that the two queues never become empty after initialization. When the active queue contains only a single element, Mobius blocks all *dequeue* actions to that queue. The queue is then switched to become a dormant one, and the *dequeue* actions are redirected to the new active queue, as illustrated by the rotation from Figure 5a to 5b. By ensuring that queues never become empty after initialization, Mobius guarantees consistency even in multi-threaded workloads. The associated pseudo-code and detailed discussion can be found in the APPENDIX A.

### 4.2 Eviction with consecutive detection

Mobius uses CAS primitives as the concurrency control mechanism in the *enqueue* and *dequeue* operations on FIFO queues. Although CAS incurs less overhead compared to exclusive locks, it can still lead to data races at the *head* and *tail* of the FIFO queue, which becomes the primary factor limiting concurrent throughput. To reduce the frequency of these data races, we introduce a second critical design element of Mobius: eviction with consecutive detection.

Figure 6a illustrates a typical eviction scenario involving three cache items. First, Mobius dequeues the cache item from the head of the active queue (①). Since the item has been accessed, Mobius enqueues it into the dormant queue (②). Then, Mobius repeats the operations for the second item which is also accessed (③, ④). Finally, Mobius dequeues an unaccessed item (⑤), which is the eviction victim. In this process, Mobius dequeues items three times and enqueues items twice, totaling 5 operations.

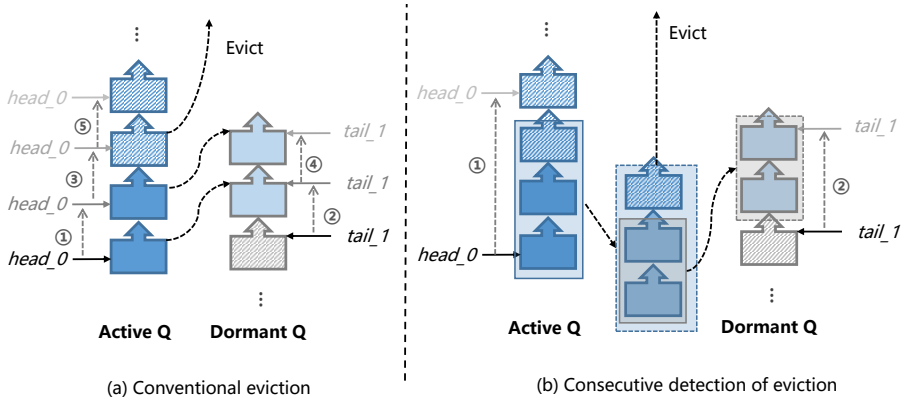



Fig. 6. Process comparison between two eviciton methods. The count of *dequeue* and *enqueue* in (a) is 5, and in (b) it is only 2.

However, we can optimize the eviction process by reducing the total number of operations. Notice that the two *enqueue* operations are consecutive, as are the three *dequeue* operations. By merging the consecutive enqueue or dequeue operations into a single engueue or dequeue operation respectively, we introduce the concept of eviction with consecutive detection.

The process of the new eviction method is shown in Figure 6b. Mobius first detects consecutive *dequeue* candidates from the head of the active queue. If the current item has been accessed, Mobius skips it and moves to the next one. This detection continues until the first unaccessed item is encountered. Then, Mobius dequeues all detected items as a sub-list (①). The last node of the sub-list is the eviction victim. After removing the eviction victim, Mobius sets the items in the sub-list as unaccessed, and enqueues them into the dormant queue (②). With this approach, Mobius

dequeues items once and enqueues items once, reducing the total number of operations from 5 to 2. Eviction with consecutive detection proves to be more efficient than the conventional eviction process.

The efficiency of consecutive detection mechanism is affected by the cache hit ratio. The higher the cache hit ratio, the more accessed items in the cache, and the more frequent the consecutive detection. In the worst-case scenario where the cache hit ratio is so low that there are no accessed items in the cache, the eviction with consecutive detection degrades to the conventional eviction process. This implies that the eviction with consecutive detection is never inferior to the conventional eviction. The associated pseudo-code and detailed discussion can be found in the APPENDIX B.

## 5 Evaluation

### 5.1 Evaluation Setup

This section introduces the experimental setup. All experiments are conducted on the same commercial machine, equipped with 2 CPUs and 512 GB of memory. Each CPU has 26 cores. As shown in Table 1, the machine runs Ubuntu 20.04.5 and utilizes GCC 9.4.0 as the compiler.

We implement Mobius in CacheLib [42] and rocksDB [49] to evaluate its throughput and cache hit ratio [1]. The evaluation encompasses four aspects: performance comparison with the state-of-the-arts in synthetic workloads (§5.2), performance breakdown analysis to assess effectiveness of Mobius' individual design features (§5.3), performance evaluation of Mobius in real-world workloads (§5.4), and performance gain of applying Mobius to RocksDB (§5.5).

Table 1. Environment Configuration

| | |
|---|---|
| CPU | Intel(R) Xeon(R) Gold 5320 (26 cores) |
| Memory | 512 GB |
| Disk | Intel D7-P5510 Series NVMe SSD 3.84TB |
| OS | Ubuntu 20.04.5 LTS |
| Kernel | 5.15.0-91-generic |

We utilize CacheLib as the platform to evaluate the performance of Mobius and comparison schemes. CacheLib is a high-throughput, low-overhead caching service that provides a thread-safe API [3]. We adopt the code publicized by previous work [65, 75], which implemented some novel cache eviction policies in CacheLib. Building upon this codebase, we implement Mobius and comparison schemes in CacheLib to facilitate multi-threaded tests.

**Workloads**. Two types of workloads are used to evaluate the performance of Mobius. The first type comprises synthetic workloads. We create 1 million objects with unique keys, where the keys are 64-bit unsigned integers, and the values are 4KB in size. Subsequently, we generate 100M requests to the dataset following a heavy-tailed power-law distribution (also known as a generalized Zipfian distribution) with a skewness parameter of $\alpha = 1$ [4]. The second type is real-world workloads from Meta [50].

**Comparison Schemes**. Several cache eviction policies are implemented for comparison with Mobius. The first is SIEVE [79], the precursor to Mobius. The second is S3FIFO [75], a FIFO-based eviction policy that utilizes multiple FIFO queues. Both S3FIFO and Mobius utilize multiple FIFO queues, but for different purposes. In S3FIFO, three lock-free FIFO queues are used to enhance

[1]The code is available at https://github.com/Anonymous-chaos/Cachelib-with-Mobius and https://github.com/Anonymous-chaos/RocksDB-with-Mobius

cache hit ratio. Queue 1 filters one-hit data, Queue 2 stores multi-hit data, and Queue 3 is a ghost queue that retains metadata of recently evicted items. While the ghost queue improves hit ratios, it also incurs additional memory overhead. The third scheme is FIFO, and the fourth is CLOCK [10]. These two policies are implemented with lock-free mechanisms. Additionally, OPTLRU and TinyLFU [14] are included for comparison purposes. OPTLRU is selected as it serves as the default eviction policy of CacheLib, balancing throughput and hit ratio. In OPTLRU, cache items can only be promoted once within a certain time period, reducing data races during access. TinyLFU is chosen due to its demonstrated high cache hit ratio in our pretesting, as shown in Figure 2.

### 5.2 Mobius under Synthetic workloads

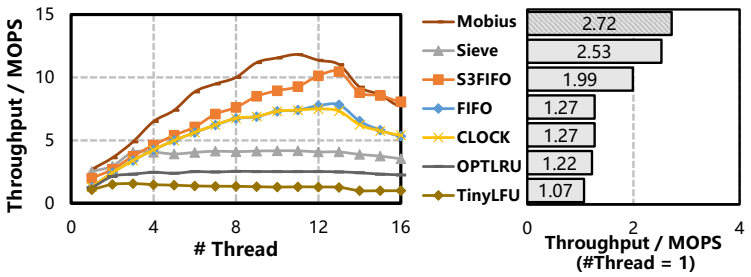
This section uses synthetic workloads to evaluate Mobius in terms of throughput and cache hit ratio. We employ a Zipfian distribution with a skewness parameter $\alpha = 1$ to generate synthetic traces. Each thread accesses the same trace, but with a unique prefix added to the keys to scale the datasets. The size of the shared cache increases as the number of threads grows. We set the cache size to 40 MB per thread, constituting 1% of the dataset size (1M * 4 KB). We then vary the thread count from 1 to 16 within the same NUMA node and record the corresponding throughput and cache hit ratio. Subsequently, we repeat the test with a cache size of 400 MB per thread while keeping other configurations constant. In the second test, the cache size accounts for 10% of the dataset size.

*5.2.1 Throughput.* Figure 7 presents the throughput results. In Figure 7a, the left graph illustrates throughput as a function of thread count when the cache size is 1% of the dataset. The right graph depicts single-thread throughput. Figure 7b shows throughput when the cache size is 10% of the dataset. From these figures, we can draw three observations.

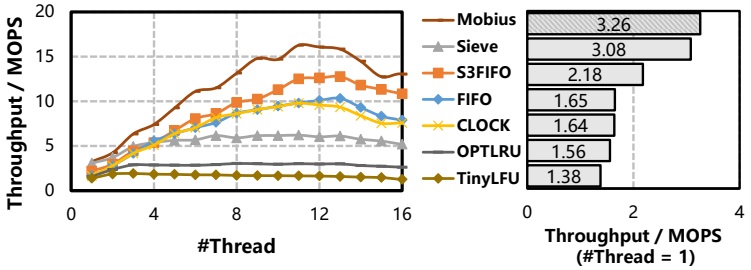
1) Mobius consistently achieves the highest throughput in multi-thread evaluations. As shown in Figure 7a, when the cache size is 1% of the dataset, Mobius peaks at 11.82 MOPS with 11 threads, surpassing SIEVE by 2.83× and OPTLRU by 4.67×. S3FIFO comes closest, achieving a peak throughput of 10.46 MOPS, still 12% less than Mobius. With a cache size of 10% of the dataset, the performance gap widens further. Mobius peaks at 16.29 MOPS, surpassing OPTLRU by 5.3× and S3FIFO by 1.27×. Mobius's superior performance stems from its lock-free design and consecutive detection of eviction.

2) Mobius also exhibits the highest single-thread throughput. As shown in Figure 7a, when the thread count is 1 and the cache size is 1% of the dataset, Mobius achieves a throughput of 2.72 MOPS, while SIEVE follows closely at 2.53 MOPS, ranking as the top two eviction policies. Mobius outperforms SIEVE due to its use of a singly linked list and no locks. Compared to the doubly linked list that SIEVE uses, a lock-free singly linked list incurs lower update overhead. Additionally, Mobius and SIEVE outperform other policies due to their simplicity and efficiency. LRU-based policies, such as TinyLRU and OPTLRU, induce data structure updates even on cache hits, resulting in the lowest single-thread throughput. CLOCK, FIFO, and S3FIFO perform better because they do not move cache items on cache hits. However, CLOCK and FIFO exhibit lower cache hit ratios, leading to more frequent *enqueue* and *dequeue* operations than SIEVE and Mobius. Similarly, S3FIFO's operation on multiple queues increases the frequency of *enqueue* and *dequeue* operations, resulting in inferior throughput to SIEVE and Mobius.

3) SIEVE outperforms LRU-based policies (OPTLRU and TinyLFU) but lags behind other FIFO-based policies. In Figure 7a, SIEVE achieves a maximum throughput of 4.17 MOPS, surpassing OPTLRU by 1.64× and TinyLFU by 2.64×. This outcome aligns with expectations, as SIEVE outperforms LRU-based policies due to its lock-free cache hits. However, SIEVE's requirement for locking




(a) Cache size = 1% dataset size




(b) Cache size = 10% dataset size

Fig. 7. Multi-thread throughput comparison of Mobius, Sieve, S3FIFO, FIFO, CLOCK, OPTLRU, and TinyLFU under two cache size scenarios. (a) 1% of dataset size, and (b) 10% of dataset size. Thread count ranges from 1 to 16.

on cache misses lowers its throughput compared to lock-free eviction policies like Mobius, S3FIFO, FIFO, and CLOCK.

*5.2.2 Hit Ratio.* Next, we analyze cache hit ratio. We record the cache hit ratio of Mobius and comparison schemes when the cache size constitutes 1% and 10% of the dataset size respectively. Specifically, we record the cache hit ratio of each policy when the thread count is 1. The results are depicted in Figure 8. Notably, in single-thread evaluation, since Mobius and SIEVE share the same algorithm, Mobius and SIEVE always exhibit the same cache hit ratio.

Although cache hit ratio is not the primary focus of this paper, we observe that Mobius, SIEVE, S3FIFO, and TinyLFU have similar hit ratios higher than other eviction policies. SIEVE and Mobius demonstrate high cache efficiency, consistent with previous research [79]. This efficiency is attributed to SIEVE's ability to fast remove unpopular cache items.

*5.2.3 Latency.* Finally, we analyze the latency of different cache eviction policies. We record the latency of each request in a 16-thread test when the cache size constitutes 10% of the dataset size. Then, we sort the latency values for each policy and compute the Cumulative Distribution Function (CDF) of latency, as shown on the left side of Figure 9. The Y-axis represents latency, from the lowest to the highest values on a logarithmic scale, while the X-axis shows the percentage of requests whose latency is below the corresponding value, ranging from 0 to 1. To facilitate comparison, we also display the P50, P90, P99, and P999 tail latencies on the right side of Figure 9. Here, the P50 value indicates that 50% of the requests have a latency lower than this threshold.

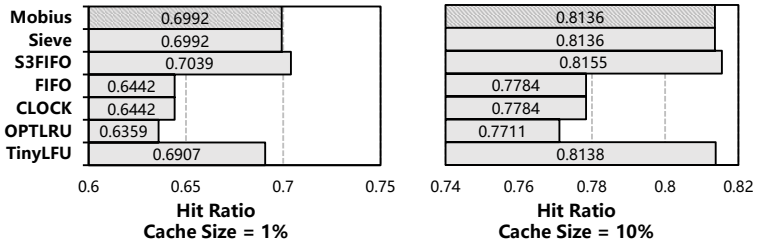



Fig. 8. Cache hit ratio comparison of Mobius, Sieve, S3FIFO, FIFO, CLOCK, OPTLRU, and TinyLFU with the cache size at 1% and 10% of the dataset size.

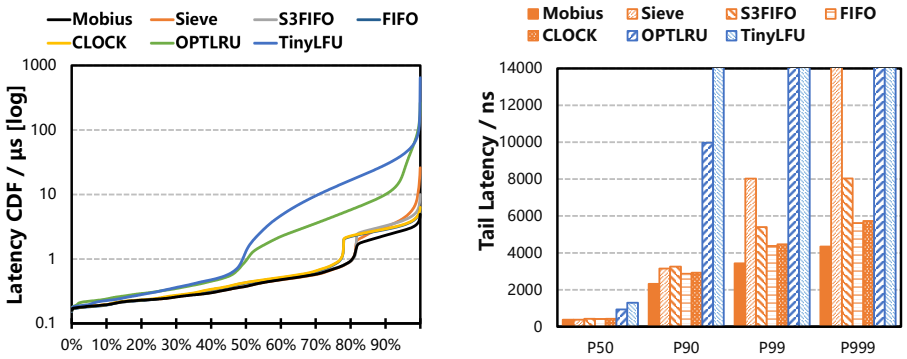



Fig. 9. Latency comparison of Mobius, Sieve, S3FIFO, CLOCK, TinyLFU, 2Q, and OPTLRU with the cache size at 10% of the dataset size and 16 threads.

From the analysis, we observe that the latency of Mobius is significantly lower than that of the comparison schemes at all measured percentiles. In the left figure, Mobius consistently shows the lowest latency. In the right figure, the P99 latency of Sieve and S3FIFO is 2.3× and 1.6× higher than Mobius, respectively, while the P999 latency is 4.6× and 1.9× higher. The low latency of Mobius is attributed to its simple and lock-free implementation.

### 5.3 Effectiveness of Mobius Design Features

In this section, we evaluate how each design feature of Mobius contributes to the overall throughput of Mobius. Mobius incorporates two key designs: lock-free FIFO queues and consecutive detection of eviction. Since these designs operate independently, we can assess the impact of each one separately. We use the original implementation of SIEVE as the baseline for comparison. We conduct tests on Mobius with and without consecutive detection to isolate the effects of each design feature, denoted by *Mobius* and *Mobius-w/d*, respectively. We also include S3FIFO, a cache eviction policy with state-of-the-art throughput optimization, for comparison purposes.

Because the effectiveness of consecutive detection is influenced by the cache hit ratio, we adjust the cache size to vary the hit ratio. We conduct tests with cache sizes ranging from 10% to 90% of the dataset size. For each test, we scale the number of threads from 1 to 32 and record the maximum throughput. The workloads consist of 100 million requests to 1 million objects, each sized at 4 KB. The maximum memory footprint remains below 115.2 GB (1 million objects * 4 KB/object * 32 threads * 90%), which is within the 512 GB physical memory limit.

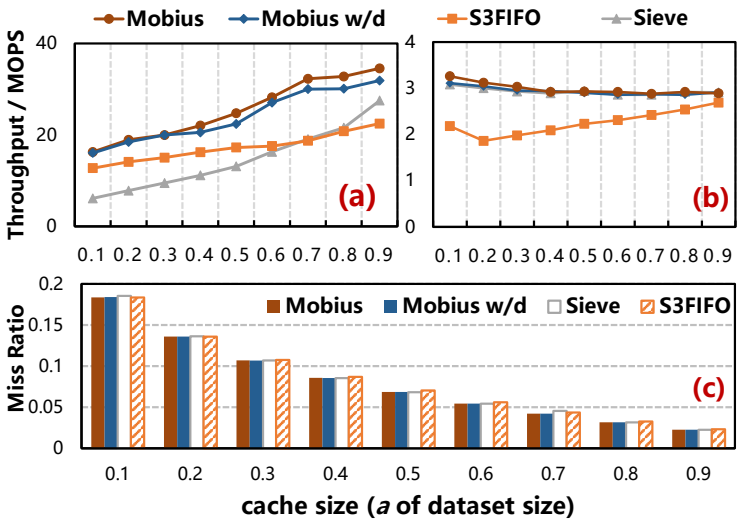



Fig. 10. Performance comparison of Mobius, Mobius without consecutive detection, SIEVE, and S3FIFO. (a) Maximum throughput in multi threads as cache size ranges from 10% to 90% of the dataset size. (b) Single-thread throughput with cache size from 10% to 90%. (c) Single-thread cache hit ratio with cache size from 10% to 90%.

We present the maximum throughput and single-thread throughput during the tests, as shown in Figures 10a and 10b, respectively. Additionally, Figure 10c illustrates the cache hit ratio in the single-thread test. From these figures, we draw five main observations.

1) Mobius consistently outperforms other schemes. When sorting schemes in Figure 10a by throughput from highest to lowest, Mobius with both lock-free structure and consecutive detection ranks highest, followed by Mobius without consecutive detection, and then S3FIFO or SIEVE. This indicates that both design features of Mobius contribute to its superiority across various cache sizes.

2) The performance gap between Mobius and SIEVE is more obvious when the cache size is not too large. At a cache size of 10% of the dataset size in Figure 10a, the maximum throughput of Mobius in multi-thread evaluations is 2.6× higher than SIEVE's. However, as the cache size approaches 90% of the dataset size, the maximum throughput of Mobius is only 1.15× higher than that of SIEVE. Mobius outperforms SIEVE due to its lock-free structure with two FIFO queues. When the cache size is smaller, the cache hit ratio is lower, and data races are more frequent. Mobius is more efficient than SIEVE in handling frequent data races.

3) Consecutive detection shows greater effectiveness with larger cache sizes. As depicted in Figure 10a, when the cache size is small (10% of the dataset size), the impact of consecutive detection on Mobius' throughput is minimal. As the cache size increases, the maximum throughput of Mobius with consecutive detection becomes noticeably higher than that of Mobius without consecutive detection (Mobius-w/d). When the cache size is 90% of the dataset size, the maximum throughput of Mobius is 8% higher. This is because consecutive detection takes effect when the cache hit ratio is high. From Figure 10c, we can observe that the cache hit ratio increases as the cache size grows larger. With a higher cache hit ratio, more items are accessed in the cache. Consequently, consecutive detection reduces the frequency of *enqueue*s and *dequeue*s in eviction, thereby increasing throughput.

4) S3FIFO consistently underperforms Mobius in all tests, particularly in single-thread evaluations and with larger cache sizes (90%). This is attributed to the complex operations involved in S3FIFO, which require more metadata management than other eviction policies. In scenarios with fewer data races for eviction, such as single-thread or large cache sizes, S3FIFO's performance suffers.

This evaluation demonstrates that both designs of Mobius contribute significantly to its performance superiority. The lock-free structure is more effective when cache hit ratio is low, while consecutive detection is more effective when cache hit ratio is high. These two designs complement each other, making Mobius efficient in different scenarios.

### 5.4 Mobius under real-world workloads

In this section, we evaluate the throughput of Mobius using four types of real-world workloads sourced from Meta [50]. The four workloads were collected from high-concurrency clusters with thousands of hosts. The first trace spans 5 consecutive days from Meta's key-value cache cluster in 2022, referred to as *KV-1* in this paper. The second trace, *KV-2*, is collected from the same cluster in 2024. The third trace, *CDN*, originates from Meta's CDN cache cluster over a 7-day period in 2023. Lastly, the *Block* trace comes from a Meta block storage cluster over 5 days in 2023. The numbers of requests in the four workloads range from 65 million to 1 billion. Similar to the evaluations with synthetic workloads, we scale the traces by adding a unique prefix to the keys in each thread. We vary the thread count from 1 to 16. The size of the shared cache is increased by the number of threads. Each thread corresponds to 4,000 MB cache space (as well as 1M cached items) to keep the cache size less than 5% of the dataset size. We repeat the evaluations for each dataset to measure their throughput.

Table 2. Features of Meta's traces

| | #Req | #Obj | Cache / Data |
|---|---|---|---|
| KV-1 | 1B | 54M | 1.85% |
| KV-2 | 782M | 63M | 1.59% |
| CDN | 250M | 86M | 1.16% |
| Block | 65M | 30M | 3.33% |

*5.4.1 Throughput.* We compare Mobius with SIEVE, S3FIFO, and OPTLRU. Figure 11 shows the throughput as a function of thread count. The results mirror those of the synthetic workloads. Mobius consistently outperforms other eviction policies, with a peak throughput more than twice that of SIEVE and three times that of OPTLRU across all traces. Only S3FIFO shows comparable performance. Compared to S3FIFO, Mobius achieves a 10% higher maximum throughput in the *KV-1* and *KV-2* traces, and a slightly higher performance in the *CDN* and *Block* traces. These findings indicate that Mobius performs well under real-world workloads.

*5.4.2 Hit Ratio.* Next, we measure the hit ratio of Mobius in real-world workloads, comparing it with S3FIFO, CLOCK, OPTLRU, TinyLFU, and 2Q. The cache hit ratio is recorded using a single thread. Since SIEVE shares the same hit ratio as Mobius in a single thread, we exclude it from the test. The results are depicted in Figure 12. We draw two observations from the figure:

1) The cache hit ratios of the evaluated cache eviction policies are closely aligned. For instance, under the *KV-1* workload, 2Q achieves the highest cache hit ratio at 84.34%, while CLOCK records the lowest at 82.82%. The difference between the highest and lowest hit ratios is 1.8%, which is

an inconspicuous gap whose performance impact (latency) may be relatively mild compared to throughput, especially as the performance gap between storage and cache narrows.

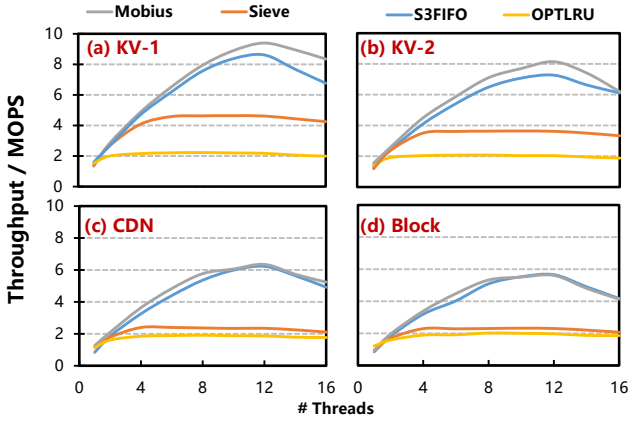



Fig. 11. Throughput comparison of Mobius, SIEVE, S3FIFO, and OPTLRU across four real-world traces: (a) KV-1, (b) KV-2, (c) CDN, (d) Block.

2) Under the block trace, the hit ratio of Mobius is slightly higher than 2Q (+0.2%). Given that the block trace includes a significant number of scan requests, and since 2Q is designed as a scan-resistant eviction policy, the fact that Mobius outperforms 2Q suggests that Mobius is also effective in resisting scan-related performance issues.

### 5.5 Mobius in RocksDB

Finally, we apply Mobius to RocksDB [49]. RocksDB is a high-performance embedded key-value database widely used in production systems at Meta [49], Yahoo[17], LinkedIn[20], Alluxio[37], and others[6]. RocksDB utilizes a cache to accelerate read performance, known as the block cache, with LRU cache being its default eviction scheme implementation. We replace the LRU cache in RocksDB with Mobius to create a Mobius cache. We benchmark the performance of the modified RocksDB against the original one.

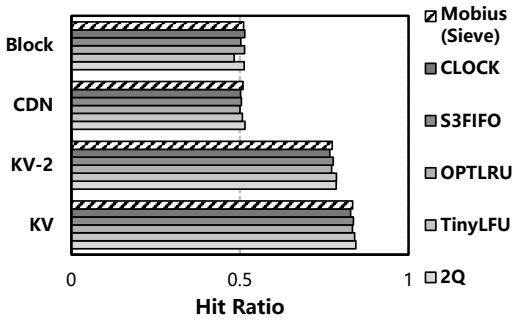



Fig. 12. Cache hit ratio comparison of Mobius, SIEVE, S3FIFO, and OPTLRU across four real-world traces.

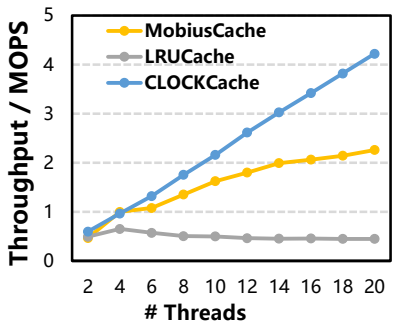



Fig. 13. Throughput comparison of Mobius cache and LRU cache.

*5.5.1 Cache benchmark.* First, we evaluate the throughput of different versions of block cache using *cache_bench*. *Cache_bench* is a benchmark tool embedded in RocksDB's source code for testing the block cache's performance. The block cache in RocksDB supports cache shards for higher scalability, with each cache shard having its own eviction queues. To explore the scalability of the eviction policy, we set the number of cache shards to 1. The cache occupies 1 GB of memory, capable of containing a maximum of 127K cache entries. We vary the thread count from 2 to 20, with each thread issuing 10M read requests to the cache. 95% of the requests are insertions and accesses, while the remaining are other operations such as erasing. All other configurations are kept at defaults in *cache_bench*. We plot the throughput as a function of the number of threads in Figure 13.

As the number of threads increases, the throughput of the LRU cache remains at 0.5 MOPS. The peak throughput of LRU cache is 0.65 MOPS when the thread count is 4. In contrast, Mobius exhibits better scalability. Its throughput increases continuously with the number of threads. When the thread count is 20, the throughput reaches its maximum value, around 2.3 MOPS, which is 3.4× that of the LRU cache. The *cache_bench* test further confirms Mobius's high throughput.

The poor scalability of LRU cache is due to its locking mechanism. As the number of threads increases, data races become more frequent, leading to increased lock contention in LRU cache. Benefiting from the lock-free design, Mobius cache demonstrates better scalability. In fact, the developers of RocksDB have already noticed the poor scalability of LRU cache. In the latest version of RocksDB, the block cache offers an alternative implementation with a variant of CLOCK, known as CLOCK cache, designed for high throughput scenarios.

We also evaluate the performance of the CLOCK cache in Figure 13. The CLOCK cache outperforms Mobius in throughput due to its hash-based design. Specifically, RocksDB implements the CLOCK cache using a lock-free hash table, rather than FIFO or LRU queues. The inherent concurrency advantages of hash tables, compared to lists, contribute to the higher throughput of the CLOCK cache. However, these hash table advantages come at the expense of not preserving the insertion order of cached items. As a result, the CLOCK cache can only evict items based on their hash order, rather than their insertion order, which reduces the cache hit ratio. This limitation explains why the CLOCK cache is not yet the default block cache in RocksDB.

The evaluation with *cache_bench* demonstrates that the Mobius cache has better scalability than the LRU cache. Moreover, unlike the CLOCK cache, Mobius maintains a higher cache hit ratio than the LRU cache in extensive evaluations [79]. The combination of high throughput and cache hit ratios suggests that the Mobius cache has the potential to be the candidate for the block cache in RocksDB.

*5.5.2 System Performance.* We evaluate the overall performance of RocksDB with Mobius cache using *db_bench*, the official benchmark tool embedded in RocksDB. Before running the tests, we pre-create a database with 8 billion objects on our disk. Since the block cache primarily targets reads, we utilize three types of read-only workloads: randomized single-point reads without skewness (*randread* with $read\_random\_exp\_range = 0$), randomized single-point reads with high skewness (*randread* with $r.r.e.r. = 10$), and randomized range reads for 10 adjacent objects (*fwdrange*). We set the number of cache shards to 16 and the thread count to 64. All tests run for 4,000 seconds, with other configurations set to their default values in the embedded script *benchmark.sh*. We compare the performance against the original RocksDB with LRU cache.

The throughput measurement of RocksDB with the Mobius cache and LRU cache over 4,000 seconds is depicted in Figure 14. The results of single-point reads are illustrated in Figure 14a, and those of range reads are shown in Figure 14b. The tail latency comparison is shown in Figure 15.

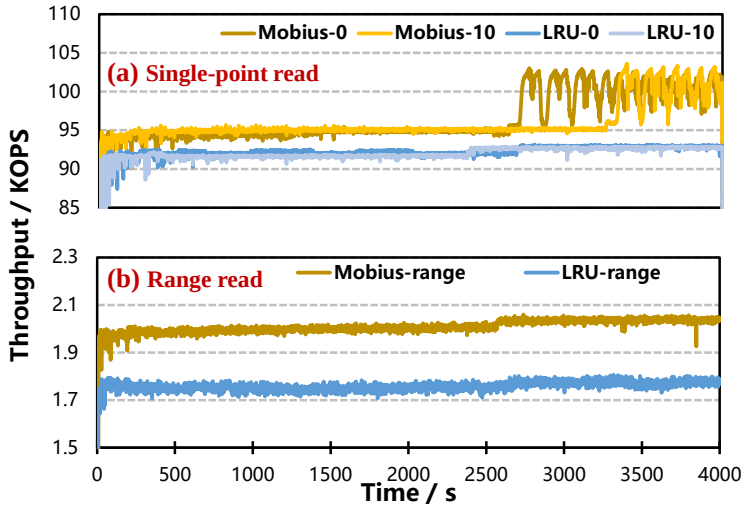



Fig. 14. Throughput comparison between RocksDB with the Mobius cache and the LRU cache in 4,000s. (a) Workloads consist of random single-point reads with no skewness (0) and high skewness (10). (b) Workloads comprise random range reads, with each operation accessing 10 objects.

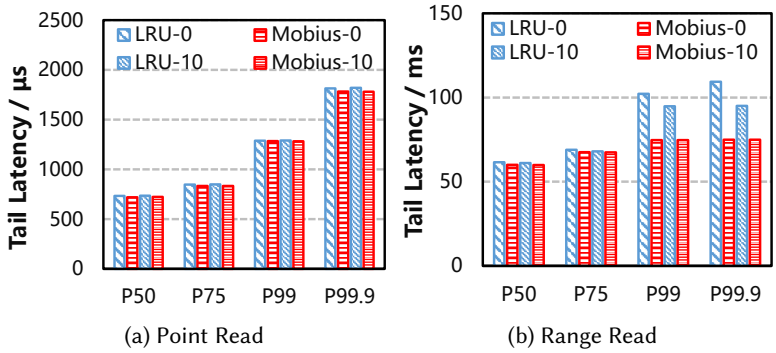



Fig. 15. Tail latency comparison between RocksDB with the Mobius cache and the LRU cache. (a) Point read. (b) Range read. Each scheme is tested using the workloads with no skewness(0) and high skewness(10).

Figure 15a depicts the tail latency of single-point reads while Figure 15b illustrates the tail latency of range reads. From these figures, we can draw three observations.

1) The throughput of RocksDB with the Mobius cache consistently surpasses that with the LRU cache. As depicted in Figure 14a, the throughput of the former during single-point reads ranges from 95 KOPS to 102 KOPS, while that of the latter hovers around 92 KOPS. On average, the former is 4% higher than the latter, demonstrating that Mobius enhances overall system performance.

2) As shown in Figure 15, the tail latency of the Mobius cache are also lower than the LRU cache.

3) The improvement brought by Mobius is particularly noticeable during range reads. As shown in Figure 14b, the throughput of RocksDB with the Mobius cache is around 2 KOPS, while that with the LRU cache is less than 1.8 KOPS. On average, the former is 14% higher than the latter. Figure 15b

also shows that the Mobius cache reduces 20-30% P99 latency compared to the LRU cache. This may be attributed to the fact that range reads often access the same cache item repeatedly. In the block cache, cache items are blocks, which are typically larger than individual objects. Consequently, a single block may contain multiple adjacent objects. A range read operation may access the same block multiple times, resulting in significant contention in LRU cache. However, Mobius, being free from data races during cache access, proves more efficient in range reads.

## 6 Conclusions

This paper focuses on studying the scalable throughput of cache eviction policies. From our preliminary research and evaluations, we identified scalability as a key factor and found that the primary bottleneck is the locking mechanism. To address this, we replace the locking mechanism with a lock-free design, by introducing Mobius, which is built on SIEVE - a recent eviction policy known for its high cache efficiency but limited scalability due to its use of locks [79]. Unlike SIEVE, Mobius uses lock-free data structures with two FIFO queues and improves scalability by selecting eviction victims through consecutive detection. Compared to other cache eviction policies, Mobius achieves a high hit ratio, better concurrent throughput, and lower latency. As demonstrated by its implementation in CacheLib and RocksDB, Mobius shows potential for broad application.

## Acknowledgments

We thank the anonymous reviewers for their valuable feedback and our shepherd for her suggestions. This work was supported by the National Key Research and Development Program of China (No. 2022YFB4501300) and the National Natural Science Foundation of China (No. U22A2027 and 61821003).

## References

[1] 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, page 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[2] 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, Renton, WA, April 2018. USENIX Association.

[3] 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.

[4] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 126–134 vol.1, 1999.

[5] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., USA, 2013.

[6] Cheng Chang. Users of rocksdb and their use cases, 2021. https://github.com/facebook/rocksdb/blob/master/USERS.md Accessed April 12, 2024.

[7] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, San Diego, CA, January 1996. USENIX Association.

[8] Carl J. Conti, Donald H. Gibson, and Stanley H. Pitkowsky. Structural aspects of the system/360 model 85, i: General organization. *IBM Systems Journal*, 7(1):2–14, 1968.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[10] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A PAGING EXPERIMENT WITH THE MULTICS SYSTEM.* Project MAC. Massachusetts Institute of Technology, 1968.

[11] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, page 143–152, New York, NY, USA, 1990. Association for Computing Machinery.

[12] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[14] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 146–153, 2014.

[15] Nada Elgendy and Ahmed Elragal. Big data analytics: A literature review paper. In Petra Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects*, pages 214–227, Cham, 2014. Springer International Publishing.

[16] Marc Eshel, Roger Haskin, Dean Hildebrand, Manoj Naik, Frank Schmuck, and Renu Tewari. Panache: A parallel file system cache for global file access. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. USENIX Association.

[17] Kay Ewbank. Rocksdb on steroids, 2024. https://www.i-programmer.info/news/84-database/8542-rocksdb-on-steroids.html Accessed April 12, 2024.

[18] 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.

[19] 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, Lombard, IL, April 2013. USENIX Association.

[20] Tao Feng. Benchmarking apache samza: 1.2 million messages per second on a single node, 2015. https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node Accessed April 12, 2024.

[21] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, aug 2004.

[22] Akira Goda. Recent progress on 3d nand flash technologies. *Electronics*, 10(24), 2021.

[23] Guidotag. Lock-free clock cache, 2022. https://github.com/facebook/rocksdb/issues/10306 Accessed March 11, 2024.

[24] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for orms. In Fabio Kon and Anne-Marie Kermarrec, editors, *Middleware 2011*, pages 329–349, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[26] Jim Handy. *The cache memory book (2nd ed.): the authoritative reference on cache design.* Academic Press, Inc., USA, 1998.

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

[28] Gerhard Hasslinger, Konstantinos Ntougias, Frank Hasslinger, and Oliver Hohlfeld. Scope and accuracy of analytic and approximate results for fifo, clock-based and lru caching performance. *Future Internet*, 15(3), 2023.

[29] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991.

[30] Intel. Intel® optane™ ssd dc p5800x series, 2024. https://ark.intel.com/content/www/us/en/ark/products/201840/intel-optane-ssd-dc-p5800x-series-3-2tb-2-5in-pcie-x4-3d-xpoint.html Accessed March 11, 2024.

[31] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. Hpc i/o throughput bottleneck analysis with explainable local models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[32] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[33] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA, 2002. Association for Computing Machinery.

[34] 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, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX

Association.

[36] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[37] Haoyuan Li. *Alluxio: A Virtual Distributed File System*. PhD thesis, 2018. Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Updated 2023-03-03.

[38] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, page 356–361, New York, NY, USA, 2014. Association for Computing Machinery.

[39] Kevin Marks. An nvm express tutorial. *Flash Memory Summit*, 2013.

[40] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[41] 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)*, San Francisco, CA, March 2003. USENIX Association.

[42] Meta. The official website of cachelib, 2024. https://cachelib.org Accessed March 11, 2024.

[43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[44] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. Share interface in flash storage for relational and nosql databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 343–354, New York, NY, USA, 2016. Association for Computing Machinery.

[45] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[46] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 128–141, New York, NY, USA, 2022. Association for Computing Machinery.

[47] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[48] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data, and web data only, 2023.

[49] Meta Platform. The official website of rocksdb, 2022. https://rocksdb.org Accessed March 11, 2024.

[50] Meta Platform. Evaluating ssd hardware for facebook workloads, 2024. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval/ Accessed March 12, 2024.

[51] S. Prybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. *SIGARCH Comput. Archit. News*, 16(2):290–298, may 1988.

[52] S. Prybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, page 290–298, Washington, DC, USA, 1988. IEEE Computer Society Press.

[53] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and Andre Brinkmann. Combining buffered I/O and direct I/O in distributed file systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 17–33, Santa Clara, CA, February 2024. USENIX Association.

[54] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 557–573, New York, NY, USA, 2023. Association for Computing Machinery.

[55] Redis. The official website of redis, 2024. https://redis.io Accessed March 12, 2024.

[56] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. *SIGMETRICS Perform. Eval. Rev.*, 18(1):134–142, apr 1990.

[57] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A deep integration of device and application for flash based Key-Value caching. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 391–405, Santa Clara, CA, February 2017. USENIX Association.

[58] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*,

SIGMETRICS '99, page 122–133, New York, NY, USA, 1999. Association for Computing Machinery.
[59] Software. The official website of ehcache, 2024. https://www.ehcache.org/ Accessed March 12, 2024.
[60] SOLIDIGM. Solidigm d7-p5620, 2024. https://www.solidigm.com/products/data-center/d7/p5620.html#configurator Accessed March 11, 2024.
[61] Håkan Sundell and Philippas Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In Teruo Higashino, editor, *Principles of Distributed Systems*, pages 240–255, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
[62] 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, Santa Clara, CA, February 2015. USENIX Association.
[63] Ruixiang Tang, Yu-Neng Chuang, and Xia Hu. The science of detecting llm-generated text. *Commun. ACM*, 67(4):50–59, mar 2024.
[64] Facebook Database Engineering Team. The source code of rocksdb, 2024. https://github.com/facebook/rocksdb Accessed March 11, 2024.
[65] Thesys-lab. The modified source code of cachelib with novel cache eviction algorithms, 2023. https://github.com/Thesys-lab/cachelib-sosp23 Accessed March 11, 2024.
[66] D. Thiebaut, J. Wolf, and H. Stone. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(06):665–676, jun 1992.
[67] Devang Tomar. How cloudflare achieved 55 million requests per second with just 15 postgresql clusters!, 2024. https://dev.to/devangtomar/how-cloudflare-achieved-55-million-requests-per-second-with-just-15-postgresql-clusters-3mm8 Accessed March 11, 2024.
[68] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 214–222, New York, NY, USA, 1995. Association for Computing Machinery.
[69] Demian Vöhringer and Viktor Leis. Write-aware timestamp tracking: Effective and efficient page replacement for modern hardware. *Proc. VLDB Endow.*, 16(11):3323–3334, July 2023.
[70] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. Transactional nvm cache with high performance and crash consistency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
[71] D. Wind. *Instant Effective Caching with Ehcache*. Packt Publishing, 2013.
[72] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. AC-Key: Adaptive caching for LSM-based Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615. USENIX Association, July 2020.
[73] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 70–79, New York, NY, USA, 2023. Association for Computing Machinery.
[74] 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.
[75] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.
[76] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: a learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proc. VLDB Endow.*, 13(12):1976–1989, jul 2020.
[77] Yue Yang and Jianwen Zhu. Write skew and zipf distribution: Evidence and implications. *ACM Trans. Storage*, 12(4), jun 2016.
[78] Guoqiang Zhang, Yang Li, and Tao Lin. Caching in information centric networking: A survey. *Computer Networks*, 57(16):3128–3141, 2013. Information Centric Networking.
[79] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. 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)*, Santa Clara, CA, April 2024. USENIX Association.
[80] Dongfang Zhao and Ioan Raicu. Hycache: A user-level caching middleware for distributed file systems. In *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1997–2006, 2013.
[81] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 91–104, USA, 2001. USENIX Association.

## APPENDIX

## A Pseudo-code of two FIFO queues

**Algorithm 1:** Mobius - two FIFO queues in rotation

```
   /* The modified enqueue function in Mobius.                          */
 1 Function Enqueue(item):
 2     item.next = null;
 3     repeat
 4         oldTail ← tail;
 5     until CAS_tail(oldTail, item) is True;
 6     if oldTail is null then
 7         head ← item;
 8     else
 9         oldTail.next ← item;
10     return;
   /* This function inserts a cache item into Mobius.                   */
11 Function Insert(item):
12     item.isVisited ← 0;
13     Q[whichQ].Enqueue(item);
14     return;
   /* This function evicts a cache item from Mobius.                    */
15 Function Evict():
16     w ← whichQ;
17     Using activeQ represents Q[w];
18     Using dormantQ represents Q[!w];
19     while True do
20         repeat
21             repeat
22                 item ← activeQ.head;
23             until item is not null;
24             if Item is the last element in activeQ then
25                 CAS_whichQ(w, !w);
26                 return Evict();
27         until activeQ.CAS_head(item, item.next) is True;
28         if item.isVisited is 1 then
29             item.isVisited ← 0 ;
30             dormantQ.Enqueue(item);
31         else
32             return item;
```

Algorithm 1 outlines the pseudocode for the *Insert* and *Evict* operations of Mobius. The *Access* operation is not included here because it is straightforward, which simply updates a reference bit.

Mobius consists of three data members: two FIFO queues and an atomic boolean variable indicating the active queue. In the pseudocode, the queues are referred to as *Q[0]* and *Q[1]*, and the variable is called *whichQ*. Each FIFO queue manages its own *head* and *tail*.

First, lines 1-10 depict the modified *Enqueue* operation. We've added a condition to check if the queue is empty. After inserting an item into the queue, Mobius checks if the original *tail* is null, indicating an empty queue (line 6). If so, Mobius sets the *head* to the item (line 7) to ensure subsequent *dequeue*s function properly. If the queue is not empty, Mobius links the item after the original tail (line 9).

Lines 11-14 illustrate the *Insert* operation in Mobius. Mobius initializes the reference bit (line 12) and then enqueues the item into the active queue (line 13).

The *Evict* function is the most complex part of Mobius, as shown in lines 15-32. Mobius first determines the active queue and the dormant queue based on the *whichQ* variable (lines 16-18). Then, Mobius enters a loop to find the eviction victim. Within the loop, Mobius dequeues the oldest cache item from the head of the active queue (lines 20-27). This *dequeue* operation is modified to handle boundary cases. At lines 21-23, Mobius repeats the check for the *tail* of the active queue until a valid candidate is found, avoiding dequeuing from an empty queue. At line 24, Mobius checks if the dequeued candidate is the last element in the active queue. The decision basis is that: If the next item after the candidate is null (*item.next is null*), and the candidate is the *head* of the active queue (*item is activeQ.head*), it is the last element in the active queue. In this case, Mobius switches the active queue by negating the value of *whichQ* (line 25). Then, Mobius calls the *Evict* function again to find the eviction victim from the new active queue (line 26). If the active queue contains more than one element, Mobius continues to dequeue cache items until it succeeds (line 27). Once the modified *dequeue* operation is completed, the remaining process is similar to that of CLOCK. Mobius checks the reference bit of the dequeued item (line 28). If the item has been accessed, Mobius clears its reference bit (line 29), inserts it into the dormant queue (line 30), and continues the loop for the next candidate. If the item has not been accessed, it is deemed the eviction victim (line 32).

The races between *Enqueues* are thread-safe. The critical resource involved in multi-thread *Enqueues* is the *tail*, with its atomic updates ensured by the *CAS_tail* instruction (line 4). Therefore, we only need to consider the remaining parts of Mobius. The critical resources in Mobius are the boolean variable *whichQ*, and the *head* and *tail* of the two queues. Lines 20-27 in function *Evict* is the associated code block other than *Enqueue*, referred to as the *Evict* for brevity. We'll discuss two possible race conditions: between *Enqueue* and *Evict*, and between two *Evict*s.

The first race condition may occur when *Enqueue* and *Evict* operate on the same empty active queue. This scenario aligns with the boundary cases we've discussed. Since *Enqueue* updates the *head* last (line 7), and *Evict* keeps blocking until the *head* is valid (lines 21-23), *Evicts* always occur after the first *Enqueue*, preventing inconsistency. After initialization, the queues are never empty, ensuring thread safety.

The second race condition arises from multiple *Evicts*. *Evicts* (lines 20-27) involve two critical resources: *whichQ* and the *head* of the active queue. Mobius operates on *whichQ* at line 25, switching the active queue by assigning the logical negation of *w* to *whichQ*. The CAS primitive ensures a consistent result in parallel. Even if multiple threads run the code simultaneously, only one thread can update *whichQ*, while other threads will fail because the updated *whichQ* is not equal to *w*. Thus, the update to *whichQ* is thread-safe. Similarly, the *head* is also thread-safe due to the CAS primitives at line 27. The thread safety of Mobius is confirmed.

## B Pseudo-code of consecutive detection

Algorithm 2 presents the pseudo-code for the operations with consecutive detection. In comparison to the previous methods outlined in Algorithm 1, we make adjustments to the functions of *Enqueue* and *Evict* to support range operations. The modified *Enqueue* function is depicted in lines 1-10. It now accepts two parameters, *beginItem* and *endItem*. *beginItem* represents the first item of a sub-list, while *endItem* signifies the end. This function links the sub-list to the tail of a FIFO queue. Notably, if *beginItem* and *endItem* are the same, this *Enqueue* operation merely inserts one item, degenerating to the previous version. Thus, the modification of *Insert* involves only changing the parameter count of *Enqueue* from one to two (line 13).

The *Evict* function (lines 15-39) is more complex, involving a two-level loop. Several issues must be addressed within the *Evict* function. The first issue is how to identify a sub-list from the head of the active queue. This sub-list should consist of consecutive accessed items, with the next cache item after the sub-list being unaccessed. The solution is depicted in the internal loop (lines 21-34) in Algorithm 2.

Within the loop, Mobius first initializes associated variables (lines 23-25). It records the *head* of the active queue in a local variable, referred to as *oldHead* (line 23). *oldHead* represents the head of the original queue and the beginning item of the operated sub-list. The search begins from the *head* (line 24). If the active queue is empty (*item is null*), or it contains only one element (line 26), Mobius updates *whichQ* to switch the active queue (line 27) and restarts the eviction process (line 29). The criterion for identifying the last element is as follows: the next item after the candidate is null (*item.next is null*), and *OldHead* remains the *head* of the active queue (*OldHead is activeQ.head*). When Mobius determines that there's no need to switch the active queue, it checks the reference bit of the current item (line 30). If this item is accessed, Mobius records it in the variable *prev* (line 31), which represents the end item of the operated sub-list. Subsequently, Mobius searches for the next item (line 32). The internal loop continues until the first unaccessed cache item is found (lines 33, 34), which serves as the eviction victim. The sub-list to be enqueued into the dormant queue is then determined, starting from *oldHead* and ending at *prev*.

The second issue in the *Evict* function is how to dequeue the sub-list from the active queue. The solution is the external loop (lines 19, 35). After identifying the sub-list, Mobius utilizes the CAS primitive to update the *head* of the active queue. If *head* differs from *OldHead*, it indicates that the current sub-list has been dequeued by another thread. Mobius re-enters the internal loop (lines 21-34) to find the next sub-list. If the *head* remains the same as *oldHead*, Mobius updates the head to the next item after the eviction victim (line 35) and proceeds to the subsequent step.

The third issue is how to enqueue the sub-list into the dormant queue. The solution is depicted in lines 36-39. Mobius first checks the variable *prev* (line 36). If *prev* is valid, it indicates that the sub-list contains at least one element. Mobius sets the cache items in the sub-list to unaccessed (line 37) and enqueues the sub-list into the dormant queue (line 38). If *prev* is null, it signifies that the sub-list is empty, and no *Enqueue* operation is necessary. Finally, the eviction victim is returned (line 39). It is worth noting that the reference bits are cleared in batches in the new *Evict* function. There are two conditions for clearing the reference bits. The first condition is at line 37. When a thread identifies an eviction victim, it should clear the reference bits in the corresponding sub-list. The second condition is at line 28. When a thread searches for the last element and successfully switches the active queue, it should set the searched items as unaccessed.

There are new race conditions introduced by the consecutive detection mechanism. Assuming two threads invoke the *Evict* function simultaneously, and they are detecting the same sub-list. The first thread (denoted by $T_1$) has dequeued the sub-list from the active queue (after line 35), while the

**Algorithm 2:** Mobius - consecutive detection

```
   /* Modified enqueue function in Mobius                                  */
 1 Function Enqueue(beginItem, endItem):
 2     endItem.next = null;
 3     repeat
 4         oldTail ← tail;
 5     until CAS_tail(oldTail, endItem) is True;
 6     if oldTail is null then
 7         head ← beginItem;
 8     else
 9         oldTail.next ← beginItem;
10     return;
   /* Insert a cache item into Mobius.                                     */
11 Function Insert(item):
12     item.isVisited ← 0;
13     Q[whichQ].Enqueue(item, item);
14     return;
   /* Evict a cache item from Mobius                                       */
15 Function Evict():
16     w ← whichQ;
17     Using activeQ represents Q[w];
18     Using dormantQ represents Q[!w];
19     repeat
20         item ← null;
21         while True do
22             if item is null then
23                 oldHead ← activeQ.head;
24                 item ← oldHead;
25                 prev ← null;
26             if item is null or item is the last element in activeQ then
27                 if CAS_whichQ(w, !w) then
28                     Clear reference bits from oldHead to item;
29                 return Evict();
30             if item.isVisited is 1 then
31                 prev ← item;
32                 item ← item.next;
33             else
34                 break;
35     until activeQ.CAS_head(oldHead, item.next) is True;
36     if prev is not null then
37         Clear reference bits from oldHead to prev;
38         dormantQ.Enqueue(oldHead, prev);
39     return item;
```

second thread (denoted by $T_2$) is still detecting along the sub-list (in lines 21-34). There are three possible conditions.

In the first condition, $T_1$ has cleared the reference bits (after line 37), so $T_2$ finds the current item in the sub-list unaccessed (line 30). In this case, $T_2$ will attempt the CAS primitive (line 33). Since the head of the active queue has been changed by $T_1$, the CAS of $T_2$ will always fail. $T_2$ will re-initialize the detection (line 20) and start from the latest *head* (lines 22-25), ensuring thread safety when CAS fails.

In the second condition, $T_1$ has not updated the reference bits (after line 35 and before line 37), while $T_2$ has arrived at the last item of the sub-list and checked the reference bit (after line 30). Then, $T_1$ runs to line 2 and makes the next item *null*. In this scenario, $T_2$ will fetch a null item (line 32) and continue the loop. The invalid item will trigger re-initialization at lines 22-25, avoiding incorrect results.

The third condition is similar to the second. When $T_1$ has not updated the reference bits (after line 35 and before line 37), $T_2$ has checked the reference bit (after line 30). However, $T_1$ then finishes the *Evict* function and enqueues the sub-list to the dormant queue (line 38). Since the sub-list is linked to the dormant queue, $T_2$ will detect the dormant queue for the eviction victim, which is an undesired behavior. Nevertheless, $T_2$ will eventually find an unaccessed item and attempt the CAS primitive (line 35). Since the *head* has been changed by $T_1$, this CAS primitive will always fail. $T_2$ will re-initialize the detection from the latest head of the active queue. In all three conditions, the *Evict* behavior eventually becomes consistent.

In addition to the possible inconsistencies, the consecutive detection mechanism may lead to another boundary case. After initialization and before the first eviction, new cache items are inserted into $Q_0$, while $Q_1$ remains empty. There is a probable condition where all cache items in $Q_0$ have been accessed. In this case, the first *Evict* behavior cannot find a victim after consecutively detecting $Q_0$. Mobius can only switch the active queue to search in $Q_1$ (lines 27-28), which, however, is still an empty queue. To ensure the *Evict* function correctly runs on an empty active queue, we modify line 26. If the current active queue is empty, Mobius cannot obtain a valid item ($item$ is null). Mobius then switches the active queue to search in another queue (lines 27-28).

Received January 2025; revised April 2025; accepted April 2025