
# Clock2Q+: A Simple and Efficient Replacement Algorithm for Metadata Cache in VMware vSAN

Yiyan Zhai
Carnegie Mellon University
Pittsburgh, United States
yiyanz@andrew.cmu.edu

Bintang Dwi Marthen
Harvard University
Cambridge, United States
mmarthen@g.harvard.edu

Sarath Balivada
Broadcom Inc.
Bangalore, India
sarath.balivada@broadcom.com

Vamsi Sudhakar Bojji
Broadcom Inc.
Bangalore, India
vamsi-sudhakar.bojji@broadcom.com

Eric Knauft
Broadcom Inc.
Palo Alto, United States
eric.knauft@broadcom.com

Jitender Rohilla
Broadcom Inc.
Palo Alto, United States
jitender.rohilla@broadcom.com

Jiaqi Zuo
Broadcom Inc.
Palo Alto, United States
alvin.zuo@broadcom.com

Quanxing Liu
Broadcom Inc.
Palo Alto, United States
quanxing.liu@broadcom.com

Maxime Austruy
Broadcom Inc.
Pully, Switzerland
maxime.austruy@broadcom.com

Junlong Gao
Broadcom Inc.
Palo Alto, United States
junlong.gao@broadcom.com

Wenguang Wang
Broadcom Inc.
Palo Alto, United States
wenguang.wang@broadcom.com

Juncheng Yang
Harvard University
Cambridge, United States
juncheng@g.harvard.edu

## ABSTRACT

Cache replacement is central to data system performance, yet metadata caches exhibit access patterns that differ substantially from data caches. In particular, metadata blocks often exhibit dense bursts of correlated references (e.g., multiple lookups that touch tuples that co-reside on the same metadata page). Treating such bursts as repeated independent hits can misclassify cold metadata as hot, polluting the cache and degrading hit ratio. We present Clock2Q+, a cache eviction algorithm deployed in the metadata cache of VMware vSAN, a production-grade distributed storage system. Clock2Q+ is the latest stage of nearly a decade of iterative evolution of vSAN's caching layer, shaped by real-world deployment, customer-driven feature requests, and an emphasis on simplicity as a first-class requirement for long-term maintainability. Clock2Q+ has a three-queue structure similar to S3-FIFO, and introduces a correlation window at the head of the Small FIFO in which accesses do not set the reference bit, preventing short-term correlated bursts from triggering premature promotion; accesses beyond this window set the bit so genuinely hot blocks are promoted efficiently. Clock2Q+ achieves the highest median and mean miss-ratio improvement among the state-of-the-art policies we evaluate on both metadata and data workloads, with the largest per-trace gap over S3-FIFO reaching a 28.5% relative miss-ratio reduction on metadata traces. To support production deployment, Clock2Q+ is engineered for low per-hit overhead and multi-core scalability, with allocation-free, array-based structures, fine-grained synchronization, dirty-page-aware eviction and promotion, bounded per-eviction work, and online cache resizing.

**PVLDB Reference Format:**

Yiyan Zhai, Bintang Dwi Marthen, Sarath Balivada, Vamsi Sudhakar Bojji, Eric Knauft, Jitender Rohilla, Jiaqi Zuo, Quanxing Liu, Maxime Austruy, Junlong Gao, Wenguang Wang, and Juncheng Yang. Clock2Q+: A Simple and Efficient Replacement Algorithm for Metadata Cache in VMware vSAN. PVLDB, 19(12): 3874 - 3887, 2026.
doi:10.14778/3827998.3828002

**PVLDB Artifact Availability:**

The source code, data, and/or other artifacts have been made available at https://github.com/1a1a11a/libCacheSim.

## 1 INTRODUCTION

Storage systems span multiple performance tiers, from CPU caches and DRAM to persistent media such as SSDs and HDDs. Caches bridge these tiers by keeping a working set in fast memory and avoiding device I/O. This paper focuses on *block caches* (also called page caches, buffer caches, or buffer pools): memory-resident caches that manage fixed-size blocks (e.g., 4KB pages). In a block cache, the fast tier is DRAM and the slow tier is one or more storage devices.

**Operational context.** This work is implemented in a deployed storage system, Broadcom's VMware vSAN. vSAN maintains separate caches for data and metadata. It includes a small data cache at the Client layer and multiple large metadata caches at its distributed metadata and local file system layers [49, 50]. In this setting, the


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828002

metadata cache is on a particularly hot path: metadata lookups mediate common operations that locate, validate, and coordinate access to data. As a result, the eviction policy runs at very high frequency, under high concurrency, and often in regimes where hit ratios are already high. Small per-hit overheads, scalability bottlenecks, or rare corner-case behaviors can therefore translate directly into throughput loss, tail-latency regressions, and operational instability. For a deployed system, an eviction policy must be judged not only by miss ratio, but also by whether it can be implemented and operated safely: predictable CPU cost, robust multi-threaded behavior, and compatibility with practical constraints such as dirty blocks and online resizing.

A central challenge is the design of the *cache eviction algorithm*, which decides which blocks remain in the cache and which are evicted when space is needed. An effective policy for production storage systems should aim to achieve:

- *High hit ratio* across diverse workloads, avoiding pathological behaviors that sharply degrade hit ratio.
- *Low CPU overhead* on cache hits, so the cache does not become the bottleneck when hit ratios are high (often close to 100%).
- *Scalability to multiple CPUs*, enabling high throughput on modern multi-core systems.
- *Minimal tuning*, with few configuration parameters and robust performance over a wide range of workloads.
- *Ease of implementation and debugging*, which is critical for reliability and maintainability in production deployments.

Block caches typically store heterogeneous content that strongly influences access patterns: (1) *Data cache*: stores user payload (e.g., file or block data); and (2) *Metadata cache*: stores system metadata used to locate and manage data (e.g., $B^+$ tree blocks).

Many systems separate these caches. For example, Linux uses a buffer cache for block-device metadata and a page cache for file data. [8]. vSAN, a distributed block storage system, similarly includes both metadata and data caches: it features a small data cache at the Client layer and multiple large metadata caches at its distributed metadata and local file system layers [49, 50]. In such systems, metadata caches are often on an even hotter path than data caches: metadata is consulted to find data, enforce consistency, and coordinate distributed operations. This makes metadata eviction policy especially consequential.

This paper identifies a key metadata-cache access pattern that complicates eviction: *correlated references*. In metadata caches, it is common for a block to be referenced many times within a short interval and then remain unused for a long period. Such bursts often arise when a single higher-level operation (e.g., traversing or updating an index) touches multiple items that reside in the same metadata page. From the standpoint of eviction, a burst of correlated references conveys limited evidence of long-term popularity: once the operation completes, the block may not be accessed again for a long time. An eviction policy that treats each access in the burst as an independent "vote" for hotness can misclassify a cold metadata block as hot, retain it far beyond the burst, and displace truly hot blocks—reducing overall hit ratio. This behavior contrasts with many data-cache workloads, where cold blocks are often one-hit wonders and are naturally filtered by policies designed to suppress single-reference items.

Most existing policies do not explicitly account for correlated references. Consider S3-FIFO [60], which uses a Small FIFO to filter unpopular blocks: blocks that receive no hits while in the Small FIFO are evicted early. Under correlated references, however, a cold metadata block can receive multiple hits while still near the head of the Small FIFO, be misclassified as hot, and be promoted to the Main Clock—even though it may receive no further requests after the correlation window ends. In a production metadata cache, this systematic misclassification is particularly harmful: it accumulates across many operations, increases cache pollution, and can create sustained miss-ratio regressions.

In this paper, we present *Clock2Q+*, a cache eviction algorithm designed for metadata caches and implemented in vSAN. Clock2Q+ combines key ideas from Clock2Q (§3.2, previous design in vSAN) and S3-FIFO [60] while addressing their weaknesses for metadata workloads. The core change is simple: Clock2Q+ conceptually partitions the Small FIFO into two "virtual" segments and introduces a *correlation window* at the head of the queue in which hits *do not* set the reference bit. This prevents short-lived correlated bursts from triggering premature promotion. Once a block moves beyond this window, subsequent hits set the reference bit as usual, allowing genuinely hot blocks to be promoted efficiently. Compared to Clock2Q, this design reduces misses when hot blocks first enter the cache; compared to S3-FIFO, it reduces misses caused by correlated references. Across 106 production traces from CloudPhysics, Clock2Q+ achieves the highest median and mean miss-ratio improvement among the evaluated state-of-the-art policies on both metadata and data workloads, with the largest per-trace gap over S3-FIFO reaching a 28.5% relative miss-ratio reduction on metadata.

Because Clock2Q+ is deployed, the design and implementation also incorporate production constraints that materially shape the system. Clock2Q+ targets low per-hit overhead and multi-core scalability, avoids complexity that impedes debugging, and supports operational requirements including safe concurrent access to cache structures, predictable bounded work per eviction (to avoid CPU spikes), practical handling of dirty blocks, and online cache resizing. These considerations are essential for reliability in a real data system and are treated as first-class goals alongside miss ratio.

This paper makes the following contributions:

- Identifies correlated references as a common and performance-critical access pattern in metadata caches, and explains why this pattern defeats common "one-hit-wonder filtering" heuristics.
- Proposes a simple method to convert data traces into metadata traces, enabling broader evaluation of metadata-cache policies when native metadata traces are unavailable.
- Describes Clock2Q+, a scalable, low-overhead cache eviction algorithm implemented in vSAN, including key production considerations such as concurrency, resizing, and dirty-block handling.
- Shares design lessons from nearly a decade of evolving vSAN's cache layer, including why simplicity is a first-class production requirement.

## 2 METADATA CACHES AND CORRELATED REFERENCES

This section explains why metadata caches behave differently from data caches in deployed storage systems, and introduces *correlated*

*references*—a prevalent access pattern that can mislead common eviction heuristics. We also describe a practical method to derive metadata traces from widely available data traces, enabling trace-driven evaluation when native metadata traces are unavailable.

## 2.1 Data Cache vs. Metadata Cache

A *data cache* is the most common form of block cache: it stores the actual payload blocks served to the upper layer (e.g., file data or virtual disk blocks). Most existing cache designs and eviction algorithms implicitly target this setting, where cold data is frequently accessed once ("one-hit wonders"), and temporal locality is primarily driven by repeated requests to the same payload blocks.

In addition to data caches, many deployed systems maintain a *metadata cache* (often called an *index cache*) that stores blocks belonging to internal metadata structures used to locate, validate, and manage data. A common example is an ordered key-value structure (e.g., $B^+$ tree) used to map a Logical Block Number (LBN) to a Physical Block Number (PBN) or to map logical extents to physical extents. This design appears across many production systems. For example, Linux XFS and Btrfs use $B^+$ tree-like structures [37, 38, 51]; Unix FFS and Linux ext2/ext3 use direct/indirect block maps similar to page tables; and Linux ext4 uses an extent tree [35]. Independent of the exact structure, these designs share a key property: *metadata is organized into fixed-size blocks/pages*, and each block aggregates many smaller entities (e.g., key-value tuples, extent descriptors, bitmap bits, or inodes). Since the full metadata structure is often much larger than DRAM, the metadata cache typically holds only a small fraction of metadata blocks, making eviction decisions critical.

Operationally, metadata caching is often more performance-sensitive than data caching. Metadata blocks are touched as part of many common operations (lookups, writes, allocation, recovery), and metadata accesses are generated internally by the system rather than directly requested by applications. This makes metadata traces harder to obtain and makes metadata eviction policy errors more likely to persist unnoticed until they manifest as higher miss ratios, higher device I/O, or tail-latency regressions.

## 2.2 Correlated References

Storage stacks frequently consist of multiple layers. Consider a two-layer setting in which an upper file system (e.g., ext4) sits on top of a lower storage system that implements its own metadata structures. Within the lower system, there may be both a data cache and a metadata cache. The data cache in the lower layer often exhibits weaker temporal locality because the upper layer already filters repeated payload accesses via its own cache [20, 55, 56]. However, the metadata cache can still experience frequent short-term re-references to the same metadata blocks, even when the corresponding data accesses show little or no re-reference.

We call this phenomenon *correlated references*: a metadata block receives multiple references within a short interval because multiple distinct higher-level accesses happen to touch entities that co-reside in the same metadata block. Importantly, such bursts do *not* necessarily indicate long-term popularity. After the burst ends, the metadata block may not be referenced again for a long time, and treating the burst as repeated independent evidence of hotness can lead to cache pollution.

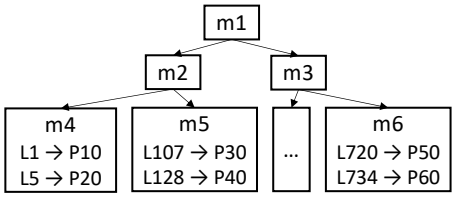



**Figure 1: Example $B^+$ tree for mapping LBN to PBN. Lookup LBN 5 will read metadata blocks $m1$, $m2$, $m4$ and find that LBN 5 is mapped to PBN 20.**

We illustrate correlated references using a $B^+$ tree that maps LBN to PBN. Figure 1 shows metadata blocks with addresses $m1$, $m2$, etc., and data blocks with LBN $L1$, $L2$, etc.

Suppose a data trace accesses four LBNs in the following order: 1, 5, 107, and 720. The corresponding metadata accesses (assuming the relevant pages are not already cached) are: $m1, m2, m4$ for LBN 1; $m1, m2, m4$ for LBN 5; $m1, m2, m5$ for LBN 107; and $m1, m3, m6$ for LBN 720. Thus the metadata block sequence is: $m1, m2, m4, m1, m2, m4, m1, m2, m5, m1, m3, m6$.

The internal (non-leaf) blocks ($m1$, $m2$, $m3$) are accessed frequently across many lookups. Any reasonable eviction algorithm will retain these blocks, and in many implementations, they are effectively pinned or have extremely high reuse. Therefore, when analyzing metadata-cache behavior relevant to eviction decisions, we focus on leaf blocks, which typically account for 99% or more of $B^+$ tree blocks. Excluding non-leaf blocks, the leaf access sequence becomes: $m4, m4, m5, m6$. Even though the upper-layer accesses are all distinct, the leaf page $m4$ appears multiple times because LBN 1 and LBN 5 fall into the same leaf—a correlated reference burst driven by structural locality, not by long-term hotness.

While we use $B^+$ tree as a concrete example, correlated references arise broadly across metadata designs. Metadata blocks aggregate many smaller entities (tuples, extent records, bitmap entries, inodes), so accesses to distinct entities often map to the same metadata block. As a result, even if the upper layer visits each entity only once, the underlying metadata block can receive multiple references in a short window. We therefore argue that correlated references are an inherent trait of metadata caching in many deployed systems.

## 2.3 Metadata Cache in vSAN

VMware vSAN provides a primary example where the metadata cache exhibits correlated references, as it uses multiple $B^+$ tree in its metadata path. vSAN is a block storage system that manages virtual disks for Virtual Machines (VMs) running on the ESXi hypervisor. Each virtual disk is addressed by Logical Block Addresses (LBAs) with a 4 KiB block size. In the top layer, vSAN uses a metadata cache to translate LBAs to Physical Block Addresses (PBAs) [53, 54]. Specifically, an LBA is first translated to a Middle Block Address (MBA) using a Logical Tree, and subsequently, the MBA is translated to a PBA using a Middle Tree. The architectural decision to utilize two distinct trees, rather than a direct LBA-to-PBA mapping, is driven by the need for storage virtualization and indirection:

- *Log-structured updates.* vSAN uses a log-structured file system where data is frequently relocated for garbage collection. The

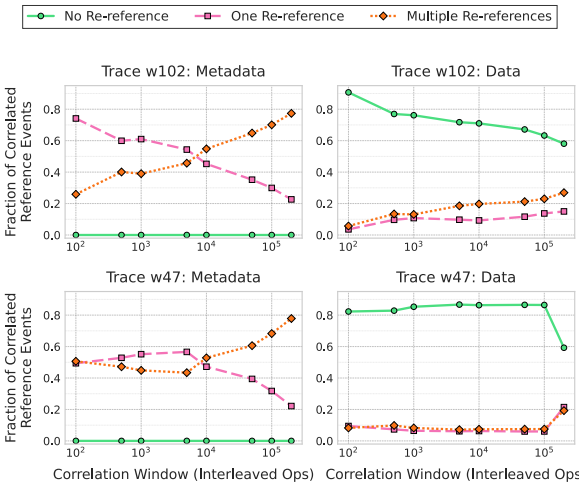



**Figure 2: Re-reference event distribution for CloudPhysics $w102$ and $w47$ across varying window sizes. While data traces are dominated by one-hit wonders, metadata traces exhibit nearly zero single-access events. Instead, metadata blocks show dense, multiple re-references, reflecting the structural locality inherent in the vSAN two-tree lookup process.**

MBA acts as a stable identifier; when data moves physically, only the MBA-to-PBA entry in the Middle Tree requires an update, leaving the higher-level LBA-to-MBA mappings in the Logical Tree untouched.

- *Snapshot support.* Copy-on-Write $B^+$ tree in the Logical Tree enables high-performance snapshots. Multiple Logical Trees can reference the same MBAs, allowing instantaneous snapshot operations without redundant metadata overhead.

These two B+ trees dominate the metadata cache in this layer. Because each leaf page contains hundreds of mapping tuples, accesses to different LBAs can repeatedly hit the same leaf pages, producing high short-term re-reference counts that are not reflected in the LBA stream itself. Policies that do not account for these correlated references can therefore over-promote these leaf pages and reduce overall cache effectiveness.

We characterized the $B^+$ tree access pattern in vSAN by replaying production block traces in vSAN [1]. We used two representative data traces from the CloudPhysics dataset [47], which comprises high-fidelity VM block request traces. By replaying these LBA-level requests, we can observe the resulting metadata traversals through the Logical and Middle trees. Figure 2 reports the distribution of re-reference events for these workloads observed at the data and metadata layers under varying correlation window sizes. We observe data traces show a strong one-hit-wonder component (often exceeding 80% in small windows), whereas metadata traces exhibit effectively zero single-access events; every accessed metadata block is re-referenced at least once. As the correlation window increases, multiple re-references dominate, reflecting structural locality during metadata traversal (e.g., repeated hits to the same

[1]Because vSAN is deployed in private customer data centers, we are not able to directly collect traces or telemetry due to privacy reasons.

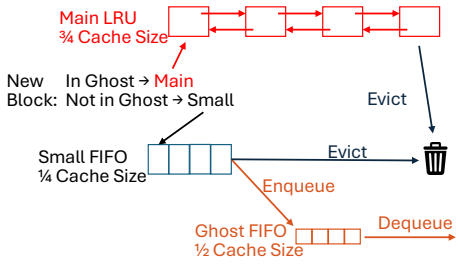



**Figure 3: Illustration of 2Q eviction algorithm.**

leaf page as different tuples are accessed). This result motivates a different design goal for metadata caches: rather than primarily filtering one-hit wonders, a metadata-cache policy must tolerate dense short-term re-references without incorrectly treating them as evidence of long-term hotness.

## 2.4 Deriving Metadata Traces

Trace-driven simulation is a standard approach for evaluating cache eviction algorithms, but evaluating *metadata* caches faces a practical obstacle: *most publicly available traces are data traces.* Data traces are straightforward to capture because they record external I/O requests entering a storage system. Metadata traces, in contrast, require instrumenting internal metadata accesses within the storage stack. This instrumentation is intrusive, and the resulting traces are typically much larger because a single data request can trigger multiple metadata reads (e.g., several page touches during an index traversal).

To expand evaluation coverage, we derive an approximate metadata trace from any data trace using a simple mapping. Using Figure 1 as an example, assume each $B^+$ tree leaf page stores $FanOut$ LBN-to-PBN tuples (e.g., $FanOut = 100$). Then LBNs 0-to $FanOut-1$ reside in leaf page 0, LBNs $FanOut$ to $2 \times FanOut - 1$ reside in leaf page 1, and so on. We convert each LBN request into a leaf-page identifier: $\text{LeafID} = \left\lfloor \frac{\text{LBN}}{FanOut} \right\rfloor$ For instance, the data requests 1, 5, 107, and 720 map to the metadata sequence 0, 0, 1, and 7. This mapping captures the essential source of correlated references: different LBNs can fall into the same leaf page, producing short-term re-references even when the data stream itself contains no repeats.

Our derivation is intentionally lightweight. It requires only an estimate of $FanOut$, obtainable from the metadata page size and tuple format (or chosen conservatively to reflect typical leaf capacities). While it does not model all implementation details (e.g., splits/merges, key skew, or caching of internal nodes), it is sufficient to exercise eviction policies under the bursty re-reference behavior that dominates metadata leaf access patterns. In our evaluation, we use these derived metadata traces to complement native metadata traces when available, enabling systematic comparison across a broader set of workloads (we empirically validate the high fidelity of this derivation method in subsection 5.2).

# 3 CLOCK2Q+ DESIGN

## 3.1 2Q

2Q [26] divides the cache into three logical components: a *Main LRU* occupying 75% of the cache, a *Small FIFO* occupying 25%

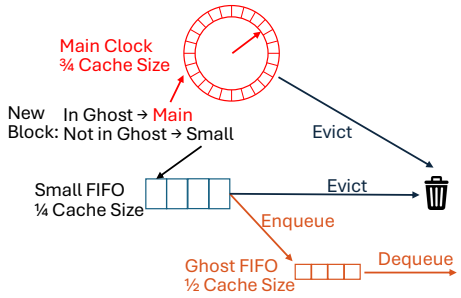



**Figure 4: Illustration of Clock2Q eviction algorithm.**

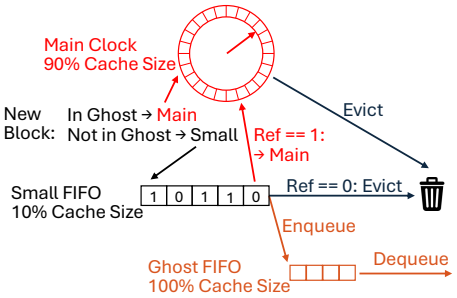



**Figure 5: Illustration of S3-FIFO eviction algorithm.**

of the cache, and a *Ghost FIFO* that stores metadata-only entries (block identifiers) with capacity $\frac{1}{2}$ of the cache (Figure 3). The ghost structure consumes little memory because it does not store data contents.

On a miss, if the requested block identifier is not in the Ghost, the block is inserted at the head of the Small FIFO. If the Small FIFO is full, its tail block is evicted and its identifier is appended to the Ghost; if the Ghost is also full, its tail identifier is dropped.

If a miss finds the block identifier in the Ghost FIFO, 2Q interprets this as evidence that the block was prematurely evicted from the Small FIFO and is therefore likely hot. The block is admitted directly into the Main LRU (and may trigger an LRU eviction from the tail). On a hit, blocks in the Main LRU are promoted in standard LRU fashion; hits in the Small FIFO do not change position.

Conceptually, the Small FIFO acts as an admission filter: correlated or short-lived bursts tend to occur while blocks remain in the Small FIFO and thus do not immediately displace long-term hot blocks in the Main LRU. However, 2Q has three practical drawbacks in modern deployed systems:

- *High LRU overhead.* Maintaining a fully ordered LRU list imposes non-trivial CPU and memory overhead and complicates scalable concurrent implementations.
- *Oversized Small FIFO.* Allocating 25% of the cache to the Small FIFO was reasonable when caches were small. In today's systems, multi-GB caches make this fraction disproportionately large, often far exceeding what is needed to absorb short correlation bursts. A smaller admission segment can increase effective main capacity and reduce misses.
- *Unnecessary misses for hot blocks.* A block can enter the main segment only after first being evicted from Small FIFO into Ghost FIFO and then being referenced again. This forces an extra miss for blocks that are actually hot, which becomes noticeable when overall miss ratios are low.

## 3.2 Clock2Q

Clock2Q is the eviction policy previously used in vSAN. As shown in Figure 4, Clock2Q addresses the first limitation of 2Q by replacing the Main LRU with a *Main Clock* [49]. This reduces per-access overhead by approximating LRU using a circular list and a reference bit, and it enables a more scalable implementation. Clock2Q has been used in vSAN OSA and earlier versions of VDFS and vSAN ESA [49, 50]. However, Clock2Q retains the other two limitations of 2Q: the Small FIFO remains relatively large, and hot blocks still suffer an unnecessary miss because promotion to the main segment relies on a ghost-mediated re-reference.

## 3.3 S3-FIFO

S3-FIFO [60] is a recent state-of-the-art policy that is structurally similar to Clock2Q but modifies admission and sizing (Figure 5). It reduces the Small FIFO from 25% to 10% of the cache, increases the Main Clock from 75% to 90%, and enlarges the Ghost FIFO from 50% to 100% of the cache. Most importantly, S3-FIFO introduces a *Ref bit* for entries in the Small FIFO.

The Ref bit addresses the extra miss incurred by 2Q and Clock2Q. When a block in the Small FIFO is referenced again, S3-FIFO sets its Ref bit. Later, when that block reaches the tail of the Small FIFO and is selected for eviction, the policy interprets Ref=1 as evidence of popularity and *promotes the block directly into the Main Clock* rather than evicting it to the Ghost FIFO. The Ghost FIFO still serves its original role: remembering recently evicted blocks that might be re-referenced later.

S3-FIFO thus resolves the three issues described for 2Q. However, for metadata caches it introduces a new failure mode: it can *over-promote* blocks that experience correlated references. A cold metadata block may be referenced multiple times within a short correlation burst while still near the head of the Small FIFO; this sets Ref=1, causing the block to be promoted into the Main Clock despite receiving no further accesses after the burst ends. This behavior is often benign for data caches, where correlated references are uncommon, but it is problematic for metadata caches where correlated references are frequent.

## 3.4 Clock2Q+

Clock2Q+ builds on the Clock2Q/S3-FIFO structure while explicitly accounting for correlated references. As shown in Figure 6, the key change relative to S3-FIFO is the introduction of a *Correlation Window* within the Small FIFO. We set the Correlation Window to 50% of the Small FIFO, i.e., 5% of the total cache size under the default 10% Small FIFO allocation.

When a block in the Small FIFO is hit, Clock2Q+ checks the block's position relative to the head of the Small FIFO:

- If the block is still within the Correlation Window (i.e., among the most recently inserted entries), the hit *does not* set the Ref bit.

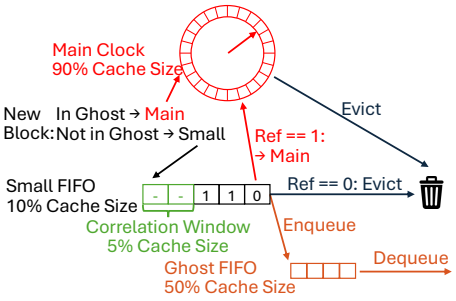



**Figure 6: Illustration of Clock2Q+ eviction algorithm.**

- Once the block has advanced beyond the Correlation Window, a subsequent hit *does* set the Ref bit, ensuring that genuinely hot blocks can still be promoted directly to the Main Clock when they reach the tail of the Small FIFO.

This gating preserves S3-FIFO's direct-promotion path for genuinely hot blocks while filtering out correlated-burst promotions.

Finally, Clock2Q+ reduces the Ghost FIFO size to 50% of the cache. Our empirical results show that this smaller ghost allocation matches the miss-ratio benefits of a 100% ghost in our workloads while reducing metadata overhead.

## 4 MAKING CLOCK2Q+ PRODUCTION READY

Before Clock2Q+, Clock2Q served as the core cache eviction algorithm used in vSAN and vSAN File Services for several years. The improved Clock2Q+ algorithm has been implemented and will be included in future releases. This section discusses the engineering efforts required to make the cache eviction algorithm production-ready.

### 4.1 Implementation

To reduce memory fragmentation and keep the hot path predictable, Clock2Q+ performs no dynamic memory allocation after cache initialization. The Main Clock, Small FIFO, and Ghost FIFO are each represented as arrays carved from contiguous memory regions. The Main Clock and Small FIFO allocate arrays of cache entries (which store state and point to cached content), while the Ghost FIFO allocates an array of keys only. This layout improves cache locality, avoids allocator overheads in the hit path, and eliminates fragmentation caused by frequent insertions and evictions.

Lookup is implemented with chained hash tables for both resident entries (Main Clock and Small FIFO) and ghost entries. We allocate an array of bucket heads (pointers) for the resident hash table and a separate bucket array for the ghost table. Each cache entry contains a next pointer for collision chaining. In our configuration, the load factor is kept below one, so collision chains remain short, and the per-operation work stays bounded.

*4.1.1 Concurrency.* Caches are accessed concurrently by many threads, so thread safety must be efficient. Clock2Q+ uses simple data structures—arrays, FIFO indices, and chained hash tables—which enable fine-grained synchronization with modest overhead.

**Hash table synchronization.** Each hash bucket has a mutex. The lock is held only while traversing or updating that bucket's collision chain. With a low load factor ($< 1$), the expected chain length is small, and our internal tests have not shown measurable contention on these locks.

**Entry-level synchronization.** Cache entries in the Main Clock and the Small FIFO each have a mutex. The lock is held briefly for in-memory state transitions (e.g., updating status bits). When a miss triggers I/O to fill an entry, the entry is marked as `Doing I/O` and inserted into the hash table, and then the per-entry lock is released for the duration of the I/O. A concurrent request that finds an entry in the `Doing I/O` state waits on the entry and resumes once the I/O completes. This prevents duplicate fills and keeps locks out of the long-latency path.

**Deadlock avoidance via a single lock order.** Operations involve both a hash-bucket lock and an entry lock. To avoid deadlocks, Clock2Q+ enforces a single global order: *entry lock first.* This is natural for insert/evict operations (lock entry, update status, then lock bucket to insert/remove). Lookups, however, must first consult the hash table. Clock2Q+ resolves this by releasing the bucket lock before taking the entry lock and validating that the entry still matches the requested key; otherwise it retries. Figure 7 shows the lookup flow.

```
1  CacheGetEntry(key)
2  {
3  retry:
4      LockHashBucket();
5      entry = SearchHash(key);
6      UnlockHashBucket();
7      LockEntry(entry);
8      if (entry.key != key) { // lost race
9          UnlockEntry(entry);
10         goto retry;
11     }
12     work on the entry;
13     UnlockEntry(entry);
14     ...
15 }
```

**Figure 7: Lock ordering for cache lookup. The bucket lock is released before taking the entry lock; a key check detects races and triggers a retry.**

**Concurrent queue/clock traversal.** The Small FIFO head/tail index, the Ghost FIFO head/tail index, and the Main Clock hand are implemented as atomic indices. Since the FIFO structures are kept full, a single index is sufficient to represent both head and tail positions. Multiple threads can advance these indices atomically to find eviction candidates or to append new entries without requiring coarse-grained locks.

*4.1.2 Debugging.* The retry-on-race behavior in Figure 7 is rarely exercised in production, but correctness depends on it. To test these low-probability interleavings deterministically, we built a *race enforcement framework* that can pause a thread at named breakpoints and resume it under test control. Our unit tests pause a thread between the hash lookup and the entry lock, allow a second thread

to evict and reuse the entry, and then verify that the first thread detects the key mismatch and retries correctly. This approach provides repeatable coverage of otherwise elusive races.

*4.1.3 Dirty Pages.* Deployed systems must handle dirty pages, which are typically constrained by write-ahead logging (WAL): before relevant log records are persisted, the corresponding dirty pages cannot be written back [22]. Even when flushing is permitted, evicting a dirty page is substantially more expensive than evicting a clean page because it requires I/O. For these reasons, the Clock2Q+ implementation in vSAN and vSAN File Services *skips dirty pages when selecting eviction candidates.* Operationally this is equivalent to deferring eviction (i.e., treating the page as temporarily ineligible) and continuing the scan.

Dirty pages also complicate promotion from the Small FIFO to the Main Clock. In the ideal algorithm, a Small FIFO page with Ref=1 at the tail should be promoted. Our implementation leaves a dirty page in the Small FIFO even if it would otherwise be promoted. This choice is driven by production constraints: the Main Clock and Small FIFO are array-based, so moving an entry between them requires copying state rather than pointer manipulation. In addition, the transaction subsystem tracks dirty pages in a dirty list for flushing; copying dirty pages would require additional coordination and locking around this list. The page remains in the Small FIFO until it is revisited in a later cycle and becomes clean. This means that if the page referenced by the head pointer of the Small FIFO, which is the tail page, is dirty, the head pointer increments to the next entry in the Small FIFO array without attempting to evict or promote this dirty page. Section 5.5.1 shows that this simplification has negligible impact on miss ratio.

Finally, workloads can transiently produce many dirty pages. In the worst case, every entry in the Small FIFO can be dirty. A naive implementation that scans the entire Small FIFO to find a clean victim can loop indefinitely. Clock2Q+ avoids this failure mode by scanning only a fixed number of entries in the Small FIFO; if all scanned entries are dirty, it falls back to the Main Clock to find eviction candidates.

We implement two complementary flushing mechanisms:

(1) **Time-based flushing:** dirty pages older than a configured threshold are flushed.
(2) **Proportion-based flushing:** the cache maintains low/high watermarks for the fraction of dirty pages. When the fraction exceeds the high watermark, pages are flushed (oldest to newest) until the fraction drops below the low watermark.

## 4.2 Live Cache Resizing

A system like vSAN must support diverse deployments and working-set sizes; operators, therefore, require the ability to grow or shrink cache capacity online. Cache resizing must preserve correctness and performance while avoiding disruptive reallocations. Our resizing mechanism is designed to satisfy:

- *Live operation:* cache hits/misses continue during resizing.
- *Low overhead:* resizing work should not measurably degrade steady-state cache performance.
- *Minimal data movement:* most cached content remains in place.
- *Fine granularity:* capacity changes occur in small increments.
- *No fragmentation:* repeated resizing does not fragment memory.

*4.2.1 Cache Growth.* Growing the cache requires enlarging the Main Clock, Small FIFO, Ghost FIFO, and their associated hash-bucket arrays. During initialization we reserve unused virtual address space after each array, allowing growth without relocating the existing arrays. Cache growth therefore proceeds by extending each array into its reserved region, initializing the new entries, and updating cache capacity and hash sizing.

A background thread then progressively migrates entries to the new hash layout. During migration, an entry may temporarily reside in the "old" location while lookups are in flight. Searching both old and new hash locations on every lookup is complex because lookups do not lock entries while traversing the hash table, and the background thread can move entries concurrently. We therefore shift complexity from lookups to insertions: lookups search only the new hash location. If this produces a false negative because an entry has not yet been migrated, the insertion path detects the old placement, rehashes and moves the entry under locks, and then the retry logic in §4.1.1 causes the lookup to succeed on retry. This keeps the hot lookup path simple and confines migration complexity to code paths that already take locks.

*4.2.2 Cache Shrinking.* Shrinking mirrors growth. The background thread rehashes entries into the smaller bucket arrays and then releases the excess memory at the end of the bucket arrays. Shrinking the Ghost FIFO is straightforward because it stores keys only and entries at the end can be dropped.

Shrinking the Main Clock and Small FIFO requires care. Clean pages at the end can be discarded, but dirty pages cannot be dropped and are difficult to copy for the reasons described in §4.1.3. When the background thread encounters a dirty page in the region to be removed, it triggers a flush and retries later, allowing shrinking to proceed once the page becomes clean.

## 4.3 Lessons Learned

**A shared cache substrate across products.** A cache eviction algorithm is critical to the storage data path, and many VMware products depend on an effective one. Clock2Q+ evolved from the simpler Clock2Q design over multiple years, driven by product needs and deployment requirements. It is now consolidated into a shared codebase reused across multiple products.

**Kernel constraints: near-100% hit ratio, scan resistance, and debuggability.** In 2015, a new vSAN OSA feature needed a cache eviction policy for a core kernel data path. The workloads were unusual but operationally common for storage: the cache hit ratio was close to 100%, so per-hit CPU overhead was the dominant concern. The code executed as a state machine on a single core, so locking and concurrency were not required. At the same time, kernel-mode debugging is notoriously difficult, so simplicity of implementation was a primary selection criterion. Finally, the algorithm had to be scan-resistant because certain workloads periodically scan the entire address space.

**Why Clock2Q: replacing LRU in 2Q with Clock.** Although several scan-resistant Clock-family algorithms exist, including Clock-Pro [24] and CAR/CART [2], they were too complex to be attractive in this kernel setting. 2Q is among the simplest scan-resistant policies, but it relies on LRU for its main segment, which carries higher CPU overhead. This context made the Clock2Q design decision

straightforward: replace the Main LRU in 2Q with a Main Clock, leveraging the long-known observation that Clock approximates LRU behavior at lower cost. Nearly a decade later, the S3-FIFO results showed that Clock can outperform LRU in many cases, which was an unexpected but welcome confirmation: Clock2Q can plausibly deliver both a lower miss ratio and lower CPU/memory overhead than 2Q.

**Array-friendly primitives reduce complexity.** Clock also simplified implementation relative to Clock-Pro and CAR/CART: it does not require dynamically adjusting the Clock size or inserting/deleting elements in the middle of the structure. That property enables an array-based implementation for the Main Clock, Small FIFO, and Ghost FIFO rather than linked lists, which reduces memory overhead and further simplifies the system.

**Reusing the design in VDFS via a templated implementation.** Around the same period, a new project, VDFS, also required a cache algorithm. This setting needed multiple cache instances to store different data types, so we generalized the cache implementation using a C++ template. Clock2Q was selected again, primarily because it met the simplicity requirements while providing strong baseline performance.

**Concurrency evolution: from a global lock, to sharding, to fine-grained locking.** In the VDFS setting, concurrency became a hard requirement: multiple threads accessed the cache simultaneously. We initially adopted the simplest correct approach—a single global lock protecting the entire cache. Although the lock was released during disk I/O on misses, it still caused substantial contention on the hot hit path. We next tried sharding, which reduced contention until the transaction subsystem introduced a new constraint: transactions needed to reserve cache blocks to guarantee forward progress and avoid deadlock. Because blocks needed by a transaction could map to any shard, reserving $N$ blocks for a transaction required reserving $N$ blocks in *each* shard to be safe, which was too wasteful to be viable. To address this, we designed the highly concurrent version of Clock2Q described in §4.1.1 and replaced sharding with fine-grained synchronization. After this change, cache access was no longer a performance bottleneck.

**Operational feature pressure: online resizing with minimal disruption.** vSAN must support a wide variety of workloads and deployment profiles, so online resizing with minimal data movement is an operational necessity. Here again, the simplicity of Clock2Q proved valuable: because the design uses only arrays and a chained hash table, we were able to add online growth and shrinking with relatively small incremental complexity.

**From S3-FIFO to Clock2Q+: correlation windows for metadata workloads.** Following the S3-FIFO publication, its structural similarity to Clock2Q was the key factor motivating further investigation. While studying S3-FIFO, we formed the initial hypothesis behind this paper: introducing a correlation window into the Small FIFO could combine the best properties of Clock2Q and S3-FIFO and outperform both, particularly for metadata caches with correlated references.

**Hardening against production corner cases: dirty pages and bounded scanning.** As we expanded workload coverage, we also encountered production-driven corner cases. One notable bug was that if every page in the Small FIFO is dirty, the eviction logic can enter an infinite loop while searching for an evictable page. This

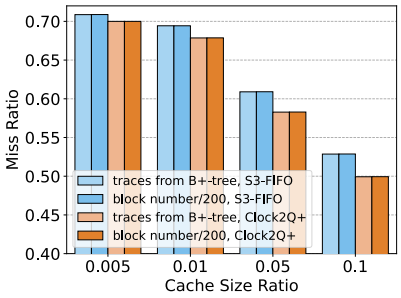



**Figure 8: Derived metadata traces and traces collected from TLX B$^+$ tree leaf block accesses are similar.**

directly motivated our study of bounding the amount of work the Small FIFO head or the clock hand performs before giving up, as discussed in §5.5.2. Implementing S3-FIFO in vSAN also revealed a practical mismatch in dirty-page behavior: dirty blocks should ideally be moved from the Small FIFO to the Main Clock when promoted, a behavior that did not occur in the original Clock2Q implementation. We again prioritized operational simplicity over theoretical accuracy, as discussed in §4.1.3, and we evaluate the negligible impact of this simplification in subsection 5.5.

**Simplicity as a first-class production requirement.** Overall, the evolution from Clock2Q to Clock2Q+ illustrates a typical path by which an academic idea is adapted into a production component. At each stage, we applied minimal engineering changes to avoid premature optimization, seeking a balanced design with reasonable hit ratio, low CPU overhead, high concurrency, and operational robustness. Among these characteristics, *simplicity* proved the most important in practice, even though it is rarely emphasized in academic discussions. Cache eviction is a class of algorithms where simplicity is especially critical: when performance is poor, it is difficult to determine whether the root cause is an implementation bug or simply an algorithm-workload mismatch. Debugging is further complicated by the lack of access to real customer workloads and the difficulty of constructing repeatable tests that preserve realistic access patterns. Widely used benchmarks often rely on simplistic distributions (e.g., uniform random or Zipf), which do not reflect production behavior. In our experience, starting from a simple design paid long-term dividends—most notably when we implemented and validated complex operational features such as online cache resizing years later.

## 5 EVALUATION

We focus on miss-ratio comparisons against state-of-the-art policies and then quantify the cost of the production simplifications in §4.

**Production goals.** Clock2Q+ retains the low hit-path overhead typical of clock-based designs: on a cache hit it performs a hash lookup and, at most, updates a Ref bit. It also scales well on multi-core systems because synchronization is fine-grained: each cache entry and each hash bucket has its own lock. In our performance testing in vSAN ESA and VDFS, cache lock contention did not appear as a bottleneck. The implementation is also intentionally simple. Clock2Q+ uses a small number of arrays (for the Main Clock,

the FIFOs, and the hash buckets) and singly linked lists only for hash collisions. This is simpler than other scan-resistant designs that require more complex data structures or adaptive resizing logic. Finally, Clock2Q+ requires no manual tuning in practice: the default parameterization is robust, which we validate explicitly in §5.6.

**Evaluation plan.** We compare Clock2Q+ to state-of-the-art eviction algorithms across multiple workload types, including derived metadata traces, original data traces, and non-block (object/key-value) traces. We quantify the impact of dirty blocks and evaluate whether the production simplifications described in §4.1.3 preserve cache efficiency. We then analyze why Clock2Q+ makes better promotion/eviction decisions than S3-FIFO, and we study parameter sensitivity to confirm robustness.

## 5.1 Experiment Setup

**Workloads.** Because vSAN is deployed on enterprise customers' hardware, we cannot run experiments on customer clusters or collect or release vSAN cache traces due to privacy and compliance requirements. We therefore use the open-source block cache traces from CloudPhysics [47]. The dataset contains 106 traces spanning 7 days, totaling 2,114 million requests over 82 TB of data. Clock2Q+ is designed for metadata caching in vSAN, but these public traces are data-layer traces. We thus derive metadata traces from the production data traces using the method in §2.4, by dividing LBNs by 200. We choose a fan-out of 200 to match the fan-out of the production $B^+$ tree in vSAN ESA. Although the proprietary vSAN traces themselves cannot be released, this fan-out-division conversion lets any reader regenerate the metadata traces used in this paper from any publicly available block trace, so the experiments below can be reproduced end-to-end.

**Benchmarks.** We implement Clock2Q+ in the libCacheSim simulator [59] to ensure fair and reproducible comparisons against prior and state-of-the-art algorithms. Our Clock2Q+ implementation has been upstreamed and is publicly available at https://github.com/1a1a11a/libCacheSim. Unless otherwise stated, we evaluate each algorithm at four cache sizes: 0.005, 0.01, 0.05, and 0.1 of the trace footprint.

**Baseline and metrics.** We compare Clock2Q+ with state-of-the-art eviction algorithms. Prior work shows that Clock can achieve both lower overhead and lower miss ratio than LRU [58], so we use Clock as the baseline and report miss-ratio improvement as:

$$\text{Improvement}_{\text{algo}} = \left( \frac{\text{MR}_{\text{Clock}} - \text{MR}_{\text{algo}}}{\text{MR}_{\text{Clock}}} \right)$$

## 5.2 Metadata Trace Fidelity

We validate the realism of our derived metadata traces by generating and comparing two metadata traces for the same workload. First, we construct a $B^+$ tree using the TLX library [6], replay a data trace, and record the accessed *leaf* blocks during the replay. Second, we generate a derived metadata trace by applying our mapping: for each request, divide the LBN by 200. We then evaluate S3-FIFO and Clock2Q+ on both traces. Figure 8 shows that the miss-ratio differences are less than 0.01% between the two traces, indicating that the simple fan-out mapping is an effective method for deriving metadata traces from data traces for our purposes.

## 5.3 Miss Ratio Improvements

**Metadata traces.** Figure 9a reports the per-trace relative miss-ratio improvement of ten state-of-the-art algorithms over Clock2Q+ on the derived metadata traces. Almost all evaluated algorithms exhibit negative or negligible improvement. This is consistent with our earlier analysis: many policies mis-handle correlated references and therefore promote transient metadata blocks into the main cache segment, leading to cache pollution and suboptimal evictions.

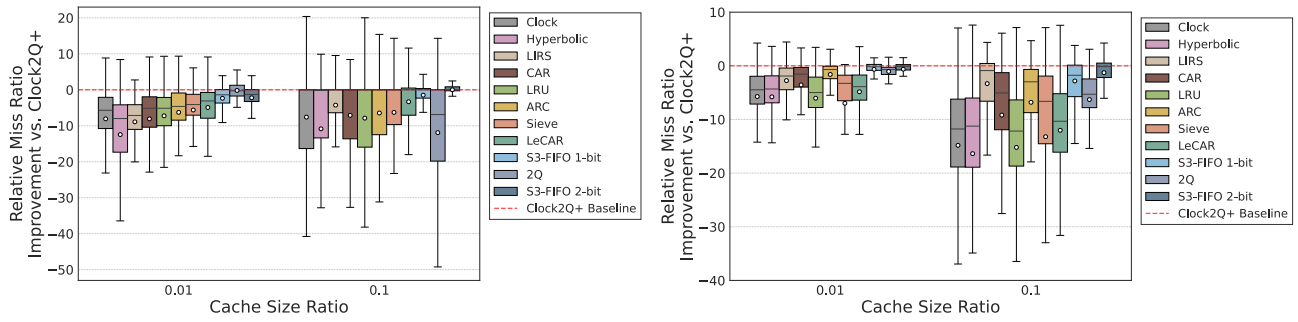
Among the evaluated policies, only S3-FIFO 2-bit (the default S3-FIFO configuration) and 2Q show improvements compared to Clock2Q+ at both cache sizes. S3-FIFO 2-bit uses a 2-bit frequency counter: an item must be re-referenced at least twice before it can be promoted into the Main Clock. This reduces (but does not eliminate) the chance that cold blocks experiencing correlated references are promoted. In contrast, S3-FIFO 1-bit promotes blocks after a single re-reference and is therefore more susceptible to correlated bursts, yielding higher miss ratios on metadata traces. 2Q addresses correlated references in a different way: entries in the Small FIFO are evicted (to ghost) rather than promoted based on short-term hit frequency, which naturally avoids over-promotion of bursty cold blocks.

Clock2Q+ is explicitly designed for correlated-reference workloads. As described in §3.4, its correlation window prevents short-term bursts from setting the Ref bit and thereby prevents cold metadata blocks from being promoted into the Main Clock. Across the 106 CloudPhysics traces, Clock2Q+ achieves the highest median and mean miss-ratio improvement at every tested cache size on both metadata and data traces; S3-FIFO 2-bit is the second-best performer at the larger cache size on metadata traces, and the largest per-trace gap over it reaches 28.5% at the larger cache size.

**Data traces.** For comparison, we evaluate the same ten algorithms on the original data traces, with results shown in Figure 9b. Even though many of these policies were designed and tuned specifically for data caches, Clock2Q+ achieves a higher median and mean miss-ratio improvement than every other evaluated policy at both cache sizes; the bulk of the per-trace distributions for the other algorithms sits at or below zero. This suggests that some data workloads may also contain correlated-reference-like effects (e.g., due to application-level clustering), and that Clock2Q+'s admission filtering can be beneficial beyond metadata caching.

**Consistent improvement across cache sizes.** Figure 10 shows miss-ratio curves for CloudPhysics workload w56, used here as a representative case study. The left column shows derived metadata traces and the right column shows data traces. Each subplot compares Clock, ARC, S3-FIFO, and Clock2Q+ on the same workload and object set. Across all panels, Clock2Q+ yields the lowest curve: it remains below the competing algorithms over the **entire** cache-size range (from 1% to 99%). On metadata traces, Clock2Q+ shifts the curve downward across the range. On data traces, absolute miss ratios are higher, yet Clock2Q+ again achieves the lowest miss ratio at every cache size.

In summary, Clock2Q+ consistently achieves lower miss ratios than state-of-the-art algorithms, with the improvement being more pronounced on metadata traces.




(a) Metadata traces (b) Data traces

**Figure 9: Comparing Clock2Q+ with state-of-the-art eviction algorithms.**

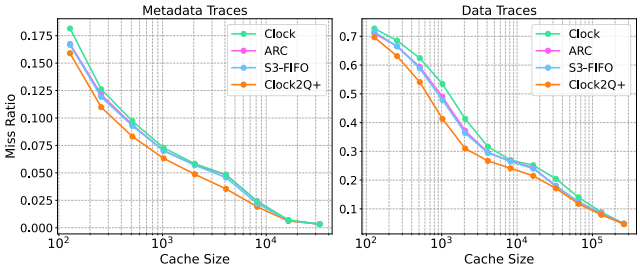
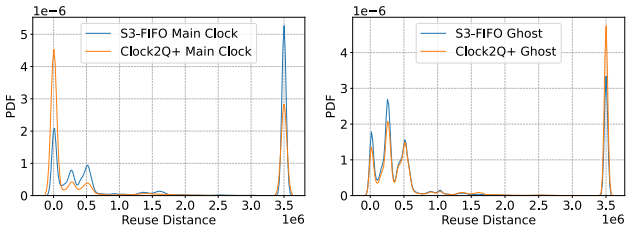



**Figure 10: Miss-ratio curves for CloudPhysics w56. The x-axis is cache size (log scale, unit is byte), and the y-axis is miss ratio. Lower curves indicate better performance (fewer misses) at a given cache size.**




(a) Blocks to Main Clock (b) Blocks to Ghost FIFO

**Figure 11: Probability distribution function (PDF) of NRD of blocks from Small FIFO. Clock2Q+ moves more cold blocks (large NRDs) to the Ghost FIFO and moves more hot blocks (small NRDs) to the Main Clock. The rightmost peak represents blocks that are not accessed again.**

## 5.4 The Secret Behind High Efficiency

To explain Clock2Q+'s advantage, we analyze how it differs from S3-FIFO in deciding which blocks are promoted into the Main Clock versus sent to the Ghost FIFO. We use one trace as a case study throughout this section. We compute the *next reuse distance* (NRD) for blocks that leave the Small FIFO and are either (i) moved into the Main Clock or (ii) moved into the Ghost FIFO. Small NRD indicates imminent reuse and suggests the block should stay in the Main Clock; large NRD indicates distant or no reuse and suggests eviction is appropriate.

Figures 11a and 11b show that Clock2Q+ sends more small-NRD blocks to the Main Clock and fewer to Ghost, retaining soon-to-be-reused blocks while filtering cold bursts.

Table 1 reports the number of blocks that are moved from the Small FIFO into the Main Clock and Ghost FIFO, respectively. Clock2Q+ promotes fewer than 25% as many Small-FIFO blocks into the Main Clock as S3-FIFO, while moving a comparable number to Ghost. This highlights that *which* blocks are admitted into the Main Clock matters as much as how many: by being more selective, Clock2Q+ reduces pollution in the main segment. These observations support the use of the correlation window as an effective mechanism for controlling admission from the Small FIFO.

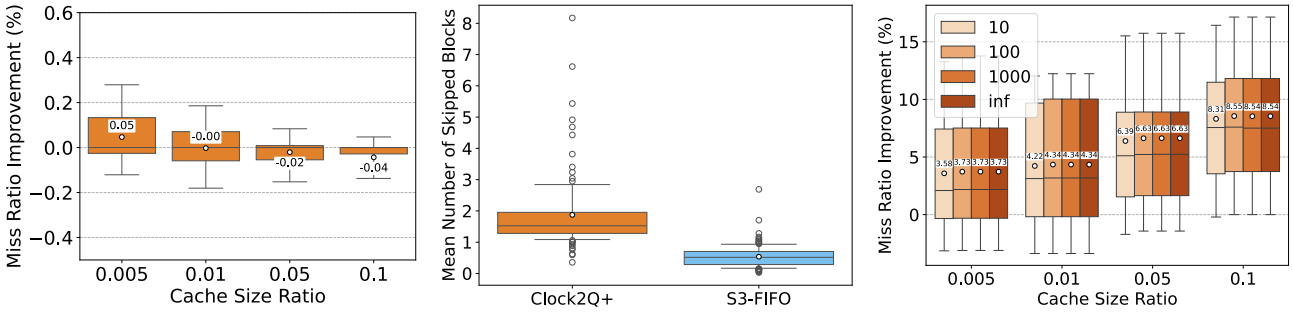
**Table 1: Counts of inserted block in the Main Clock and Ghost FIFO for Clock2Q+ and S3-FIFO.**

| Algorithm | Small → Main | Small → Ghost | Ghost → Main |
|---|---|---|---|
| Clock2Q+ | 20,762 | 186,959 | 33,068 |
| S3-FIFO | 88,529 | 126,897 | 29,909 |

## 5.5 Impact of Production Optimizations

*5.5.1 Dirty Blocks.* In Clock2Q+, the Small FIFO skips dirty blocks when searching for an eviction candidate. After skipping a threshold number of dirty blocks, it gives up on the Small FIFO for that eviction and inserts the new block into the Main Clock. This prevents dead loops when the Small FIFO becomes entirely dirty.

As discussed in §4.1.3, a dirty block at the tail of the Small FIFO with Ref=1 should ideally move to the Main Clock. Implementing this movement in vSAN is not trivial, so the production design reinserts the dirty block into the Small FIFO and revisits it in a later cycle when it becomes a candidate again. To quantify the cost of this simplification, we implement two Clock2Q+ variants in libCacheSim [59]: the production design described above, and an idealized baseline that moves Ref=1 dirty blocks at the Small




**(a) Simplified dirty block handling** **(b) Mean skipped block count distribution** **(c) Limit skipped blocks per eviction**

**Figure 12: (a) Impact of simplified dirty block handling. Miss ratio improvement is calculated against a variant of Clock2Q+ that moves dirty pages from the Small FIFO to the Main Clock. (b) The number of skipped blocks (frequency higher than 1 or dirty) per eviction. (c) Impact on the maximum skipped blocks on the miss ratio. A small limit, i.e., 10, does not significantly increase the miss ratio.**

FIFO tail directly into the Main Clock. The idealized baseline is not among the algorithms compared in Figure 9a.

We set the dirty block flush time to 30 seconds. The low and high watermarks for background flushing are set to 10% and 20% of cache size, matching Linux kernel page-cache defaults[2].

Figure 12a shows that simplified dirty-block handling has minimal impact (negative values indicate a miss-ratio increase relative to the idealized baseline defined above). At the smallest cache size, it can even reduce miss ratio for some traces; we conjecture that the Small FIFO window is too short and potentially hot blocks can be prematurely evicted, so skipping dirty blocks incidentally protects them. At larger cache sizes, simplified dirty handling slightly increases miss ratio for some traces, but the overall effect remains negligible. This supports the production choice to prioritize simplicity for dirty-page handling. This analysis was conducted only in simulation, because it is more difficult to experiment with dirty page handling in real systems under various workloads.

*5.5.2 Limiting Clock Hand Movement.* When evicting from the Main Clock, the clock hand can skip blocks whose Ref bit is set. In the worst case, a single eviction could skip many blocks, increasing CPU cost and creating latency spikes. Figure 12b shows the mean number of skipped blocks across the 106 CloudPhysics traces. The mean is low ($< 2$) for most traces for both Clock2Q+ and S3-FIFO. Clock2Q+ skips more blocks per eviction than S3-FIFO, likely because it retains more truly hot blocks in the Main Clock.

To bound CPU cost, we evaluate thresholds of 10, 100, 1000, and $\infty$ (unbounded). Once the threshold is reached, the algorithm evicts a non-dirty block regardless of its Ref bit. Figure 12c shows that a threshold of 10 only slightly increases miss ratio, suggesting that a small reinsertion limit is safe for production.

## 5.6 Sensitivity Analysis

Clock2Q+ sets the correlation window to half of the Small FIFO by default. We evaluate sensitivity to this choice by testing correlation windows of 10%, 30%, and 50% of the Small FIFO. Figure 13a shows that at small cache sizes a larger correlation window slightly improves miss ratio, while at larger cache sizes the differences are small. Our conjecture is that the correlation window is primarily a workload property and should not scale strongly with cache size; hence, when the cache is very small, a larger window can better suppress correlated bursts. Overall, miss ratio is not highly sensitive to the window size: all tested settings significantly outperform the Clock baseline.

Finally, Figure 13b evaluates three open-source object/key-value cache datasets [60] (Wikimedia, Meta Key-Value, and Tencent Photo), which together include 10 traces. Clock2Q+ still improves over the Clock baseline, but its gains are slightly smaller than S3-FIFO's. We conjecture that these workloads contain fewer correlated references, so Clock2Q+'s correlation window offers limited filtering benefit and can occasionally postpone promotion of truly hot objects. This result is consistent with our hypothesis that the correlation window is most useful when correlated references are common.

# 6 RELATED WORK

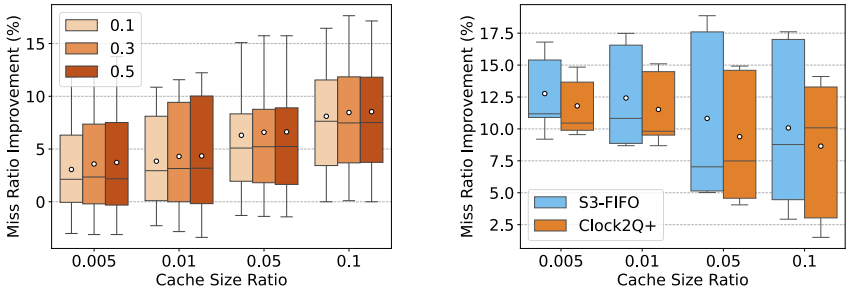
## 6.1 Cache eviction Algorithms

Cache eviction policies have been extensively studied [10, 19, 25, 26, 32, 45, 46, 57, 62]. LRU remains the most widely deployed due to its simplicity [18], while Clock provides a lower-overhead approximation [16, 60]. However, both are vulnerable to scans and pollution.

To improve scan resistance, policies use reuse distance (LIRS [25], LIRS2 [63], Clock-Pro [24], DLIRS [31], LRU-K [34]), adaptive recency/frequency lists (ARC [32], CAR [2], CART [2]), segment protection (SLRU [23, 28]), or explicit scan detection (SEQ [21], EELRU [27]). Frequency-based policies (LFU [42], LFU* [1], LFU-Aging [36], LFU*-Aging [52], LRFU [30], LeCaR [46], Cacheus [39]) balance recency and frequency. Recent work explores learning-based ranking (LRB [43], HALP [44], 3L-cache [64], GL-Cache [57]) and systems approaches (S3-FIFO [60], SIEVE [62]).

Variable-size caches use size-aware techniques such as GDSF [12], AdaptSize [5], LHD [3], and Hyperbolic [7], or admission control such as TinyLFU [19]. Other work includes application hints to

[2] `dirty_background_ratio` and `dirty_ratio` in `/proc/sys/vm/`.




**(a) Impact of correlation window size** **(b) Non-block traces**

**Figure 13: (a) Clock2Q+ is not sensitive to correlation window size. While 0.5 of the Small FIFO queue works the best, even a very small correlation window size, e.g., 0.3, does not reduce the miss ratio improvement. (b) Comparing S3-FIFO and Clock2Q+ on non-block cache workloads (object and key-value) shows that the correlation window is not useful for these workloads because they do not show correlated references.**

guide eviction decisions (DEAR [13], AFC [14], UBM [29], ILRU [40], OLRU [40], Hot Set [41], QLSM [15]) and contention reduction approaches (Mr. Cache [48]).

BP-Wrapper [17] reduces lock contention through batching and prefetching, but it does not remove the CPU and memory overhead of LRU as Clock does. Its batching model may also be less effective for vSAN threads, where a thread can block on the first contended cache access before its per-thread FIFO becomes useful.

A fundamental challenge for production systems is lock contention from object promotions [58]. Lazy promotion techniques strategically delay or suppress promotions, improving throughput without significantly increasing miss ratio [11].

## 6.2 Correlated References

Correlated references were identified in FBR [36]. Subsequent policies introduced mechanisms to detect and filter correlated bursts so that short-term clustering does not masquerade as long-term popularity. 2Q [26] uses a Small FIFO to absorb correlated bursts, but it achieves this by evicting every block that exits the FIFO to a ghost structure, which can impose an extra miss on hot blocks before they reach the main segment. CART [2] extends CAR with correlated-reference handling using an idea similar in spirit to 2Q: it uses the length of $T_1$ (analogous to 2Q's Small FIFO) as a correlation window, and blocks found in ghost queue $B_1$ are treated as being outside that window. This design, like 2Q, can still induce extra misses as part of establishing that a block should be promoted.

Clock2Q+ targets the same problem but avoids this penalty. Rather than requiring an eviction-and-ghost re-reference to establish that a block is beyond the correlation window, Clock2Q+ suppresses Ref-bit setting within the correlation window and enables direct promotion once a block demonstrates reuse outside the window. This preserves the benefits of correlated-reference filtering while avoiding additional misses for hot blocks.

## 6.3 Cache Production Systems

Several papers have reported experience building and operating large-scale caching systems in production. Nisthala *et al.* describe the challenges of scaling Memcache at Meta [33]. Bronson *et al.* present TAO, a specialized cache for the social graph at Meta [9]. Huang *et al.* study Meta's CDN cache design with an emphasis on eviction policy [23]. More recently, Berg *et al.* demonstrate the benefits of consolidating multiple production caches onto a shared caching library at Meta [4]. In addition, prior work has analyzed large in-memory KV-cache workloads at Twitter and highlighted the diversity of access patterns in real deployments, and Yang *et al.* discuss a flash-cache admission policy deployed at Google [61].

Clock2Q+ is complementary to these efforts. Rather than presenting a full caching stack or a workload characterization study, Clock2Q+ focuses on the *eviction-policy design* itself, and explains the specific design choices needed to handle metadata-cache access patterns such as correlated references. Moreover, most prior deployed-system papers target *application data caches*; in contrast, vSAN's metadata cache is more sensitive to overhead because it sits on a particularly hot path, and it runs in environments where *manual tuning is impractical* (vSAN is deployed on customers' hardware). These differences drive different engineering priorities: our design emphasizes low CPU overhead, minimal tuning, simplicity, and multi-core scalability.

# 7 CONCLUSION

Metadata-cache workloads frequently exhibit *correlated references*, where short bursts of repeated accesses reflect structural locality rather than sustained popularity. This paper presents Clock2Q+, a simple cache eviction algorithm tailored to this behavior and implemented in Broadcom VMware vSAN. Clock2Q+ achieves the highest median and mean miss-ratio improvement among the state-of-the-art policies we evaluate on both metadata and data workloads, with the largest per-trace gap over S3-FIFO reaching a 28.5% relative miss-ratio reduction on metadata. Beyond cache efficiency, Clock2Q+ is engineered for deployment: it preserves the low CPU and memory overhead of clock-based designs, scales well under high concurrency, requires no tuning in practice, and remains straightforward to implement and debug—properties that are essential for reliable operation in production systems.

# REFERENCES


[1] Martin F. Arlitt. 1996. *A performance study of Internet web servers.* Master's thesis. Department of Computer Science, University of Saskatchewan.

[2] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA) *(FAST '04)*. USENIX Association, USA, 187–200.

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

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

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

[6] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, retrieved Oct. 7, 2020.

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

[8] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3 ed.). O'Reilly Media, Inc.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13) (ATC'13)*. 49–60.

[10] Richard W. Carr and John L. Hennessy. 1981. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles (SOSP'81)*. Association for Computing Machinery, New York, NY, USA, 87–95. https://doi.org/10.1145/800216.806596

[11] Qinghan Chen, Muhammad Haekal Muhyidin Al-Araby, Ziyue Qiu, Zhuofan Chen, Rashmi Vinayak, and Juncheng Yang. 2025. Demystifying and Improving Lazy Promotion in Cache Eviction. *Proc. VLDB Endow.* 19, 4 (Dec. 2025), 549–562. https://doi.org/10.14778/3785297.3785299

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

[13] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, , and Yookun Cho. 1998. An adaptive block management scheme using on-line detection of block reference patterns. In *Proceedings of the 1998 International Workshop on Multimedia Database Management Systems (IW-MMDBMS 1998)*. Dayton, OH, 172–179.

[14] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, , and Yookun Cho. 2000. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*. Santa Clara, CA, 286–295.

[15] Hong-Tai Chou and David J. DeWitt. 1985. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB 1985)*. Stockholm, Sweden, 174–188.

[16] Fernando J. Corbat and Victor A. Vyssotsky. 1969. A Paging Experiment with the Multics System. *Honeywell Inc.* (1969).

[17] Xiaoning Ding, Song Jiang, and Xiaodong Zhang. 2011. BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*. IEEE, 369–380.

[18] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of Database Buffer Management. In *ACM Transactions on Database Systems (TODS)*, Vol. 9. ACM, 560–595. Issue 4.

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

[20] Kevin W. Froese and Richard B. Bunt. 1996. The effect of client caching on file server workloads. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS 1996)*. ACM, Kihei, HI, USA, 150–159.

[21] Gideon Glass and Pei Cao. 1997. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1997)*. ACM, Seattle, WA, USA, 115–126.

[22] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.

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

[24] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. Clock-Pro: An Effective Improvement of the Clock Replacement. *Proceedings of the USENIX Annual Technical Conference* (2005), 323–336.

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

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

[27] Yannis Smaragdakis Scott Kaplan and Paul Wilson. 1999. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*. ACM, Atlanta, GA, USA, 122–133.

[28] Ramakrishna Karedla, Spencer Love, and B. Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 27, 3 (1994), 38–46. https://doi.org/10.1109/2.268884

[29] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, , and Chong Sang Kim. 2000. Unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*. San Diego, CA.

[30] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* 50, 12 (2001), 1352–1361.

[31] Cong Li. 2018. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018, HAIFA, Israel, June 04-07, 2018*. ACM, 59–64. https://doi.org/10.1145/3211890.3211891

[32] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 115–130.

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

[34] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. ACM, 297–306.

[35] Hal Pomeranz. [n.d.]. Understanding EXT4 (Part 3): Extent Trees. https://web.archive.org/web/20190818050155/https://digital-forensics.sans.org/blog/2011/03/28/digital-forensics-understanding-ext4-part-3-extent-trees. Accessed June 23, 2026.

[36] John T. Robinson and Murthy V. Devarakonda. 1990. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*. ACM, Boulder, CO, USA, 134–142.

[37] Ohad Rodeh. [n.d.]. COW B-trees. https://github.com/ohad-rodeh/bt. Accessed June 23, 2026.

[38] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage* 9, 3 (2013), 1–32.

[39] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 341–354.

[40] Giovanni Maria Sacco. 1987. Index access with a finite buffer. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 1987)*. Brighton, England, 301–309.

[41] Giovanni Maria Sacco and Mario Schkolnick. 1986. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)* (1986).

[42] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208.

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

[44] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. 2023. HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*. 1149–1163.

[45] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*. 373–386.

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

[47] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *13th USENIX conference on file and storage technologies (FAST'15)*. USENIX Association, Santa Clara, CA, 95–110.

[48] Wenguang Wang. 2004. *Storage Management for Large Scale Systems*. Ph.D. Dissertation. Department of Computer Science, University of Saskatchewan.

[49] Wenguang Wang. 2017. System and methods of a CPU-efficient cache replacement algorithm. U.S. Patent 9760493B1. Filed on March 14, 2016, issued on September 12, 2017.

[50] Wenguang Wang, Mounesh Badiger, Abhay Kumar Jain, Junlong Gao, Zhaohui Guo, and Richard P. Spillane. 2021. System and method of a highly concurrent cache replacement algorithm. U.S. Patent 11086779B2. Filed on November 11, 2019, issued on August 10, 2021.

[51] Wikipedia. [n.d.]. XFS. https://en.wikipedia.org/wiki/XFS. Accessed June 23, 2026.

[52] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. 1993. Disk cache replacement policies for network fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems(ICDCS 1993)*. Pittsburgh, PA, 2–11.

[53] Enning Xiang and Wenguang Wang. 2022. Updating stored content in an architecture utilizing a middle map between logical and physical block addresses. U.S. Patent 11487456B1. Filed on April 23, 2021, issued on November 1, 2022.

[54] Enning Xiang and Wenguang Wang. 2023. Architecture utilizing a middle map between logical to physical address mapping to support metadata updates for dynamic block relocation. U.S. Patent 11579786B2. Filed on April 23, 2021, issued on February 14, 2023.

[55] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. 2011. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Transactions on Computer Systems* 29, 2 (May 2011), 5:1–5:51. https://doi.org/10.1145/1963559.1963561

[56] Gala Yadgar, Michael Factor, and Assaf Schuster. 2007. Karma: Know-it-All Replacement for a Multilevel Cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07) (FAST'07)*.

[57] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23) (FAST'23)*. 115–134.

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

[59] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208.

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

[61] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. 2022. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference (ATC'22)*. USENIX Association, Carlsbad, CA, 1021–1036.

[62] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. 2024. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1229–1246.

[63] Chen Zhong, Xingsheng Zhao, and Song Jiang. 2021. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 19, 12 pages. https://doi.org/10.1145/3456727.3463772

[64] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 2025. 3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 237–254.