Evaluating Cache-Replacement Policies Under a Mixed Workload

P. Novak

August 27, 2026

Contents

1 Objective **3**

1.1 Background 3

2 Method **4**

2.1 Workload 4

2.2 Implementation 4

3 Results **6**

4 Discussion **7**

5 Conclusion **8**

Bibliography **9**

Chapter 1

Objective

This report evaluates three cache-replacement policies — LRU (Least Recently Used), LFU (Least Frequently Used) and CLOCK — on a simulated single-level cache under a workload that mixes a scanning access pattern with a small set of hot keys. The goal is to identify which policy best resists *cache pollution* from the scan.

1.1 Background

A pure scan (each key touched exactly once, in sequence) is the textbook failure case for LRU: every scanned key evicts a still-useful hot key, even though the scanned keys themselves are never reused [1]. LFU and CLOCK are both designed to be more resistant to this, at the cost of extra per-entry bookkeeping [2].

Chapter 2

Method

2.1 Workload

Each trial issues 200,000 accesses to a keyspace of 10,000 keys split into two groups:

- A **hot set** of 200 keys, together receiving 80% of all accesses (Zipf-distributed within the set).
- A **scan set** of the remaining 9,800 keys, each touched at most twice, interleaved with the hot-set accesses at a fixed ratio of one scan access per four hot accesses.

The cache is sized at 500 entries (5% of the keyspace, 2.5$\times$ the hot set), so a policy immune to scan pollution should converge to a near-100% hit rate on the hot set alone.

2.2 Implementation

Listing 2.1: CLOCK eviction core

```
class ClockCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.ref_bit = {}
        self.order = []
        self.hand = 0

    def access(self, key):
        if key in self.ref_bit:
            self.ref_bit[key] = 1
            return True # hit
        self._evict_if_full()
        self.order.append(key)
        self.ref_bit[key] = 1
        return False # miss
```

LRU was implemented with an ordered dict (O(1) move-to-front on access); LFU with a min-heap keyed on access count, ties broken by insertion order; CLOCK as shown above, with a circular scan and a single reference bit per entry.

Chapter 3

Results

Table 3.1: Overall and hot-set hit rate by policy

Policy	Overall hit rate	Hot-set-only hit rate
LRU	61.2%	76.5%
LFU	78.9%	98.1%
CLOCK	74.4%	93.7%

LRU's hot-set hit rate of 76.5% falls well short of the near-100% a scan-immune policy should achieve at this cache size, confirming the expected pollution effect: each scan access evicts one hot key whether or not it will ever be touched again. LFU comes closest to immune, since a key touched only once or twice can never outrank a hot key's accumulated count. CLOCK sits between the two — its single reference bit protects a hot key for one extra sweep past its last access, but a long enough run of scan accesses between hot touches still evicts it.

Chapter 4

Discussion

The results reproduce the standard LRU-scan-pollution story quantitatively rather than just qualitatively: at this hot/scan ratio and cache size, LRU loses roughly 1 in 4 hot-set requests to scan-induced eviction that LFU recovers entirely. CLOCK's middle-ground result matches its design intent as a cheap approximation to LFU-like resistance without a full frequency counter [2], and the gap between CLOCK and LFU here (4.4 percentage points on the hot set) is the price of that approximation under this specific workload.

A caveat: LFU's advantage assumes the hot set is genuinely stable over the measurement window. A workload whose hot set shifts over time would erode LFU's advantage, since old high-count keys can persist in cache long after they stop being useful — a failure mode this benchmark does not exercise.

Chapter 5

Conclusion

For a workload with a stable hot set contaminated by a one-shot scan, LFU gives the best cache utilisation of the three policies tested, with CLOCK a reasonable lower-overhead compromise and plain LRU the clear loser.

Bibliography

[1] L. A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems Journal* 5.2 (1966), pp. 78–101.

[2] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. “Caching in the Sprite Network File System”. In: *ACM Transactions on Computer Systems*. Vol. 6. 1. 1988, pp. 134–154.