Time-Decayed Caching: A Novel Rule Benchmarked Against Established Policies

Cătălin Pângăleanu

Abstract:

AMS Mathematical Subject Classification: 68M20, 68P05

ACM Computing Reviews Categories and Subject Descriptors: Theory of computation, Design and analysis of algorithms, Online algorithms, Caching and paging algorithms

Contents

1. Introduction

We propose a time-decayed policy based on the nuclear physics concept of half-life. We introduce a one-parameter eviction rule that maintains a per-key decayed score using a half-life (H) constant. The policy blends recency and frequency and evicts the item with the lowest current decayed score.

1.1. Motivation

1. Summarize classical policies such as LRU, LFU, SLRU
2. Explain the need for a new policy that combines both recency and frequency

1.2. Contributions

1. Go into more detail for half-life policy. In short, H = the horizon where an old hit is worth half a new hit. If H = 100 requests, then a hit that happened 100 requests ago counts ½; 200 requests ago counts ¼, etc.
2. Explain automatic adjustment of H-constant. $H = c * R$, where c is a predetermined constant and R = exponential weighted moving average ($R[i + 1] = (1 - \eta) * R[i] + \eta * gap$, where gap = now - last_hit[key] in requests)

1.3. Research Questions

1. On different workloads and fixed capacity, does HL-Cache improve hit rate and tail latency over LRU/SLRU/LFU?
2. Does the new policy offer CPU/memory savings?
3. How sensitive is the policy to half-life mis-specification, and does online auto-tuning (formula in section above) keep performance better?

2. Background and Related Work

2.1. Terminology and Metrics

1. Explain cache-related terminology (hit etc.)
2. Explain what we would count as success for the proposed policy (possible metric is hit-rate gain compared to other classical policies etc.)

2.2. Classical Policies

1. Go into detail on LRU, LFU etc.

2.3. Other Related Work

1. Go over some of the more recent policies and optimizations (from the bibliography)
2. Explain general use of caching policies (in OS, databases etc.)

3. The Proposed Policy: Time-Decayed Caching

3.1. Formulation

We formalize the time-decayed caching policy in terms of a per-key decayed popularity score and an associated eviction rule. The key idea is that each access to an object contributes a unit of “popularity mass” that decays exponentially over subsequent requests, parameterized by a half-life H.

Problem setting and notation

We work in discrete time, indexed by request number. Let:

- $\mathcal{U}$ be the universe of objects.
- $r_1, r_2, \ldots, r_t, \ldots$ be the request sequence, where each $r_t \in \mathcal{U}$.
- The cache have capacity C (considered in number of objects).
- $\mathcal{C}(t) \subseteq \mathcal{U}$ denote the set of objects in the cache immediately before serving request r_t.

For each object $i \in \mathcal{U}$, let

$$t_1^i < t_2^i < \cdots < t_{k_i(t)}^i \leq t$$

be the times (request indices) at which i has been requested up to and including time t, and let $k_i(t)$ be the number of such requests.

Time-decayed popularity score

The policy associates to each object i a real-valued popularity score $S_i(t)$ at time t. This score is defined as an exponentially decayed sum of past accesses:

$$S_i(t) \; = \; \sum_{j=1}^{k_i(t)} e^{-\lambda\,(t - t_j^i)}, \lambda > 0.$$

Equivalently, if we parameterize the decay in terms of a **half-life** $H > 0$ measured in number of requests, we set

$$\lambda \; = \; \frac{\ln 2}{H}$$

and obtain the equivalent form

$$S_i(t) \;=\; \sum_{j=1}^{k_i(t)} 2^{-(t-t_j^i)/H}.$$

In this formulation:

- Each request to object i at time t_j^i contributes a term that is initially 1 and then decays over subsequent requests.
- After exactly H requests have passed since that access, its contribution is halved:

$$2^{-H/H} = \frac{1}{2}.$$

- After $2H$ requests, the contribution is $2^{-2} = \frac{1}{4}$, after $3H$ it is $2^{-3} = \frac{1}{8}$, and so on.

Thus, H defines an effective horizon over which past hits remain significantly influential: a hit that occurred nH requests ago counts as 2^{-n} of a new hit. Small values of H emphasize very recent activity, while large values integrate information over a longer history.

Conceptually, this popularity score has the following properties:

- It increases whenever the object is requested (each new hit contributes an additional term of size 1).
- It decreases smoothly as time passes without accesses, since every term in the sum is multiplied by a factor smaller than 1 at each step.
- It captures both frequency (more hits add more terms) and recency (more recent hits are less decayed and thus have larger weight).

For later implementation and analysis, it is helpful to note that this definition admits a recursive characterization. Let $\Delta_i(t)$ denote the number of requests that have occurred since the previous request for object i (i.e., the gap in request indices). Then, immediately before processing a new request for i at time t, we can write

$$S_i(t) \;=\; S_i(t^-) \cdot 2^{-\Delta_i(t)/H} + 1,$$

where $S_i(t^-)$ denotes the score right before applying the current hit. This shows that the full history of accesses to i can be compressed into a single scalar S_i, updated by exponential decay plus a unit increment on each hit.

Eviction rule

The time-decayed policy uses the scores $S_i(t)$ to make eviction decisions. At a cache miss at time t, when the cache is already full (i.e., $|\ \mathcal{C}(t)\ | = C$), the policy proceeds conceptually as follows:

1. For each object $j \in \mathcal{C}(t)$, consider its current time-decayed popularity score $S_j(t)$.
2. Identify an object with minimal score:

$$j = \ \arg\min_{j \in \mathcal{C}(t)} S_j(t).$$

3. Evict j from the cache.

4. Insert the newly requested object r_t into the cache and initialize its score with 1.

The half-life parameter H controls how rapidly the cache "forgets" past popularity:

- For small H, scores concentrate on recent accesses, and the policy behaves more like a recency-driven strategy.
- For large H, scores retain the influence of past accesses over longer spans, and the policy behaves more like a smoothed frequency-based strategy.

In subsequent sections we describe how to maintain the popularity scores efficiently in an online setting and how to adapt the half-life H to the observed workload.

3.2. Complexity

A naïve implementation would maintain, for each object i in the cache, its explicit decayed score

$$S_i(t) = \sum_{j=1}^{k_i(t)} e^{-\lambda(t-t_j^i)}$$

and, at a miss, recompute all $S_j(t)$ for $j \in \mathcal{C}(t)$ in order to select the victim with minimal score. This leads to $\mathcal{O}(|\ \mathcal{C}(t)\ |)$ work per miss, which is unacceptable for large caches. Instead, we exploit the algebraic structure of $S_i(t)$ to maintain a time-normalized key K_i for each cached object and store these keys in a priority queue (min-heap), so that extracting the victim remains efficient.

For an object i, let t_i denote the time of its most recent update (hit) and $S_i(t_i)$ the score immediately after that update. By definition of exponential decay,

$$S_i(t) = S_i(t_i)\, e^{-\lambda(t-t_i)}$$

for any later time $t \geq t_i$. We can rewrite this as

$$S_i(t) = e^{-\lambda t}(S_i(t_i)\, e^{\lambda t_i}).$$

This suggests defining the time-normalized key

$$K_i := S_i(t_i)\, e^{\lambda t_i}.$$

Then, for any current time t,

$$S_i(t) = e^{-\lambda t}\, K_i.$$

The global factor $e^{-\lambda t}$ is the same for all objects, so for any fixed t the ordering of objects by true score $S_i(t)$ is exactly the same as the ordering by key K_i. In particular, an object with minimal K_i also has minimal current score $S_i(t)$. This allows us to store only K_i for each object and to use K_i as the heap key, without explicitly recomputing all $S_i(t)$ at eviction time.

Moreover, the key representation yields a simple update rule on hits that does not require storing $t_i$or $S_i(t_i)$ separately. Suppose object i is present in the cache at time t, with current key K_i. Its true score just before processing the new hit is

$$S_i^{\text{current}} = S_i(t) = K_i e^{-\lambda t}.$$

After the hit, the new score is

$$S_i^{\text{new}} = S_i^{\text{current}} + 1 = K_i e^{-\lambda t} + 1.$$

The corresponding new key is

$$K_i^{\text{new}} = S_i^{\text{new}} e^{\lambda t} = \left(K_i e^{-\lambda t} + 1\right) e^{\lambda t} = K_i + e^{\lambda t}.$$

Thus, on a hit at time t, we can update the key using the simple recurrence

$$K_i \leftarrow K_i + e^{\lambda t},$$

without explicitly tracking $t_i$or $S_i(t_i)$. On a miss for a never-seen-before object i, its first request occurs at time t, and by the definition of $S_i(t)$ we have $S_i(t) = 1$, so we initialize

$$K_i = e^{\lambda t}.$$

To support efficient eviction, we maintain a hash table from object identifiers to heap positions, and a binary min-heap keyed by K_i over the objects currently in the cache. For each request, the operations are:

- Hit: we look up the object in the hash table (expected $\mathcal{O}(1)$), compute $e^{\lambda t}$ once for the current time step, update $K_i \leftarrow K_i + e^{\lambda t}$, and perform a key-increase operation in the heap. In a binary heap this takes $\mathcal{O}(\log\ C)$ time, where C is the cache capacity.
- Miss with free space: we insert a new object with key $K_i = e^{\lambda t}$ into the hash table and heap, which costs $\mathcal{O}(\log\ C)$.
- Miss with full cache: we extract the minimum key from the heap to obtain the victim

  $$\mathrm{j} = \arg \min_{j \in \mathcal{C}(t)} K_j$$

  in $\mathcal{O}(\log\ C)$, remove j from the hash table, and then insert the new object as above, for a total $\mathcal{O}(\log\ C)$ time.

Therefore, each request is processed in $\mathcal{O}(\log\ C)$ worst-case time, dominated by heap operations, compared to the $\mathcal{O}(1)$ updates of pointer-based LRU or counter-based LFU. In practice, cache capacities are bounded and $\log\ C$ is small (e.g., $\log_2 10^4 \approx 14$), so the additional overhead is modest, while the policy gains the ability to track a smooth time-decayed popularity measure without scanning the full cache. The space complexity is $\mathcal{O}(C)$: for each cached object we store its key K_i, the cached value (or a pointer to it), and the metadata required by the hash table and heap.

4. Implementation

4.1. Data Structures and Maintained State

The implementation follows the formulation from section 3.2. Thus, we store only a single scalar key K_i per cached object, rather than explicitly maintaining its decayed score $S_i(t)$. We recall that for an object i updated last at time t_i with score $S_i(t_i)$, we define

$$K_i := S_i(t_i)\, e^{\lambda t_i},$$

so that for any current time t,

$$S_i(t) = e^{-\lambda t} K_i.$$

Because the factor $e^{-\lambda t}$ is the same for all objects, the ordering of objects by K_i coincides with the ordering by their true decayed scores $S_i(t)$ at time t.

The cache maintains the following global state:

- A global time index $t \in \mathbb{N}$, interpreted as the number of read requests processed so far. Each GET (access) operation increments t by one.
- The cache capacity C, measured in number of objects.
- The decay parameter $\lambda > 0$, computed based on half-life H: $\lambda = \ln\ 2/H$.

For the objects currently stored in the cache, we maintain two core data structures:

- A hash table `hashTable` that maps object identifiers (keys) to internal cache entries. This provides expected $\mathcal{O}(1)$ lookup by key.
- A binary min-heap `minHeap` containing all cached entries, ordered by their time-normalized key K_i. The heap supports extracting the entry with minimal K_i and updating an existing entry in $\mathcal{O}(\log\ C)$ time.

Each cached object i is represented by a cache entry that stores:

- `key`: the object identifier,
- `value`: the cached value,
- `K`: the current time-normalized key K_i.

The explicit score $S_i(t)$ and the last update time t_i are not stored. Conceptually, they can be recovered from K_i, but the reconstruction is not required by the core operations of the policy.

4.2. Core Operations

We model cache usage through a standard key–value interface with two primary operations:

- `GET(key)`: read access that may hit or miss in the cache and does contribute to the popularity score.
- `PUT(key, value)`: write/update operation that adds or modifies a cached entry but does not contribute to the popularity score.

In the pseudocode below, `exp(·)` denotes the natural exponential $e^{(\cdot)}$. `HEAP_UPDATE`, `HEAP_INSERT`, and `HEAP_POP_MIN` represent classic heap operations on `minHeap`. Similarly, `HASH_TABLE_SEARCH`, `HASH_TABLE_INSERT`, and `HASH_TABLE_REMOVE` represent hash table operations on `hashTable`.

GET operation (read request)

```
procedure GET(key):
    t ← t + 1    // advance read-time counter
    entry ← HASH_TABLE_SEARCH(hashTable, key)
    if entry ≠ NONE then    // cache hit
        UPDATE_POPULARITY(entry)
        return entry.value
    else    // cache miss
        return NONE
```

On a hit, we update the time-normalized key K_i for the corresponding entry. Based on previous sections, if the current time is t and the entry has key K_i, then after accounting for the decayed past and adding one new hit, the updated key is

$$K_i \leftarrow K_i + e^{\lambda t}.$$

```
procedure UPDATE_POPULARITY(entry):
    delta ← exp(λ · t)
    entry.K ← entry.K + delta
    HEAP_UPDATE(minHeap, entry)
```

Here `HEAP_UPDATE (minHeap, entry)` performs the appropriate sift-up or sift-down operation to restore the min-heap invariant after `entry.K` changes; in a binary heap this runs in $\mathcal{O}(\log\ C)$ time.

PUT operation (write/update request)

The PUT operation updates or creates a cached value without incrementing the popularity score. Writes therefore do not act as “hits” for the replacement policy; they only affect the stored value and, for new keys, their presence in the cache. When inserting a new key into a full cache, PUT triggers eviction.

```
procedure PUT(key, value):
    entry ← HASH_TABLE_SEARCH(hashTable, key)
    if entry ≠ NONE then    // update cache entry
        entry.value ← value
    else    // create cache entry
```

```
        if SIZE(minHeap) = C then
            EVICT()

        entry ← new cache entry
        entry.key ← key
        entry.value ← value
        entry.K ← 0

        HEAP_INSERT(minHeap, entry)
        HASH_TABLE_INSERT(hashTable, key, entry)
```

Since PUT does not contribute to the score $S_i(t)$, newly inserted keys start with $K_i = 0$ and only gain popularity when subsequently accessed via GET. If the cache becomes full, such write-only entries are natural eviction candidates.

```
procedure EVICT():
    victim ← HEAP_POP_MIN(minHeap)
    HASH_TABLE_REMOVE(hashTable, victim.key)
```

During eviction, the cache entry with the smallest time-normalized key K_i, i.e. the least popular object according to the time-decayed score, is removed from `minHeap`. Its mapping from `hashTable` is removed as well.

5. Evaluation

5.1. Methodology

1. Implement classical policies (LRU, LFU etc.)
2. Get metrics such as hit-rate, CPU/memory usage, tail latency etc.
3. Benchmark current policy to classical ones based on metrics above. Use at least 2 different generated/pre-existing datasets
4. Determine sensitivity of half-life policy to changing constants (see c, η in formulas above)

5.2. Results

1. Summarize results and provide information visually (graphs etc.)

6. Future Work

7. Conclusion

8. Bibliography

[1] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, C.-S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," IEEE Transactions on Computers, 1352–1361, 2001.

[2] N. Megiddo, D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," USENIX FAST, 116–130, 2003.

[3] T. Johnson, D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," VLDB, 439–450, 1994.

[4] G. Einziger, R. Friedman, B. Manes, "TinyLFU: A Highly Efficient Cache Admission Policy," arXiv, 2015.

[5] A. Blankstein, S. Sen, M. J. Freedman, "Hyperbolic Caching: Flexible Caching for Web Applications," USENIX ATC, 499–511, 2017.

[6] O'Neil, Elizabeth J.; O'Neil, Patrick E.; Weikum, Gerhard. The LRU-K Page Replacement Algorithm for Database Disk Buffering. Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD '93), Washington, DC, USA, 297–306, 1993.

[7] Jiang, Song; Zhang, Xiaodong. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02), 31–42, 2002.

[8] Bansal, Sorav; Modha, Dharmendra S. CAR: Clock with Adaptive Replacement. Proc. USENIX Conference on File and Storage Technologies (FAST '04), 187–200, 2004.

[9] Jiang, Song; Chen, Feng; Zhang, Xiaodong. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. Proc. USENIX Annual Technical Conference (USENIX ATC '05), 323–336, 2005.

[10] Zhou, Shuhui; Philbin, John; Li, Kai. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. Proc. USENIX Annual Technical Conference (USENIX ATC '01), 91–104, 2001.

[11] Smaragdakis, Yannis; Kaplan, Scott F.; Wilson, Paul R. EELRU: Simple and Effective Adaptive Page Replacement. Proc. ACM SIGMETRICS '99, 122–133, 1999.

[12] Robinson, John T.; Devarakonda, Murthy V. Data Cache Management Using Frequency-Based Replacement. Proc. ACM SIGMETRICS '90, 134–142, 1990.

[13] Cao, Pei; Irani, Sandy. Cost-Aware WWW Proxy Caching Algorithms (GreedyDual-Size). Proc. USITS ’97, 193–206, 1997.

[14] Smith, Alan Jay. Disk Cache—Miss Ratio Analysis and Design Considerations. ACM Transactions on Computer Systems (TOCS) 3(3), 161–203, 1985.

[15] Ruemmler, Chris; Wilkes, John. UNIX Disk Access Patterns. Proc. USENIX Winter ’93, 405–420, 1993.