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 - η) * R[i] + η * 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 Related Work

2.1. Background

Caching is a standard mechanism for reducing latency in data-intensive systems. A cache stores a subset of objects from a larger universe and serves future requests directly from this faster memory tier whenever possible. Each request is either a hit (the requested object is already in the cache) or a miss (the object must be fetched from the backing store and possibly inserted into the cache). Over a finite request trace, the primary effectiveness metric is the hit ratio, defined as the fraction of requests that are served as hits, with the complementary miss ratio measuring the fraction of requests that incur a miss.

When the cache is full, a replacement policy decides which object to evict to make room for a newly requested one. Classical policies such as Least Recently Used (LRU) and Least

Frequently Used (LFU) exploit different forms of locality, and more sophisticated adaptive schemes (e.g. ARC) attempt to balance recency and frequency automatically. In this work, we adopt the standard setting of fixed-capacity object caches and regard a replacement policy as successful if, for a wide range of workloads and cache sizes, it achieves high hit ratios and small gaps to strong baselines such as LRU, LFU, and ARC, while remaining implementable with modest time and space overhead.

2.2. Existing Policies

We review the established caching policies selected as baselines for evaluation: Least Recently Used (LRU), Least Frequently Used (LFU), and Adaptive Replacement Cache[3] (ARC). We analyze their mechanisms in relation to our proposed Time-Decayed Caching (TDC) policy, highlighting key differences, similarities, and the specific scenarios where our approach is expected to outperform them.

Least Recently Used (LRU)

LRU is one of the most ubiquitous caching algorithms due to its simplicity and low overhead. It operates on the heuristic that items accessed recently are likely to be accessed again in the near future (temporal locality). The policy maintains a list of items ordered by access time; upon a cache miss, the item at the tail of the list (the least recently accessed) is evicted.

Comparison with Proposed Policy

- Similarities: Both LRU and TDC account for recency. In TDC, a recent access resets or significantly boosts an item's score, ensuring that recently requested items are unlikely to be evicted immediately.
- Differences: The fundamental difference lies in history retention. LRU is "memoryless" regarding frequency; it does not distinguish between an item accessed once and an item accessed a thousand times if both were last accessed at the same moment. In contrast, TDC integrates frequency by accumulating score mass from multiple accesses.
- Expected Performance: We expect TDC to outperform LRU on most workloads, especially ones where frequency is a strong predictor of future access. While LRU excels at adapting to sudden changes, it often suffers from "scan resistance" issues where a one-time scan of items flushes valuable popular content. TDC’s accumulated score mass provides resistance against such scenarios.

Least Frequently Used (LFU)

LFU operates on the premise that items with a high frequency of access in the past are likely to be accessed frequently in the future. It maintains a counter for every item, incrementing it upon every hit. When the cache is full, the item with the lowest frequency count is evicted.

Comparison with Proposed Policy

- Similarities: Both policies utilize frequency as a core metric. Mathematically, LFU can be viewed as a special case of Time-Decayed Caching where the decay rate approaches zero (infinite half-life), meaning past accesses never lose their weight.
- Differences: The critical flaw of standard LFU is its inability to handle "cache pollution" caused by non-stationary workloads. An item that was popular in the past but is no longer relevant may retain a high count and block new, currently popular items. TDC solves this by introducing the time-decay factor which ensures that old popularity mass naturally fades over time.
- Expected Performance: We anticipate similar results to LFU on purely stationary workloads where the popular items never change. However, on non-stationary workloads (where the popularity distribution shifts periodically), we expect TDC to yield significantly better hit rates. Our policy is designed specifically to "forget" outdated history, allowing it to detect and adapt to new hot sets much faster than LFU.

Adaptive Replacement Cache (ARC)

ARC is a sophisticated, self-tuning algorithm that dynamically balances between recency (LRU) and frequency (LFU). It maintains two lists: one for items accessed once (recency) and another for items accessed at least twice (frequency), along with "ghost" lists that track evicted items to adjust the cache partition size adaptively.

Comparison with Proposed Policy

- Similarities: Both ARC and TDC aim to solve the recency-frequency trade-off. Both are considered "scan-resistant" and high-performance policies.
- Differences: ARC achieves this balance structurally (by resizing lists), whereas TDC achieves it mathematically (through the half-life decay formula). TDC offers a simpler implementation model, requiring only a priority queue (min-heap) and a single scalar score per item, avoiding the complexity of maintaining four distinct lists and shadow directories.
- Expected Performance: We consider ARC the strongest baseline. We expect TDC to achieve competitive hit rates comparable to ARC across most traces.

Summary of Expectations

To summarize our comparative expectations, the proposed Time-Decayed Caching policy is designed to improve on existing literature (particularly LFU) in non-stationary environments. By mathematically coupling recency and frequency, TDC retains the high hit rates of frequency-based policies while matching the adaptability of recency-based ones.

Additionally, we expect TDC to surpass LRU in hit rate performance across most workloads because LRU ignores frequency entirely, whereas TDC's score accumulation preserves valuable, frequently accessed content. Against LFU, we anticipate a significant advantage on non-stationary workloads. Finally, compared to ARC, we expect TDC to remain competitive and comparable. While ARC achieves a similar recency-frequency balance through structural complexity, TDC offers a simpler implementation model while targeting the same high-performance metrics.

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.
- $r_1, r_2, \dots, r_t, \dots$ 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., $\mid \mathcal{C}(t) \mid = C$), 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 $\mathcal{O}(\mid \mathcal{C}(t) \mid)$ 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 $O(1)$), 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 $O(\log\ C)$ 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 $O(\log\ C)$.
- 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 $O(\log\ C)$, remove j from the hash table, and then insert the new object as above, for a total $O(\log\ C)$ time.

Therefore, each request is processed in $O(\log\ C)$ worst-case time, dominated by heap operations, compared to the $O(1)$ 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 $O(C)$: 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

This chapter aims to describe how the time-decayed caching policy is implemented in practice. We first outline the data structures and state maintained by the cache, and then present the core operations in pseudocode. A full C++ implementation is available as open-source code on GitHub [1].

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 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 to internal cache entries. This provides expected $O(1)$ 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:

- object: the identifier of the cached object,
- K: the current time-normalized key K_i associated with that object.

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 single access operation on objects. Each request in a trace corresponds to one call to ACCESS(object), which either hits or misses in the cache. The cache tracks only which objects are present and their time-decayed popularity scores.

On a hit, the cached entry's time-normalized key K_i is updated. On a miss, if the object has never been seen before, its first request at time t should give a decayed score $S_i(t) = 1$. By the definition, this corresponds to initializing

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

If the cache is full, the least popular entry is evicted before inserting the new object.

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.

```
procedure ACCESS(object):
  t ← t + 1 // advance time counter
  entry ← HASH_TABLE_SEARCH(hashTable, object)

  if entry ≠ NONE then // cache hit
    UPDATE_POPULARITY(entry)
    return HIT

  else // cache miss
    if SIZE(minHeap) = C then
      EVICT()

    entry ← new cache entry
    entry.object ← object
    entry.K ← exp(λ · t)
    HEAP_INSERT(minHeap, entry)
    HASH_TABLE_INSERT(hashTable, object, entry)

    return MISS
```

On a hit, we update the time-normalized key K_i for the corresponding entry. If the current time is t and the entry has key K_i, then after accounting for the decayed past and adding one new access, 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 $O(\log\ C)$time.

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

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

We aim to describe how we evaluate the proposed time-decayed cache replacement policy ("Proposed") against three baselines: LRU, LFU and ARC[3]. We first define the validation metrics, then describe the experimental design, and finally present the workloads and the choice of decay parameters.

Validation metrics and baselines

We compare four algorithms:

$$\mathcal{A} = \{\text{LRU}, \text{LFU}, \text{ARC}, \text{Proposed}\}.$$

A request trace is a finite sequence

$$R = (r_1, r_2, \ldots, r_T),$$

where r_t is the identifier of the object requested at time t, and T is the trace length.

For each algorithm $A \in \mathcal{A}$, cache capacity C, and trace R, we simulate the trace and record, for every request $t \in \{1, \dots, T\}$, a hit indicator

$$H_t(A, C, R) \in \{0,1\},$$

which is 1 if the request r_t is a cache hit under algorithm A with capacity C, and 0 otherwise.

The hit ratio and miss ratio are defined as

$$\mathrm{HR}(A, C, R) = \frac{1}{T}\sum_{t=1}^{T} H_t(A, C, R)\,,$$
$$\mathrm{MR}(A, C, R) = 1 - \mathrm{HR}(A, C, R).$$

The hit ratio HR is the main effectiveness metric.

To normalize performance across heterogeneous traces and cache capacities, we use a relative gap-to-best measure. Let traces be indexed by j, and capacities by an index c corresponding to a capacity level (defined precisely further in this section). For a fixed trace R_j and capacity level c, we define the best hit ratio among all algorithms as

$$\mathrm{HR}_{\max}(j, c) = \max_{A' \in \mathcal{A}} \mathrm{HR}\,(A', C_{j,c}, R_j),$$

where $C_{j,c}$ is the absolute cache capacity (number of objects) corresponding to level c on trace j.

For the proposed algorithm we define, on each trace j and capacity level c, the relative gap

$$\Delta(j, c, \text{Proposed}) = \frac{\mathrm{HR}(\text{Proposed}, C_{j,c}, R_j) - \mathrm{HR}_{\max}(j, c)}{\mathrm{HR}_{\max}(j, c)} \cdot 100\%.$$

A value $\Delta(j, c, \text{Proposed}) > 0\%$ means that the proposed policy is better than another policy.

To study adaptivity on non-stationary traces (i.e., traces with periodically changing popular items), we additionally use a sliding-window hit ratio. We fix a window length W. For $t \geq W$, the windowed hit ratio is

$$\mathrm{HR}_W(A, C, R, t) = \frac{1}{W} \sum_{\tau=t-W+1}^{t} H_\tau(A, C, R).$$

On non-stationary traces, we plot $\mathrm{HR}_W(A, C, R, t)$ over time for all algorithms $A \in \mathcal{A}$ and visually compare:

- the drop in hit ratio after each phase change, and
- the speed at which each algorithm's HR_W recovers.

This highlights how quickly the proposed policy adapts relative to LRU, LFU and ARC.

Experimental design

We assume a finite object universe

$$\mathcal{O} = \{o_1, o_2, \dots, o_N\},$$

where each o_i is a distinct object identifier. Each trace $R = (r_1, \dots, r_T)$ is a sequence of requests with $r_t \in \mathcal{O}$.

For a given trace R and length T, we define the footprint

$$U(R, T) = \mid \{r_t : 1 \leq t \leq T\} \mid,$$

the number of distinct objects requested in the first T requests. In our experiments, T is the full trace length; we write $U_j = U(R_j, T_j)$ for trace j with length T_j.

Cache capacities are expressed as percentages of the footprint. We fix a set of capacity fractions

$$\alpha_c \in \{0.01, 0.02, 0.04, 0.08, 0.16\},$$

corresponding to 1%, 2%, 4%, 8% and 16% of the footprint. For trace j and capacity level c, the absolute cache capacity (in number of objects) is

$$C_{j,c} = \lfloor \alpha_c \cdot U_j \rfloor.$$

All algorithms are evaluated on the same pairs $(R_j, C_{j,c})$.

We consider two main experiments:

1. **Overall performance experiment**:
   For each trace $R_j$and each capacity level α_c:

a. Compute U_j and the corresponding capacity

$$C_{j,c} = \lfloor \alpha_c \cdot U_j \rfloor.$$

b. For each algorithm $A \in \mathcal{A}$, simulate the full trace R_j with capacity $C_{j,c}$, record the hit indicators $H_t(A, C_{j,c}, R_j)$ and compute the hit ratio $\mathrm{HR}(A, C_{j,c}, R_j)$.

c. From the resulting set of hit ratios, compute $\mathrm{HR}_{\max}(j, c)$ and the relative gap $\Delta(j, c, \text{Proposed})$ for the proposed algorithm.

d. The statistics defined in the previous subsection are then computed over all values $\Delta(j, c, \text{Proposed})$ obtained in this experiment.

2. **Adaptivity experiment:**

To isolate adaptivity, we consider non-stationary traces with a known number of phases P, each of equal length L, so that the total length is

$$T = P \cdot L.$$

Each phase $p \in \{1, \ldots, P\}$ has its own hot set and request distribution; phase changes occur at times

$$t = L, 2L, \ \ldots, (P-1)L.$$

We fix a capacity C and window length W. Then for each algorithm $A \in \mathcal{A}$:

a. Simulate the full non-stationary trace R with capacity C, recording the hit indicators $H_t(A, C, R)$.

b. For each $t \geq W$, compute the sliding-window hit ratio

$$\mathrm{HR}_W(A, C, R, t) = \frac{1}{W} \sum_{\tau = t-W+1}^{t} H_\tau(A, C, R).$$

c. We then plot $\mathrm{HR}_W(A, C, R, t)$ over time for all algorithms. Adaptivity is judged qualitatively by the magnitude of the drop in HR_W immediately after phase changes, and the recovery speed (i.e., how quickly HR_W returns to its steady level in each new phase).

Workloads and decay parameter selection

We use four synthetic workloads and one real-world trace. Synthetic workloads let us control the popularity dynamics; the real trace demonstrates behavior under realistic access patterns. For all experiments, the baselines use standard parameter settings, and the proposed policy uses a decay parameter λ chosen according to the rules below.

Stationary synthetic workloads

In the stationary workloads, popularity does not change over time. In total, we will be using two workloads generated from a Zipf distribution over $\mathcal{O}$. We index objects as $o_1, \ldots, o_N$, and define

$$\mathbb{P}(r_t = o_i) \propto i^{-\alpha},$$

for indices $i \in \{1, \ldots, N\}$, where the rank i corresponds to object o_i. More specifically, the object with rank 1 will be hottest, rank 2 is colder, and so on. This gives us a heavy-tailed popularity: a few objects get a lot of requests, and there is a long tail of rarely used ones. In order for the probabilities to sum up to 1 and maintain the proportionality described above, we use the following probability formula

$$\mathbb{P}(r_t = o_i) = \frac{i^{-\alpha}}{\sum_{j=1}^{N} j^{-\alpha}}.$$

We will use a different choice of α for each of the two workloads (e.g., moderately and highly skewed values).

Non-stationary synthetic workloads

In the non-stationary workloads, popularity changes over time in a controlled way. We will be using two workloads that consist of P phases of equal length L, so that $T = P \cdot L$. For each phase $p \in \{1, \ldots, P\}$, we choose a hot set $H_p \subset \mathcal{O}$ of size k, with limited overlap between consecutive hot sets H_p and H_{p+1}. Within phase p, requests are generated as

$$r_t = \begin{cases} \text{a key chosen uniformly from } H_p \text{ with probability } p_{\text{hot}} \\ \text{a key chosen uniformly from } \mathcal{O} \setminus H_p \text{ with probability } 1 - p_{\text{hot}} \end{cases}.$$

The hot set thus “jumps” at times $t = L, 2L, \ldots$, creating clear phase changes.

Real-world trace

In addition to the synthetic workloads, we evaluate the algorithms on a real-world trace drawn from a web proxy cache. This serves as a realistic counterpart to the previous non-stationary workloads, since web-access traces are known to exhibit strong temporal locality and evolving hot sets as user interests shift over time.

Decay parameter selection

The proposed policy uses exponential decay with rate $\lambda > 0$, equivalently described by a half-life H:

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

We choose λ based on a desired memory horizon A (in units of requests), and then keep λ fixed across all traces within a given regime:

1. Stationary workloads
   Let T be the number of simulated requests per stationary trace. We choose a memory horizon

$$A = \beta \cdot T,$$

with a small β(for example $\beta = 0.01$, corresponding to a horizon of approximately 1% of the trace length). We then set

$$H = \frac{A}{3}, \lambda = \frac{\ln 2}{H}.$$

With this choice, hits that occurred about A requests ago have their weight reduced by a factor of

$$2^{A/H} = 2^3 = 8,$$

i.e. to roughly $1/8 \approx 12.5\%$ of their original contribution. In our setting, we consider contributions at this level (around ten percent of the original weight) to be effectively negligible, so A can be interpreted as the point where past hits become largely irrelevant.

2. Non-stationary workloads

   Let L denote the characteristic length of a stable hot phase. We choose a memory horizon A on the order of L (for example $A = L$) and again set

$$H = \frac{A}{3}, \lambda = \frac{\ln 2}{H}.$$

   This aligns the effective memory of the policy with the time scale over which each hot set remains stable.

5.2. Case Study: Synthetic Cache Traces

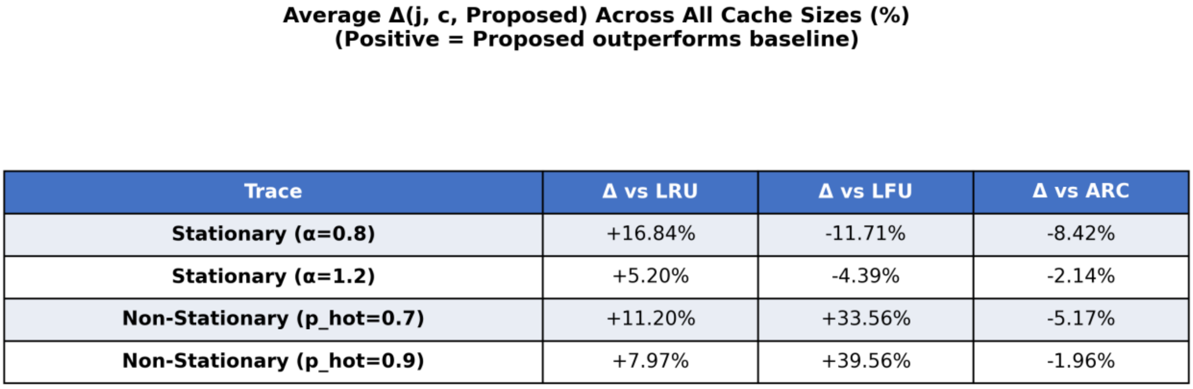
To evaluate the proposed Time-Decayed Caching (TDC) policy, we conducted experiments on four synthetic traces representing different access patterns. The traces were generated with 1,000 unique items and 50,000 requests each:

- Stationary Zipf (α = 0.8): A moderately skewed distribution where popular items receive more accesses, but the popularity is relatively spread out.
- Stationary Zipf (α = 1.2): A highly skewed distribution with strong concentration of accesses on a small set of popular items.
- Non-Stationary (p_hot = 0.7): A trace with 10 phases where the hot set changes between phases. In each phase, 70% of requests target the hot set (25% of items).
- Non-Stationary (p_hot = 0.9): Similar phase-based workload but with 90% of requests targeting the hot set, creating more pronounced locality shifts.

We compared the proposed TDC algorithm against three baseline policies: Least Recently Used (LRU), Least Frequently Used (LFU), and Adaptive Replacement Cache (ARC). Experiments were conducted for cache sizes ranging from 1% to 16% of the total item space.

Experiment 1: Hit Ratio Performance

Table 1 presents the average Δ(j, c, Proposed) values across all cache sizes, measuring the relative difference between the proposed algorithm's hit ratio and each baseline. Positive values indicate that the proposed algorithm achieves a higher hit ratio than the baseline.



Average Δ(j, c, Proposed) Across All Cache Sizes (%)
(Positive = Proposed outperforms baseline)

Trace	Δ vs LRU	Δ vs LFU	Δ vs ARC
Stationary (α=0.8)	+16.84%	-11.71%	-8.42%
Stationary (α=1.2)	+5.20%	-4.39%	-2.14%
Non-Stationary (p_hot=0.7)	+11.20%	+33.56%	-5.17%
Non-Stationary (p_hot=0.9)	+7.97%	+39.56%	-1.96%

Table 1

The results demonstrate TDC's performance characteristics across different workload types:

- **vs LRU**: TDC consistently outperforms LRU across all traces, with improvements ranging from +5.20% to +16.84% on stationary traces and +7.97% to +11.20% on non-stationary traces. This demonstrates that incorporating frequency information through time-decay provides significant benefits over pure recency-based eviction.

- **vs LFU**: On stationary traces, LFU outperforms TDC (Δ of -11.71% and -4.39%), which is expected since LFU is optimized for stable access patterns. However, on non-stationary traces, TDC significantly outperforms LFU with improvements of +33.56% and +39.56%. This highlights the critical weakness of LFU in dynamic environments where access patterns change over time.

- **vs ARC**: ARC, being an adaptive algorithm, performs competitively. TDC shows a very small gap to ARC on most traces (-8.42% to -1.96%), with the gap narrowing on non-stationary workloads where TDC's time-decay mechanism proves beneficial.

Experiment 2: Adaptability Analysis

To further investigate the adaptability of each algorithm, we analyzed the sliding window hit ratio over time on the non-stationary traces using a 5% cache size and a window of 500 requests.

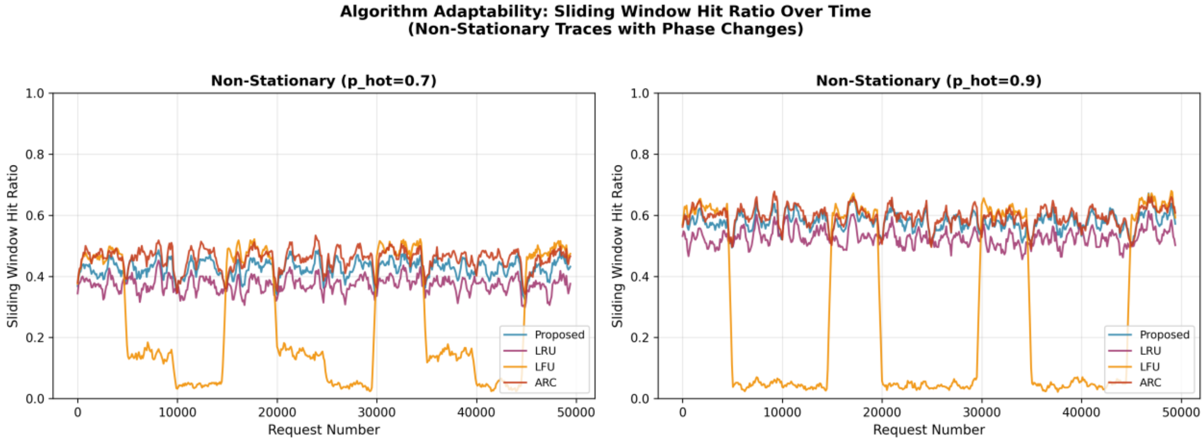


Figure 1

Figure 1 reveals several important observations:

- LFU struggles with phase changes: The LFU algorithm shows significant performance drops after each phase transition, as its frequency counters retain stale information from previous phases.
- LRU adapts quickly but has lower peak performance: LRU recovers rapidly from phase changes due to its focus on recency, but its steady-state performance is limited.
- ARC demonstrates strong adaptive behavior: The ARC algorithm balances recency and frequency effectively, showing competitive performance across phases. This is expected, as ARC is near-SOTA on most cache traces.
- TDC achieves balanced adaptation: The proposed algorithm demonstrates smooth transitions between phases while maintaining competitive hit ratios. The time-decay mechanism naturally deprioritizes stale frequency information, enabling faster adaptation than pure LFU while preserving frequency benefits over pure LRU. The TDC performance was closest to ARC.

Discussion

The experimental results validate the design goals of the Time-Decayed Caching policy:

- Unified recency-frequency handling: By using exponential time decay, TDC captures both recency (recent accesses have higher scores) and frequency (repeated accesses accumulate score) in a single, elegant mechanism.
- Adaptation: The time-decay factor allows the algorithm to naturally "forget" outdated access patterns, enabling smooth adaptation to workload changes.
- Performance: While TDC may not achieve the absolute best performance on some workloads (e.g., purely stationary high-skew traces where LFU excels), it consistently performs within a small margin of the best algorithm across diverse workload types.

6. Future Work

7. Conclusion

8. Bibliography

[1] C. Pângăleanu, „Time-Decayed Caching Implementation (GitHub Repository)”, 2025. Available: https://github.com/PanCat26/time-decayed-caching.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

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

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