# Sketching the Path to Efficiency: Lightweight Learned Cache Replacement

**Rana Shahout**[1]
Harvard University, Cambrdige, MA, USA

**Roy Friedman**
Technion, Haifa, Israel

## Abstract

Cache management policies are responsible for selecting the items that should be kept in the cache, and are therefore a fundamental design choice for obtaining an effective caching solution. Heuristic approaches have been used to identify access patterns that affect cache management decisions. However, their behavior is inconsistent, as they can perform well for certain access patterns and poorly for others. Given machine learning's (ML) remarkable achievements in predicting diverse problems, ML techniques can be applied to create a cache management policy. Yet a significant challenge arises from the memory overhead associated with ML components. These components retain per item information and must be invoked on each access, contradicting the goal of minimizing the cache's resource signature.

In this work, we propose *ALPS*, a light-weight cache management policy that takes into account the cost of the ML component. *ALPS* combines ML with traditional heuristic-based approaches and facilitates learning by identifying several statistical features derived from space-efficient *sketches*. ALPS's ML process derives its features from these sketches, resulting in a lightweight and highly effective meta-policy for cache management. We evaluate our approach over real-world workloads run against five popular heuristic cache management policies as well as a state-of-the-art ML-based policy. In our experiments, ALPS always obtained the best hit ratio. Specifically, ALPS improves the hit ratio compared to LRU by up to 20%, Hyperbolic by up to 31%, ARC by up to 9% and W-TinyLFU by up to 26% on various real-world workloads. Its resource requirements are orders of magnitude lower than previous ML-based approaches.

**2012 ACM Subject Classification** Information systems → Data streams

**Keywords and phrases** Data streams, Memory Management, Cache Policy, ML

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2023.34

**Supplementary Material** *Software (Source Code)*: `https://github.com/r4n4sh/sketch_learning` archived at `swh:1:dir:bcbd252fd90d7f6377a0443811576d59ca79e1fd`

**Funding** *Rana Shahout*: Supported in part by Schmidt Futures Initiative and Zuckerman Institute. *Roy Friedman*: Israel Science Foundation grant #3119/21.

**Acknowledgements** We thank Ohad Eytan for helping run Caffeine's simulator.

## 1 Introduction

Caching is a fundamental performance boosting technique, widely used by middleware, operating systems, databases, data-stores, edge servers, and content delivery networks [10, 14, 15, 21, 23, 34, 19, 17, 35, 12]. A cache improves the system's average response time by storing certain items closer to their consumers. This way, future accesses to cached items are served faster than serving them from their main storage. Caching of responses can also save communication and computations needed to re-calculate remote invocations.

[1] Corresponding author

Alas, caches usually cannot hold all accessed items, meaning that a *cache management policy* is needed to decide which items should be stored in the cache. An access to a cached item is called a *hit*; otherwise it is a *miss*. In particular, the cache management policy must predict what items are likely to be accessed in the future, in order to maximize the ratio of hits to all accesses (*hit ratio*), thereby minimizing the expected access latency of the entire system.

Traditionally, caching relies on heuristic approaches to identify patterns and statistically significant signals within the access history. A notable example is LRU [25], which operates under the assumption that the recency of an item's last access serves as a reliable indicator of its future access likelihood. LRU incorporates newly arrived items into the cache while evicting the least recently accessed item when the cache reaches its capacity limit.

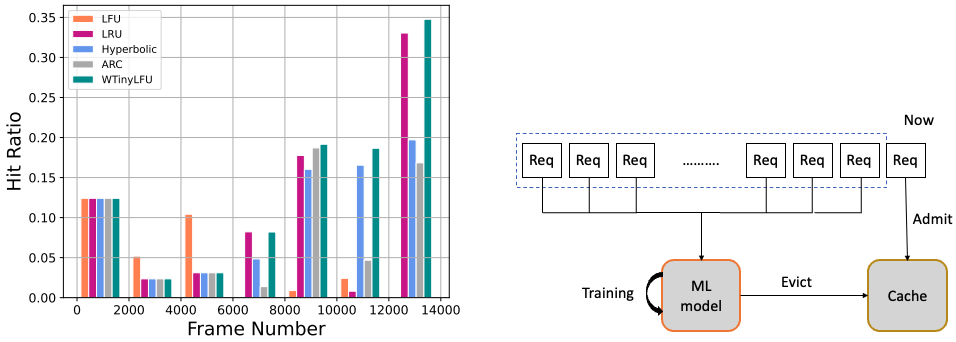
Frequency is another important signal, at least for certain workloads. The respective LFU policy [2] and its variants [13, 29, 3, 2, 20, 21] prioritize items based on their access frequency. To accommodate dynamic changes in item popularity over time, some variants incorporate aging or freshness mechanisms. However, predictions based solely on recency or frequency may not always yield good cache performance. The inclusion of inter-access or inter-reference, which captures temporal item access relationships, has shown promise in improving predictions. Further, it is now a common belief that adaptive combinations of recency, inter-reference, and frequency yield superior cache performance across a wide range of workloads [28, 39, 37, 33, 9, 20].

Given the success of machine learning (ML) in prediction tasks, we may apply ML for cache management. This involves training models to predict which items are likely to have the lowest hit rate and therefore should be prime candidates for eviction [47, 5]. However, a drawback of these ML-based approaches is the associated memory overhead. These methods store information per item and require invoking ML mechanisms for every access, resulting in substantial memory and computational requirements. Alternatively, CACHEUS [43] utilizes ML to decide which among a few cache management experts to use at any given moment. Yet, CACHEUS invokes ML on every access, and its learning and decisions are based directly on the access history.

Since caches are meant to serve as performance optimizations, one aspires to keep the cache's resource signature as low as possible. Also, learning and predicting are based on the raw access stream, which lengthens the learning process and makes predictions expensive.

**Motivation.** When analyzing the hit ratio performance of various widely deployed caching algorithms across a comprehensive set of workloads, we see that there is no silver bullet across all workloads. The effectiveness of different policies varies depending on workload characteristics and cache size. Moreover, as the workload evolves over time, the most suitable heuristic caching algorithm for the same workload may change. To investigate this dynamic behavior, we partition the access sequence into frames and assess the performance of different caching techniques within each frame. Figure 1a illustrates this analysis using the Mergep trace, where each frame consists of 10K accesses. It is shown that the winning heuristic approach changes as the trace progresses, but no single policy dominates across all frames. Notably (not shown here due to lack of space), subsequent frames often exhibit the same winning scheme. In summary, despite numerous attempts to identify an ideal caching policy, state-of-the-art algorithms demonstrate inconsistent performance across diverse workloads and even within the same workload, primarily due to their inherent variability.

The application of ML to cache management has been limited, despite its wide application across several domains. A previous approach, as illustrated in Figure 1b, involved utilizing ML to derive new caching policies. When the cache reaches full capacity and an item is accessed,




**(a)** Hit ratio.

**(b)** Existing ML based policies.

**Figure 1** (a) Hit ratio of different heuristic approaches as a function of the frame number on sampled frames using the Mergep trace with a cache size of 100K items (b) General architecture of existing ML based policies. The ML model is invoked on each access to advise which item to evict.

**Table 1** The allocated space of the metadata for LRB in different traces (values taken from [47]).

| Cache size | Wiki | A1 | A2 |
|---|---|---|---|
| 128 GB | 2.68 GB | 0.76 GB | 1.79 GB |
| 256 GB | 3.5 GB | 1.28 GB | 3.07 GB |
| 512 GB | 5.12 GB | 2.04 GB | 5.12 GB |
| 1 TB | 6 GB | 3 GB | 7 GB |

this approach leverages the ML model to make eviction decisions. However, it faces two primary issues. First, storing **per item** past information within a sliding memory window for training and prediction incurs significant memory overhead. Second, the prediction overhead associated with each eviction operation further adds to the computational burden. To quantify the memory overhead, Table 1 presents the metadata requirements for LRB [47]. The memory overhead is contingent upon the cache size and the specific trace, and it increases as the cache size grows larger.

**Our approach.** To reduce ML costs, ALPS divides the access stream into frames and applies ML to predict the most suitable heuristic caching policy for the subsequent frame. By doing so, we reduce computational overhead by executing ML only once per frame. This is significantly more efficient than applying ML on every cache access. Further, to maintain per-frame information memory efficiently, ALPS utilizes memory frugal sketches. These sketches serve as concise and structured representations of previously identified heuristic caching patterns. At the end of each frame, the condensed and structured information is fed into ALPS, which applies the predicted most appropriate policy to the next frame. This approach combines the benefits of known heuristic approaches with ML in an efficient manner. We highlight that ALPS learns from sketches' output rather than the raw access stream, which reduces memory overhead. The sketches employed in ALPS include HLL [22] for estimating the number of unique items accessed in a frame, Count-Min sketch [16] for estimating item frequencies and generating a frequency histogram, and the Space Saving algorithm [38] to identify heavy hitters. Finally, we developed a novel sketch called ISketch (Inter-Reference Sketch), which captures the inter-reference data of the most frequently accessed items within each frame and is fed into ALPS. We compared ALPS with established heuristic- and ML-based cache

management policies. Our analysis reveals that ALPS outperforms these approaches by achieving a higher hit ratio while significantly reducing memory overhead compared to LRB [47] and CACHEUS [43].

**Contributions.** We make the following contributions: (1) We design ALPS, a generic caching framework that achieves a high hit ratio while reducing memory overheads compared to prior ML-based approaches. (2) As a building step, we present ISketch, a novel algorithm for tracking inter-arrival times, which allows us to track recency during each frame. (3) We evaluate ALPS against well-known heuristic based cache algorithms and learning based algorithms. ALPS exceeds most efficient heuristic-based cache policies' hit rates, improving LRU by up to 20%, Hyperbolic by up to 31%, ARC by up to 9% and W-TinyLFU by up to 26% depending on workloads. In addition, ALPS decreases the memory overhead by an order of magnitude compared to the state-of-the-art ML-based approaches while improving the hit ratio by $6\% - 9\%$ compared to LRB and $7.5\% - 12\%$ compared to CACHEUS. (4) ALPS achieves a low training and inference overhead by extracting the most effective features from highly memory-efficient sketches and applying ML only once per frame.

To summarize our key insights, we argue that: (1) combining existing cache management policies in a smart way is likely to yield better improvements than inventing another one, (2) succinct sketches can capture data access patterns and serve as effective features for ML, (3) operating at time frame granularity rather than single access granularity yields more efficient ML based solutions.

# 2 Background and Related Work

## 2.1 Caching Algorithms

**Least Frequently Used (LFU)** [11, 44] aims at maintaining the most frequently used items in the cache. To that end, when the cache is full, LFU removes the item with the lowest reference frequency from the cache. This is done by tracking how many times each item is referenced in the cache.

Yet, in most practical workloads, the access frequency radically changes over time. Hence, there is no point in keeping an item in the cache once its popularity has faded, just because it was once very popular. As a result, LFU variants include aging algorithms or focus on a small window of the last W accesses alone, as done, e.g., in Window LFU (WLFU) [29].

**Least Recently Used (LRU)** [25] always inserts the last accessed item into the cache, and the Least Recently Used item is evicted when the cache is full. LRU adapts automatically to variations in data access patterns. Practical systems often only realize an approximation of LRU, e.g., through sampling [42] or Clock [4, 26] to reduce execution overheads and eliminate concurrency hot-spots. Segmented LRU (SLRU) [30] distinguishes between items that are temporarily popular and are accessed twice or more in a short period of time and those that are accessed just once during that period. LRU-K [39] combines concepts from LRU and LFU. 2Q [28] is an effective practical approximation of LRU-K.

**Hyperbolic** [9] is a recent proposal for combining frequency with recency in a holistic manner. With the same internal data structures, hyperbolic caching can act like a number of different eviction policies, changing its behavior based on the workload. The main idea is that an item's temporal priority is set to its frequency since it was last inserted into the cache, divided by the time that it spent in the cache, multiplied by a parameterized factor. The cache victim is the item with the lowest temporal priority.

**Adaptive Replacement Cache (ARC)** [37] is an adaptive caching algorithm that takes both recency and frequency of accesses into account. The cache is divided into two LRU lists, T1 and T2. T1 contains items that have been accessed only once, whereas T2 contains items that have been accessed multiple times after admission. In addition, ARC maintains ghost entries for both T1 and T2, which aid in deciding how to dynamically adapt the relative sizes of T1 and T2. Due to the fact that ARC uses an LRU list for T2, it is not possible to get the full frequency distribution of the workloads and perform well under LFU-friendly workloads.

**Adaptive W-TinyLFU** is the management policy of the Caffeine Java 8 cache [36], Go based Ristretto [18], and Rust based Moka [31]. W-TinyLFU has three parts: the Main cache, TinyLFU – an approximated LFU based admission filter, and a Window cache. New items are added to the Window cache; it can be kept using any known policy, but all known implementations employ LRU. The Main cache can use any cache management strategy, although known realizations employ SLRU. The filter utilizes a space-efficient sketch [16] to approximately track the access frequencies of a large number of items, well beyond the cache size. Whenever an item is evicted from the window cache, it is compared by the TinyLFU filter to the would be main cache victim; the item with the highest estimated frequency gets to be in the main cache and the other is deleted.

**Least Hit Density (LHD)** [5] is based on hit density, a workload-agnostic metric for ranking objects during eviction. It monitors objects online and uses conditional probability to predict their likely behavior. LHD predicts the expected number of hits per space unit consumed by each object (hit density) and eliminates objects that contribute little to the cache's hit rate. LHD does not rely on heuristics but rather rigorously models objects' behavior using conditional probability to modify its behavior in real time.

**Learning Relaxed Belady (LRB)** [47] approximates a variant of Belady's algorithm [7], called Belady's MIN (oracle) algorithm, using ML to find objects to evict based on past access patterns. To reduce the cost of ML, the authors first came up with a relaxed Belady algorithm that evicts an object whose next request is above a certain threshold but not necessarily the farthest in the future. For this, LRB keeps information **about an object** within a defined sliding memory window. The information within the sliding memory window is used for training and prediction. Thus, LRB invokes ML on each access, and the predictions are invoked once the full cache has to evict an object.

**CACHEUS** [43] identifies several cache management experts (ARC, LIRS, LFU, SR-LRU, and CR-LFU) and uses ML to predict which one to use at any given time based on workload primitive types. This is based on classifying workload primitives into: LRU-friendly, LFU-friendly, scan, and churn. CACHEUS uses online reinforcement learning with regret minimization to provide a caching method that aims to optimize for dynamically manifesting workload primitive types. CACHEUS invokes ML on each access and bases its learning and prediction decisions directly on the access history.

**MiniSim** [51] is a generic framework that uses multiple scaled-down simulations to explore candidate cache configurations simultaneously. It consists of multiple shadow caches, each activating a different management policy for a sample of the workload. Periodically, MiniSim checks which shadow cache works best, and switches the real cache to the policy that performed best among the shadow caches during the last such period. Additionally, MiniSim

**Table 2** Comparison of the sketches that appear in the paper. $\epsilon$ is the estimation accuracy parameter.

| Algorithm | Space | Update Time | Randomization | Comments |
|---|---|---|---|---|
| SS [38] | $O(\epsilon^{-1})$ | $O(1)$ | Deterministic | implemented according to [8] |
| HLL [22] | $O((\log\log D)/\epsilon^2)$ | $O(1)$ | Randomized | $D$ is distinct elements number |
| Count-Min [16] | $O(\epsilon^{-1}\log\delta^{-1})$ | $O(\log\delta^{-1})$ | Randomized | $\delta$ is the probability of failure |
| ISketch | $O(k)$ | $O(1)$ | Deterministic | $k$ is the number of entries |

performs a Talus-like [6] performance cliff removal transparently for complex policies. The main shortcoming of MiniSim is that to be memory and computationally viable, the sampling probability needs to be very small, which misses out on certain phenomena, resulting in sub-optimal behavior [21].

## 2.2 Sketches

A sketch is a space efficient data structure that provides a fast data synopsis of the dataset, often sacrificing accuracy for space frugality. The tradeoffs one has to consider in designing sketches are accuracy error, space consumption, and processing time complexity. Table 2 lists the analytical performance summary of several algorithms mentioned below.

**Space Saving (SS)** [38] finds the most frequently occurring elements in a data stream, a.k.a., *heavy hitters*. SS processes a stream of identifiers in order to determine their frequency. It keeps track of a collection of $\frac{1}{\epsilon}$ integer counters, each with its own unique item ID. When a new item is received, SS increments its counter, if it has one. Otherwise, SS gives the item a minimal-valued counter before incrementing it (disassociating the previous ID). As an example, suppose the smallest counter is associated with ID $x$ and has a value of 4; if $y$ comes and does not have a counter, it will take over $x$'s counter and increment it to 5 (thereby leaving $x$ without a counter). When an item's frequency is queried, SS returns the value of its counter if it has one, or the value of the minimal counter otherwise. Suppose the total number of insertions handled by SS is $Z$, then the sum of counters equals $Z$ and hence the minimal counter is at most $Z\epsilon$. Hence, SS frequency estimates have a maximum error of $Z\epsilon$.

**Count-Min Sketch** [16] can be used to estimate items' frequencies over a stream without explicitly remembering any identifiers. It consists of a set of $d$ independent hash functions $\{h_j()|j \in [1,\ldots,d]\}$ and a two-dimensional array of counters of width $w$ and depth $d$. To add an item $x$ with a value of $v_x$, we increment the counters at $CM[j,h_j(x)]$ by $v_x$ for $1 \le j \le d$.

A query on an item is returns the minimum of the respective counters. For a stream of size $N$, CM sketch guarantees that its frequency estimation is correct up to an additive $N \cdot \epsilon$-error with a probability of at least $1-\delta$ where $d = \log\delta^{-1}$.

**HyperLogLog (HLL)** [22] is a probabilistic data structure that counts the number of distinct elements in a multiset. HLL applies a hash function $h()$ to the identifier of each element, and remembers the maximal number of leading zeros in all hash values. The more leading zeros there are, the higher the cardinality is. If the bit pattern $0^{L-1}$ is at the beginning of the remembered value, a good estimate for the size of the multiset is $2^L$. HLL guarantees a relative accuracy of $1.04/\sqrt{m}$ given a memory budget of $m$ units.

## 3 ALPS DESIGN

This section presents the design of Adaptable Learned Policy Selection (ALPS), a framework that utilizes machine learning (ML) to dynamically switch between a number of cache management algorithms. The core objective of ALPS is to address the cache replacement problem by predicting the optimal heuristic-based algorithm and configuring the cache management policy for the subsequent time frame.

Formally, the prediction problem can be defined as follows: given the latest $W$ accesses, the task is to predict the cache management policy that maximizes the hit ratio in the next $W$ arrivals, where $W$ represents the frame size. This prediction-based approach offers several advantages. Firstly, it consistently outperforms heuristic-based methods in cache hit ratio. Secondly, since the prediction algorithm is executed once every $W$ accesses, the inference time is short, and the associated cost is amortized over $W$ accesses.

**ALPS structure.** Figure 2 illustrates ALPS's workflow. It comprises two independent data structures: the *Sketches component* (Section 3.1) and the *ML submodel* (Section 3.3). Unlike previous ML-based caching approaches, ALPS divides the access sequence into fixed-sized intervals called frames to facilitate its operation. The Sketches component consists of four sketches, namely SS, HLL, CM, and ISketch (a novel sketch introduced in this work to track inter-arrival times), as discussed in Section 3.1. With each item's access, these sketches capture relevant statistical indicators related to recency and frequency bias within the workload during the last frame, as depicted in Figure 2. At the end of each frame, the statistical indicators extracted from the sketches are used as input features for the ML mechanism. Following that, the sketches are flushed, and all their counters are reset.

To train the ML submodel, we perform supervised learning offline using real-world traces. Ground truths are derived using an established caching simulator [36]. To minimize system overhead, effective features are extracted from the data and utilized as input for the trained model.

A straightforward implementation involves maintaining only the metadata required by all policies and only maintaining a single copy of the data itself for the winning policy. This approach is feasible as metadata is typically much smaller than the actual data, often requiring only a counter or pointer per item. Hyperbolic, for instance, necessitates two counters, but they can be repurposed from LFU's frequency counter and a recency timestamp from LRU. Switching policies at the end of a frame simply involves changing the metadata version and employing a different replacement algorithm for eviction decisions. More sophisticated optimizations are left for future work.

**ALPS Update Workflow.** Each access to the cache triggers updates to all sketches within the Sketches component. According to Table 2, the update operation for HLL has a constant time complexity of $O(1)$. SpaceSaving (and ISketch (Section 3.2)) can be implemented with a linked list data structure by keeping items with equal counts in a group, resulting in an $O(1)$ update time. The $O(1)$ time complexity means the algorithm consistently executes a small number of operations, rather than requiring thousands. On the other hand, the update operation for CM has a time complexity of $O(\log \delta^{-1})$, where $\delta$, which is a tunable parameter, represents the configuration parameter for CM. By performing these updates, ALPS effectively maintains **per frame** information in the form of *statistical indicators* (Section 3.1). This approach differs from previous learning-based approaches that stored **per item** information.

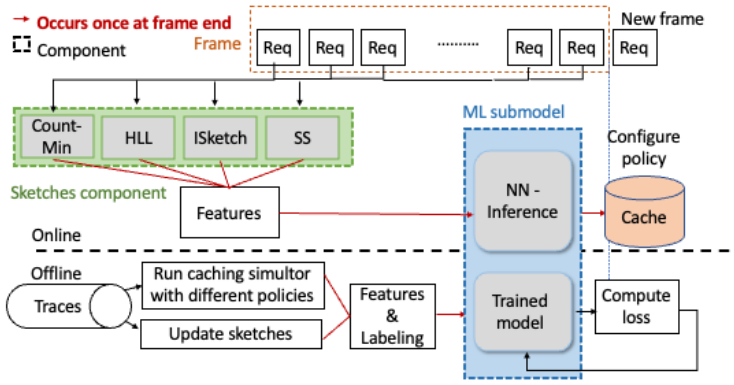



**Figure 2** Detailed architecture overview of ALPS.

## 3.1 Features and Sketches Component

*Recency* and *frequency* are two prominent signals utilized for designing cache management policies. Recency denotes the time that has elapsed since an item's last access. It provides insight into the likelihood of future accesses based on locality principles. Empirically, there is often a strong correlation between item access frequency and the probability of future access. Additionally, the number of unique items within a single frame offers a valuable indication of access distribution skew. Our objective is to identify the most informative features while maintaining high accuracy and imposing minimal space overhead. To assess the recency versus frequency bias within a workload frame, we define a set of *statistical indicators* as features for the ML submodel. In the context of ALPS, the following indicators are identified:

**Number of most accessed items:** refers to the count of heavy-hitters in the workload frame.

**Maximum frquency:** the maximal frequency count.

**Average frequency:** the average frequency among the most accessed items.

**Unique count:** the number of distinct items in the frame.

**Frequency distribution:** this vector captures the distribution of frequencies, with each entry $i$ representing the count of occurrences for frequencies with a value of $i$.

**Minimum inter-arrival time:** denotes the shortest inter-arrival time observed.

**Average inter-arrival time:** calculates the average inter-arrival time among items with the lowest inter-arrival time.

We classify the above into two categories based on their relation to recency or frequency. The set of recency indicators comprises the minimum and average inter-arrival times, whereas the set of frequency indicators includes the rest (unique count, number of most accessed items, maximum frequency, average frequency, and frequency distribution).

**Sketches Component.** We utilize several sketch data structures, namely Space Saving (SS) [38], Count-Min Sketch [16], HyperLogLog [22], and our novel ISketch. Each sketch is configured with identical values of $\epsilon$ and $\theta$, resulting in a total space overhead of $O(\epsilon^{-1} \log \delta^{-1})$, as depicted in Table 2 (all constants in the table are small). We maintain all sketches upon arrival and feed all of their information into the learning model. Consequently, the estimation error associated with each indicator is $\epsilon W$, where $W$ denotes the frame size. Our experiments, as discussed in Section 4, demonstrate that this estimation error has minimal impact on prediction accuracy.

**Deriving Statistical Indicators from Sketches.** For SS, item frequency is determined by maintaining a collection of counters, as described in Section 2.2. The *number of most accessed items* indicator is obtained by counting the items in the collection with a count exceeding $\theta W$, where $W$ represents the frame size and $\theta \in [0, 1]$ is a frequency threshold. Further, SS maintains two variables: the maximum and average counter values, which provide the *maximum frequency* and *average frequency* indicators, respectively. Using the HyperLogLog algorithm from [22], we can estimate the *unique count*. At the end of each frame, we generate the frequency vector from the CMS sketch array to obtain the *frequency distribution* indicator. Each entry $i$ represents the count of frequency $i$ occurrences. Section 3.2 below elaborates on the retrieval of the *minimum* and *average inter-arrival times* indicators using our ISketch sketch.

## 3.2 ISketch: Interarrival Sketch

To our knowledge, we are the first to use sketches to approximate recency in a workflow. To this end, we present the Interarrival Sketch (ISketch) algorithm, specifically designed for tracking inter-arrival times. The inter-arrival time of an item $x$ refers to the time elapsed between its last two occurrences within the system. This measurement has been widely recognized as one of the preferred methods for quantifying workload recency [27, 40]. However, maintaining precise inter-arrival times for each item consumes a significant amount of storage. Therefore, ISketch focuses on tracking inter-arrival times only for items with low inter-arrival values, as these items have the greatest impact on estimating workload recency. To accomplish this without knowing these items' identities, ISketch employs a Space Saving-inspired table structure with $k$ entries. Each entry consists of an item ID, inter-arrival time, and last arrival timestamp.

Upon the arrival of an item $x$, if $x$ has no allocated entry, we replace the item whose inter-arrival value is maximal in the table with $x$ and reset its associated fields. Alternatively, if $x$ already has an allocated entry, ISketch calculates its last inter-arrival time by subtracting the last arrival timestamp from the current timestamp. If the result is lower than the inter-arrival field, we update the inter-arrival value. The QuerySTAT() function retrieves the average and minimum inter-arrival times of all items in the trace whose inter-arrival times are shortest. We maintain the average inter-arrival time and minimum value with each arrival and satisfy QuerySTAT() by returning these pre-computed values.

ISketch includes the exact inter-arrival times of items that exhibit frequent arrivals and possess short recency periods. Specifically, when an item arrives at least twice and its inter-arrival time does not exceed the maximum time stored in ISketch, it is allocated an entry within the data structure. Once an item is inserted, it remains inside ISketch as long as its inter-arrival time remains less than the maximum value. This condition holds for items that consistently have a short inter-arrival time. On the other hand, ISketch replaces items with infrequent arrivals, resulting in their inter-arrival time no longer being tracked. Consequently, ISketch does not provide an error bound on inter-arrival times for unmonitored items. Moreover, low-frequency items encountered towards the end of the workload have a higher probability of being monitored by ISketch. This is because items that were frequently accessed in the past but have not been accessed for an extended period are likely to be evicted from ISketch to make room for items with shorter recent inter-arrival times. For a detailed understanding of the ISketch algorithm, refer to Algorithm 1 in the appendix.

▶ **Lemma 1.** *An item with an inter-arrival time* $< interArr_{max}$*, must exist in the SS table.*

▶ **Theorem 2.** *IS requires* $k(2\log W + \log\frac{W}{k} + 2\omega)(1 + o(1))$ *space where* $k$ *the number of entries,* $\omega$ *is the number of bits required to represent an item in* $\mathcal{R}$*, and* $W$ *is the frame size. ISketch performs updates and answers queries as well as computes* QUERYSTAT() *in* $O(1)$ *time.*

The proof of Theorem 3.2 is in the appendix.

## 3.3 ML submodel

We use a neural network (NN) as the ML submodel. In particular, we use a 4-layer fully-connected NN [24] with two hidden layers and ReLU activation that implements multi-class regression. Formally, denote the output of a 4-layer fully connected neural network as: $N_{i,j}(x) = A(A(x \cdot w_1 + b_1) \times w_2 + b_2) \times w_3 + b_3$, where $x$ is the input (statistical indicators), $w_1, b_1$ are the weight and bias vectors for layer 1 (first hidden layer), $w_2, b_2$ are the weight and bias vectors for layer 2 (second hidden layer), and $w_3, b_3$ are the weight and bias vectors for the output layer. The ReLU function $A$ applies a function $a$ on each element of an input vector: $a(x) = \left\{ \begin{array}{ll} x, & x \geq 0 \\ 0, & x < 0 \end{array} \right\}$. The submodel output, denoted $M_i, j(x)$, is defined as: $M_{i,j}(x) = H(N_{i,j}(x))$, where $H$ is the softmax function, which generates a probability distribution for our policy classes given their respective prediction scores. The result of $H$ is a vector with elements in $[0, 1]$ that all sum up to 1.

The training process of ALPS is depicted in Figure 2. To train the NN submodel ($model_{ml}$), we utilize traces from real-world workloads and employ supervised learning with a cross-entropy loss function. For feature generation and label creation, we partition each trace into frames and retain the four sketches, namely $SS_{train}$, $HLL_{train}$, $CM_{train}$, and $ISketch_{train}$. We also utilize a caching simulator [36] to simulate various heuristic caching algorithms. At the end of a frame, we take the statistical indicators from the sketches and use them as features. Ground-truth hit ratios for each caching algorithm are obtained from the simulator. Users of ALPS can either directly use our pre-trained model, utilize our model as a starting point for further training, or perform training from scratch using their own traces.

## 3.4 Frame Size Selection

Determining the optimal frame size is a critical component of the current challenge. In order to perform the task, the user must have a preliminary understanding of the temporal and scalar alterations within the data stream. Several research works have delved into the domain of detecting alterations within data streams, as discussed in [1, 32, 49]. These methodologies fundamentally depend on an understanding of the inherent probability distribution, which forms the basis of the incoming data stream. However, due to the inherently unpredictable dynamics of data streams, acquiring preliminary information is rarely straightforward.

Intuitively, when the frame size is small, the sketch counters are relatively small, lacking statistical significance. In such a scenario, the sketch counters are so close together that it becomes difficult to distinguish real differences between them. Such a situation results in an inaccurate approximation of the relevant statistical indicators. Conversely, exceedingly large frames may hide dynamic transformations occurring within a singular frame. In general, if a user assigns a frame size that either exceeds or falls short of the necessary scale, the frame fails to effectively reflect the latest workload changes. The notion of frame size can be discussed in two contexts: as a hyperparameter or as a model parameter. According to the first perspective, the frame size is adjusted to a value proportionate to the cache size. Alternatively, in the second perspective, the frame size is viewed as a parameter, and its

determination is subject to the model's learning capabilities. For our experimental setup, the default frame size is set to ten times the cache size; we analyze the effect of the frame size on the hit ratio in Section 4.

### 3.5 Putting All Together

We summarize all the steps of ALPS:

**Training:** (1) Divide the trace into frames and maintain the sketches for each frame. (2) Run the cache simulator with the various caching algorithms. (3) Train on each frame.

**Cache selection (inference):** (1) Query the NN submodel at the end of the frame. (2) Configure the cache policy according to the returned policy when it differs from the current cache policy.

ALPS pseudocode appears in Algorithm 2 in the appendix.

## 4 Evaluation

In this section, we compare ALPS with five heuristic caching algorithms: LFU, LRU, Hyperbolic, ARC, and W-TinyLFU, and with the state-of-the-art learning algorithms LRB [47] and CACHEUS [43].

### 4.1 Implementation

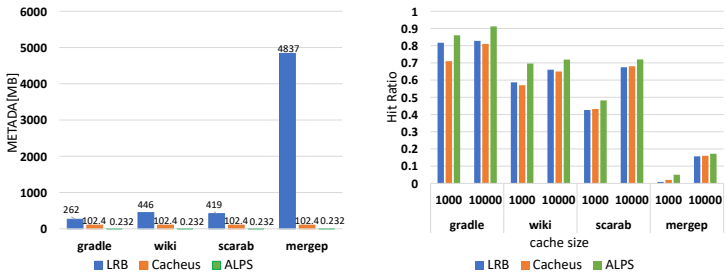
We have implemented ALPS in C++ and employed Caffeine's simulator [36] to derive cache hit ratios for various management policies. Our simulations encompassed five cache management algorithms, namely LFU [44], LRU [25], Hyperbolic [9], ARC [37], and W-TinyLFU [20]. The implementations of these algorithms were sourced from the Caffeine project's repository.

Regarding the SS, HLL, and CM sketches, we integrated the original author's implementation and realized ISketch's implementation ourselves. As previously mentioned, we utilized a fully connected MLP neural network (NN) consisting of two hidden layers with ReLU activation. The input features were constructed using statistical indicators introduced in Section 3.1, combined with a frequency distribution vector of length 32. The hidden layers comprised 512 neurons each, and the output size corresponded to the number of policies. The output layer consisted of 64 neurons. The total number of trainable parameters in this NN was computed as $38 \times 512 + 512 \times 512 + 512 \times 64 + 512 + 512 + 64 = 315,456$.

For training the NN submodel, we employed PyTorch [41] and trained it on various real-world workloads using supervised learning with cross-entropy error loss function. The labels for the training phase were obtained from the Caffeine simulator [36]. To be more specific, we partitioned each workload into frames based on the customizable frame size option. Subsequently, we executed the Caffeine simulator to calculate the per-frame hit ratio for each policy, identify the optimal policy within each frame, and assign the corresponding label to the frame.

### 4.2 Experimental Setup

The experiments were run on a DGX-A100 cluster using Slurm to schedule work. Each job was limited to a single A100 card with 40GB memory. It is important to note that ALPS can also be run on CPUs due to its low computational and memory overheads.




**(a)** Metada overhead. **(b)** Hit Ratio.

**Figure 3** (a) LRB, CACHEUS and ALPS memory overhead using various traces (b) LRB, CACHEUS and ALPS hit ratio using previously stated traces separately with two cache sizes $(1000, 10000)$. ALPS's frame size is 10000 items.

**Traces.** Our evaluation is based on real-world workloads from a variety of different domains: databases, analytic systems, transaction processing, search engines, and Windows servers are just a few examples. These workloads exhibit a wide range of underlying characteristics: some display a strong bias towards recency, others exhibit a bias towards frequency, and some present a combination of both. It is important to note that we meticulously partitioned the traces into separate subsets for training, validation, and testing purposes. To prevent overfitting, we trained our model on the interleaved prefixes of all traces. Subsequently, we conducted independent testing on each suffix of every trace. The workloads reported below include:

**P1–P14, Mergep:** 14 traces obtained from Windows NT workstations using Vtrace, which captures disk operations with the use of device filters. The traces were collected over a period of several months [37], containing about 491 million accesses.

**Gradle:** A trace from a distributed build cache, the Gradle project, that holds the compiled output so that subsequent builds on different machines can fetch the results instead of building anew. Since machines leverage local build caches, the distributed cache is recency-biased as only the latest changes are requested. It includes $\approx$ 2M accesses.

**Scarab:** A one-hour trace from Scarab Research of product recommendation lookups for several e-commerce sites of varying sizes worldwide. This trace includes $\approx$ 28M accesses.

**Wikipedia:** Wikipedia trace containing 10% of the traffic to Wikipedia during September–October 2007 [50]. This trace size is about 12 million accesses.

**Cache configurations.** Cache hit rate is the prime metric when evaluating caching algorithms. We compare our algorithm's performance across a range of cache and frame sizes when handling the above traces.

**Sketches overhead.** The memory overhead for ALPS is dominated by four sketches. We configure each sketch with the same $\epsilon$ and $\delta$ (if needed) values. We set the number of entries $(k)$ for ISketch to $\epsilon^{-1}$. As shown in Table 2, the overall space overhead of all sketches is $O(\epsilon^{-1} \log \delta^{-1})$.

**Comparison to LRB and CACHEUS.** We compare ALPS to LRB and CACHEUS, considering different cache sizes of 1000 and 10000. We examined the hit ratio and memory overhead of these algorithms. The implementation code for LRB was obtained from [46], while the code for CACHEUS was sourced from [48].

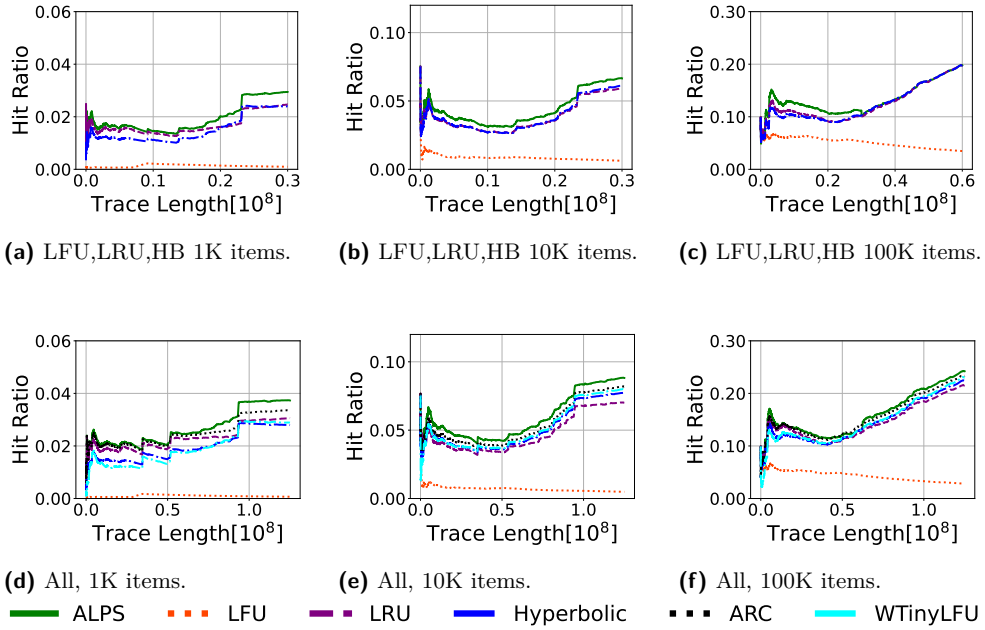
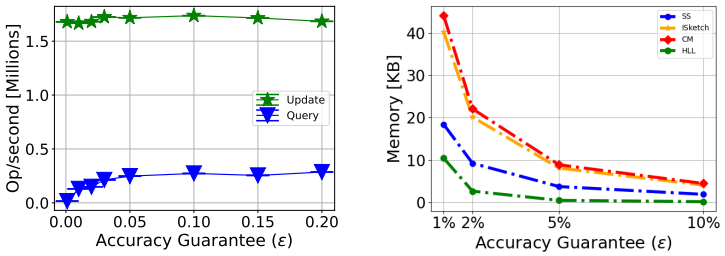



**Figure 4** Detailed comparison of ALPS against LFU, LRU, Hyperbolic, ARC and W-TinyLFU in 3 cache sizes when each frame contains $10 * \textit{cache size}$ accesses when ALPS is trained on (LFU, LRU, Hyperbolic), and on all the five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The plots show the hit ratio as a function of the trace length. The used trace in this experiment is **mergep**, which is a merge of 14 traces obtained from Windows NT workstations. The first column has a cache size of 1K items, the second column has a cache size of 10K items, and the third column has a cache size of 100K items.

Figure 3a illustrates the memory overhead of the LRB, CACHEUS, and ALPS policies across four distinct traces. In this experiment, the cache size was set to 100000 items, and the frame size for ALPS was also 100000 items. It is evident that ALPS demonstrated superior memory efficiency than the other algorithms. LRB maintains item-specific information within a sliding memory window, resulting in memory overhead that varies depending on the cache size and average item size in a trace. On the other hand, CACHEUS consumes approximately twice the cache size in memory for metadata, which tracks cache-resident items and historical items. Therefore, CACHEUS memory overhead primarily depends on cache size. In contrast, ALPS utilizes frame-specific information derived from compact sketches, making it insensitive to cache size and average item size in a trace. Consequently, ALPS's memory overhead remains constant across all traces, while LRB's memory overhead varies significantly, reaching magnitudes higher than ALPS.

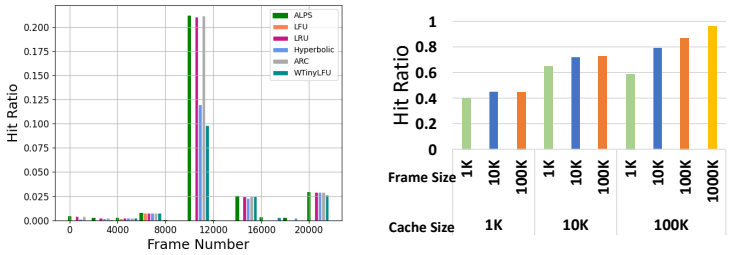
Additionally, Figure 3b presents a comparison of the hit ratios achieved by LRB, CACHEUS, and ALPS using cache sizes of 1000 and 10000 using previously stated traces separately. In this comparison, ALPS was trained using the five heuristic-based algorithms, and its frame size set to 10000 items. It is notable that ALPS consistently outperformed both LRB and CACHEUS in terms of hit ratio, particularly when the cache size was small.

**Hit ratio.** We conducted experiments to measure the hit ratio as a function of trace length using the Mergep trace (P1-P14). The page size for these traces was set to 512 bytes, while the frame size was defined as 10 times the cache size. The hit ratios for different cache sizes




**(a)** Sketches performance. **(b)** Sketches overhead.

**Figure 5** (a) Sketches update and query performance of the four sketches (SS, ISketch, CM and HLL) using the previously stated traces and with frame size of 10K items.(b) Sketches space comparison as a function of $\epsilon$ when the frame size to 10000 items.




**(a)** Hits per frame. **(b)** Hits vs. frame sizes.

**Figure 6** (a) Hit ratio of ALPS, LFU, LRU, Hyperbolic, ARC and W-TinyLFU per sampled frames when ALPS is trained on the five policies with cache of size 100K using mergep trace (b) Hit ratio of ALPS using scarab trace with different cache sizes (1K,10K,100K items) and different frame sizes.

are presented in Figure 4, where ALPS was trained on (LFU, LRU, Hyperbolic) as well as on all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The first column represents a cache size of 1K items, the second column corresponds to a cache size of 10K items, while the third represents a cache size of 100K items.

Our findings indicate that ALPS achieves the highest hit ratio when trained on all five policies with a cache size of 100K items. LFU has the lowest hit ratio. As expected, the hit rate for all algorithms increases with cache size. Notably, Figure 4 demonstrates that training ALPS on all five policies yields a more substantial improvement in the hit ratio than training it solely on (LFU, LRU, Hyperbolic). This is because ALPS benefits from a wider range of caching algorithms adapted to the frames' features. This observation is evident when comparing Figure 4a with Figure 4d, for example. Furthermore, the improvements in the hit ratio are more pronounced for small caches, as illustrated by the comparison of the columns in Figure 4. These enhancements arise from smaller caches, resulting in smaller frames and more frequent evictions. To that end, ALPS dynamically adjusts the cache management strategy, benefiting from the varying choices of evicted items by the heuristic-based policies.

In Table 3, we present the percentage increase in the hit ratio achieved by ALPS over LRU, Hyperbolic, ARC, and W-TinyLFU. The evaluations were performed using the aforementioned traces and a frame size of 10 times the cache size for three cache sizes: 1K items, 10K items, and 100K items. The results demonstrate that ALPS outperforms the other algorithms across all cache sizes. The most significant improvements were observed in the smallest cache size of 1K items. As mentioned earlier, ALPS enhances the hit ratio by predicting

**Table 3** Hit ratio increase by ALPS in three cache sizes (1K, 10K, and 100K items) against LRU, Hyperbolic, ARC and WTinyLFU when trained on (LFU, LRU,Hyperbolic), (ARC, W-TinyLFU), and with all the five policies using the previously stated traces with frame size of $10 * |cache|$ items.

| Training Set | Cache Size | LRU | Hyperbolic | ARC | WTinyLFU |
|---|---|---|---|---|---|
| LFU, LRU, HB | 1000 | 18.9% | 22.4% | – | – |
| LFU, LRU, HB | 10000 | 10.6% | 6.7% | – | – |
| LFU, LRU, HB | 100000 | 5% | 1% | – | – |
| ARC, WTinyLFU | 1000 | – | – | 10.4% | 18.8% |
| ARC, WTinyLFU | 10000 | – | – | 3.4% | 4% |
| ARC, WTinyLFU | 100000 | – | – | 2.6% | 5% |
| All | 1000 | 20% | 31% | 9% | 26% |
| All | 10000 | 19.3% | 8.4% | 2.17% | 4.84% |
| All | 100000 | 19.1% | 14.5% | 9.8% | 11.5% |

the best-performing cache policy for the next frame and dynamically configuring cache management accordingly. This behavior is particularly impactful in smaller caches due to smaller frames and increased evictions. This results in larger differences in heuristic-based policy choices. Specifically, the hit ratio percentage increase of ALPS compared to LRU is 20%, Hyperbolic is 31%, ARC is 9%, and W-TinyLFU is 26%.

Figure 6a depicts the hit ratios on the Mergep trace for ALPS, LFU, LRU, Hyperbolic, ARC, and W-TinyLFU when ALPS is trained on the five policies, using a cache size of 100K items, as a function of frame number. To enhance clarity, only samples of frames are displayed. The hit ratio obtained by ALPS in the sampled frames closely aligns with the highest hit ratio.

**Effect of frame size.** Figure 6b illustrates the impact of frame size on the hit ratio of ALPS with varying cache sizes (1K, 10K, 100K items) using the Scarab trace. As expected, both cache and frame sizes positively influence the hit ratio. Comparing the frame sizes with a 100K cache, we observe that smaller frames (relative to the cache size) lead to a reduced hit ratio. This is due to statistical indicators exhibiting minimal variations between two small frames. Conversely, extremely large frames may compromise the efficiency of the "winning" heuristic caching algorithm at the beginning of the frame. This indicates that it is advisable to avoid excessively long frames, as demonstrated in the case of 1K cache items.

**Inference time of the NN submodel.** Regarding the inference time of the NN submodel in ALPS, it is worthwhile to note that inference is executed only once per frame. The inference time represents the forward propagation duration. To ensure synchronized execution, we implement synchronization between the host (CPU) and the device (GPU) to record time only after GPU-based activity. This is achieved by performing a "GPU warm-up" by running dummy examples, which initializes the GPU and prevents it from entering power-saving mode during time measurement. As a result, we employ tr.cuda.event to measure GPU time. Table 4 in the appendix presents the inference time of the model for various training policies: (LFU, LRU), (LFU, LRU, Hyperbolic), (ARC, W-TinyLFU), and all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU). The values represent the mean of 300 iterations used to compute the inference time. It is observed that the inference time increases as the number of training policies expands. For instance, when ALPS is trained on five policies with a frame size of $10^4$, the inference is performed once every $10^4$ access and takes 0.309 milliseconds.

Note that while these measurements were obtained using a GPU, the inference in ALPS can also be executed on CPUs, due to its moderate computational and memory overheads.

**Sketches overhead.** Figure 5b presents the space occupied by the sketches for a given $\epsilon$, using the previously mentioned traces. The number of entries ($k$) for ISketch was set to $\epsilon^{-1}$. It is evident that as $\epsilon$ decreases, all sketches require more space. In comparison to the SS sketch, ALPS consumes additional space because ISketch maintains the exact item IDs instead of their fingerprints. This is necessary to accurately report inter-arrival times for items with low inter-arrival times.

The performance overhead of the update and query operations for the four sketches (SS, ISketch, CM, and HLL) is depicted in Figure 5a. The update performance is exceptionally efficient and remains unaffected by changes in $\epsilon$ values, as the update complexity is reported to be $O(1)$, as shown in Table 2. However, query performance in certain sketches depends on $\epsilon$. As $\epsilon$ decreases, more entries are included in the SS sketch, leading to slower query performance. Yet a query is executed only once per frame.

## 5 Conclusions

We have introduced ALPS, a lightweight cache management meta-policy that effectively combines ML with traditional heuristic-based approaches while addressing memory overhead challenges associated with ML components. The key idea behind ALPS is to divide the access sequence into frames and employ ML to predict the most effective cache management policy for each frame. Unlike existing ML caching algorithms that store item-specific information within a sliding window, ALPS utilizes space-efficient sketches to maintain per-frame information, making it a highly resource efficient algorithm. The features derived from these sketches are fed into the ML process, which is invoked only once per frame.

We have also introduced ISketch, an efficient algorithm for tracking inter-arrival times, enabling us to efficiently monitor frame recency. Subsequently, we presented the design, implementation, and evaluation of ALPS. Our experiments clearly demonstrate that ALPS significantly improves the hit ratio across various cache sizes when compared to traditional heuristic-based approaches, as well as state-of-the-art learning algorithms such as LRB and CACHEUS. Furthermore, the memory overhead of ALPS is orders of magnitude smaller than that of LRB and CACHEUS. As part of our future work, we plan to expand ALPS training to incorporate additional heuristic-based caching algorithms. This will enable the ML process to predict the optimal cache size and support weighted items. This will enhance ALPS's capabilities in optimizing cache performance. All code is available online [45].

## References


1 Charu C Aggarwal. A framework for Diagnosing Changes in Evolving Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 575–586, 2003. doi:10.1145/872757.872826.

2 Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *In Proc. of the 2nd Workshop on Internet Server Performance*, 1999.

3 Martin Arlitt, Rich Friedrich, and Tai Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. *Perform. Eval.*, 39(1-4):149–164, feb 2000. doi:10.1016/S0166-5316(99)00062-0.

4 Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proc. of the 3rd USENIX Conf. on File and Storage Technologies (FAST)*, pages 187–200, 2004. URL: http://www.usenix.org/events/fast04/tech/bansal.html.

5 Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 389–403, 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/beckmann.

6 Nathan Beckmann and Daniel Sanchez. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, 2015. doi:10.1109/HPCA.2015.7056022.

7 L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi:10.1147/SJ.52.0078.

8 Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy Hitters in Streams and Sliding Windows. In *The 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9, 2016. doi:10.1109/INFOCOM.2016.7524364.

9 Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 499–511, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

10 Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, and Willy Zwaenepoel. Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2006. doi:10.1007/11925071_1.

11 Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, pages 126–134, 1999. doi:10.1109/INFCOM.1999.749260.

12 Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX. In *Proc. of the 20th ACM/IFIP International Middleware Conference*, pages 14–27, 2019. doi:10.1145/3361525.3361533.

13 Ludmila Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical report, In HP Tech. Report, 1998.

14 Gregory V. Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2000. doi:10.1007/3-540-45559-0_1.

15 Wonil Choi, Bhuvan Urgaonkar, Mahmut Taylan Kandemir, and George Kesidis. Multi-Resource Fair Allocation for Consolidated Flash-Based Caching Systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pages 202–215, 2022. doi:10.1145/3528535.3565245.

16 Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58–75, apr 2005. doi:10.1016/J.JALGOR.2003.12.001.

17 Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 24–44, 2000. doi:10.1007/3-540-45559-0_2.

18 Dgraph. Ristretto: A High Performance Memory-Bound Go Cache, 2020. URL: https://github.com/dgraph-io/ristretto.

19 Xiaoming Du and Cong Li. SHARC: Improving Adaptive Replacement Cache with Shadow Recency Cache Management. In *Proc. of the 22nd ACM/IFIP International Middleware Conference*, pages 119–131, 2021. doi:10.1145/3464298.3493389.

20 G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage (TOS)*, 2017. doi:10.1145/3149371.

21 Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018. doi:10.1145/3274808.3274816.

22 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

23 Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A Trigger-Based Middleware Cache for ORMs. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2011. doi:10.1007/978-3-642-25821-3_17.

24 Simon Haykin. *Neural Networks: a Comprehensive Foundation*. Prentice Hall PTR, 1994.

25 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

26 Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an Effective Improvement of the CLOCK Replacement. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2005. URL: http://www.usenix.org/events/usenix05/tech/general/jiang.html.

27 Song Jiang and Xiaodong Zhang. LIRS: an Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems SIGMETRICS*, pages 31–42, jun 2002. doi:10.1145/511334.511340.

28 Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 1994. URL: http://www.vldb.org/conf/1994/P439.PDF.

29 G. Karakostas and D. N. Serpanos. Exploitation of Different Types of Locality for Web Caches. In *Proc. of the 7th Int. Symposium on Computers and Communications (ISCC)*, pages 207–212. IEEE, 2002. doi:10.1109/ISCC.2002.1021680.

30 Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 27(3):38–46, 1994. doi:10.1109/2.268884.

31 Tatsuya Kawano. A High Performance Concurrent Caching Library for Rust, 2021. URL: https://github.com/moka-rs/moka.

32 Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting Change in Data Streams. In *VLDB*, volume 4, pages 180–191, 2004. doi:10.1016/B978-012088469-8.50019-X.

33 Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001. doi:10.1109/TC.2001.970573.

34 Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and Analysis of a Container-Based Flash Cache for Compound Objects. *ACM Trans. on Storage (ToN)*, 13(3), sep 2017. A preliminary version appeared in ACM/IFIP Middleware 2015. doi:10.1145/3094785.

35 Tanu Malik, Xiaodan Wang, Philip Little, Amitabh Chaudhary, and Ani Thakar. A Dynamic Data Middleware Cache for Rapidly-Growing Scientific Repositories. In *Proc. of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pages 64–84, 2010. doi:10.1007/978-3-642-16955-7_4.

36 Ben Manes. Caffeine: A High Performance Caching Library for Java 8. *https://github.com/ben-manes/caffeine*, 2017.

37 Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST)*, pages 115–130, 2003. URL: http://www.usenix.org/events/fast03/tech/megiddo.html.

38 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-K Elements in Data Streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005. doi:10.1007/978-3-540-30570-5_27.

39 Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *ACM SIGMOD Rec.*, 22(2):297–306, jun 1993. doi:10.1145/170035.170081.

40 Sejin Park and Chanik Park. FRD: A Filtering Based Buffer Cache Algorithm that Considers both Frequency and Reuse Distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

41 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

42 Redis-Labs. Using Redis as an LRU cache, 2020. URL: `https://redis.io/topics/lru-cache`.

43 Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST)*, pages 341–354, 2021. URL: `https://www.usenix.org/conference/fast21/presentation/rodriguez`.

44 Dimitrios N Serpanos, George Karakostas, and Wayne Hendrix Wolf. Effective Caching of Web Objects Using Zipf's Law. In *IEEE International Conference on Multimedia and Expo (ICME): Latest Advances in the Fast Changing World of Multimedia (Cat. No. 00TH8532)*, volume 2, pages 727–730, 2000. `doi:10.1109/ICME.2000.871464`.

45 Rana Shahout. Open Source Code. URL: `https://anonymous.4open.science/r/sketch_learning-7A60`.

46 Zhenyu Song. webcachesim2: A Simulator for CDN Caching and Web Caching Policies, 2019. URL: `https://github.com/sunnyszy/lrb`.

47 Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 529–544, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/song`.

48 Systems Research Laboratory (SyLab). Cacheus Project, 2021. URL: `https://github.com/sylab/cacheus`.

49 Yingying Tao and M Tamer Ozsu. Mining Data Streams with Periodically Changing Distributions. In *Proceedings of the 18th ACM conference on Information and Knowledge Management*, pages 887–896, 2009. `doi:10.1145/1645953.1646065`.

50 Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks*, 53(11):1830–1845, 2009. `doi:10.1016/J.COMNET.2009.02.019`.

51 Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *USENIX Annual Technical Conference (ATC)*, pages 487–498, 2017. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger`.

# A ISketch and ALPS Algorithms

## A.1 ISketch Pseudocode

**Algorithm 1** ISketch.

```
    Init: ts ← 0, interArr_min ← 0, interArr_max ← 0, avrg ← 0
 1: function INSERT(x, ℓ)
 2:     ts ← ts + 1
 3:     if x is monitored then
 4:         x.interArr ← ts − x.lastArr
 5:         x.lastArr ← ts
 6:         update interArr_min, avrg
 7:     else
 8:         if Less than k items are monitored then
 9:             x.lastArr ← ts
10:             x.interArr ← ∞
11:             update interArr_min, avrg
12:         else
13:             Let x′ be the element with largest inter-arrival
14:             Start monitoring x instead of x′;
15:             x′.lastArr ← ts
16:             x′.interArr ← x.interArr
17:             update interArr_min, avrg
18:         end if
19:     end if
20: end function

21: function QUERYSTAT()
22:     return (interArr_min, avrg)
23: end function

24: function QUERY(x)
25:     if x is monitored then
26:         return x.interArr
27:     else
28:         return interArr_max
29:     end if
30: end function
```

## A.2 Proof of Theorem 3.2

The ISketch implementation is built on the CSS implementation [8] with $k$ entries. It maintains $k$ entries with three values: item ID counters, latest arrival timestamps, and inter-arrival times. In this implementation, we substitute the frequency counter with inter-arrival time and add the last arrival timestamp to each allocated entry in the ID-Index data structure, which adds $k \log W$ to the space overhead compared to CSS. We have $k(2 \log W + \log \frac{W}{k} + 2\omega)(1 + o(1))$ space in total. The implementation in [8] allows item additions and point queries in $O(1)$ time (w.h.p.). The ISketch implementation is symmetric to CSS, with the exception that we replace the maximal counter rather than the minimal counter and keep the average and minimal inter-arrival time updated on each arrival, which also takes $O(1)$ time. QUERYSTAT() simply returns the average and minimum inter arrival times that have been kept. In total, the three operations take $O(1)$ time.

## A.3 ALPS Pseudocode and Inference Time

**Algorithm 2** ALPS.

Initialization: $fs \leftarrow 0$, *initialize* $model_{ml}, SS, HLL, ISketch, CM,$
$SS_{train}, HLL_{train}, ISketch_{train}, CM_{train}$.
**function** TRAIN()
  **for** $trace \in traces$ **do**
    *break trace into frames*
    **for** $req \in frame$ **do**
      $UpdateSketches(SS_{train}, HLL_{train}, ISketch_{train}, CM_{train})$
    **end for**
    Run caching simulator
    **if** frame ends **then**
      $features =$ *Extract from* $SS_{train}, HLL_{train}, ISketch_{train}, CM_{train}$
      $labels =$ *Get labels from the caching simulator*
      *train* $model_{ml}$ *with* $(features, labels)$
    **end if**
  **end for**
**end function**

**function** UPDATE($Req_i$)
  $fs \leftarrow (fs + 1) \mod W$
  $SS.Add(Req_i)$
  $HLL.Add(Req_i)$
  $ISketch.Add(Req_i)$
  $CM.Add(Req_i)$
  **if** $fs \mod W = 0$ **then**
    $ConfigureCache()$
    *flush sketches*
  **end if**
**end function**

**function** CONFIGURECACHE()
  $features = GetFeatures(SS, HLL, ISketch, CM)$
  $policy_{new} = model_{ml}(features)$
  **if** $Cache.GetPolicy() != policy_{new}$ **then**
    $Cache.SetPolicy(policy_{new})$
  **end if**
**end function**

**Table 4** Mean inference time of ALPS when trained on (LFU, LRU), (LFU, LRU, Hyperbolic), (ARC, W-TinyLFU), and finally with all five policies (LFU, LRU, Hyperbolic, ARC, W-TinyLFU) with frame size of 10000 items. Inference is executed only once per frame.

| Trained Policies | Mean Inference Time |
|---|---|
| LFU, LRU | 0.304 milliseconds |
| LFU, LRU, Hyperbolic | 0.308 milliseconds |
| ARC, WTinyLFU | 0.3026 milliseconds |
| All | 0.309 milliseconds |