Performance Evaluation 169 (2025) 102488

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Optimizing latency for caching with delayed hits in non-stationary environment

Bowen Jiang, Yubo Yang, Bo Jiang *

School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China,

ARTICLE INFO

Keywords:
Cache algorithm
Delayed hits
Non-stationary
Recurrent neural network
Multilayer perceptron

ABSTRACT

Caching plays a crucial role in many latency-sensitive systems, including content delivery networks, edge computing, and microprocessors. As the ratio between system throughput and transmission latency increases, delayed hits in cache problems become more prominent. In real-world scenarios, object access patterns often exhibit a non-stationary nature. In this paper, we investigate the latency optimization problem for caching with delayed hits in a non-stationary environment, where object sizes and fetching latencies are both non-uniform. We first find that given known future arrivals, evicting the object with the larger size, a higher aggregate delay due to miss and arriving the farthest in the future brings more gains in reducing latency. Following our findings, we design an online learning framework to make cache decisions more effectively. The first component of this framework utilizes historical data within the training window to estimate the object's non-stationary arrival process, modeled as a mixture of log-gaussian distributions. Subsequently, we predict future arrivals based on this estimated distribution. According to these predicted future arrivals, we can determine the priority of eviction candidates using our defined rank function. Experimental results on four real-world traces show that our algorithm consistently reduces latency by 2% – 10% on average compared to state-of-the-art algorithms.

1. Introduction

Caching is very popular in many systems such as content delivery network (CDN) [1], wireless communication network [2], and recommendation systems [3]. When the requested content is available in the cache near the user, it results in a cache hit, reducing the user's perceived latency. If the content is not in the cache, a cache miss occurs, requiring the content to be fetched from a remote server, significantly increasing latency. The capacity of a cache is typically much smaller than the total number of object types, making it essential to select which objects to store carefully. This allows users' requests to be directly satisfied by their nearby caches as much as possible, without the need to retrieve them from elsewhere. In other words, this directly relates to the metric known as the hit ratio, defined as the proportion of user requests that are directly satisfied by the cache. In recent years, numerous studies [4–11] have focused on improving the hit ratio, operating under the assumption that the subsequent request for a missed object suffers zero delay. The assumption shows that the delay in retrieving missing requests from the remote server is negligible compared to the time interval between two consecutive requests for the same object. However, the assumption of zero delay becomes unrealistic, particularly in high-throughput systems where multiple requests for the same object can occur before it is retrieved from the remote server. For example, the fetching time might be greater than 100 ms to retrieve the missing content

* Corresponding author.
E-mail addresses: bwjjhc@sjtu.edu.cn (B. Jiang), yyb515616@sjtu.edu.cn (Y. Yang), bjiang@sjtu.edu.cn (B. Jiang).

https://doi.org/10.1016/j.peva.2025.102488
Received 26 September 2024; Received in revised form 2 April 2025; Accepted 5 May 2025
Available online 20 May 2025

from the remote server, while about 1 million requests for objects are sent during the fetching process [12]. The latency experienced by these subsequent requests during the fetching process becomes increasingly significant as the ratio between system throughput and latency evolves larger.

The unrealistic assumption of zero delay has inspired the exploration of novel algorithms to optimize the latency in cache problems. The concept of the **delayed hit** is first proposed in [13], which claims that during the period when a missing file is being retrieved from the remote server, the subsequent requests for this file are queued to be satisfied until the missed object is fetched. Thus, the subsequent requests suffer delayed hits. The experiment further revealed that traditional algorithms aimed at maximizing hit ratio do not effectively reduce latency, demonstrating that optimizing hit ratio and latency in the presence of delayed hits are not equivalent. The heuristic algorithms incorporated with existing algorithms LRU and LHD [14] are further proposed to optimize the total latency with delayed hits when the size and latency of requests are uniform. The lower bound of caching with delayed hits for deterministic solutions is introduced in [15]. [16] accounts for variations in request sizes and assumes the request arrival process follows a Poisson distribution. Under this assumption, the expression for the mean latency experienced by requests for a particular object is derived as a weighted sum of cache miss latency and delayed hit latency. Building on this, a novel timer-based ranking function is proposed to reduce overall latency. The online caching problem with delayed hits and bypassing is addressed in [12], where user requests can be bypassed and served directly by the remote server. [12] argues that estimating total latency due to a miss in [13] is aggressive while using the upper bound of fetch latency is conservative. To balance these approaches, they propose a well-defined combination of aggregate delay and upper bound to estimate the weight of each object, achieving a trade-off between the two methods. By continuously learning the weights of different objects, traditional algorithms can also effectively address the issue of delayed hits.

Although the aforementioned work addresses the issue of delayed hits in caching and proposes algorithms to minimize overall latency in various scenarios, several essential aspects of delayed hits still warrant further consideration. Firstly, the size of objects in real-world traces can go beyond eight orders of magnitude [5]. Thus, the fetching delay of various objects could be quite different, and assuming that the latency for all missing requests is identical in [13] does not fully reflect the real-world scenario. Although it is assumed that request sizes vary in [12], its method for estimating object weights cannot effectively differentiate between objects with equal weights but different sizes. What is more, the real-world requests are often non-stationary, and regular stochastic processes such as the Poisson process [16] cannot characterize its features thoroughly. The above considerations inspire us to ask the following question in this paper:

For caching with delayed hits when the size and latency of objects are both non-uniform, can we design an online algorithm that shows superior performance over the existing algorithms in a non-stationary environment?

To answer this question, we first focus on the offline scenario that assumes the total request sequences are known. We find that evicting the object with higher aggregate delay caused by a miss, arriving farthest in the future, and with a larger object size can further reduce latency. Following the insights from our findings, we design an online learning framework with recurrent neural network (RNN) and multilayer perception (MLP) to learn the objects' non-stationary arrival process, which is modeled as a mixture of log-gaussian distributions. The combination of RNN and a mixture of log-gaussian distribution can approximate any non-stationary arrival process [17]. The learned arrival process enables us to predict future arrivals and make cache evictions based on future predictions. In summary, our contributions in this paper are as follows:

- For the scenario where all request sequences are known, we find that evicting the object with a larger size that suffers the higher aggregate delay due to missing and will arrive the farthest in the future offers a new insight to reduce latency. (see Section 3)
- Inspired by our findings, we develop an online learning framework to learn the arrival process, which is modeled as a mixture of log-normal distributions. We can generate future arrival predictions by continuously sampling from these learned mixture distributions and stacking the samples. Using these predicted future arrival times, we can establish the priority of eviction candidates based on our defined ranking function. (see Section 4)
- We conduct simulations on a synthetic dataset and four real-world traces. The results show that compared with the state-of-the-art algorithms considering delayed hits, our algorithm consistently reduces latency by 2%–10% on average. (see Section 5)

The rest of this paper is structured as follows. Section 2 discusses the related work, including cache algorithms and delayed hits. In Section 3, we model the delayed hits and show our findings motivated by a toy example. Then, we propose our learning framework in Section 4. Moreover, in Section 5, we set up our experiment and discuss the simulation results. Section 6 gives conclusions.

2. Related work

2.1. Cache algorithm

The term cache was first introduced in computer systems to describe a type of memory that can be accessed rapidly but typically has limited capacity [18]. Even small caches can significantly enhance system performance by exploiting the correlation in memory access patterns. Over time, caching has become widespread across various domains, including content delivery networks [19], edge computing [20], recommendation systems [3], and web pages [21], among others. However, due to the cache's limited capacity, it can only store a small subset of all possible objects, making it necessary for caching algorithms to decide which contents to retain.

When a request arrives, if the requested object is found in the cache, it results in a hit; otherwise, it is a miss. In the case of a miss, the object must be retrieved from a remote server or neighboring caches, leading to additional processing overhead and longer response times, which can degrade the Quality-of-Service and overall system performance. The primary metric used to evaluate a caching algorithm is the **hit ratio**, defined as the proportion of requests directly served by the cache.

To determine what to store in the cache, caching algorithms must be particularly attuned to the characteristics of the request arrival process. Some traditional studies assume requests follow the well-known Independent Reference Model (IRM), where content requests are independently drawn according to static (stationary) probabilities. Under this assumption, numerous algorithms have been proposed to maximize the hit ratio, as seen in works such as [7–11]. However, it has become evident that the assumption of static request probabilities does not accurately reflect real-world scenarios [22]. In practice, requests that are frequent during one time period may become infrequent or even vanish in the next, while new content requests continuously emerge. Consequently, the arrival characteristics of requests are often **non-stationary**. To address this, some studies have specifically designed caching algorithms tailored for non-stationary request arrival processes [23–25].

Over the years, numerous caching algorithms have been proposed to utilize various object features for improved cache decision-making. Different LRU variants [26–28] apply diverse interpretations of recency to determine eviction candidates. Some algorithms combine both frequency and recency to rank the objects in unique ways [29,30], while others factor in both frequency and object size. Some studies design caching algorithms within the framework of online learning. By modeling the caching problem as an online learning task, they develop algorithms using techniques such as online gradient descent [31–33] or the follow-the-perturbed-leader approach [34,35], among others. With the rise of neural network technology, people have also begun to design caching algorithms based on machine learning [5,36–42]. Several studies have focused on predicting an object's features, including residual time [5,36–39] and object popularity [43,44]. By accurately predicting residual time, the cache algorithm can approximate Belady's algorithm [45], which optimally evicts the object that will arrive the furthest in the future. Several studies have employed reinforcement learning to select among multiple simple cache algorithms. For instance, LeCaR [46] utilizes two cache algorithms including LRU and LFU. LeCaR can follow the cache decision made by LRU or LFU each time according to the different weights of these two algorithms. Another work models the request probability distribution and bases its decisions on this distribution. For example, LHD [14] utilizes the request probability distribution to compute hit density (hits per space consumed) as a criterion for eviction. Specifically, LHD learns the request probability as a function of object age and then adjusts this probability by considering the object's size to determine the hit density.

Some recent studies have considered scenarios involving variable object sizes. Greedy Dual Size Frequency (GDSF) [29] is among the first algorithms to consider size when making cache replacement decisions by combining recency, frequency, cost, and size. It assigns each item a dynamic score based on these factors and removes the items with the lowest scores when adding new content. Recently, three notable size-aware caching strategies have been proposed: AdaptSize [47], LHD [14], and LRB [5]. AdaptSize adopts a probabilistic, size-based admission policy where larger items are less likely to be admitted than smaller ones. Specifically, it utilizes Markov models to refine and optimize the admission process. LHD [14] introduces the concept of hit density, which combines frequency and size metrics to evaluate each object's contribution. It estimates the hit density for each item and filters out those that have minimal impact on the cache's overall hit ratio. LRB [5] is an advanced cache replacement policy designed to maximize hit ratios and reduce bandwidth consumption in CDN. It introduces a variant called relaxed Belady, where the algorithm evicts objects whose next access occurs beyond a defined Belady boundary. The LRB policy employs machine learning techniques to emulate the relaxed Belady algorithm, which has proven to be both practical and highly effective in achieving superior hit ratios. In addition, by making minimal modifications to the popular TinyLFU [48] admission policy to handle variable-sized items, [49] significantly improves the overall hit ratio under variable object sizes.

2.2. Delayed hit

Many existing algorithms typically assume that user-perceived latency is negligible (i.e., zero latency) in case of a cache miss. Although this assumption is common in the literature, recent studies suggest that assuming zero latency can degrade the caching algorithms' performance when the goal is to minimize user-perceived latency [13,16]. The delayed hit is first proposed in [13] and the authors find that in the scenario of delayed hit, the goal of optimizing the hit ratio in traditional work is not equivalent to minimizing delay. The authors point out that as the ratio of network throughput to fetch delay becomes larger and larger, the delay hit problem becomes more prominent in many scenarios such as CDN [19]. For offline scenarios, the authors propose a near-optimal algorithm based on Minimum-Cost Multi-Commodity Flow (MCMCF) [50]. For online scenarios, the authors propose a new ranking function that takes into account the aggregate delay caused by a miss and the time interval from the current time to the next arrival time. However, the authors do not consider the effect of object size on the caching decision when designing the algorithm. The work in [16] considers scenarios with different request sizes while assuming that the objects' arrival process follows the stationary Poisson distribution. The authors derive the expression of the mean latency experienced by a specific content theoretically. Based on this expression, a new ranking function is given, which combines the arrival rate of the content, the aggregate delay caused by a miss, and the size of the request. The author [12] claims that the method of predicting the actual aggregate delay in [13] is radical, and they estimate the actual aggregate delay as the combination of the average aggregate delay of all the past requests and the upper bound of the total latency caused by the object's miss. Then, the existing general file caching algorithms can handle the delayed hits by constantly updating the weight of each requested file, which is this novel estimation.

2.3. Arrival process estimation

Estimating arrival processes is crucial for many systems, including content delivery networks [44,51,52] and transportation systems [53–55] and so on. In content delivery networks, much of the existing works assumes a Poisson arrival process [8,10,56–60],

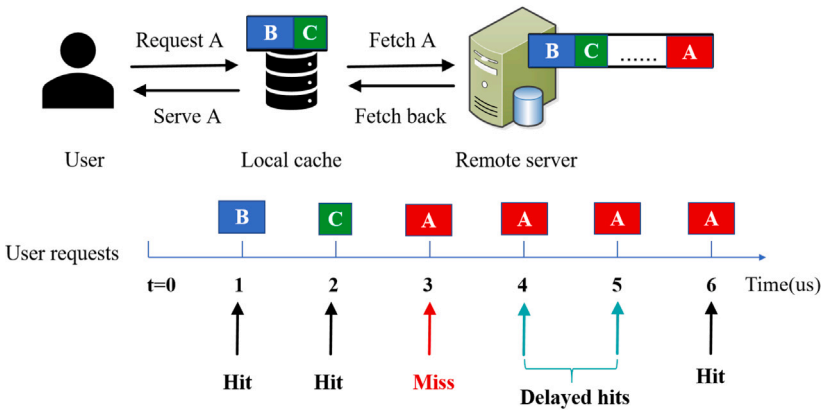

Fig. 1. An example of a cache-enabled network, where the hit, miss, and delayed hits occur.

estimating the arrival process essentially an estimation of the arrival rate. A straightforward way to do this is by taking the reciprocal of the average historical inter-arrival time, which is an unbiased estimator [61]. [44] does not directly estimate the distribution of inter-arrival times; instead, it employs a Long Short-Term Memory (LSTM) framework that uses historical request arrivals to predict future arrivals for maximizing the hit ratio. [62] proposes a survival-based approach using RNN to directly model inter-arrival times and partially observed information, outputting the parameters of distribution instead of point estimates for predicting purchase patterns across multiple products. [55] implies the Pareto distribution to fit the inter-arrival time distribution and then derives the hit ratio expression in the vehicular edge caching problem. [63] proposes fitting unknown inter-arrival time distributions with a mixture of Gamma distributions and then learning the mixture parameters. [51] approximates the system's average age of information (AoI) using the first and second-order moments of unknown inter-arrival time distributions since moments are easier to obtain via counting and statistics. [52] focuses on estimating Transmission Control Protocol (TCP) flow inter-arrival times by fitting them to a Weibull distribution; a window-based approach then detects anomalies through significant deviations in the Weibull parameters. [53] collects passenger inter-arrival data from multiple subway stations, fits eight distributions (including the newly introduced Hyper-Erlang distribution, HErD) via maximum likelihood estimation, and identifies HErD as the best-fitting model; an improved parameter estimation method—-based on long-term peak-hour volume and peak-hour factor—is further proposed to forecast future inter-arrival distributions for capacity design.

3. Modeling delayed hits and motivation

3.1. Delayed hits

We consider a cache-enabled network where there is a user, a local cache close to the user, and a remote server far away from the user. Time is divided into units of size, and there is only one request in each time slot. Denote the size of each object i as s_i, and the delay to fetch object i from the remote server when a miss occurs is d_i. For the request r_i^t for object i at time t, there are three cases to handle it:

- **Hit.** If r_i^t is already in the local cache, then it can be served immediately with no latency.
- **Miss.** If not, it will experience a delay of d_i in retrieving this object from the remote server.
- **Delayed Hit.** If r_i^t is already in the fetch process, then it is queued to be served until the object is fetched into the cache. Assume the last missing time before time t for content i is t'. Thus, the corresponding latency of r_i^t is $t' + d_i - t$.

It can be seen that all the same requests for object i arrive during $[t', t' + d_i)$ suffer delayed hits. We denote the D_i as the aggregate delay to capture the total latency caused by a miss for object i as follows:

$$D_i = d_i + \sum_{1 \le \tau \le d_i - 1} (d_i - \tau)\, \mathbb{I}(r^{t+\tau} = i), \tag{1}$$

where $r^{t+\tau}$ is the requested object at time $t + \tau$ and $\mathbb{I}(r^{t+\tau} = i)$ equals one if the requested object is i. In fact, the aggregate delay defined in (1) is related to time t. However, for the sake of simplicity, we do not explicitly include t in the definition of D_i. Once the missed object is retrieved, we should decide which cached object to evict to make room for the new one. The goal is to reduce the total latency suffered by all requests.

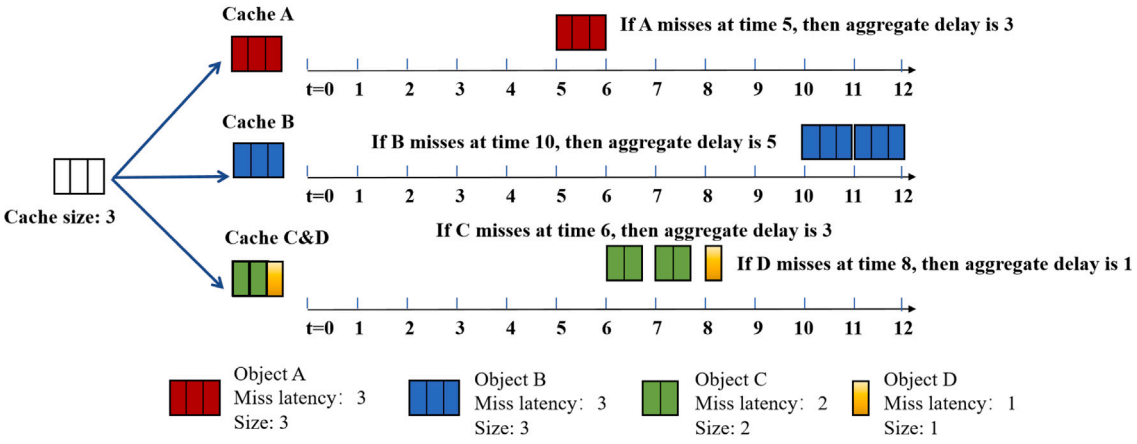

Fig. 2. A toy example considering objects A, B, C and D.

The delayed hits are illustrated in Fig. 1. We assume the miss latency between the cache and the remote server is 3us. As can be seen, the requests that arrive at $t = 1us$ and $t = 2$ are all hits because the cache stores B and C. The first request for object A at time $t = 3$ is a miss due to it not being cached in the local cache. The miss triggers the fetching A from the remote server, which will take some time. Therefore, the successive requests for A arriving at $t = 4$ and $t = 5$ suffer the delayed hits. Object A is fetched back at time $t = 6$, resulting in a true hit.

3.2. Motivation

Given precise knowledge of future requests, how should we design a caching strategy to efficiently reduce latency? Let us first consider a simple example illustrated in Fig. 2. The cache capacity is 3, and we consider four objects—A, B, C, and D—with sizes of 3, 3, 2, and 1, respectively. The fetch latencies for missing objects A, B, C, and D are also 3, 3, 2, and 1, respectively. In the future, object A will be requested once after five time slots. Object B will be requested two times consecutively after ten time slots, while object C will be requested twice in succession after six time slots. Finally, object D will be requested once after eight time slots. If the first request for object B results in a cache miss, the subsequent request will experience a delayed hit, leading to an aggregate delay of 5. Similarly, the aggregate delays for requests to objects A, C, and D are 3, 3, and 1, respectively.

The question is which object to cache at the current time, assuming we retain the cached object until its next request. If we cache object A or B, the cache will be fully occupied, leaving no room for other objects. However, if we cache object C, there will be enough space to also store object D. A key insight is that cache policies must consider both (i) the total delay an object will incur if its first request results in a miss and (ii) the resources it consumes, including the cache space it occupies and the duration it remains in the cache. For example, caching object A would reduce an aggregate delay of 5, but it would remain in the cache for five time slots, saving an average of $\frac{3}{5}$ units of delay per time slot. Similarly, caching object B saves an average of $\frac{5}{10}$ units of delay per time slot, while caching object C yields an average of $\frac{3}{6}$ units of delay per time slot. Additionally, because of C's smaller size, object D can be cached alongside it, saving $\frac{1}{8}$ units of delay on average. Caching A is more efficient when the object size is not considered; however, once the object size is factored in, caching C becomes more efficient.

The toy example in Fig. 2 inspires us to cache evict the object with higher aggregate delay due to miss, arrive the farthest in the future, and larger size when the cache is full. Thus, we propose a rank function as follows,

$$f_i = \frac{D_i}{R_i s_i}, \tag{2}$$

where D_i is the sum of the delay due to miss and any delayed hits when the object is being fetched, R_i is the interval from the current time t to the next request arrival (i.e., residual time), and s_i is the size of content i. We evict the cached object with the smallest rank when eviction is needed. Compared to [13], we consider the impact of object size on caching decisions. The idea is similar to the Belady algorithm [45], which evicts the object that arrivals farthest in the future when eviction is needed and has proven to be offline optimal with uniform object size and no delayed hits. The effectiveness of evicting objects based on (2) with the known future requests is illustrated in Sections 5.2 and 5.3.

Remark 1 (*Offline Optimal Algorithm*)**.** The offline algorithm that addresses delayed hits in cache, while accounting for different object sizes, remains an open problem. One promising approach is to utilize the Minimum-Cost Multi-Commodity Flow (MCMCF) method [50]. However, requests of variable sizes introduce different latencies. Consequently, the time that the missed contents

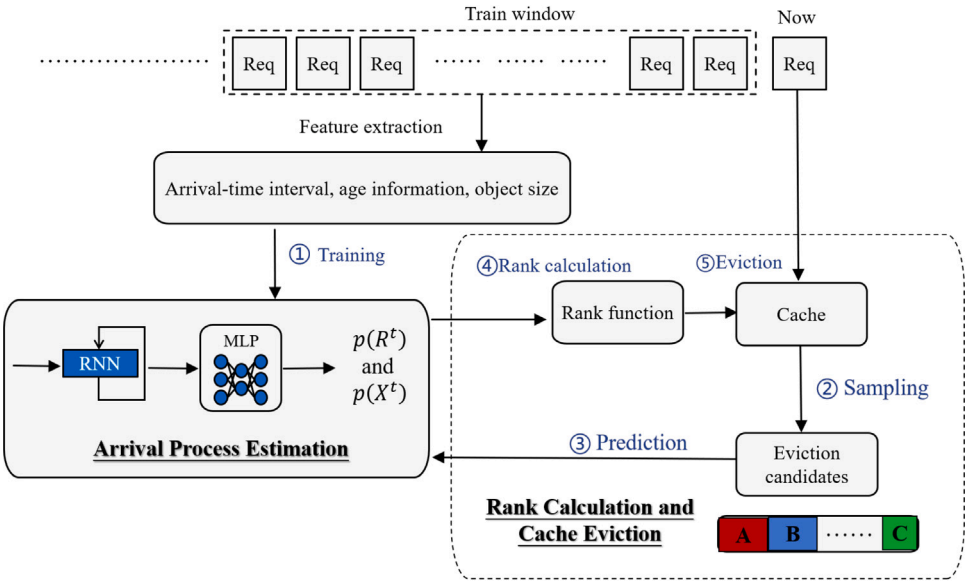

Fig. 3. Overview of our online learning framework.

are fetched from the remote server (i.e., the decision-making time) may not align precisely with the order in which the requests arrive. This discrepancy may necessitate the addition of new auxiliary edges or constraints to construct an appropriate flow graph. Overall, developing an offline optimal algorithm that accounts for delayed hits with different object sizes deserves further research. While studying offline optimal solutions can provide upper bounds, these solutions are impractical for real-time applications since the optimal decision for a request is unavailable unless the process for the entire request trace is completed. Our proposed offline algorithm overcomes this limitation by utilizing only near-future (a window related to miss latency) information to make informed decisions.

4. Our online learning framework

The rank function in Eq. (2) considers the aggregate delay due to a miss, the residual time, and the object size simultaneously when making evictions. The aggregate delay and residual time both depend on the future arrivals. However, the exact knowledge of future requests is often unknown in practice. Thus, we need to predict the objects' future arrivals and then resort to Eq. (2) to make evictions. Following this idea, we propose an online learning framework depicted in Fig. 3 to predict future arrivals and make cache decisions based on the predictions, which consists of two major components:

- Arrival Process Estimation uses the features extracted from the train window (including the age information, object size, and history embedding) to estimate the object's arrival process. Specifically, we use the combination of RNN and MLP to learn the parameters of objects' arrival process.
- Rank Calculation and Cache Eviction first samples the eviction candidates from the cache objects and makes the objects' future arrivals prediction based on the estimated arrival process. Then, compute the rank of eviction candidates based on our proposed rank function. Finally, evict the objects with the lowest rank to insert the fetched object into the cache.

This paradigm in Fig. 3 is widely used in modern machine learning algorithm applications [5,38] and has demonstrated good performance. For instance, LRB [5] predicts the future arrival time of requests by learning from historical arrival patterns and mimics the offline optimal algorithm Belady by evicting requests with the furthest future arrival time. OA-cache [38] leverages a temporal convolutional network to model both long-term and short-term dependencies between content requests within the training window. It subsequently applies an imitation learning framework to estimate the eviction probability distribution, approximating the oracle policy using the learned dependencies. Following this approach, we also learn request patterns from historical data and make decisions based on the learned characteristics. However, we introduce a specialized design to address the problem of delayed hits. It is important to note that the arrival process estimation module requires training two separate RNN and MLP networks to estimate the distributions of inter-arrival and residual time, respectively.

4.1. Arrival process estimation

4.1.1. Model the arrival process

Let $\{0 < T_{i,1} < T_{i,2} < \cdots\}$ represent the successive time slots when object i is requested independently. Define the inter-arrival time $X_i^k = T_{i,k} - T_{i,k-1}$ for $k \geq 2$ and $X_i^1 = T_{i,1}$. Define the residual time at time t as $R_i^t = T_{i,k} - t$ for $T_{i,k-1} < t < T_{i,k}$. For $t > 0$, define $\mathcal{H}_i^t = \{T_{i,k}, k \geq 1 : T_{i,k} < t\}$ as the history of the point process $\{T_{i,k}\}_k$ in $[0, t)$. Let $p(X_i)$ denote the density distribution of the inter-arrival time X_i^k for object i. The arrival times of requests for the same object are independent. At the same time, the arrival processes for different objects are also independent of each other. It is common to assume that the arrival process of each object is independent of each other, such as [17,64]. Let $\{0 < T_1 < T_2 < \cdots\}$ be the arrival process resulting from the superposition of $\{T_{i,k}\}_k, i = 1, \ldots, n$. We do not make particular stationary assumptions about the point process $\{T_{i,k}\}_k$, such as poisson process [16] and it can be a non-stationary pattern which changes slowly with time. The non-stationary means that $p(X_i)$ can be time-varying for all object $i \in n$.

As discussed in Section 3.2 and illustrated in Fig. 3, it is essential to predict both the residual time and the future arrival time sequence. These two factors depend on the residual time distribution and the inter-arrival time distribution. Consequently, it is crucial to model and accurately estimate these distributions. We employ the mixture of log-gaussian distribution to model these two processes separately similar to [6]. The residual time R_i^t at the current time t is indeed a conditional distribution that depends on age, size, and arrival history. In particular, we model R_i^t for the content i as follows:

$$p(R_i^t \mid s_i, a_i^t, \mathcal{H}_i^t) = \sum_{k=1}^{K} \frac{\omega_i^k}{R_i^t \sigma_i^k \sqrt{2\pi}} \exp\left(-\frac{\left(\log R_i^t - \mu_i^k\right)^2}{2(\sigma_i^k)^2}\right) \tag{3}$$

where K is the number of mixture components, ω_i^k, μ_i^k, and σ_i^k, respectively, represent the weight, mean, and the standard deviation of the kth Gaussian component. The term a_i^t is the time interval between the current time t and the last arrival time of content i (i.e., age). For clarity of readability, we use $p(R_i^t)$ to represent the distribution of residual time R_i^t. Similarly, we model the inter-arrival time distribution $p(X_i^t)$ as the mixture of log-normal processes.

4.1.2. Learning arrival process

We learn the residual time distribution $p(R_i^t)$ and inter-arrival time distribution $p(X_i^t)$ for all objects similar to [6]. We use the samples collected within the training window to learn the request arrival process. After receiving a number of requests equal to the size of the training window (since the last training), we retrain the model to adapt to changes in the arrival process. In other words, we train the model periodically to update the estimated $p(R_i^t)$ and $p(X_i^t)$. Next, we demonstrate the learning process for $p(X_i^t)$; the method for learning $p(R_i^t)$ is analogous. The input to this framework includes the historical inter-arrival times $\mathcal{H}_i^t$, the object size s_i, and the time elapsed from the last arrive time a_i^t (i.e., age). In particular, we firstly utilize a single-layer RNN to capture temporal dependencies by embedding the historical inter-arrival times into a fixed-dimensional vector $\boldsymbol{h}_i^t = \mathrm{RNN}\left(X_i^{t-1}, \boldsymbol{h}_i^{t-1}\right)$. Then the triple $\boldsymbol{x} = \{\boldsymbol{h}_i^t, s_i, a_i^t\}$ including hidden state $\boldsymbol{h}_i^t$, object size s_i, and age a_i^t are used as inputs to the three-layer fully connected MLP to learn the parameters $\phi_i = [\boldsymbol{\omega_i}, \boldsymbol{\mu_i}, \boldsymbol{\sigma_i}]$ of $p(X_i^t)$. The parameters ϕ_i is obtained as follows,

$$\boldsymbol{\omega_i} = \mathrm{softmax}\left(\boldsymbol{W}_\omega \boldsymbol{c} + \boldsymbol{b}_\omega\right) \quad \boldsymbol{\mu_i} = \boldsymbol{W}_{\boldsymbol{\mu}} \boldsymbol{c} + \boldsymbol{b}_\mu \quad \boldsymbol{\sigma_i} = \exp\left(\boldsymbol{W}_\sigma \boldsymbol{c} + \boldsymbol{b}_\sigma\right), \tag{4}$$

where $\boldsymbol{c}$ is the output of MLP,

$$\boldsymbol{c} = \mathrm{ReLU}\left(\boldsymbol{W}_2\, \mathrm{ReLU}\left(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1\right) + \boldsymbol{b}_2\right), \tag{5}$$

and $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_\omega, \boldsymbol{W}_{\boldsymbol{\mu}}, \boldsymbol{W}_\sigma, \boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_\omega, \boldsymbol{b}_\sigma, \boldsymbol{b}_\mu\}$ are the parameters that need to be learned. Note that the exp operation ensures that the parameters $\boldsymbol{\sigma_i}$ are in the range $(0, +\infty)$.

Assuming the length of the training window is T, there are M different objects within the training window. For object i, we assume there are a total of k historical inter-arrival time $\left[X_i^1, X_i^2, \ldots, X_i^k\right]$ and the last arrival time is t_i^{k+1}. In this subsection, we redefine the term k to represent the number of arrival time intervals, replacing the previously used definition. At time t, We train the neural networks by maximizing the log-likelihood and survival probability of object request sequences that happened during the training window. The specially introduced survival probability is used to solve the parameter estimation problem under data scarcity [6,65]. We find the optimal parameters by maximizing the following loss function:

$$\boldsymbol{\theta}^\star = \max_{\theta} \frac{1}{T} \sum_{i=1}^{M} \left(\sum_{j=1}^{k} \log p\left(X_i^j \mid \phi_i\right) + \log \Pr\{X_i^{k+1} > t - t_i^{k+1} \mid \phi_i\} \right), \tag{6}$$

where $\phi_i = [\boldsymbol{\omega_i}, \boldsymbol{\mu_i}, \boldsymbol{\sigma_i}]$ is the learned parameters for object i and is determined by neural network parameters $\boldsymbol{\theta}$, $p\left(X_i^j \mid \phi_i\right)$ is the likelihood of observing inter-arrival time X_i^j within the parameter ϕ_i, the term X_i^{k+1} is sampled from our estimated inter-arrival time distribution, and $\Pr\{X_i^{k+1} > t - t_{k_{i+1}} \mid \phi_i\}$ is the survival probability that the next inter-arrival time is greater than the survival time.

Similarly, we employ the same framework to learn the parameters of the inter-arrival time distribution $p(R_i^t)$. It is important to note that we assume the arrival time interval distribution and the residual time distribution for a given object both follow a mixture log-Gaussian distribution, though their parameters differ. At the current time t, the residual time and age for object i are related. Additionally, it is necessary to obtain samples of residual time from historical arrival requests. Our method is similar to the approach described in [6], and the core idea of obtaining residual time samples from historical arrival intervals is based on sampling. Using object i as an example, we outline how to obtain its residual time samples. Recall that there are k historical inter-arrival time for

object i and the set is $[X_i^1, X_i^2, \ldots, X_i^k]$. Specifically, for the first interval X_i^1, we generate the age sample a_i^1 by uniformly sampling within the range $[0, X_i^1]$. Then the corresponding residual time sample R_i^1 is $X_i^1 - a_i^1$. The same logic is applied to other intervals, and then we can obtain k residual time samples. For the last incomplete arrival (from t_i^{k+1} to t), we also apply a similar logic: we uniformly sample a value a_i^{k+1} from the interval $[0, t - t_i^{k+1}]$, and then the difference $t - t_i^{k+1} - a_i^{k+1}$ is used as the residual time. We then find the optimal parameters of the residual time distribution by maximizing the following loss function:

$$\theta_{\boldsymbol{R}}^{\star} = \max_{\theta_{\boldsymbol{R}}} \frac{1}{T} \sum_{i=1}^{M} \left(\sum_{j=1}^{k} \log p\left(R_i^j \mid \Psi_i\right) + \log \Pr\{R_i^{k+1} > t - t_i^{k+1} - a_i^{k+1} \mid \Psi_i\} \right), \tag{7}$$

where Ψ_i (including the weights, means and standard deviations) is the learned parameters for the residual time distribution of object i and is determined by neural network parameters $\boldsymbol{\theta_R}$, $p\left(R_i^j \mid \Psi_i\right)$ is the likelihood of observing residual time R_i^j within the parameter Ψ_i, the term R_i^{k+1} is sampled from our estimated residual time distribution, and $\Pr\{R_i^{k+1} > t - t_i^{k+1} - a_i^{k+1} \mid \Psi_i\}$ is the survival probability that the residual time is greater than the survival time minus age.

Remark 2 (*Data Scarcity*)**.** In real-world, especially those with long-tailed distributions [66–68], a small number of popular contents account for the majority of requests, while the vast majority of contents are characterized by significantly fewer requests. For contents with small amounts of arrival, effective features may not be learned from this limited data. Traditional methods [69,70] that rely solely on the log-likelihood of observed intervals may risk overfitting to contents with abundant data while neglecting those with limited data. Therefore, we have to consider the issue of data scarcity when estimating parameters. In our estimation method, we use survival probability to solve this problem. The second term in our loss functions represents the survival probability of the next request arriving after a certain time interval, which serves as a critical regularization term that balances the estimation for both frequently and infrequently requested items similar to [6]. By incorporating survival probability, the model learns to map the history embeddings of infrequent objects to distributions where large residual times and inter-arrive time have higher probabilities. Therefore, our model can effectively generalize to infrequent objects, mitigating the impact of limited observations and improving its ability to estimate inter-arrival time and residual time distributions in data-scarce scenarios.

Remark 3 (*Correlated Arrival Process*)**.** Overall, our model effectively handles scenarios where the arrival process of individual objects exhibits dependencies. However, additional design is required to address situations where different arrival processes of objects are interrelated. Our model indeed captures temporal dependencies within the arrival process of each object. Specifically, the residual time distribution $p\left(R_i^t \mid s_i, a_i^t, \mathcal{H}_i^t\right)$ incorporates the history of arrivals $\mathcal{H}_i^t$, which allows the model to account for temporal patterns and dependencies inherent in the request behavior of each object. To extend our model to account for correlations between different objects, we can consider Graph Neural Networks (GNN) [71] as a feasible approach. Each object is represented as a node, and edges are established based on relationships such as co-occurrence or similarity. By employing architectures like Graph Attention Networks (GATs) [72], the model can dynamically weigh the influence of neighboring nodes, capturing both individual object behaviors and inter-object dependencies. The residual time distribution now depends on both the object's history and the enriched embedding that captures inter-object correlations, which is defined as $p\left(R_i^t \mid s_i, a_i^t, \mathcal{H}_i^t, \mathbf{h}_i^{(L')}\right)$ where $\mathbf{h}_i^{(L')}$ is the final embedding after L' GNN layers. This integration allows the model to effectively capture complex dependencies across objects, improving accuracy in scenarios with correlated arrival processes. However, integrating graph-based models to capture inter-object correlations increases computational complexity and poses scalability challenges with large numbers of objects. Constructing accurate graphs requires extensive domain knowledge and careful relationship selection, which can be time-consuming. Therefore, while extending the model to incorporate inter-object correlations is a promising direction, it necessitates additional careful design to address these complexities.

4.2. Rank calculation and cache eviction

We can predict future arrivals with estimated $p(R_i^t)$ and $p(X_i^t)$ after training. Following (2), we propose the rank function based on objects' future arrival prediction as follows,

$$f_i = \frac{\hat{D}_i}{\hat{R}_i s_i}, \tag{8}$$

where $\hat{D}_i$ and $\hat{R}_i$ is the estimated aggregate delay for the next request to content i and the estimated residual time of content i, respectively. The specific process for rank calculation is as follows. Firstly, we sample residual time $\hat{R}_i$ from $p(R_i^t)$ and take the sum of $\hat{R}_i$ and current time t as the first arrival time $\hat{T}_{i,1}$ in the future. Then, we sample from $p(X_i^t)$ and take the sum of the sampled value and the first arrival time $\hat{T}_{i,1}$ as the second arrival time $\hat{T}_{i,2}$ in the future. We repeat this process until we obtain the set of future arrival time $\tau = \{\hat{T}_{i,1}, \hat{T}_{i,2}, \ldots, \hat{T}_{i,k-1}, \hat{T}_{i,k}\}$, where $\hat{T}_{i,k} \geq \hat{T}_{i,1} + d_i$ and $\hat{T}_{i,k-1} < \hat{T}_{i,1} + d_i$. Next, we assume $\hat{T}_{i,1}$ is a miss and calculate the total aggregate latency $\hat{D}_i$ for arrival time set τ. Then, we can calculate the rank function using $\hat{D}_i$ and $\hat{R}_i$ based on (8). For object i, the pseudocode to calculate its rank is presented in 1. In lines 5–9 of Algorithm 1, we predict and store the future arrival time of object i, while in lines 10–13, we compute its aggregate delay based on the set of predicted arrival time. Finally, we calculate the rank function of object i in line 14.

Algorithm 1 Estimating the rank function of object i

Input: learned inter-arrival time distribution $p(X_i^t \mid \phi_i^t)$, learned residual time distribution $p(R_i^t \mid \Psi_i^t)$, the miss latency d_i, the object size s_i, the current time t_0
Initialization: the future arrival time recorder $\tau = \emptyset$, $\hat{D}_i = 0, \hat{R}_i = 0$;
$\hat{R}_i \sim p(R_i^t \mid \Psi_i^t)$ ▷ sample the first residual time
$t = t_0 + \hat{R}_i$ ▷ save the first future arrival time
while $t < t_0 + d_i$ **do**
  $\tau \leftarrow \tau \cup t$ ▷ save the future arrival time in order
  $X_i \sim p(X_i^t \mid \phi_i^t)$; ▷ sample the inter-arrival time
  $t \leftarrow X_i + t$; ▷ obtain the predicted arrival time
end while
for k in $1, 2, \ldots, |\tau|$ **do**
  $t_k = \hat{T}_{i,1} + d_i - \hat{T}_{i,k}$ ▷ calculate the latency for k-th delayed hit
  $\hat{D}_i \leftarrow \hat{D}_i + t_k$
end for
calculate the priority score $f_i = \frac{\hat{D}_i}{\hat{R}_i s_i}$
return: the priority score $f(i)$ for object i

To reduce the error introduced by random sampling, we consider the probability that the rank of object i is greater than that of any other cached objects, which is approximated as follows,

$$p_i \approx \frac{\sum_{q=1}^{Q} \mathbb{I}\left\{f_{iq} > f_{jq}, \forall j, j \neq i\right\}}{Q}, \tag{9}$$

where Q is the total sets of predicted arrival time, $\mathbb{I}\left\{f_{iq} > f_{jq}, \forall j, j \neq i\right\}$ is 1 if the qth calculated rank of object i is greater than the qth calculated rank of any other objects and is 0 otherwise. When eviction is necessary, we randomly sample N cached objects as eviction candidates. Subsequently, we conduct a batch prediction for all candidates to forecast their future arrivals and determine their respective probabilities. We then evict the object with the lowest probability and insert the fetched object into the cache.

The neural networks cannot initially learn the arrival processes of objects absent from the training set. For any such object i absent from the training set, we define a heuristic rank $f_i = -s_i$ to determine its eviction. To adapt to the non-stationary request patterns, we employ a sliding window that leverages the latest arrival data. The neural network is retrained from scratch instead of updating its parameters after accumulating a fixed number of requests equal to the training window size T. This retraining involves initializing the neural network parameters and solving the optimization problem defined in Section 4.1.2 to update the estimation of the arrival process. After retraining, the model jointly learns the neural network parameters and the arrival process parameters. The heuristic rank $f_i = -s_i$ prioritizes the eviction of larger objects. If a positive ranking function is used, these objects may become intermixed with other objects whose ranks are calculated using estimated distribution, making it difficult to ensure the quick eviction of objects with large sizes. Our experimental results also show that the negative sorting function we defined outperforms certain positive sorting functions (e.g., $f_i = \frac{1}{s_i}$).

Remark 4 (*Complexity Discussion*)**.** In algorithm 1, for a particular content i, predicting the arrival times of future requests and calculating the aggregate latency are the most time-consuming operations. All other operations are completed in constant time with $\mathcal{O}(1)$ time complexity. The prediction of future arrival times relies on the learned request arrival process. When the cardinality of the set of predicted arrival times is $|\tau|$, the time complexity for prediction is $\mathcal{O}(|\tau|)$. Similarly, calculating the aggregate latency involves traversing the whole set of predicted arrival times, which also has a complexity of $\mathcal{O}(|\tau|)$. Based on our assumption that the request arrival process follows a mixture of log-normal distributions, the average inter-arrival time can be approximated by $\sum_{k=1}^{K} w^k \cdot e^{\mu^k + \frac{(\sigma^k)^2}{2}}$ (omit content i here) [73], resulting in overall time complexity of $\mathcal{O}(\frac{d}{\sum_{k=1}^{K} w^k \cdot e^{\mu^k + \frac{(\sigma^k)^2}{2}}})$ for algorithm 1, where d is the miss latency. The algorithm's time complexity scales linearly with the cardinality of the set of predicted arrival times $|\tau|$, which ensures efficient computations for moderate values of $|\tau|$ (i.e., low miss latency d). In cases with high miss latency, $|\tau|$ can become large, potentially impacting performance. Optimization techniques such as batching [74,75] or parallel processing [76,77] might be necessary in such scenarios. For example, parallel processing techniques, such as distributing rank computations across multiple CPU cores, can significantly decrease execution time by concurrently handling independent operations, thereby making our algorithm more scalable and efficient for high miss latency.

Remark 5 (*Learning Process*)**.** To illustrate the whole learning process: at time t, the model is initially trained on the most recent T requests (forming window W_1). Let $\mathbb{M}$ be the set of objects present in W_1. This training yields models for the arrival process, specifically $p\left(X_i^t \mid \phi_i^t\right)$ and $p\left(R_i^t \mid \Psi_i^t\right)$, for each object $i \in \mathbb{M}$. During the subsequent T arrivals (window W_2), the model uses these learned distributions to infer the arrivals for these known objects ($i \in \mathbb{M}$). For any object $i \notin \mathbb{M}$ encountered during this W_2 period, we assign a heuristic rank $f_i = -s_i$, prioritizing smaller objects for retention. Once the T requests in W_2 are processed, the model is retrained using the data from W_2. This step incorporates objects that were present in W_2 but absent in W_1. Consequently, in subsequent time intervals, the model can infer arrivals for all objects encountered thus far (both original and newly learned) using their respective learned distributions, and their ranks are determined according to Eq. (8).

5. Evaluation

5.1. Methodology

Baseline algorithms. We consider the baseline algorithms, including LRU, LAC [16], CALA [12],RAVEN [6], ADAPTSIZE [47], LHD [14], LRB [5], LHDMAD [13], and LRUMAD [13]. The detailed description of these algorithms are as follows.

- LRU: The most recent used (arrival) time of each cached content is recorded. When the missed objects are fetched from the remote server, it will replace the least recently used objects with new ones if the cache is full.
- LAC [16]: This algorithm calculates the expected aggregated latency of all delayed hits in the fetch process under a Poisson process assumption and subsequently proposes a ranking function for each object as i as $f_i = \frac{\lambda_i L_i(1+\lambda_i L_i)}{(2+\lambda_i L_i)s_i}$, where L_i is the fetch latency when the object i is missed, λ_i is the arrival rate of object i and s_i is the object size. This algorithm uses the reciprocal of the average of past arrival time intervals as an unbiased estimate of the λ_i. When new contents are fetched from the remote server, it will replace the content with minimum rank if the cache is full.
- CALA [12]: The algorithm calculates the aggregate delay as $W_i = (1-\gamma)D_i + \gamma d_i^2$, which combines the upper bound of miss latency with the average cumulative delay of all previous requests for a specific object. The conservative parameter γ is used to get a trade-off between these two methods. Then, simulate an existing general file caching algorithm while continuously updating the weight of each requested object.
- LRUMAD [13]: The algorithm assumes that all past requests for a particular content were misses and calculates the average aggregate delay, $\hat{D}_i$, per miss. Similar to the LRU algorithm, LRUMAD uses recency as an estimator for the residual time, $\hat{R}_i$. The algorithm then maintains the rank for each content i as $f_i = \frac{\hat{D}_i}{\hat{R}_i}$. When new contents are fetched from the remote server to the cache, it will replace the content with minimum rank with new ones if the cache space is full.
- LHD [14]: Propose a novel metric named hit density to rank the objects, defined as $H_i = \frac{h_i}{s_i l_i}$ where h_i is the hit ratio of content i, s_i is the content size, and l_i is its expected time in the cache. When eviction is necessary, LHD evicts the object with the lowest rank.
- LHDMAD [13]: The algorithm presumes that every past request for a given content in history resulted in a miss and computes the average aggregate delay per miss, expressed as $\hat{D}_i$, per miss. Similar to the LHD algorithm [14], LRUMAD uses hit density as an estimator for the residual time, $\hat{R}_i$. Subsequently, the algorithm maintains a rank for each content i, defined as $f_i = \frac{\hat{D}i}{(s_i l_i)/h_i}$. When new contents are fetched from the cloud to the cache, it will replace the content with minimum rank with new ones if the cache space is full.
- RAVEN [6]: Utilize the estimated objects' arrival distributions to compute the probability of an object that arrives farther than any other objects in the cache. This algorithm does not consider the delayed hit and approximate the optimal Belady [45] to make evictions. When new contents are fetched from the contents to the cache, it will replace the content that is predicted to arrive farthest if the cache space is full.
- ADAPTSIZE [47]: The algorithm optimizes the cache hit ratio in scenarios with heterogeneous object sizes. It is based on a Markov chain to determine the optimal object size admission threshold, and when the cache is full, it uses the LRU algorithm for eviction.
- LRB [5]: It approximates the Belady algorithm by using Gradient Boosting Machines (GBM) to identify objects for eviction based on past access patterns. This approach implements a relaxed version of the Belady algorithm, evicting objects whose next request occurs beyond a specified threshold rather than those that are farthest in the future.

Notably, the algorithms LAC [16], CALA [12], LHDMAD [13], and LRUMAD [13] address the issue of delay hits, while RAVEN [6], ADAPTSIZE [47], LHD [14], and LRB [5] focus on maximizing cache hit ratios in environments with heterogeneous object sizes.

Simulation settings. The design of our algorithm includes numerous hyperparameters, such as the size of the training window, the number of log-Gaussian mixtures, the number of hidden layers, the learning rate, and others. The selection of these hyperparameters significantly impacts the performance of our algorithm. In Section 5.4, we investigate the effects of hyperparameters on the algorithm's performance. Unless otherwise stated, we use the following default values to evaluate our algorithm. In our simulation, the number of log-gaussian mixture components K is 64, and the RNN hidden state size is 64. The learning rate is 0.001, the size of the train window is 20K, and the total sets of predicted arrival time Q is 50 according to [78]. The epochs for each training are 100. Similar to previous works, we choose random sample size N as 64 [5,6].

Optimization algorithm. We use the Adam algorithm [79] to optimize the objective function because it effectively combines adaptive learning rates with momentum, facilitating efficient and reliable convergence. We provide a detailed explanation of the optimization (6), and the optimization of (7) follows a similar approach. To apply the Adam optimization algorithm, begin by initializing the neural network parameters $\boldsymbol{\theta}$ and setting the first and second moment ($\mathbf{m}$ and $\mathbf{v}$) vectors to zero. Select the hyperparameters, including the learning rate, decay rates, and a small constant ϵ to prevent division by zero. In each training iteration, compute the loss function defined in (6) and calculate its gradients with respect to $\boldsymbol{\theta}$. Update the first and second-moment vectors using these gradients and apply bias correction to obtain unbiased estimates $\hat{\mathbf{m}}$ and $\hat{\mathbf{v}}$. Adjust the parameters $\boldsymbol{\theta}$ by adding the scaled first moment divided by the square root of the second moment plus ϵ, effectively maximizing the loss function. Repeat these steps iteratively until the loss function converges to its maximum value.

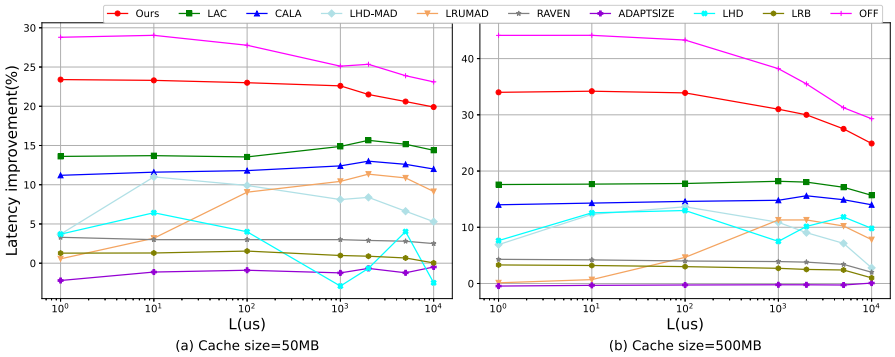

Fig. 4. Comparison of latency improvement between offline and state-of-the-art algorithms relative to LRU under the synthetic dataset(Poisson).

Performance metric. The performance of algorithm A is measured by the latency improvement relative to LRU [12], which is defined as the ratio of the latency difference between LRU and algorithm A to the latency of LRU as follows,

$$\text{Latency Improvement of A} = \frac{\text{Latency(LRU)} - \text{Latency(A)}}{\text{Latency(LRU)}}, \tag{10}$$

where Latency(LRU) is the total latency of LRU algorithm, Latency(A) is the total latency of algorithm A. Better performance is achieved with higher improvement.

5.2. Synthetic dataset

We first conduct the simulations on synthetic non-stationary datasets. The synthetic dataset contains requests for 1000 objects divided into different sessions. We assume that requests' popularity $p_i \in \{1, 2, \dots, n\}$ follows a Zipf distribution. Under Zipf's law, the popularity p_i of the ith ranked object can thus be expressed as

$$p_i = \frac{i^{-\alpha}}{\sum_{i=1}^{n} i^{-\alpha}}, \tag{11}$$

where n represents the total number of distinct objects, and α denotes the Zipf exponent, which characterizes the skewness of the request pattern. The rank of objects (and hence their popularity) is permutated in each session, which means an object popular in the first session may have a different popularity in the second session. The arrival rate of an object is directly proportional to its popularity. We consider three different scenarios:

- **Poisson:** The arrival process of requests follows a Poisson distribution and remains consistent over time.
- **Pareto:** The arrival process of requests follows a Pareto distribution and remains unchanged.
- **Mixed:** The arrival process alternates, with requests following a Poisson distribution in one session and a Pareto distribution in the next, switching in sequence.

The datasets in these three scenarios are non-stationary, and the mixed scenario is even more non-stationary because the popularity of requests changes, as does their arrival process. The total number of requests corresponding to the three different scenarios are 164,518, 149,810, and 103,608, respectively. The object sizes are uniformly sampled between 1 MB–10 MB. The miss latency for each object is composed of a constant term L and a part proportional to its size. The constant term L represents the propagation delay, while the remaining part corresponds to transmission delays [80,81]. This assumption of a fixed miss latency is consistent with [12,16]. We measure the cache size as a fraction of the average total size of active objects as [13]. For these three datasets, the average size of active objects is approximately 5000MB. We consider fractions of 1% and 10%, so the cache sizes are 50MB and 500MB, respectively. The simulation results are shown in Figs. 4, 5, and 6.

The results indicate that our online learning algorithm outperforms state-of-the-art algorithms under settings with variable fetch delays and cache sizes. Our algorithm outperforms the state-of-the-art algorithms across different request arrival processes, whether they follow a Poisson distribution, Pareto distribution, or a mixed distribution. This demonstrates that the combination of RNN and mixture of log-gaussian distribution can effectively estimate non-stationary processes, as validated in [6,17]. Furthermore, the experimental results demonstrate that the offline algorithm based on (2) performs well. While the offline algorithm is expected to perform well due to the assumption that future requests are known, it is essential to highlight that it offers valuable insights for designing online algorithms. Additionally, the gap between our algorithm and the offline solution is minimal.

5.3. Real world traces

We consider four real-world traces as follows:

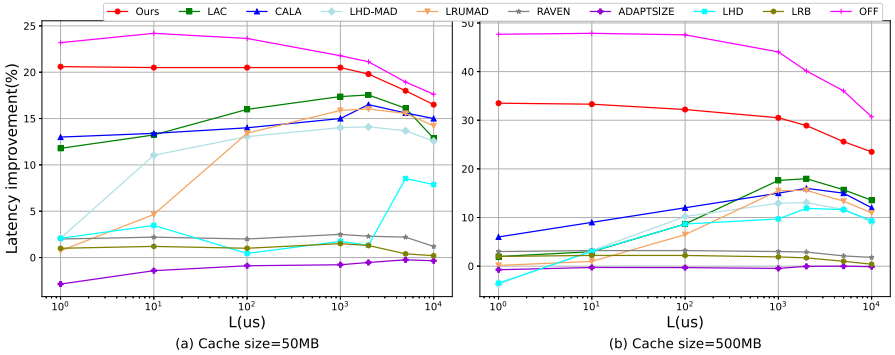

Fig. 5. Comparison of latency improvement between offline and state-of-the-art algorithms relative to LRU under the synthetic dataset(Pareto).

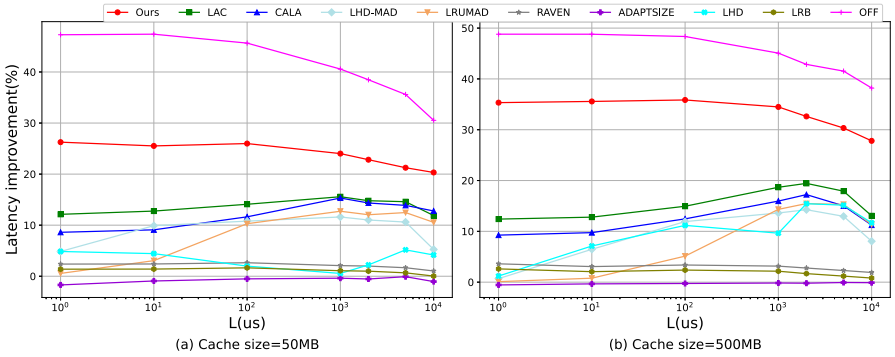

Fig. 6. Comparison of latency improvement between offline and state-of-the-art algorithms relative to LRU under the synthetic dataset(Mixed).

- **Wikipedia** (2018 and 2019): These two traces originate from CDN nodes located in a metropolitan area during 2018 and 2019 [5]. They primarily include web and multimedia content, such as images and videos.
- **CloudPhysics**: CloudPhysics [82] collects the block I/O traces from VMware virtual disks operating in customer data centers using the VMware ESXi hypervisor. A user-mode application installed on each ESXi host worked alongside the standard VMware vscsiStats utility to gather block I/O traces for the virtual disks [83].
- **Youtube** [84]: Youtube is an online platform that enables video sharing over the Internet. Users can upload videos on Youtube and share them with the public. Viewers can search for and watch videos on their devices.

We use a subset for simulation for the four datasets—Wiki2018, Wiki2019, Cloud, and YouTube. The total number of requests included in these datasets is 133,009, 155,015, 113,872, and 162,301, respectively. We present the content popularity and average inter-arrival time distributions of these four traces in Fig. 7. The distribution of average inter-arrival time illustrates that the Youtube and Cloud datasets exhibit more burstiness compared to the Wiki18 and Wiki19 datasets. We show the object size distribution in Fig. 8, demonstrating that real-world traces exhibit object sizes spanning over eight orders of magnitude. Considering object size when making caching decisions better aligns with real-world scenarios. The simulation results are shown in Figs. 9 and 10.

Impact of latency. Figs. 9 and 10 compare the latency improvement of our algorithm and state-of-the-art algorithms with different fetching latencies using a 128 GB and a 256 GB cache, respectively. Our algorithm is generally consistently better than the state-of-the-art algorithms, as discussed in Section 5.1 across various fetch latency and cache size settings. When the cache size is 128 GB, our algorithm achieves a latency reduction improvement of approximately 4%–6% for the Wiki18 dataset compared to state-of-the-art algorithms. For the Wiki19 dataset, it provides an improvement of around 3%–5%. For the Cloud dataset, the algorithm improves latency reduction by 2%–8%; for the YouTube dataset, the improvement ranges from 2%–5%. When the cache size is increased to 256 GB, our algorithm shows a latency reduction improvement of approximately 6% for the Wiki18 dataset. In the Wiki19 dataset, it delivers a 5%–6% improvement. For the Cloud dataset, the latency reduction ranges from 3%–10%, while for the YouTube dataset, the improvement is between 4%–9%, compared to state-of-the-art algorithms. The key ingredient that allows us to obtain a better performance is that our rank function, inspired by the offline algorithm, assists us in determining the priority of eviction candidates. In addition, we can make future arrival predictions for the non-stationary arrival process based on the online learning framework. Overall, our algorithm has improved more on YouTube and Cloud than the Wiki18 and Wiki19 datasets. This is attributed to YouTube and Cloud's higher burstiness relative to Wiki18 and Wiki19. Our algorithm effectively addresses the impact of delayed hits in more bursty traces, showcasing superior performance.

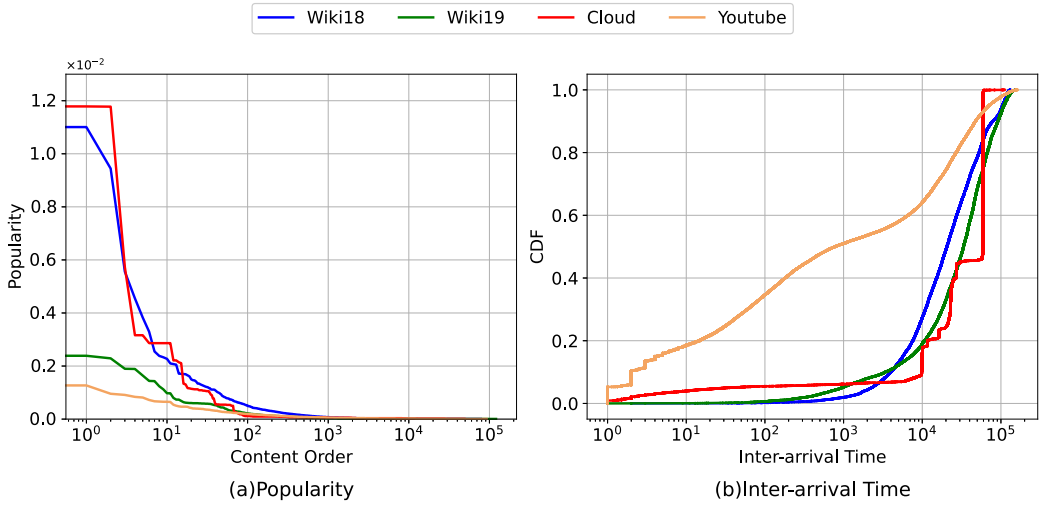

Fig. 7. The content popularity and average interval-time distributions of the four real-world traces.

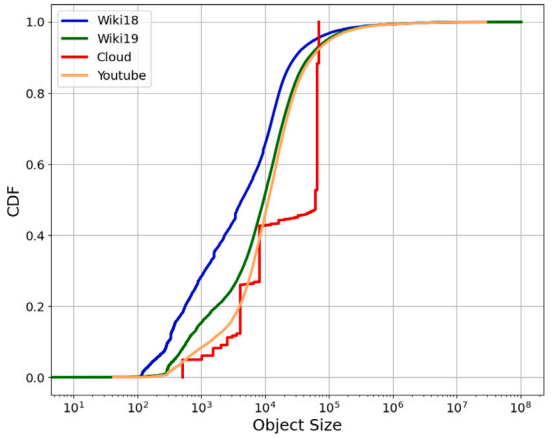

Fig. 8. The object size distribution of the four real-world traces.

Comparison with offline algorithm. In both Figs. 9 and 10, the offline algorithm based on (2) outperforms the other algorithms because it assumes the entire sequence is known and fully considers factors such as request size, arrival time, and cumulative delay. This aligns with the superior performance of the offline algorithm on the synthetic dataset compared to the others. Fig. 10 shows that the performance gap between our algorithm and the offline algorithm is slight for specific datasets, such as only 2% for Wiki18 trace when the latency $L = 10$ ms. However, there remains a performance gap across all datasets between our algorithm and the offline algorithm. Specifically, our algorithm exhibits a performance gap of approximately 3%–10% compared to the offline algorithm across various cache sizes and latency conditions. One possible improvement direction is to consider using parameters of objects with similar characteristics to new objects when making evictions [85] rather than heuristically based on the object size.

Impact of cache size. We further characterize the impact of cache size on the latency reduction of our algorithm compared to state-of-the-art algorithms. We compute the latency improvement for all traces using a fixed value of $L = 100us$. For the Cloud dataset, we consider a broader range of cache sizes because its average size of active objects (approximately 600 GB) is larger than those of the other three datasets. The results are shown in Fig. 11. It can be seen that our algorithm has the most significant improvement in latency compared to other algorithms under different cache sizes and latency settings. As shown in Fig. 11, the relationship between latency improvement and cache size in our algorithm differs for Wiki18 and Wiki19 compared to the other two datasets. This is because the inflection point for latency reduction has not yet been reached. In fact, as we continue to increase the cache size in our experiments, we observe that the latency improvement begins to show a downward trend on Cloud and Youtube dataset.

Hit ratio comparison. While our primary focus is on designing a latency-optimized caching system in the presence of delayed hits—where maximizing hit ratio and minimizing latency are distinct goals—we argue that the improvements in latency also positively impact cache hit performance. When the cache size is 256 GB, we compare the true hit ratios of different algorithms

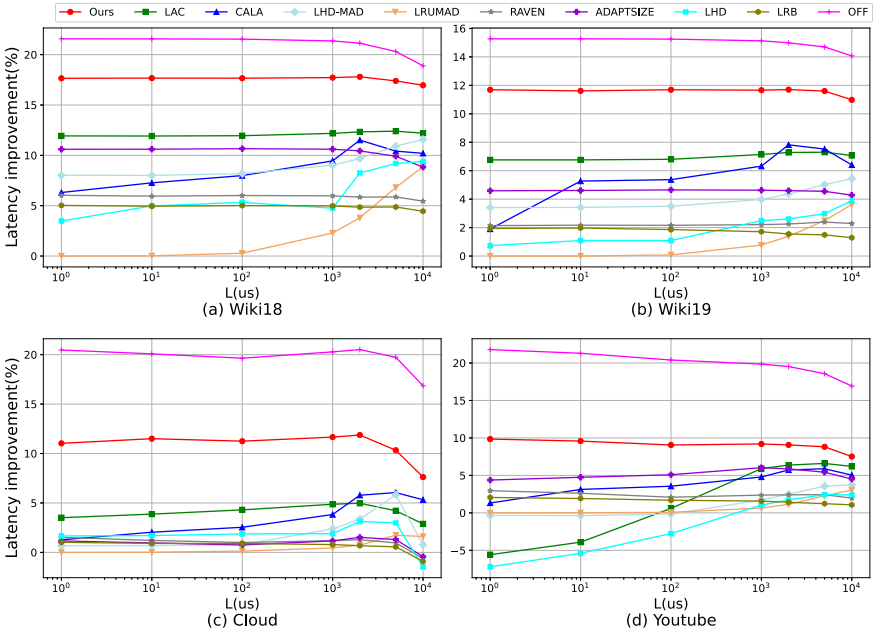

Fig. 9. Comparison of latency improvement between our algorithm and state-of-the-art algorithms relative to LRU using a 128 GB cache.

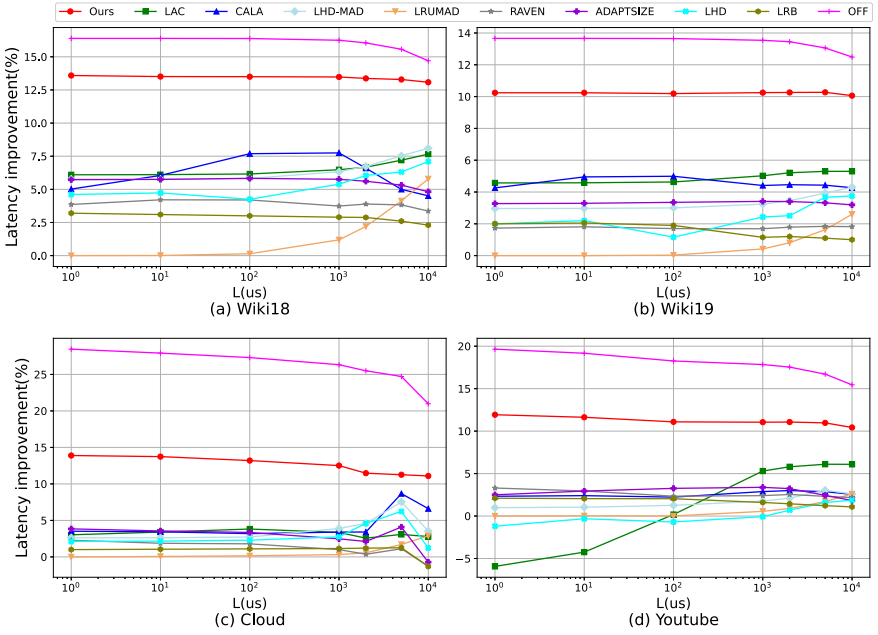

Fig. 10. Comparison of latency improvement between our algorithm and state-of-the-art algorithms relative to LRU using a 256 GB cache.

across various latencies, with the results shown in Fig. 12. As seen in Figs. 9 and 10, our algorithm achieves a significant latency reduction compared to other online algorithms. In Fig. 12, the hit ratio of our algorithm consistently exceeds that of the other online algorithms. When the average fetching latency becomes very large, in both traces almost all these algorithms tend to have a decreasing hit ratio, since nearly all the requests are misses or delayed hits.

5.4. Discussion

Sensitivity analysis of hyperparameters. We have introduced several hyperparameters in our algorithm, which significantly impact its performance. Therefore, we conduct a comprehensive sensitivity analysis for the key hyperparameters, including the size of

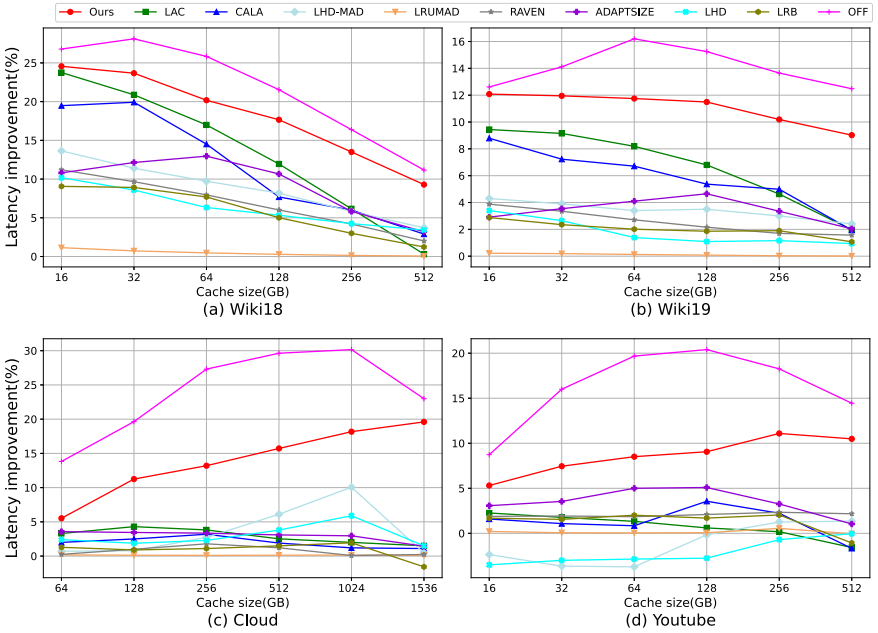

Fig. 11. Comparison of the latency improvement between our algorithm and state-of-the-art algorithms relative to LRU as a function of cache sizes(fixed latency $L = 100us$).

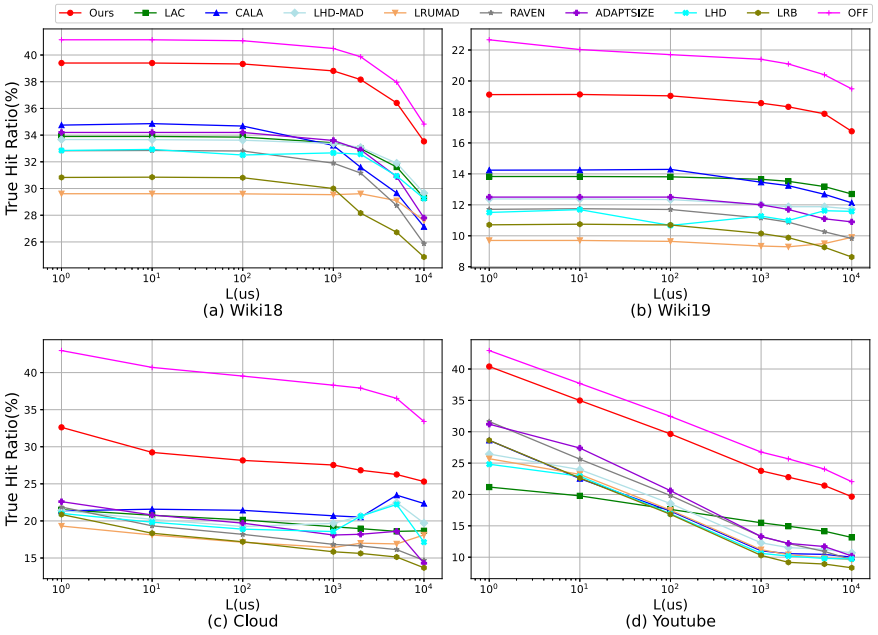

Fig. 12. Comparison of true hit ratios between our algorithm and state-of-the-art algorithms using a 256 GB cache.

the training window, the mixture components, the hidden state size, and the learning rate. We conducted experiments using a cache size of 256 GB and a latency of $L = 100us$, with the results presented in Fig. 13. Similar trends are observed under other different parameter settings. The size of the training window plays a critical role in the algorithm's performance. If the window size is too small, the learned arrival distribution will not be accurate. On the other hand, a large window size may increase memory overhead and result in more time required to train. Fig. 13(a) demonstrates that the algorithm's latency improvements vary with different training window sizes and across various datasets. Nevertheless, all results outperform the baseline algorithms, highlighting the robustness of our approach. This variability suggests that the optimal training window size may differ for each training set to achieve maximum improvement. In our experiments, we select a window length of 20K, which ensures a balanced trade-off between latency

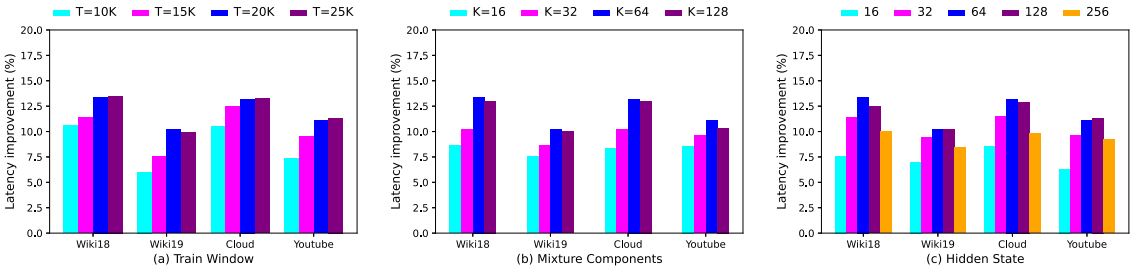

Fig. 13. The impact of hyperparameters on latency improvement with a 256 GB cache size and the fixed latency $L = 100us$.

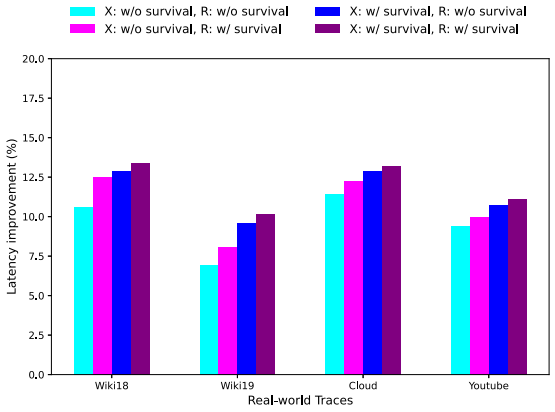

Fig. 14. Impact of survival probability on the latency improvement in the loss functions.

improvement, memory overhead, and training time across all datasets. Using too few mixture components can prevent the model from accurately capturing the complexity of the data distribution, leading to underfitting and reduced performance. Conversely, employing too many components can increase computational overhead and the risk of overfitting, potentially diminishing the model's ability to generalize effectively. Fig. 13(b) further confirms the robustness of our algorithm under various mixture component settings, suggesting that the selected number of mixture components effectively captures the necessary distributional characteristics without overcomplicating the model. We choose a mixture component of 64 to achieve a good average performance of the algorithm across different datasets. Additionally, Fig. 13(c) shows that our policy maintains better performance over the baseline algorithms across different hidden state sizes. As shown, when the hidden state size is between 32 and 128, there is a significant performance improvement. However, for smaller or larger hidden state sizes, the algorithm's performance may slightly decrease. We select a hidden state size of 64 to balance the algorithm's average performance across different datasets. We employ the Adam optimization algorithm [79], which uses a default learning rate of 0.001. Furthermore, we investigate the effects of varying learning rates on the algorithm's performance. A learning rate of 0.01 achieved performance comparable to that of 0.001. However, learning rates of 0.1 and 1 prevented the neural network from converging during training.

Computational overhead. Our algorithm uses the metadata of each object in the cache to infer its arrival distribution: its history embedding, current age, and object size. Therefore, the metadata memory of each object in the cache is 264 bytes according to our RNN hidden state size setting. The metadata used by LRB to make an inference for an object is the size of their manually defined features, which takes 176 bytes. Overall, the memory overhead of our algorithm is more than that of LRB. During the training process, our algorithm utilizes an average of 256MB of memory, while the LRB algorithm consumes an average of 319 MB. This demonstrates that our approach requires less memory during training, which can be attributed to the streamlined neural network architecture we designed. Specifically, it incorporates a single-layer RNN and a three-layer MLP, which contribute to reduced memory consumption. Both the metadata storage overhead and the memory usage during training are manageable for modern storage systems, which typically have capacities exceeding the GB level. The average training time of RNN and MLP on four datasets is about 2 h. Although this may seem time-consuming, there are several opportunities for optimization. Firstly, our implementation is currently written in Python, which generally exhibits lower performance (in latency) compared to high-performance languages such as C++. Second, leveraging optimization techniques such as batch processing [74,86] and parallel computing [76,77] can significantly reduce training time. For example, we can train neural networks for the inter-arrival time distribution and residual time distribution concurrently on separate GPUs. Additionally, in our experiments, we utilized GRU [87] as the RNN unit; however, the RNN module can be replaced with SRU [88], an efficient implementation of recurrent neural units that have demonstrated reduced training times without sacrificing performance [89–91]. We plan to explore module optimization and fine-tuning in future work.

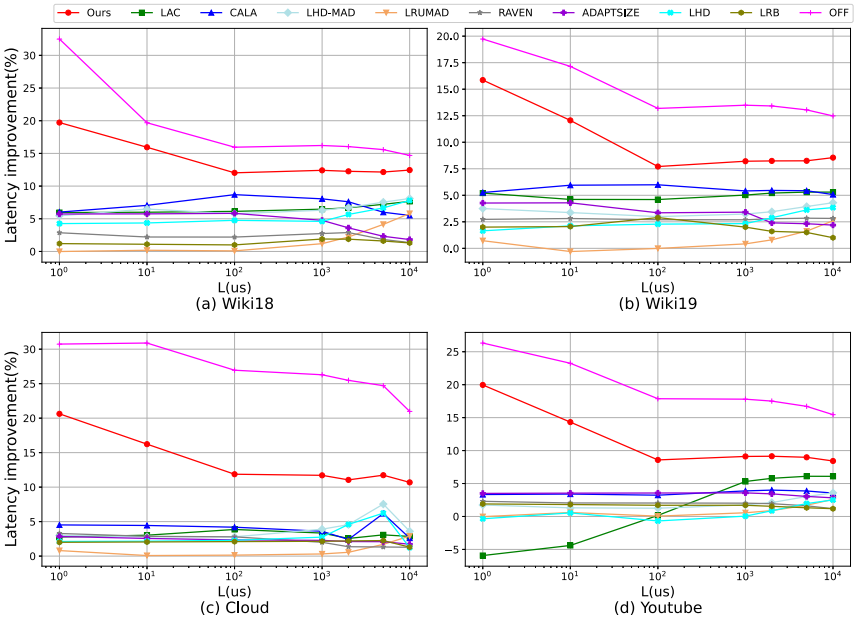

Fig. 15. Comparison of latency improvement between our algorithm and state-of-the-art algorithms relative to LRU using a 256 GB cache when the fetch latency is random.

Survival probability. We incorporate survival probability into the loss functions defined in Eqs. (6) and (7) to address data scarcity. In this subsection, we perform ablation experiments on survival probability using a cache size of 256 GB and a latency of $L = 100us$. There is also a similar phenomenon in other parameter settings. The results are presented in Fig. 14. In the figure, the legend "X: w/survival, R: w/o survival" indicates that survival probability is incorporated in the estimation of the arrival time interval but excluded from the estimation of residual time. The other legends follow similar conventions. Fig. 14 demonstrates that, compared to scenarios where survival probability is not considered, latency improvements are further enhanced when survival probability is accounted for in both the arrival time interval and residual time estimations. We observe that our algorithm's performance improves across all traces, with particularly significant gains on the Wiki 18 and Wiki 19 datasets. This enhancement is attributed to these two datasets having a higher proportion of request types with fewer arrivals compared to the Cloud and YouTube datasets, which exacerbates data scarcity. By incorporating survival probability into our loss function, we effectively address and mitigate this issue.

Non-deterministic fetch latency. Regarding the assumption that d_i is a linear function of the request size s_i, we note that this assumption is widely adopted in previous studies [13,16]. Furthermore, we have extended our experiments to include scenarios where the fetch latency is random, thereby better reflecting the variability of communication network conditions in the real-world. In this case, the fetch latency and object size do not exhibit a straightforward linear relationship. We set the cache size to 256 GB and model fetch latency using a truncated Gaussian distribution, ensuring that latency values are always positive. This distribution has a mean equal to the latency specified in Section 5.2 and a variance of 10. When a cache miss occurs, we sample from this distribution to determine the fetch latency for this miss instance. In our algorithm, when calculating the aggregate delay of the predicted request arrival sequence, we use the mean of the distribution as the delay for the first miss. Similarly, for other baseline algorithms, whenever fetch latency is needed to calculate the rank, we use the distribution's mean instead. The results, presented in Fig. 15, demonstrate that our algorithm consistently outperforms others in latency performance, even under random fetch latency. Additionally, we would like to emphasize that current works on delay hits typically all assume deterministic fetch latency [12,13,16]. When the fetch latency is random, the caching problem under delayed hits deserves further investigation, particularly regarding how the characteristics of fetch latency distributions, such as variance, impact algorithm performance.

6. Conclusion

In this paper, we consider the cache scenario with delayed hits, where the arrival process is non-stationary, and the fetch latency and object size are both non-uniform. We design an online learning framework composed of RNN and MLP to predict objects' future arrivals. Based on these predictions, we can efficiently determine the rank of eviction candidates. Simulations on a synthetic dataset and four real-world traces show that our algorithm consistently outperforms state-of-the-art algorithms across various delay and cache size settings.

CRediT authorship contribution statement

Bowen Jiang: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Yubo Yang:** Software, Methodology. **Bo Jiang:** Writing – original draft, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China under Grant No. 62072302.

Data availability

Data will be made available on request.

References

[1] Niklas Carlsson, Derek Eager, Ajay Gopinathan, Zongpeng Li, Caching and optimized request routing in cloud-based content delivery systems, Perform. Eval. 79 (2014) 38–55, http://dx.doi.org/10.1016/j.peva.2014.07.003.
[2] Nestor Michael C. Tiglao, António M. Grilo, An analytical model for transport layer caching in wireless sensor networks, Perform. Eval. 69 (5) (2012) 227–245, http://dx.doi.org/10.1016/j.peva.2011.12.002.
[3] Yaru Fu, Quan Yu, Angus K.Y. Wong, Zheng Shi, Hong Wang, Tony Q.S. Quek, Exploiting coding and recommendation to improve cache efficiency of reliability-aware wireless edge caching networks, IEEE Trans. Wirel. Commun. 20 (11) (2021) 7243–7256, http://dx.doi.org/10.1109/TWC.2021.3081682.
[4] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, Christian Tanguy, Performance evaluation of the random replacement policy for networks of caches, Perform. Eval. 40 (1) (2012) 395–396, http://dx.doi.org/10.1016/j.peva.2013.10.004.
[5] Zhenyu Song, Daniel S. Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al., Learning relaxed belady for content distribution network caching, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, 2020, pp. 529–544.
[6] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, Zhi-Li Zhang, Raven: belady-guided, predictive (deep) learning for in-memory and content caching, in: Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies, 2022, pp. 72–90, http://dx.doi.org/10.1145/3555050.3569134.
[7] Hazem Gomaa, Geoffrey G. Messier, Carey Williamson, Robert Davies, Estimating instantaneous cache hit ratio using markov chain analysis, IEEE/ACM Trans. Netw. 21 (5) (2012) 1472–1483, http://dx.doi.org/10.1109/TNET.2012.2227338.
[8] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, Don Towsley, Performance evaluation of hierarchical TTL-based cache networks, Comput. Netw. 65 (2014) 212–231, http://dx.doi.org/10.1016/j.comnet.2014.03.006.
[9] Amr Rizk, Michael Zink, Ramesh Sitaraman, Model-based design and analysis of cache hierarchies, in: 2017 IFIP Networking Conference (IFIP Networking) and Workshops, IEEE, 2017, pp. 1–9, http://dx.doi.org/10.23919/IFIPNetworking.2017.8264840.
[10] Giovanni Neglia, Damiano Carra, Mingdong Feng, Vaishnav Janardhan, Pietro Michiardi, Dimitra Tsigkari, Access-time-aware cache algorithms, ACM Trans. Model. Perform. Eval. Comput. Syst. (TOMPECS) 2 (4) (2017) 1–29.
[11] Andrés Ferragut, Ismael Rodríguez, Fernando Paganini, Optimizing TTL caches under heavy-tailed demands, ACM SIGMETRICS Perform. Eval. Rev. 44 (1) (2016) 101–112, http://dx.doi.org/10.1145/2896377.2901459.
[12] Chi Zhang, Haisheng Tan, Guopeng Li, Zhenhua Han, Shaofeng H.-C. Jiang, Xiang-Yang Li, Online file caching in latency-sensitive systems with delayed hits and bypassing, in: IEEE INFOCOM 2022-IEEE Conference on Computer Communications, IEEE, 2022, pp. 1059–1068, http://dx.doi.org/10.1109/INFOCOM48880.2022.9796969.
[13] Nirav Atre, Justine Sherry, Weina Wang, Daniel S. Berger, Caching with delayed hits, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 495–513, http://dx.doi.org/10.1145/3387514.3405883.
[14] Nathan Beckmann, Haoxian Chen, Asaf Cidon, {LHD}: Improving cache hit rate by maximizing hit density, in: 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 18, 2018, pp. 389–403.
[15] Peter Manohar, Jalani Williams, Lower bounds for caching with delayed hits, 2020, arXiv preprint arXiv:2006.00376.
[16] Gang Yan, Jian Li, Towards latency awareness for content delivery network caching, in: 2022 USENIX Annual Technical Conference, USENIX ATC 22, 2022, pp. 789–804.
[17] Oleksandr Shchur, Marin Biloš, Stephan Günnemann, Intensity-free learning of temporal point processes, Int. Conf. Learn. Represent. (ICLR) (2020).
[18] Kenneth R. Kaplan, Robert O. Winder, Cache-based computer systems, Computer 6 (3) (1973) 30–36, http://dx.doi.org/10.1109/c-m.1973.217037.
[19] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, William E. Weihl, Globally distributed content delivery., IEEE Internet Comput. 6 (5) (2002) 50–58.
[20] Quyuan Luo, Shihong Hu, Changle Li, Guanghui Li, Weisong Shi, Resource scheduling in edge computing: A survey, IEEE Commun. Surv. & Tutorials 23 (4) (2021) 2131–2165, http://dx.doi.org/10.1109/COMST.2021.3106401.
[21] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, Tao Xie, Measurement and analysis of mobile web cache performance, in: Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 691–701.
[22] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, Saverio Niccolini, Temporal locality in today's content caching: Why it matters and how to model it, ACM SIGCOMM Comput. Commun. Rev. 43 (5) (2013) 5–12, http://dx.doi.org/10.1145/2541468.2541470.
[23] Shiji Zhou, Zhi Wang, Chenghao Hu, Yinan Mao, Haopeng Yan, Shanghang Zhang, Chuan Wu, Wenwu Zhu, Caching in dynamic environments: A near-optimal online learning approach, IEEE Trans. Multimed. 25 (2021) 792–804, http://dx.doi.org/10.1109/TMM.2021.3132156.

[24] Setareh Maghsudi, Mihaela van der Schaar, A non-stationary bandit-learning approach to energy-efficient femto-caching with rateless-coded transmission, IEEE Trans. Wirel. Commun. 19 (7) (2020) 5040–5056.

[25] S. Krishnendu, B.N. Bharath, Navneet Garg, Vimal Bhatia, Tharmalingam Ratnarajah, Learning to cache: Federated caching in a cellular network with correlated demands, IEEE Trans. Commun. 70 (3) (2021) 1653–1665, http://dx.doi.org/10.1109/tcomm.2021.3132048.

[26] Song Jiang, Xiaodong Zhang, LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance, ACM SIGMETRICS Perform. Eval. Rev. 30 (1) (2002) 31–42, http://dx.doi.org/10.1145/511399.511340.

[27] Nimrod Megiddo, Dharmendra S. Modha, {ARC}: A {Self-Tuning}, low overhead replacement cache, in: 2nd USENIX Conference on File and Storage Technologies, FAST 03, 2003.

[28] Elizabeth J. O'neil, Patrick E. O'neil, Gerhard Weikum, The LRU-K page replacement algorithm for database disk buffering, Acm Sigmod Rec. 22 (2) (1993) 297–306, http://dx.doi.org/10.1145/170036.170081.

[29] Ludmila Cherkasova, Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, Hewlett-Packard Laboratories Palo Alto, CA, USA, 1998.

[30] Aaron Blankstein, Siddhartha Sen, Michael J. Freedman, Hyperbolic caching: Flexible caching for web applications, in: 2017 USENIX Annual Technical Conference, USENIX ATC 17, 2017, pp. 499–511.

[31] Tareq Si Salem, Giovanni Neglia, Stratis Ioannidis, No-regret caching via online mirror descent, ACM Trans. Model. Perform. Eval. Comput. Syst. 8 (4) (2023) 1–32, http://dx.doi.org/10.1109/ICC42927.2021.9500487.

[32] Naram Mhaisen, George Iosifidis, Douglas Leith, Online caching with optimistic learning, in: 2022 IFIP Networking Conference, IFIP Networking, IEEE, 2022, pp. 1–9, http://dx.doi.org/10.23919/IFIPNetworking55013.2022.9829806.

[33] Georgios S. Paschos, Apostolos Destounis, Luigi Vigneri, George Iosifidis, Learning to cache with no regrets, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 235–243, http://dx.doi.org/10.1109/INFOCOM.2019.8737446.

[34] Rajarshi Bhattacharjee, Subhankar Banerjee, Abhishek Sinha, Fundamental limits on the regret of online network-caching, Proc. the ACM Meas. Anal. Comput. Syst. 4 (2) (2020) 1–31, http://dx.doi.org/10.1145/3392143.

[35] R. Sri Prakash, Nikhil Karamchandani, Sharayu Moharir, On the regret of online edge service hosting, Perform. Eval. 50 (4) (2023) 35–37, http://dx.doi.org/10.1016/j.peva.2023.102367.

[36] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, Junwhan Ahn, An imitation learning approach for cache replacement, in: International Conference on Machine Learning, PMLR, 2020, pp. 6237–6247.

[37] Vladyslav Fedchenko, Giovanni Neglia, Bruno Ribeiro, Feedforward neural networks for caching: N enough or too much? Acm Sigmetrics Perform. Eval. Rev. 46 (3) (2019) 139–142.

[38] Shuting Qiu, Qilin Fan, Xiuhua Li, Xu Zhang, Geyong Min, Yongqiang Lyu, OA-cache: Oracle approximation-based cache replacement at the network edge, IEEE Trans. Netw. Serv. Manag. 20 (3) (2023) 3177–3189, http://dx.doi.org/10.1109/TNSM.2023.3239664.

[39] Gang Yan, Jian Li, RL-Bélády: A unified learning framework for content caching, in: Proceedings of the 28th ACM International Conference on Multimedia, 2020, pp. 1009–1017.

[40] Zhe Wang, Jia Hu, Geyong Min, Zhiwei Zhao, Zi Wang, Agile cache replacement in edge computing via offline-online deep reinforcement learning, IEEE Trans. Parallel Distrib. Syst. 35 (4) (2024) 663–674, http://dx.doi.org/10.1109/TPDS.2024.3368763.

[41] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, Ramesh K. Sitaraman, RL-cache: Learning-based cache admission for content delivery, IEEE J. Sel. Areas Commun. 38 (10) (2020) 2372–2385, http://dx.doi.org/10.1109/JSAC.2020.3000415.

[42] Pengmiao Li, Yuchao Zhang, Huahai Zhang, Wendong Wang, Ke Xu, Zhili Zhang, A delayed eviction caching replacement strategy with unified standard for edge servers, Comput. Netw. 230 (2023) 109794, http://dx.doi.org/10.1016/j.comnet.2023.109794.

[43] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, Tapani Ristaniemi, Learn to cache: Machine learning for network edge caching in the big data era, IEEE Wirel. Commun. 25 (3) (2018) 28–35.

[44] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, Zhi-Li Zhang, Deepcache: A deep learning based framework for content caching, in: Proceedings of the 2018 Workshop on Network Meets AI & ML, 2018, pp. 48–53.

[45] Laszlo A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Syst. J. 5 (2) (1966) 78–101, http://dx.doi.org/10.1147/sj.52.0078.

[46] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, Giri Narasimhan, Driving cache replacement with {ML-based}{LeCaR}, in: 10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 18, 2018.

[47] Daniel S. Berger, Ramesh K. Sitaraman, Mor Harchol-Balter, {AdaptSize}: Orchestrating the hot object memory cache in a content delivery network, in: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 17, 2017, pp. 483–498.

[48] Gil Einziger, Roy Friedman, Ben Manes, Tinylfu: A highly efficient cache admission policy, ACM Trans. Storage (ToS) 13 (4) (2017) 1–31, http://dx.doi.org/10.1109/PDP.2014.34.

[49] Gil Einziger, Ohad Eytan, Roy Friedman, Benjamin Manes, Lightweight robust size aware cache management, ACM Trans. Storage (TOS) 18 (3) (2022) 1–23.

[50] Daniel S. Berger, Nathan Beckmann, Mor Harchol-Balter, Practical bounds on optimal caching with variable object sizes, Proc. the ACM Meas. Anal. Comput. Syst. 2 (2) (2018) 1–38, http://dx.doi.org/10.1145/3292040.3219627.

[51] Licheng Chen, Yunquan Dong, Estimating age of information in wireless systems with unknown distributions of inter-arrival/service time, IEEE Trans. Netw. Sci. Eng. (2024) http://dx.doi.org/10.1109/TNSE.2024.3453959.

[52] Laleh Arshadi, Amir Hossein Jahangir, An empirical study on TCP flow interarrival time distribution for normal and anomalous traffic, Int. J. Commun. Syst. 30 (1) (2017) e2881.

[53] Yanru Chen, Bing Yi, Yangsheng Jiang, Jidong Sun, M.I.M. Wahab, Inter-arrival time distribution of passengers at service facilities in underground subway stations: A case study of the metropolitan city of Chengdu in China, Transp. Res. Part A: Policy Pr. 111 (2018) 227–251, http://dx.doi.org/10.1016/j.tra.2018.03.009.

[54] Ennio Cascetta, Transportation Systems Analysis: Models and Applications, vol. 29, Springer Science & Business Media, 2009, http://dx.doi.org/10.1007/978-0-387-75857-2_9.

[55] Yao Zhang, Changle Li, Tom H. Luan, Chau Yuen, Yuchuan Fu, Hui Wang, Weigang Wu, Towards hit-interruption tradeoff in vehicular edge caching: Algorithm and analysis, IEEE Trans. Intell. Transp. Syst. 23 (6) (2021) 5198–5210, http://dx.doi.org/10.1109/TITS.2021.3052355.

[56] Nitish K. Panigrahy, Jian Li, Don Towsley, Christopher V. Hollot, Network cache design under stationary requests: Exact analysis and Poisson approximation, Comput. Netw. 180 (2020) 107379, http://dx.doi.org/10.1016/j.comnet.2020.107379.

[57] Daniel S. Berger, Sebastian Henningsen, Florin Ciucu, Jens B. Schmitt, Maximizing cache hit ratios by variance reduction, ACM SIGMETRICS Perform. Eval. Rev. 43 (2) (2015) 57–59.

[58] Mathieu Leconte, Georgios Paschos, Lazaros Gkatzikis, Moez Draief, Spyridon Vassilaras, Symeon Chouvardas, Placing dynamic content in caches with small population, in: IEEE INFOCOM 2016-the 35th Annual IEEE International Conference on Computer Communications, IEEE, 2016, pp. 1–9.

[59] Yuanyuan Li, Stratis Ioannidis, Cache networks of counting queues, IEEE/ACM Trans. Netw. 29 (6) (2021) 2751–2764.

[60] Yunbo Wang, Mehmet C. Vuran, Steve Goddard, Cross-layer analysis of the end-to-end delay distribution in wireless sensor networks, IEEE/ACM Trans. Netw. 20 (1) (2011) 305–318, http://dx.doi.org/10.1109/RTSS.2009.27.

[61] A. Clifford Cohen, Estimating the parameter in a conditional Poisson distribution, Biometrics 16 (2) (1960) 203–211.

[62] Tianle Chen, Brian Keng, Javier Moreno, Multivariate arrival times with recurrent neural networks for personalized demand forecasting, in: 2018 IEEE International Conference on Data Mining Workshops, ICDMW, IEEE, 2018, pp. 810–819, http://dx.doi.org/10.1109/ICDMW.2018.00121.
[63] Bruce G. Lindsay, Ramani S. Pilla, Prasanta Basak, Moment-based approximations of distributions using mixtures: Theory and applications, Ann. Inst. Statist. Math. 52 (2000) 215–230.
[64] Nitish K. Panigrahy, Philippe Nain, Giovanni Neglia, Don Towsley, A new upper bound on cache hit probability for non-anticipative caching policies, ACM Trans. Model. Perform. Eval. Comput. Syst. 7 (2–4) (2022) 1–24, http://dx.doi.org/10.1145/3547332.
[65] Andreas Wienke, Frailty Models in Survival Analysis, Chapman and Hall/CRC, 2010.
[66] Xianzhi Zhang, Yipeng Zhou, Di Wu, Miao Hu, Xi Zheng, Min Chen, Song Guo, Optimizing video caching at the edge: A hybrid multi-point process approach, IEEE Trans. Parallel Distrib. Syst. 33 (10) (2022) 2597–2611, http://dx.doi.org/10.1109/TPDS.2022.3147240.
[67] Yin Zhang, Ranran Wang, Yiran Wang, Min Chen, Mohsen Guizani, Diversity-driven proactive caching for mobile networks, IEEE Trans. Mob. Comput. (2023).
[68] Yijing Li, Shihong Hu, Guanghui Li, CVC: A collaborative video caching framework based on federated learning at the edge, IEEE Trans. Netw. Serv. Manag. 19 (2) (2021) 1399–1412, http://dx.doi.org/10.1109/TNSM.2021.3135306.
[69] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, Yoshua Bengio, A recurrent latent variable model for sequential data, Adv. Neural Inf. Process. Syst. 28 (2015).
[70] Ruizhi Deng, Bo Chang, Marcus A. Brubaker, Greg Mori, Andreas Lehrmann, Modeling continuous stochastic processes with dynamic normalizing flows, Adv. Neural Inf. Process. Syst. 33 (2020) 7805–7815.
[71] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, Graph attention networks, 2017, arXiv preprint arXiv:1710.10903.
[72] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, Gabriele Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2008) 61–80, http://dx.doi.org/10.1109/TNN.2008.2005605.
[73] Edwin L. Crow, Kunio Shimizu, Lognormal Distributions, Marcel Dekker New York, 1987.
[74] Xiufeng Liu, Nadeem Iftikhar, Xike Xie, Survey of real-time processing systems for big data, in: Proceedings of the 18th International Database Engineering & Applications Symposium, 2014, pp. 356–361.
[75] James Warren, Nathan Marz, Big Data: Principles and Best Practices of Scalable Realtime Data Systems, Simon and Schuster, 2015.
[76] Seyed H. Roosta, Parallel Processing and Parallel Algorithms: Theory and Computation, Springer Science & Business Media, 2012.
[77] Behrooz Parhami, Introduction to Parallel Processing: Algorithms and Architectures, Springer Science & Business Media, 2006.
[78] Mohammad Rafiqul Islam, Sample size and its role in central limit theorem (CLT), Comput. Appl. Math. J. 4 (1) (2018) 1–7, http://dx.doi.org/10.31295/ijpm.v1n1.42.
[79] Diederik P. Kingma, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.
[80] Biqian Feng, Chenyuan Feng, Daquan Feng, Yongpeng Wu, Xiang-Gen Xia, Proactive content caching scheme in urban vehicular networks, IEEE Trans. Commun. 71 (7) (2023) 4165–4180, http://dx.doi.org/10.1109/TCOMM.2023.3277530.
[81] Yiming Miao, Yixue Hao, Min Chen, Hamid Gharavi, Kai Hwang, Intelligent task caching in edge cloud via bandit learning, IEEE Trans. Netw. Sci. Eng. 8 (1) (2020) 625–637, http://dx.doi.org/10.1109/TNSE.2020.3047417.
[82] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, Irfan Ahmad, Efficient {MRC} construction with {SHARDS}, in: 13th USENIX Conference on File and Storage Technologies, FAST 15, 2015, pp. 95–110.
[83] Muhammad Wajahat, Aditya Yele, Tyler Estro, Anshul Gandhi, Erez Zadok, Analyzing the distribution fit for storage workload and internet traffic traces, Perform. Eval. 142 (2020) 102121, http://dx.doi.org/10.1016/j.peva.2020.102121.
[84] Michael Zink, Kyoungwon Suh, et al., Watch global, cache local: YouTube network traffic at a campus network: measurements and implications, in: Multimedia Computing and Networking 2008, vol. 6818, 2008, pp. 35–47, http://dx.doi.org/10.1117/12.774903.
[85] Juncheng Yang, Ziming Mao, Yao Yue, K.V. Rashmi, {GL-Cache}: Group-level learning for efficient and high-performance caching, in: 21st USENIX Conference on File and Storage Technologies, FAST 23, 2023, pp. 115–134, http://dx.doi.org/10.5555/3585938.3585946.
[86] James Warren, Nathan Marz, Big Data: Principles and Best Practices of Scalable Realtime Data Systems, Simon and Schuster, 2015.
[87] Rahul Dey, Fathi M. Salem, Gate-variants of gated recurrent unit (GRU) neural networks, in: 2017 IEEE 60th International Midwest Symposium on Circuits and Systems, MWSCAS, IEEE, 2017, pp. 1597–1600.
[88] Tao Lei, Yu Zhang, Sida I. Wang, Hui Dai, Yoav Artzi, Simple recurrent units for highly parallelizable recurrence, 2017, arXiv preprint arXiv:1709.02755.
[89] Hui Liu, Chengqing Yu, Chengming Yu, A new hybrid model based on secondary decomposition, reinforcement learning and SRU network for wind turbine gearbox oil temperature forecasting, Measurement 178 (2021) 109347, http://dx.doi.org/10.1016/j.measurement.2021.109347.
[90] Juntao Liu, Chuang Yu, Zhuhua Hu, Yaochi Zhao, Xin Xia, Zhigang Tu, Ruoqing Li, Automatic and accurate prediction of key water quality parameters based on SRU deep learning in mariculture, in: 2018 IEEE International Conference on Advanced Manufacturing, ICAM, IEEE, 2018, pp. 437–440.
[91] Xingyue Cui, Zhe Chen, Fuliang Yin, Speech enhancement based on simple recurrent unit network, Appl. Acoust. 157 (2020) 107019.