Incremental Least-Recently-Used Algorithm: Good, Robust, and Predictable Performance

Jinbei Zhang, Chunpeng Chen, Kechao Cai, and John C. S. Lui, *Fellow, IEEE*

***Abstract*—This paper proposes a replacement algorithm for file caching in mobile edge computing (MEC) networks. While there are numerous schemes for file replacement, it remains a challenge to achieve good, robust, and predictable performance simultaneously. To address this challenge, we introduce a general scheme called Incremental Least-Recently-Used (iLRU), which builds on the classic Least-Recently-Used (LRU) algorithm. iLRU initially caches only a "portion" of the file upon the first request and incrementally caches more when there are more requests for the file. In this regard, the request frequency can be inferred from the cached size without incurring additional overhead, where a larger cached size represents a higher request frequency. We derive the theoretical hit ratio of iLRU based on the Time-to-Live (TTL) analysis. With the Time-to-Live (TTL) analysis, we can theoretically derive the hit ratio and properties of iLRU and notably show that iLRU allocates more cache space to popular files, resulting in a higher hit ratio than LRU. Simulation results demonstrate the superior performance of iLRU and validate the accuracy of the theoretical hit ratio. Furthermore, we conduct simulations over various real-world traces to show that iLRU outperforms existing schemes across various real-world traces, defenestrating the robustness of iLRU.**

***Index Terms*—Mobile edge computing, replacement algorithm, analytical model, performance evaluation.**

I. Introduction

MOBILE Edge Computing (MEC) has attracted significant attention in recent years for addressing the challenges of resource-intensive and delay-sensitive applications in mobile networks [2], [3], [4], [5]. MEC servers provide computing and caching resources, enabling them to function as cache nodes that store frequently requested content closer to end users [6]. Many MEC applications, such as video analytics and distributed machine learning, depend on frequent access to large data resources, including object databases and trained models [3], [7]. By storing such data at the edge, MEC significantly reduces backhaul retrieval between base stations and backend servers, alleviating network congestion and minimizing end-to-end delays [2], [8], [9]. Therefore, caching is a fundamental component of MEC networks [10].

Caching follows a basic hit/miss model. A request is served from the cache (a "hit") if the requested file is available; otherwise, it is retrieved from a remote source (a "miss"), potentially replacing an existing cached file. The hit ratio, defined as the proportion of hits to total requests, depends on the effectiveness of the cache replacement algorithm. While replacing the file least likely to be requested in the future achieves optimal performance theoretically [11], [12], predicting future request patterns is challenging due to the spatiotemporal volatility introduced by user mobility in MEC networks [8], [13].

Traditional cache replacement algorithms, such as Least Recently Used (LRU) and Least Frequently Used (LFU), may perform suboptimally under limited cache capacity and dynamic request patterns [14]. LRU, which evicts the least recently requested file, assumes temporal locality but neglects frequency, resulting in lower hit ratios under limited cache capacity [15]. In contrast, LFU evicts the least frequently used file based on historical requests, achieving optimal performance under the Independent Reference Model (IRM), where requests are independently drawn from a fixed distribution over all files [16], [17]. However, LFU is struggling with outdated statistics in dynamic environments [18], [19], and its high time complexity makes it less suitable for MEC networks.

Trying to strive for some balance between LRU and LFU, many replacement algorithms are proposed [15], [18], [19], [20], [21], [22], [23], [24], [25]. Among them, ARC and TinyLFU are two representative algorithms. ARC [19] uses two self-tuning lists for recently and frequently accessed files, which has low overhead to infer popular files in changing request patterns. TinyLFU [18] uses a space-efficient data structure called Counting Boom Filter (CBF) to store the historical frequency information, resulting in reduced query overhead. TinyLFU also introduces an aging mechanism to update the frequency statistics. Hence, ARC and TinyLFU achieve a good and robust performance. However, these algorithms may rely on complex structures that complicate theoretical analysis and increase computational overhead, limiting their applicability in MEC scenarios.

Received 5 August 2024; revised 16 December 2024; accepted 19 February 2025. Date of publication 3 March 2025; date of current version 5 June 2025. This work was supported in part by National Key R&D Program of China under Grant 2022YFB2902700, in part by the NSF China under Grant 62471505, Grant 62071501, and Grant 62202508, in part by Shenzhen Science and Technology Program under Grant JCYJ20220818102011023, Grant 20220817094427001, and Grant ZDSYS20210623091807023. An earlier version of this paper was presented at the 2024 IEEE International Conference on Communications Workshops [DOI: 10.1109/ICCWorkshops59551.2024.10615665]. Recommended for acceptance by C. M. Pinotti. *(Corresponding author: Kechao Cai.)*

Jinbei Zhang, Chunpeng Chen, and Kechao Cai are with the School of Electronic and Communication Engineering, Sun Yat-sen University, Shenzhen 519082, China (e-mail: zhjinbei@mail.sysu.edu.cn; chenchp6@mail2.sysu.edu.cn; caikch3@mail.sysu.edu.cn).

John C. S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong,, Hong Kong (e-mail: cslui@cse.cuhk.edu.hk).

This article has supplementary downloadable material available at https://doi.org/10.1109/TMC.2025.3547066, provided by the authors.

Digital Object Identifier 10.1109/TMC.2025.3547066

Motivated by the above observations, we introduce a novel cache replacement framework called Incremental Least-Recently-Used (iLRU) tailored for MEC networks to achieve robust, predictable, and high performance. Unlike traditional cache replacement algorithms, iLRU incorporates an innovative approach called the incremental caching process. Initially, iLRU caches only a portion of a requested file and incrementally increases the cached size with subsequent requests for the same file. This method is valid as partial caching of files is common in web caching and has been investigated by several studies [26], [27], [28], [29], [30], [31], [32].

Our theoretical analysis is based on the TTL analysis, which uses the hit ratio of TTL cache to approximate that of LRU cache [33], [34]. Various studies validate the robustness and accuracy of the TTL approximation [31], [35], [36], [37], [38], [39]. Within the iLRU framework, we introduce a Markov chain to illustrate the transitions between caching states, and each caching state represents one cached size. These transition probabilities are deduced through the TTL analysis. By solving each state probability, we can derive the theoretical hit ratio of iLRU, thereby making its performance predictable. Notably, we prove that iLRU allocates more cache space to popular files and achieves a higher hit ratio than LRU. Moreover, we prove that it is optimal for iLRU to cache one file wholly at the file's last caching state.

We simulate the hit ratios and cache occupancy of LRU and iLRU, empirically validating our theoretical analysis. Meanwhile, we compare the hit ratios from trace-driven simulations with the theoretical results to validate our analytical model's accuracy and investigate the relationship between the hit ratio and the number of caching states. We simulated the TTL and the duration of the content in the cache, demonstrating that our scheme has a superior performance in retaining popular files. The performance of iLRU is also simulated when the file sizes are heterogeneous. Compared with existing works over various real traces, iLRU shows superior performance, validating its robustness. Moreover, we further introduce a modified iLRU scheme (called W-iLRU) to improve its performance when the requests are burst in a short term.

Our contributions can be summarized as follows:

- We introduce a novel cache replacement framework called Incremental Least-Recently-Used (iLRU), whose performance is good, robust, and predictable. And, its time complexity is O(1).
- We derive the theoretical hit ratio of iLRU based on TTL analysis and theoretically prove that it is better than that of LRU under the Independent Reference Model (IRM). We also prove that within the iLRU framework, it is optimal to cache files wholly for the last caching state.
- The trace-driven simulations verify the superior performance of iLRU and show that iLRU allocates more cache space to high-frequency files than LRU under the IRM. The simulations also demonstrate that iLRU performs well when file sizes are heterogeneous and that it outperforms existing schemes over various real traces, verifying its robustness.
- We develop W-iLRU scheme to enhance the robustness of iLRU in the scenario of request bursts and find small buffer size of W-iLRU can perform well in real-world traces.

The remainder of the paper is organized as follows. Section II introduces related works. Section III presents the system model and assumptions used in this paper. In Section IV, we present the iLRU scheme and the corresponding analysis. The simulations and results are in Section V. Finally, Section VI concludes the paper.

II. Related Work

In this section, we introduce related works on cache replacement algorithms, TTL analysis, and partial caching.

A. Cache Replacement Algorithm

The key features of cache replacement algorithms include recency and frequency. Recency captures temporal locality, assuming recently accessed files are more likely to be accessed again soon. Frequency leverages historical access patterns, assuming frequently requested files in the past are likely to remain popular.

In Mobile Edge Computing (MEC) environments, user mobility introduces spatiotemporal volatility, resulting in dynamic request patterns that significantly challenge traditional caching algorithms [8], [13]. Frequent changes in content popularity due to user movement across different edge nodes complicate decisions regarding what files to cache and evict. This mobility-induced dynamism demands cache replacement algorithms that can adapt effectively to these changing patterns.

Least Recently Used (LRU) is a classic recency-based algorithm that always evicts the least recently used file [19], [23]. LRU is commonly implemented using a doubly linked list with a hash table, known as the LRU list, which enables its time complexity O(1) [40]. LRU performs well in changing request patterns, but it does not store any frequency information, resulting in LRU allocating a considerable amount of cache space to low-frequency files, as shown in our simulations. Hence, its hit ratio is low when the request pattern is stable.

Furthermore, LRU shows poor performance in scenarios with files of heterogeneous sizes. The most typical example is that caching a large file will result in the eviction of many small files, which may result in a rapid drop in the hit ratio. [41] introduced this issue and proposed AdaptSize, a self-tuning algorithm based on a probabilistic, size-aware admission policy. This algorithm tends to cache the smaller objects and dynamically tunes its parameters to achieve better object hit ratios.

Least Frequently Used (LFU) is a frequency-based algorithm, which is known to be optimal under the Independent Reference Model (IRM) [16], [17]. In common, LFU maintains a frequency table with counters for each file. When there is insufficient space to accommodate new files, LFU searches the file with the minimum frequency count and evicts it from the cache. The time complexity of LFU is O($\log n$), where n is the number of files in the frequency table.

Although LFU achieves the highest hit ratio under the IRM, its frequency estimates can become outdated in changing request

patterns, and the logarithmic time complexity presents a significant overhead in practical environments [18]. To mitigate this overhead, [18] proposed the TinyLFU scheme, which utilizes Bloom Filter and Sketch structure [42], [43], [44] to store the historical frequency information. TinyLFU always keeps the item with the higher frequency between the new item waiting to be cached and the candidate for eviction, which reduces time complexity to O(1) and enables efficient caching of the frequently accessed files.

Many caching algorithms have been proposed to improve performance by combining recency and frequency, including LRU-K, SLRU, 2Q, MQ, ARC, FB-FIFO, TinyLFU, LHD, S3-FIFO and others [15], [18], [19], [20], [21], [22], [23], [24], [25]. Among them, ARC and TinyLFU are two representative algorithms.

Adaptive Replacement Cache (ARC) is a self-tuning and low overhead cache replacement algorithm [19]. ARC employs two lists: L_1 for recently accessed items and L_2 for frequently accessed items and equips ghost cache for historically deleted items of each list. ARC dynamically adjusts the lengths of these lists to tune the ratio of cached recently accessed and frequently accessed files according to the changing request patterns. This mechanism allows ARC to remove outdated items and contain the currently popular items. Moreover, ARC has low overhead, and its time complexity is O(1).

TinyLFU [18] has an aging mechanism to keep the frequency statistics fresh. [18] also proposed a novel framework called W-TinyLFU, which uses an LRU buffer ahead of the TinyLFU to cache the recently accessed files. Therefore, it can adapt to the recent request pattern changes.

Although these algorithms work well across a wide range of workloads, they rely on the complex structures that complicate their theoretical analysis and increase computational overhead, limiting their applicability in MEC scenarios.

B. TTL Analysis

Time-to-Live (TTL) analysis is a method for estimating a file's expiration time, representing the expected duration a cached file remains useful before eviction [36], [45], [46]. This approach commonly utilizes the TTL concept to estimate the stationary hit ratio of LRU and its variants. TTL analysis was initially introduced in [33] and later refined using mean field approximation and stochastic process theory in [34] and [35]. This method has been extensively utilized in numerous studies [31], [36], [37], [38], [39].

TTL analysis approximates the hit ratio of an LRU cache with the hit ratio of a TTL cache. TTL cache sets a deterministic TTL timer with an initial value, denoted as T, for each cached file. When the timer expires, the corresponding file is removed from the cache. If a file is requested before its timer expires, the timer is reset. This approximation adjusts the T to guarantee that the expected number of cached items in the TTL cache is equal to the cache capacity S of the LRU cache. Assume there are n files with size one, and the request probability for file i is p_i, the cache capacity is

$$S = \sum_{i=1}^{n} \left(1 - e^{-p_i T}\right). \tag{1}$$

Additionally, [38] extends TTL analysis to evaluate the performance of two classes of cache replacement algorithms with multiple LRU lists under the Independent Reference Model (IRM). Meanwhile, [41] applies TTL approximation to scenarios with variable LRU list lengths, demonstrating its efficacy when the LRU list length changes over time.

C. Partial Caching

The topic of partial caching or chunked caching has been studied extensively in various works, including [26], [27], [28], [29], [30], [31], [32]. Among these, the works by [26], [27], [31] are closely related to ours.

In the context of multimedia scenarios, [26] introduced the performance benefits of partial caching. Building upon this work, [27] extended the investigation to three types of partial caching policies: fixed, pyramid, and skyscraper. In these studies, a media file is divided into multiple equal-sized blocks, which serve as the smallest unit of transfer. Multiple chronologically adjacent blocks are then grouped into segments. The sizes of these segments are either fixed, increase exponentially, or resemble a skyscraper from the beginning of the video. Furthermore, the cache is divided into two portions corresponding to two LRU stacks: the first portion is dedicated to the initial segments, while the second portion handles the later segments. These later segments use an admission policy determined by the relative popularity and size of each segment.

[31] proposed the GiLRU, which also divides videos into blocks/chunks and caches them progressively in a fixed pattern upon request. The eviction policy of GiLRU is similar to the LRU policy: it moves the hitting file with all its chunks to the head of the cache but allows partial chunks of the file at the tail to be evicted to free up space for new file chunks. Before all blocks of a file are evicted, the file can still append blocks and be moved to the head of the cache with subsequent requests.

The differences between our work and previous studies are as follows.

Scope of Content: The works by [26], [27], [31] primarily focus on video transmission, where typically only a portion of the video is delivered because most users do not watch videos in their entirety. Specifically, [26], [27], [31] aim to cache the initial segments or chunks to enable immediate playback for more user requests. These works prioritize caching portions at critical positions within the original file. In contrast, our approach targets mobile edge computing (MEC) applications that require complete files to support data-intensive tasks, where the positions of individual segments are not prioritized. MEC environments increasingly rely on applications driven by Big Data and machine learning models, which typically demand access to the full dataset or model. For such applications, caching whole files is essential to minimize data transfer overhead and improve service quality. Therefore, our scheme manages the size of cached portions solely by file popularity and overall cache capacity constraint without emphasizing its position within a file.

Algorithm Flexibility: Existing works [26], [27], [31] specify fixed increments in the size of files cached between nearby requests. Our approach allows for arbitrary increments, making

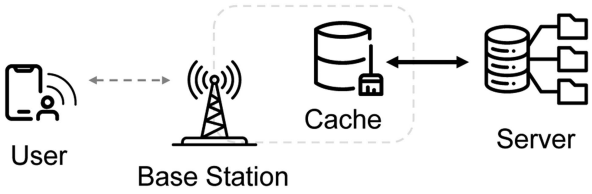

Fig. 1. Network model.

our scheme a generalization of these previous methods. Moreover, our scheme fully evicts the candidate file when space is insufficient, adhering to a pure LRU policy, which makes it easy to deploy on existing cache systems.

Performance Evaluation: Most importantly, we provide theoretical analysis to show that iLRU outperforms LRU, whereas [26], [27], [31] rely solely on simulations to evaluate performance. To our knowledge, this is the first work to theoretically demonstrate the superior performance of incremental caching. We also prove that it is optimal to cache files wholly for the last caching state and discuss how the design of the incremental caching process impacts performance. Finally, we investigate the robustness of the iLRU scheme in real-world scenarios and propose W-iLRU to enhance its robustness further.

III. System Model and Assumptions

In this paper, similar to the classic model of cache replacement schemes in the scenarios of MEC [47], we consider a model consisting of one server, one base station (BS) with a cache, and one user with multiple requests, as shown in Fig. 1. When a user request arrives at the BS, a hit occurs if the file is cached. Otherwise, a miss occurs, and the BS requests the file from the server. After receiving the request, the server sends the file to the user via the reverse path. When the file arrives at the BS, the BS cache can store a partial file according to the cache replacement scheme.

Our model follows the independent reference model (IRM), which assumes that user requests arriving at the cache are independent and identically distributed (i.i.d.) random variables [33], [38]. In this context, users request files from the server at discrete time slots $t = 0, 1, 2, \ldots$, with each request treated as an i.i.d. random variable, independent of previous requests.

User mobility is a critical factor in MEC networks, as it influences cache performance through request patterns and content popularity [2], [47], [48], [49]. Mobility causes fluctuations in local content demand as users move across different network regions, leading to dynamic changes in observed content popularity at edge caches. This variability impacts caching performance by introducing challenges in maintaining an optimal cache hit ratio and efficient resource utilization. While this study focuses mainly on optimizing caching mechanisms rather than predicting content popularity or mobility patterns, we account for the impact of mobility through the average content popularity observed at the target cache, similar to prior work [47].

Assume that the file set on the server is $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, where $|\mathcal{L}| = n$. The file l_i with size s_i is requested with probability p_i, where $\sum_{i=1}^{n} p_i = 1$. Without loss of generality, we assume that $p_i \geq p_{i+1}$ for $1 \leq i \leq n-1$. The cache capacity of the base station (BS) is S, and assume that $\sum_{i=1}^{n} s_i \gg S$, i.e., the cache of the base station cannot store all files.

Assume that each file l_i has a series of request states $I_i(t)$ for $t = 0, 1, 2, \ldots$. If file l_i is requested at time t, $I_i(t) = 1$, and $I_j(t) = 0$ for other files $j \neq i$. Otherwise, $I_i(t) = 0$. Let $r_i(t)$ denote the cached size of file l_i at time t where $0 \leq r_i(t) \leq s_i$, and the total cached size must satisfy $\sum_{i=1}^{n} r_i(t) \leq S$.

Additionally, we assume that each request is fully processed before the next one arrives. This means that any changes to the cache contents, whether due to a hit or a miss, are completed before the next request begins processing. Thus, there is no overlap or interference between the processing of consecutive requests.

The main performance metric is the average hit ratio over time, which is the ratio between the number of hit file portions and the total number of requested files and is expressed as

$$H = \lim_{m \to +\infty} \frac{\sum_{t=0}^{m} \sum_{i=1}^{n} r_i(t) I_i(t)}{\sum_{t=0}^{m} \sum_{i=1}^{n} s_i I_i(t)}, \tag{2}$$

where m is the total number of observed time slots.

IV. iLRU Scheme and Performance Analysis

In this section, we introduce the iLRU caching scheme and conduct its performance analysis.

A. iLRU Scheme

In contrast to the classic Least Recently Used (LRU) scheme, Incremental Least Recently Used (iLRU) scheme employs an incremental caching approach. Initially, iLRU caches a portion of the requested file and incrementally caches additional portions upon subsequent requests until the file is fully cached.

1) Caching State: We define the caching state by the cached size and order it by the cached size. We define $R_{i,K}$ as an ordered set of caching states for file l_i, where $R_{i,K} = \{r_{i,1}, \ldots, r_{i,k}, \ldots, r_{i,K}\}$. Here, $r_{i,k}$ is the k-th caching state for file l_i, and K is the total number of states.

For initial caching state of file l_i, $r_{i,1} = 0$, indicating that the file is not cached. As requests arrive, the file l_i changes to the next caching state, and the scheme increments the cached size of the requesting file such that $r_{i,k+1} > r_{i,k}$. The final caching state, $r_{i,K} = s_i$, indicates that the file is fully cached when receiving sufficient requests.

Note that iLRU scheme has two constraints for the caching states. First, the initial state must be zero, which is straightforward as it represents an uncached file. Second, the final state must correspond to the file being fully cached, which will be shown to be optimal in Section IV-C.

2) Procedures of iLRU: The iLRU scheme involves two main phases: the request phase and the update phase, as illustrated in Fig. 2. The pseudo code of iLRU is presented in Algorithm 1.

Request Phase: iLRU manages file requests, including request querying and caching state determination. When a request x_t arrives at time t, if x_t is in the LRU list, a (partial) hit occurs, and x_t is moved to the head of the LRU list, as in line 3 of Algorithm 1. Otherwise, if a miss occurs, iLRU inserts x_t to the head of the LRU list (Line 5).

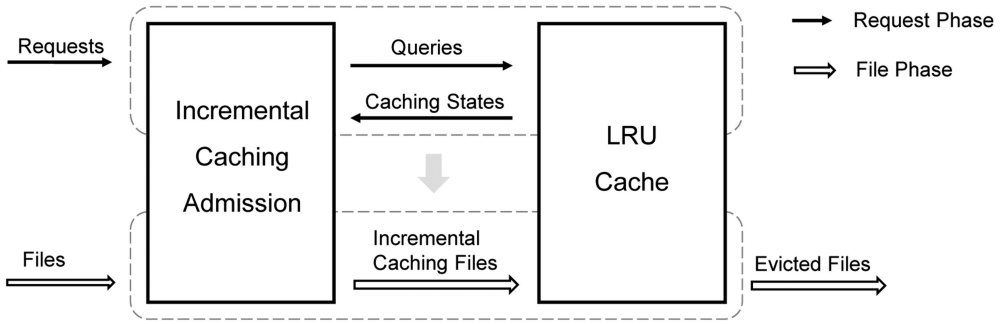

Fig. 2. Procedure of iLRU scheme.

Algorithm 1: iLRU Scheme.

Input: Sequence of requests $x_1, x_2, \ldots, x_t, \ldots$.
Set of caching states $R_{i,K} = \{r_{i,1}, r_{i,2}, \ldots, r_{i,K}\}$.
Empty LRU list L.
Initial cache occupancy $C = 0$.
Cache capacity S.
Output: Cached files in LRU list L.
Caching states of cached files.

1: **for all** x_t **do**
2: **if** x_t is in L **then**
3: A cache hit has occurred, move x_t to the head of L.
4: **else**
5: A cache miss has occurred, insert x_t to the head of L.
6: **end if**
7: CACHING(x_t).
8: **end for**
9: **SubFunction** CACHING(x_t):
10: **if** x_t is not fully cached **then**
11: Retrieve the remaining file of x_t from the server.
12: Obtain the current caching state $r_{x_t,k}$, and calculate the required cache space $d = r_{x_t,k+1} - r_{x_t,k}$.
13: **while** $C + d > S$ **do**
14: Remove x_{tail} from L, delete the linked file in cache, and update C.
15: **end while**
16: Perform incremental caching of x_t with the increased cached size d, and update C.
17: **end if**

Update Phase: iLRU manages file caching operations, including incremental caching and eviction of the least recently used (LRU) files when cache space is insufficient. If the cache has not stored the whole file (partial hit or miss occurs), the remaining portions of file are retrieved from the server (Line 11). Then, the caching state transitions from $r_{x_t,k}$ to $r_{x_t,k+1}$, increasing the cached size by $r_{x_t,k+1} - r_{x_t,k}$ (Line 12). If the cache fully caches the file, no further caching occurs during this period. When the cache has insufficient space to accommodate new file contents, it evicts the least recently used file (Lines 13–15).

3) Advantages: iLRU scheme comprises two modules: an incremental caching admission module and an LRU eviction module, as illustrated in Fig. 2. This design facilitates seamless integration with existing LRU caches, enhancing their functionality by addressing some inherent limitations.

Traditional LRU algorithm prioritizes the most recently used items but overlooks access frequency, while LFU algorithms track access frequency but suffer from an aging problem. Compared to both, iLRU can infer file request frequencies from the cached file sizes by the incremental caching mechanism, thereby addressing the limitation of LRU algorithms that do not track frequency. Additionally, iLRU ensures a similar lifetime for all recently requested files to maintain the freshness of frequently requested files using the LRU eviction policy, thus alleviating the aging problem observed in LFU algorithm.

On the other hand, schemes like ARC and TinyLFU, which use more complex mechanisms, tend to retain older frequently used files in the cache for a longer time, potentially leading to lower cache efficiency. Consequently, iLRU effectively caches recent frequently accessed files, outperforming both traditional LRU and LFU algorithms.

More importantly, the performance of iLRU is theoretically guaranteed due to its simple caching behavior. The theoretical model and analysis are provided in the next subsection.

B. Performance Metrics of iLRU

Since each request in the independent reference model (IRM) is independent, we model the changes in caching states for each file as a Discrete Markov Chain (DMC).

The DMC involves two types of transitions, i.e, one type is triggered by a request arrival, and the other type is triggered by a file eviction. When a request for file l_i arrives, its state advances to the next state with a transition probability, called the incremental caching probability, denoted as $P_{i,in}$. We assume that the next state of the last state $r_{i,K}$ is itself, indicating that once a file is fully cached, it remains in that state upon subsequent requests. Upon eviction of the file, its state resets to the initial state, with the transition probability referred to as the eviction probability, denoted as $P_{i,ev}$. The DMC is shown in Fig. 3.

Under the Time-to-Live (TTL) analysis, the time that a file remains in the cache is known as the characteristic time [34], which is deterministic and independent of the file itself, denoted

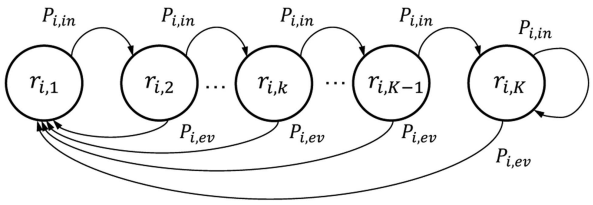

Fig. 3. Discrete Markov Chain of incremental caching process for file l_i.

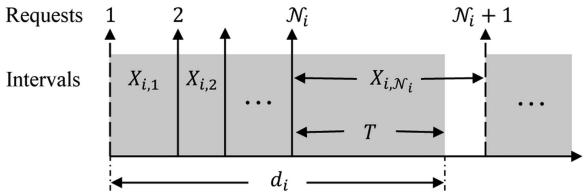

Fig. 4. Duration of file l_i in the cache.

as T. If a request for a cached file does not arrive before the time expires, the cache evicts the file, and this probability is $P_{i,ev} = (1-p_i)^T$. Otherwise, the remaining time is updated to T, and the probability is $P_{i,in} = 1-(1-p_i)^T$.

We then derive the hit ratio of iLRU with K caching states. The transition matrix for the caching process of file l_i denoted as $\mathbf{P}_{i,K}$ is

$$\begin{pmatrix} (1-p_i)^T & \left(1-(1-p_i)^T\right) & \cdots & 0 & 0 \\ (1-p_i)^T & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (1-p_i)^T & 0 & \cdots & 0 & \left(1-(1-p_i)^T\right) \\ (1-p_i)^T & 0 & \cdots & 0 & \left(1-(1-p_i)^T\right) \end{pmatrix}.$$

This matrix represents the transition probabilities between caching states. Specifically, the entry at position $(i, i+1)$ represents the incremental caching probability. The entries in the first column uniformly indicate the eviction probability from any state to the initial state. All other entries are zero.

According to the Poisson Arrival See Time Averages (PASTA) theorem [50], the probability of each state, denoted as $F_{i,k}$, $k = 1, 2, \ldots, K$, is equal to the steady state probability of the DMC, which can be solved from equations $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}_{i,K}$ and $\sum_{k=1}^{K} \pi_k = 1$.

Lemma 1: For K-state iLRU, the probability of state k ($k = 1, 2, \ldots, K$) for file l_i can be expressed as

$$F_{i,k}(T) = \begin{cases} \left(1-(1-p_i)^T\right)^{k-1}(1-p_i)^T, & k \le K-1 \\ \left(1-(1-p_i)^T\right)^{K-1}, & k = K \end{cases}. \tag{3}$$

Note that $r_{i,k}$ is the cached size of file l_i in caching state k ($k = 1, 2, \ldots, K$) in the K-state iLRU scheme. The cache occupancy of the iLRU scheme is the sum of each file's expected cached size, which is

$$C_K(T) = \sum_{i=1}^{n}\sum_{k=1}^{K} r_{i,k}F_{i,k}(T). \tag{4}$$

Then, the hit ratio of iLRU can be readily derived as follows.

Theorem 1: The hit ratio of K-state iLRU is

$$H_K(T) = \frac{\sum_{i=1}^{n} p_i \sum_{k=1}^{K} r_{i,k}F_{i,k}(T)}{\sum_{i=1}^{n} p_i s_i}. \tag{5}$$

Proof: The average hit ratio defined in (2) describes a hit ratio on a request sequence, which is

$$H = \lim_{m\to+\infty} \frac{\sum_{t=0}^{m}\sum_{i=1}^{n} r_i(t)I_i(t)}{\sum_{t=0}^{m}\sum_{i=1}^{n} s_i I_i(t)}.$$

Since each request is independent, we can express H as

$$H = \lim_{m\to+\infty} \frac{\sum_{i=1}^{n}\sum_{k=1}^{K} r_{i,k}\sum_{t=0}^{m} I_{i,k}(t)}{\sum_{i=1}^{n} s_i \sum_{t=0}^{m} I_i(t)},$$

where $I_{i,k}(t) = 1$ when file l_i is requested at time t and its caching state is k before this request. Otherwise, $I_{i,k}(t) = 0$.

Assume m_i is the number of time slots requested for file l_i. Let $m_{i,k}$ be the number of time slots requested for file l_i when its caching state is k, where $m_i = \sum_k m_{i,k}$.

Then, we have

$$H = \lim_{m\to+\infty} \frac{\sum_{i=1}^{n}\sum_{k=1}^{K} r_{i,k}m_{i,k}}{\sum_{i=1}^{n} s_i m_i}.$$

By dividing both the numerator and the denominator by m, we have

$$H = \lim_{m\to+\infty} \frac{\sum_{i=1}^{n}\sum_{k=1}^{K} r_{i,k}\cdot(m_{i,k}/m)}{\sum_{i=1}^{n} s_i \cdot (m_i/m)}.$$

As $m \to +\infty$, $m_{i,k}/m$ converges to $p_i F_{i,k}(T)$ and m_i/m converges to p_i, where $F_{i,k}(T)$ represents the steady-state probability of file l_i being in state k.

$$H = \frac{\sum_{i=1}^{n}\sum_{k=1}^{K} r_{i,k}\cdot p_i F_{i,k}(T)}{\sum_{i=1}^{n} s_i \cdot p_i}.$$

Therefore, the expected hit ratio of K-state iLRU has been derived. ■

(4) and (5) are both related to T. When S, p_i and s_i are known, letting $C = S$, we can obtain the value of T via (4), and finally calculate the hit ratio of iLRU via (5).

Additionally, we derive the mean duration of file l_i in the cache, denoted as d_i. This duration represents the interval between the insertion of a file into the cache and its subsequent eviction, as shown in Fig. 4. Assume that there are a series of requests for file l_i, where the first request results in a cache miss and the file is cached incrementally. Let $X_{i,j}$ denote the interval between the j-th request and the $j+1$-th request, where $j = 1, 2, \ldots$. Assume that the file is evicted after $\mathcal{N}_i$ requests, and the $(\mathcal{N}_i+1)$-th request causes another cache miss. Thus, we have $X_{i,j} \le T$ for $j < \mathcal{N}_i$ and $X_{i,j} > T$ for $j = \mathcal{N}_i$.

Thus, the mean duration d_i can be represented as

$$d_i = E\left[\sum_{j=1}^{\mathcal{N}_i-1} X_{i,j}\right] + T, \tag{6}$$

where T is the characteristic time of the cache, and $E[\cdot]$ represents the expected value of random variable.

Lemma 2: Under the IRM, the mean duration of file l_i in the cache is given by

$$d_i = \frac{1}{p_i(1-p_i)^T} - \frac{1}{p_i}. \tag{7}$$

Proof of Lemma 2: Since the request interval $X_{i,j}$ follows geometric distribution with success probability p_i, we have

$$E[X_{i,j}|X_{i,j} \le T] = \frac{1-(T+1)(1-p_i)^T + T(1-p_i)^{T+1}}{p_i\left(1-(1-p_i)^T\right)}. \tag{8}$$

Similarly, the random variable $\mathcal{N}_i$ also follows geometric distribution with success probability $(1-p_i)^T$, representing the file l_i is not requested for time T, thus we have

$$E[\mathcal{N}_i] = \frac{1}{(1-p_i)^T}. \tag{9}$$

Since $X_{i,j} \le T$ for $j < \mathcal{N}_i$, applying Wald's equation, we obtain

$$\begin{aligned} E\left[\sum_{j=1}^{\mathcal{N}_i-1} X_{i,j}\right] &= E[X_{i,j}|X_{i,j} \le T]E[\mathcal{N}_i - 1] \\ &= \frac{1}{p_i(1-p_i)^T} - \left(\frac{1}{p_i} + T\right). \end{aligned} \tag{10}$$

Substituting (10) into (6), we can obtain

$$d_i = \frac{1}{p_i(1-p_i)^T} - \frac{1}{p_i}. \tag{11}$$

Therefore, the lemma is proved. □

C. Properties of iLRU

In this section, we theoretically prove the superior performance of iLRU and its optimality in fully caching at the final state. Additionally, we analyze the complexity of iLRU.

First, we introduce two lemmas before proving that the Three-state iLRU (TiLRU) outperforms LRU in hit ratio when file sizes are equal. Assume the file size of all files be s. The set of caching states for TiLRU is $R_{i,3} = \{0, r, s\}$ for any file l_i, where the second caching state $r_{i,2} = r \in (0, s)$. Note that if iLRU has two states, it is equivalent to LRU, and its set of caching states is $R_{i,2} = \{0, s\}$ for any file l_i.

Next, we analyze the cache occupancy across different files. Let the TTL of LRU be T_2 and $u_{i,2} = (1-p_i)^{T_2}$. Denote the cache occupancy of LRU for file l_i as $h_{i,2}$. Then, based on Lemma 1, we can easily derive that

$$h_{i,2} = s_i(1-u_{i,2}). \tag{12}$$

Similarly, for iLRU, let the TTL be T_3 and $u_{i,3} = (1-p_i)^{T_3}$. The cache occupancy of TiLRU for file l_i is

$$h_{i,3} = ru_{i,3}(1-u_{i,3}) + s(1-u_{i,3})^2. \tag{13}$$

Then, we have the following lemma.

Lemma 3: When $h_{i,2} = h_{i,3}$, the inequality holds, i.e., $\frac{\mathrm{d}h_{i,3}}{\mathrm{d}p_i} \ge \frac{\mathrm{d}h_{i,2}}{\mathrm{d}p_i}$, where $p_i \in (0,1)$.

The proof of Lemma 3 is presented in Appendix A. Lemma 3 shows that when the cache occupancy is equal, the cache occupancy of TiLRU increases more rapidly with the request probability than that of LRU. Under a constant cache capacity, this means TiLRU allocates more cache space to high-frequency files, enhancing the hit ratio since it favors more frequently requested files.

After analyzing the cache occupancy, we focus on the hit ratio of TiLRU. The hit ratio of TiLRU can be derived from (5) as follows.

$$H_3(r) = \frac{1}{s}\cdot\sum_{i=1}^{n} p_i\left(r\cdot u_i(1-u_i) + s(1-u_i)^2\right), \tag{14}$$

where $u_i = (1-p_i)^T$, and T can be solved from (4) given the cache capacity.

Lemma 4: For a given cache capacity, we have

$$\frac{\mathrm{d}H_3(r)}{\mathrm{d}r} < 0.$$

The proof of Lemma 4 is in Appendix A. Lemma 4 shows that the hit ratio $H_3(r)$ decreases as the cached size of the second state r increases. With Lemma 4, we are ready to prove the following theorem.

Theorem 2: Under the IRM, the performance of TiLRU is superior to that of LRU when the file sizes are equal.

Proof: Denote H_2 as the hit ratio of LRU, where

$$H_2 = \frac{1}{s}\cdot\sum_{i=1}^{n} p_i\left(1-(1-p_i)^T\right), \tag{15}$$

where T can be solved from (4) given the cache capacity.

When r converges to s, iLRU degenerate into LRU. Thus, when the cache capacity is constant, $H_3(s) = H_2$. From Lemma 4,

$$\frac{\mathrm{d}H_3(r)}{\mathrm{d}r} < 0.$$

Since $0 < r < s$, we have $H_3(r) > H_2$, indicating that the hit ratio of TiLRU is higher than that of LRU.

Therefore, Theorem 2 is proved.

When the file sizes are heterogeneous, we consider a variant of Three-state iLRU, referred to as a h-TiLTU, and assume that the second caching state $r_{i,2} = \alpha s_i \in (0, s_i)$, where α is the ratio between cached size and total file size.

Similarly, the hit ratio of h-TiLRU can be derived as

$$H_{3,h}(\alpha) = \frac{\sum_{i=1}^{n} p_i s_i\left(\alpha\cdot u_i(1-u_i) + (1-u_i)^2\right)}{\sum_{i=1}^{n} p_i s_i}. \tag{16}$$

Then, we have the following lemma.

Lemma 5: For a given cache capacity, we have

$$\frac{\mathrm{d}H_{3,h}(\alpha)}{\mathrm{d}\alpha} < 0.$$

The proof of Lemma 5 is also presented in Appendix A. Lemma 5 shows that the hit ratio $H_{3,h}(\alpha)$ decreases as α increases.

Analogous to Theorem 2, based on Lemma 5, we can derive that $H_{3,h}(\alpha) > H_2$, indicating that the hit ratio of h-TiLRU is higher than that of LRU. Therefore, we have the following theorem.

Theorem 3: Under the IRM, the performance of h-TiLRU is superior to that of LRU when the file sizes are heterogeneous.

Until now, we have theoretically shown that TiLRU performs better than LRU. Additionally, we can derive the optimal configuration of TiLRU. As Lemmas 4 and 5 indicate, the smaller the portion of the cached size at the second caching state, the higher the hit ratio. Thus, for an optimal hit ratio, TiLRU only needs to cache a portion of a file as small as possible as an "indicator" upon the first request.

Then, we intuitively analyze the mechanism behind this improvement. TiLRU decides whether to cache a whole file based on its last request interval. If the request intervals are shorter than T, indicating high popularity, the incremental caching process admits the file, acting like a high-pass filter before the LRU list. Meanwhile, iLRU remains sensitive to low-frequency files. If the file becomes unpopular, i.e., the request interval becomes larger than T, iLRU evicts it.

For iLRU with any number of caching states, we theoretically illustrate why the iLRU scheme selects the whole caching as its last caching state.

Theorem 4: For an optimal iLRU scheme, no matter how many states it has, its last caching state must be whole caching.

The proof is presented in Appendix B. The intuition is that when a file transitions to the last caching state, the popularity of the file should be high. Caching these popular files can increase the total hit ratio.

Finally, we analyze the time complexity of iLRU.

Lemma 6: The time complexity of iLRU is O(1).

Proof: In our framework, iLRU comprises two modules: an incremental caching admission module and an LRU eviction module. Let $\mathcal{T}_{in}$ denote the time complexity of the incremental caching admission module, and $\mathcal{T}_{ev}$ denote the time complexity of the LRU eviction module.

First, consider $\mathcal{T}_{in}$. This module involves two main operations: accessing the current caching state and deciding the next caching state. Accessing the current caching state requires querying the LRU cache for the cached size of the current request. Given that this operation can be implemented efficiently, its time complexity is O(1). The decision for the next caching state can be facilitated using a hash structure, which also ensures O(1). Therefore, we have $\mathcal{T}_{in} = \mathrm{O}(1) + \mathrm{O}(1) = \mathrm{O}(1)$.

Next, consider $\mathcal{T}_{ev}$. This module performs the same operations as the traditional LRU eviction algorithm. The time complexity of the LRU algorithm is well-established as O(1) [19]. Therefore, we have $\mathcal{T}_{ev} = \mathrm{O}(1)$.

Combining the complexities of these two modules, the total time complexity of the iLRU algorithm is

$$\mathcal{T}_{\text{iLRU}} = \mathcal{T}_{in} + \mathcal{T}_{ev} = \mathrm{O}(1) + \mathrm{O}(1) = \mathrm{O}(1). \tag{17}$$

Therefore, we conclude this lemma. □

The O(1) time complexity of the iLRU algorithm ensures that the operations implemented within the incremental caching admission module and the LRU eviction module can be executed in constant time, regardless of the cache size or the number of cached files. This efficiency is achieved through a doubly linked list combined with a hash table [40]. The doubly linked list allows quick updates to the cache order, while the hash table enables fast lookups of file positions within the list. Specifically, when a file is accessed, the algorithm retrieves its position via the hash table in O(1) time and then moves it to the head of the list by adjusting finite pointers, also in O(1) time.

On the other hand, this design introduces a memory trade-off. Each node in the doubly linked list requires additional memory to store two pointers (previous and next), and the hash table incurs extra memory overhead for storing key-value pairs [51], [52]. Compared to the traditional LRU algorithm, the iLRU algorithm consumes more memory due to the incremental caching of partial file items. This additional memory usage is a trade-off for achieving higher caching efficiency, where the partial caching size reflects the historical request frequency of each file. These design choices are critical for maintaining high performance in Mobile Edge Computing (MEC) networks.

D. Robustness and W-ILRU Scheme

In real-world scenarios, request patterns change over time. The iLRU scheme dynamically adapts to these changes by inheriting the recency characteristic of LRU, enabling it to update cached content as request patterns evolve. This adaptability contributes to the robustness of the iLRU scheme, which we will illustrate in our simulations.

However, the iLRU scheme has some limits. iLRU caches a file incrementally, resulting in multiple requests being only partially served. This limitation may lead to deviate performance, especially during bursts of short-term requests, such as those following a Pareto-like distribution. In such scenarios, a sequence of requests often targets a small number of files, and in extreme cases, all requests may focus on a single file. During these bursts, the most recently accessed files will likely be requested again, regardless of frequencies. In this scenario, iLRU may experience deviated performance because it requires some time to adapt to the changes in request patterns through the incremental caching process. Thus, although incremental caching can be beneficial for inferring the frequencies, it also poses challenges for iLRU in the scenarios with request bursts.

To address these issues, we developed the W-iLRU scheme, which incorporates a small LRU buffer ahead of the iLRU cache. This design is inspired by the W-tinyLFU scheme [18] and aims to enhance performance by better handling short-term request bursts.

In W-iLRU, when a request arrives and the requested file is not found in either the LRU buffer or the iLRU cache, the file is

inserted at the head of the buffer. If the requested file is found in the buffer, it is moved to the head of the buffer. If the file is located in the iLRU cache, it is incrementally cached and moved to the head of the iLRU cache. When there is insufficient cache space in either the buffer or the iLRU cache, a file is evicted from the tail of the respective cache. Upon eviction from the buffer, the file transitions to the iLRU cache and begins the incremental caching process from the initial state by caching an "indicator".

The advantages of W-iLRU are as follows. If a file experiences a burst of requests, the buffer temporarily stores it while it remains popular. Once its popularity wanes, the iLRU cache prevents it from occupying excessive cache space. Conversely, if the file has stable request patterns, it will be effectively captured by the iLRU cache based on frequency. Additionally, the buffer proportion, a hyper-parameter of W-iLRU that represents the ratio between the buffer size and the total cache capacity, can be manually adjusted based on the workload requirements. The performance and efficiency of W-iLRU will also be evaluated in our simulations.

E. Complexity in MEC Networks

Then, we discuss the complexity and potential implementation challenges of the iLRU scheme in MEC networks, particularly for resource-constrained environments. The iLRU scheme requires dividing files into chunks to support incremental caching, which introduces computational overhead. For example, in a three-state iLRU mechanism, a file is cached in portions across two requests, necessitating its division into at least two chunks. Although these chunks can be divided in advance and stored on remote servers to reduce initial overhead, reassembling them into whole files increases the computational overhead on base stations or user devices. This increased overhead may impact the efficiency and practicality of the iLRU scheme.

To address these challenges, there are several strategies that could be introduced. First, iLRU can adjust the number of caching states based on available computational resources. When computational resources become scarce, iLRU reduces the number of caching states to decrease the number of chunks. Second, the system can be deployed with dedicated edge servers at the base station to handle the chunk reassembly process. When all requested chunks are available at the cache of the base station, dedicated edge servers reassemble them into a whole file and then send the file back to the user.

V. Simulation Results

In our simulations, we compare iLRU (with its variants of TiLRU, DiLRU, and EiLRU) and W-iLRU with several existing caching schemes, including Random, LRU, LFU, ARC, TinyLFU, LHD, S3-FIFO, and Sieve. Specifically, TiLRU initially caches only one unit of a file and fully caches it upon the subsequent request for the same file. DiLRU increases the cached file size exponentially until the file is fully cached, while EiLRU increases it linearly. To accelerate the incremental caching process for large files, we limited the maximum number of states to five, ensuring the entire file is cached when the caching state transitions to the fifth state. ARC and TinyLFU have been introduced in Section II. Least hit density (LHD) is an eviction policy based on hit density using conditional probability [25]. S3-FIFO is characterized by its efficient FIFO-queue-only eviction policy [15]. Sieve is an easy, fast, and surprisingly efficient cache eviction algorithm [53].

We evaluate the performance of these algorithms using various traces. Also, we validated the accuracy of our analytical model and examined the impact of changes in the number of states. Our simulations confirmed the good and robust performance of iLRU.

TABLE I
Summary of Real-World Traces

Trace Name	Data Source or Class
OLTP	CODASYL Database
DS1	Commercial Database
P1-P14	Windows NT Workstation
ConCat	Concat P1-P14
SPC1 like	Long Sequential Scans
S1-S3	Commercial Search Engine
MergeS	Merge S1-S3

A. Simulation Setting and Traces

Our simulation involves both numerical and trace-driven simulations. We implement numerical simulations using the equations detailed in Section IV-B, focusing specifically on the hit ratio. Our trace-driven simulations utilize libCacheSim [54], a high-performance cache simulator designed for running cache simulations and analyzing cache traces. It supports various state-of-the-art eviction algorithms, including ARC, TinyLFU, LHD, S3-FIFO, and Sieve.

Web cache workloads typically follow Power-law (generalized Zipf) distributions [53], [55], where a small subset of objects accounts for a large proportion of requests. To mimic user behavior in MEC networks, we assume that file popularity is drawn from the Zipf distribution, with the request probability defined as

$$p_i = Q \cdot \frac{1}{R(i)^{\beta}}, \tag{18}$$

where β is a constant value. Larger β results in a more skewed distribution. Here, $R(i)$ represents the rank of file frequency, and $Q = \sum_{i=1}^{n} 1/i^{\beta}$, where n is the total number of files.

We also collected real-world traces to test more complex request patterns, such as time-varying file popularity and request bursts, drawn from different production scenarios [19]. These patterns are typical as they reflect the dynamic nature of user requests in MEC environments. Table I summarizes the basic information about these traces.

The OLTP trace captures references to a CODASYL database over one hour, reflecting typical database access patterns common in MEC environments. The traces P1-P14 were collected

from Windows NT workstations using Vtrace, a tool that captures disk operations through device filters, providing insights into user behavior and workload characteristics in desktop environments. Additionally, we tested a trace formed by concatenating these 14 traces in sequence to simulate request mutation scenarios, representing changing user access patterns due to user mobility in dynamic MEC contexts. The trace DS1 was obtained from a commercial database server running an ERP application, showcasing the transactional workloads often encountered in enterprise applications. The SPC1-like trace contains long sequential scans and random accesses, while traces S1-S3 represent disk-read accesses initiated by a large commercial search engine in response to various web search requests, reflecting the diverse and unpredictable request patterns typical in MEC systems.

Lastly, the MergeS trace was created by merging the S1-S3 traces using timestamps of each request to simulate multi-user request environments, which is crucial for assessing cache performance in MEC networks where multiple users access shared resources concurrently. Together, these traces provide a comprehensive evaluation of iLRU's performance across various scenarios, which can also reflect the request patterns in MEC networks.

All file sizes are uniformly set to 64 units in our simulations unless otherwise specified. The file sizes depend on the dividing strategy used in the iLRU scheme. In MEC applications, this strategy can be adjusted based on specific requirements and resources, such as computational resources and data types. Additionally, we use the term "normalized cache capacity" to represent the maximum number of complete files that can be cached.

Note that our simulations begin measuring hit ratios once the cache is full, a state referred to as "transient free" by [56] or "warm start" by [57]. Starting measurements from an empty cache would result in biased misses during initial requests.

B. *Performance in IRM Synthetic Traces*

The IRM synthetic traces can represents stationary and periodic request pattern in the MEC networks. When the parameter β of Zipf distribution is large, only few files are popular, and most requests focus on these files.

We conducted tests using synthetic traces with various βs, each consisting of 10,000 files. Since the results showed consistent trends across different parameters, we only present results of $\beta = 0.9$ in Fig. 5. As the trace distribution becomes more skewed, the plots become steeper. Fig. 5 shows that DiLRU and TiLRU outperform LRU, with DiLRU consistently performing better than TiLRU. The performance improvement of iLRU is significant when cache space is limited. For example, at a normalized cache capacity of 100 and $\beta = 0.9$, the hit ratio gap between DiLRU and LRU can reach 11.69%. LFU shows the best performance. DiLRU performs similarly to TinyLFU, but it has lower complexity. The primary reason for this is that TinyLFU requires maintaining and periodically updating a frequency table, while DiLRU only manages the cached size within an LRU list. DiLRU also outperforms ARC, LHD, and S3-FIFO.

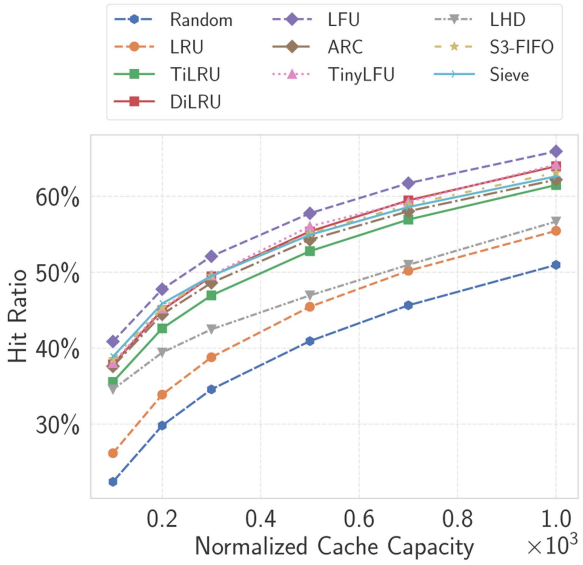

Fig. 5. Hit ratios of various algorithms in the synthetic traces with parameter $\beta = 0.9$.

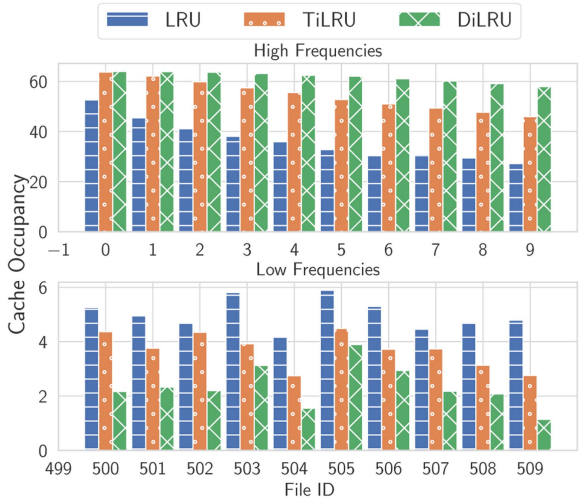

Fig. 6. Cache occupancy variation: iLRU vs. LRU with zipf distribution across files.

We observe that Sieve outperforms iLRU and other algorithms when the cache size is small. This advantage can be attributed to Sieve's capability to quickly evict unpopular files, whereas other algorithms necessitate a longer process for eviction due to their structural design. All tested algorithms, except Random, perform better than LRU, with Random demonstrating the worst performance.

We then analyzed cache occupancy differences across 1,000 files in the synthetic trace with $\beta = 0.5$ and a normalized cache capacity of 100. Fig. 6 shows the results for selected files: files 0 to 9 (most popular) and files 500 to 509 (less popular). The results indicate that iLRU allocates more cache space to popular files and less to unpopular files compared to LRU, verifying Lemma 3.

TABLE II
RELATIVE ERROR (%) OF THE HIT RATIOS BETWEEN THE TRACES AND THE MODEL

Cap.s	LRU		TiLRU		EiLRU		DiLRU	
	0.7	0.9	0.7	0.9	0.7	0.9	0.7	0.9
100	1.28	0.08	1.31	0.06	0.84	0.02	0.78	0.05
500	0.27	0.19	0.30	0.15	0.46	0.15	0.48	0.16
700	0.25	0.20	0.40	0.02	0.50	0.13	0.49	0.11
1000	0.06	0.35	0.37	0.05	0.30	0.08	0.27	0.08
1500	0.01	0.29	0.21	0.13	0.23	0.12	0.22	0.10
3000	0.02	0.13	0.22	0.06	0.10	0.14	0.10	0.13
5000	0.02	0.07	0.04	0.02	0.04	0.17	0.05	0.16

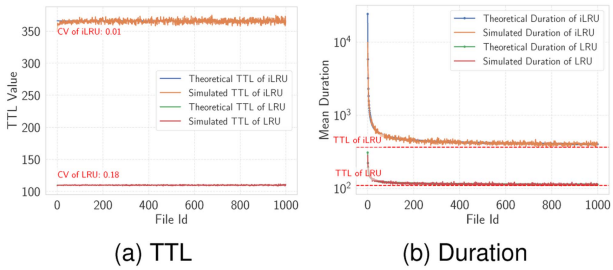

Fig. 7. Validation of theoretical results.

These results demonstrate that iLRU is superior to classic caching algorithms such as Random, LRU, and ARC, as well as some more recent algorithms like LHD and S3-FIFO. Additionally, iLRU's performance is close to the LFU algorithm under the IRM.

C. Model Accuracy Verification

We validated the theoretical hit ratio via simulations. First, we compare the hit ratios obtained from the numerical simulation and the trace-driven simulation. The simulated traces are synthetic and contain 10,000 files requested following a Zipf distribution with parameters $\beta = 0.7$ and 0.9. We ran tests across various cache capacities and calculated the relative difference between the theoretical and simulated values, which is

$$E_r = \frac{|H_{sim} - H_K|}{H_K}, \tag{19}$$

where E_r is the relative error, H_{sim} is the hit ratio from the trace-driven simulation, and H_K is the theoretical hit ratio calculated by (5).

Table II displays the relative errors. These values show a good match between the estimated hit ratios and the simulated hit ratios. In particular, the highest relative error between predicted and simulated values is 1.28%, and most values are lower than 1%, indicating that the theoretical hit ratio is valid.

Second, we validate both Che's approximation (assuming a constant TTL) and the theoretical mean duration of files in the cache. To achieve this, we simulate both LRU and TiLRU cache with 1,000 files with popularity following a Zipf distribution (parameter $\beta = 0.5$) and a capacity of 100. Fig. 7(a) shows that

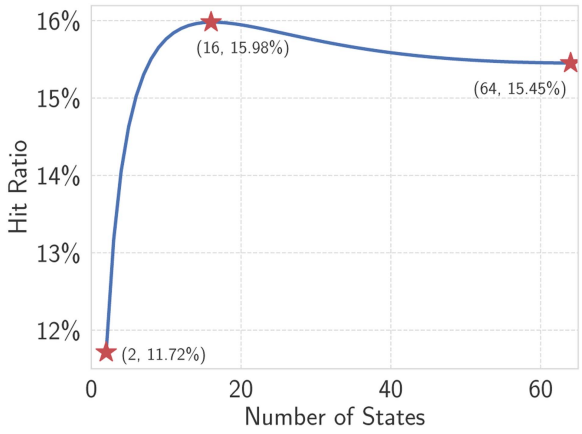

Fig. 8. Comparison of hit ratio for variants of EiLRU schemes.

although some variability exists, the Coefficient of Variation (CV) of the TTLs for all files is small, where the CV is the ratio of standard deviation to the mean. Hence, Che's approximation performs well in the performance evaluation of both LRU and iLRU schemes.

Additionally, Fig. 7(b) demonstrates that the simulated mean duration closely matches the theoretical values and is bounded below by the TTL, validating the accuracy of our theoretical results. The results also show that iLRU achieves a larger mean duration over LRU due to the incremental caching mechanism. This difference in mean duration provides an intuitive explanation for iLRU's superior performance. By retaining popular files longer, iLRU improves their occupancy and the hit ratio in the cache.

D. Performance and Caching States

To study the relationship between performance and the number of caching states, we tested a series of EiLRUs with different numbers of states, where the whole file is cached in the last caching state. The simulation includes 10,000 files, which follow the Zipf distribution with $\beta = 0.3$. The normalized cache capacity is 1000. We then calculated the predicted hit ratios across various numbers of caching states, as shown in Fig. 8. The results show that the hit ratio first increases and then decreases as the number of states increases. Specifically, the hit ratio is highest when the number of states is 16, and lowest when the number of states is 2.

Therefore, the hit ratio does not increase monotonically as the number of cache states increases. The underlying reason could be that too many states consume cache space with less popular content, resulting in performance degradation.

E. Heterogeneous File Sizes

Different types of objects, such as videos, images, and machine learning models, vary significantly in file sizes, which impacts cache performance in MEC networks. We tested two types of size distributions, Gently and Antergic, representing two extreme situations. In the simulations, the sizes are random

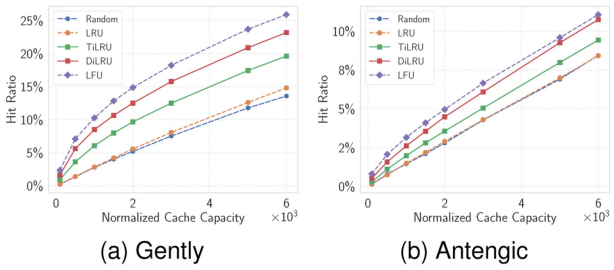

Fig. 9. Hit ratios in the scenario of heterogeneous file sizes.

in $[16, 1024]$. Gently means that the more popular the file, the larger its size, while Antergic is the opposite. The simulated IRM synthetic trace contains 10,000 files with parameter $\beta = 0.7$. We show the results in Fig. 9.

For both distributions, DiLRU and LFU consistently outperform LRU and Random; however, the improvement is less pronounced in the Antergic case. In the Gently scenario, LRU's hit ratio is slightly higher than that of Random, while in the Antergic scenario, LRU's hit ratio is nearly identical to Random's. These findings indicate that heterogeneous file sizes negatively impact the performance of cache algorithms. Notably, the results demonstrate the consistently superior performance of iLRU over LRU, even in heterogeneous file sizes.

F. Robustness of iLRU in the Real Traces

To demonstrate the robustness of iLRU, we compared its performance with existing schemes using real-world traces summarized in Table I. Fig. 10 presents representative hit ratio curves, while Fig. 11 depicts the relative difference in hit ratios between iLRU and ARC across traces P1-P14 and ConCat. The results show that iLRU consistently outperforms the other schemes in almost all tested cases. This superior performance is attributed to iLRU's ability to cache recent frequently accessed files effectively and the timely eviction of less popular files in response to changes in request patterns. Specifically, Fig. 10 shows that Sieve performs worse than iLRU algorithm in the real-world traces although it has a slight performance advantage in IRM synthetic trace when the cache capacity is limited (shown in 5). This is because Sieve takes time to initialize its eviction mechanism before evicting outdated popular files, which results in lower adaptability to dynamic request patterns. In contrast, iLRU maintains a similarly simple implementation structure but demonstrates greater robustness across various real-world traces by leveraging the LRU eviction mechanism, which ensures the timely removal of outdated files.

The traces in Table I encompass various work scenarios, including databases, long sequential scans, workstations, disk access patterns, request mutation scenarios due to user mobility, and multi-user request environments. Therefore, these tests demonstrate the robustness of iLRU across various work environments.

We also simulated the W-iLRU scheme introduced in Section IV-D, testing it with a buffer that occupies 20% of the total cache capacity and a DiLRU cache on the OLTP trace. Our results in Fig. 10–(a) show that W-iLRU outperformed other algorithms, while DiLRU only performs better when the cache capacity is smaller. These results demonstrate that W-iLRU can enhance the robustness of our scheme.

G. Impact of Request Burst

Although iLRU performs well with periodic requests, especially under IRM, it exhibits instability during bursty request patterns, as discussed in Section IV-D. To illustrate the impact, we generate Pareto-like traces, where the request intervals of individual files follow the Pareto distribution instead of the Exponential distribution used in the IRM setting. The Pareto distribution, reflecting the "80/20 rule", models burst traffic by assigning high access frequencies to a few files while most others have low frequencies [58]. To test different levels of burstiness, we configure two scenarios with severe and mild bursts. The severe burst scenario represents the requests concentrated in short, intense bursts, with intervals of successive requests following a typical Pareto distribution with a pronounced heavy tail. Conversely, the mild burst scenario spreads the requests more evenly over time with slight burstiness, and the intervals are distributed like an exponential shape. Additionally, the popularity of the files follows the Zipf distribution with parameter $\beta = 0.5$.

The simulation results are shown in Fig. 12, where the relative difference in hit ratio is defined as the difference between the hit ratios of (W)-DiLRU and LRU, divided by the hit ratio of LRU. Fig. 12 demonstrates that iLRU outperforms LRU in most test cases, particularly when cache capacity is limited. However, in the severe burst scenario, iLRU's hit ratio is slightly lower than LRU's as the cache capacity increases. Thus, the impact of the Pareto model is limited, and iLRU adapts well to this bursty patterns. Additionally, Fig. 12 also shows that the hit ratios of W-iLRU are between iLRU and LRU. This indicates that it is more robust than iLRU. Overall, the results demonstrate that iLRU and its variants can maintain good and robust performance in scenarios with request bursts.

H. Impact of Buffer Size of WiLRU

We examined the impact of buffer size on the performance of the W-iLRU scheme. Fig. 13 presents the instantaneous hit ratios with a normalized capacity of 800 for the OLTP trace and 300 for the Zipf trace, using $10,000$ files drawn from a distribution parameter of 0.9. The instantaneous hit ratios are calculated over a window of size $5,000$. Fig. 14 illustrates the changes of hit ratios with varying buffer proportions that represent the ratio between the buffer size and the total cache capacity, where a buffer proportion of zero corresponds to iLRU, and a proportion of one corresponds to LRU.

Fig. 13 shows the instantaneous hit ratios of W-iLRU with various buffer proportions in the OLTP and Zipf traces, while Fig. 14 shows the hit ratio changes by the buffer proportions. The plots in Fig. 13(a) show significant fluctuations, reflecting changing request patterns over time. The results of Figs. 13 and 14 indicate that W-DiLRU, with a small buffer proportion of 0.2, achieves the highest hit ratio on the OLTP trace but performs poorly on the Zipf trace. This indicates that the buffer of 0.2

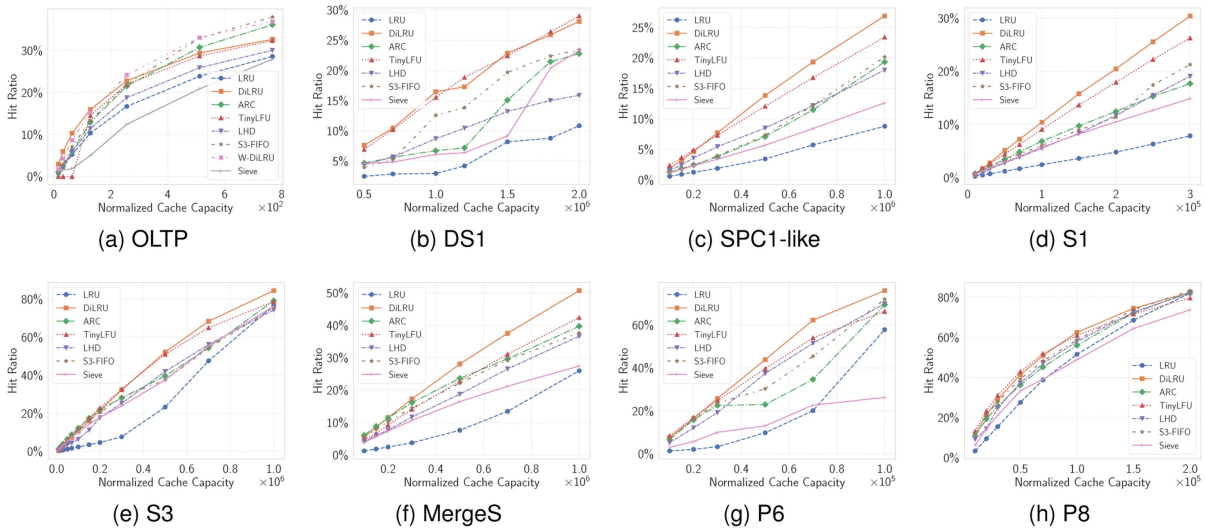

Fig. 10. Hit ratios in some real traces.

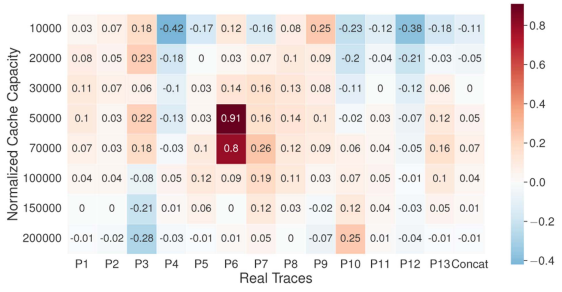

Fig. 11. Relative difference (%) of hit ratios between iLRU and ARC in the P1-P13 and Cancat traces.

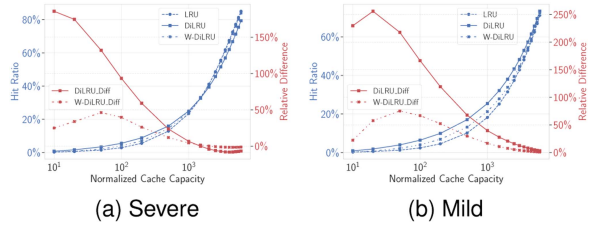

Fig. 12. Performance comparison in the Pareto-like traces.

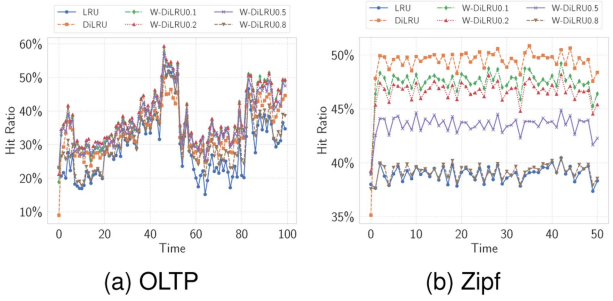

Fig. 13. Instantaneous hit ratios of W-DiLRU with various buffer proportions in the OLTP and Zipf traces.

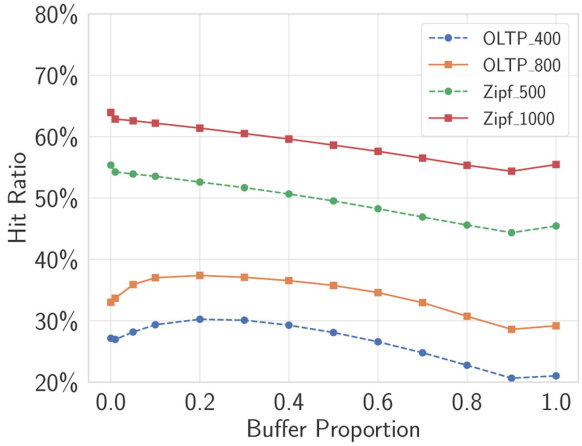

Fig. 14. Impact of buffer size of WiLRU.

is just enough to absorb the request bursts in the OLTP trace. Notably, the hit ratio decreases as the buffer size increases on the Zipf trace. Thus, the buffer can enhance iLRU's robustness in dynamic request patterns but sacrifice performance under IRM conditions, which is a trade-off. Therefore, the buffer proportion should be dynamically adapted to the changes of request patterns.

VI. Conclusion

In this paper, we introduced Incremental Least-Recently-Used (iLRU), an enhancement of the traditional LRU algorithm, and developed an analytical model to calculate its cache occupancy and hit ratio. We theoretically proved that iLRU allocates more cache space to high-frequency files and achieves a higher hit ratio than LRU in scenarios with homogeneous and heterogeneous file sizes. We discussed the robustness and

limits of iLRU in the MEC networks and developed the W-iLRU scheme to enhance its robustness.

Our simulation results demonstrated that iLRU performs close to the optimal algorithm LFU and outperforms other existing algorithms under the IRM. We validated the accuracy of our analytical model and highlighted the importance of appropriately setting the number of states. Our findings also revealed iLRU's robustness, showing its superior performance compared to existing algorithms in multiple real-world traces. Moreover, we conducted simulations to validate the effectiveness of W-iLRU and to investigate the impact of its buffer size. Our simulation results indicated that burst requests can negatively affect the performance of iLRU; however, the W-iLRU scheme effectively mitigates this issue.

For future work, it would be interesting to investigate a self-tuning mechanism to dynamically adjust the configuration of iLRU and W-iLRU for optimal performance. Exploring the cooperative consideration of both segment priority and content popularity could also be an insightful direction. Furthermore, extending iLRU to more general settings, such as multi-cache environments and delay-sensitive scenarios, would provide valuable avenues for further research.

REFERENCES

[1] C. Chen, J. Zhang, and K. Cai, "Incremental least-recently-used algorithm: Good, robust, and predictable performance," in *Proc. Int. Conf. Commun. Workshops*, Denver, USA, 2024, pp. 514–519.

[2] T. X. Tran and D. Pompili, "Adaptive bitrate video caching and processing in mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 18, no. 9, pp. 1965–1978, Sep. 2019.

[3] V. Farhadi et al., "Service placement and request scheduling for data-intensive applications in edge clouds," *IEEE/ACM Trans. Netw.*, vol. 29, no. 2, pp. 779–792, Apr. 2021.

[4] A. Lekharu, M. Jain, A. Sur, and A. Sarkar, "Deep learning model for content aware caching at MEC servers," *IEEE Trans. Netw. Serv. Manage.*, vol. 19, no. 2, pp. 1413–1425, Jun. 2022.

[5] Z. Jin, T. Song, and W.-K. Jia, "An adaptive cooperative caching strategy for vehicular networks," *IEEE Trans. Mobile Comput.*, vol. 23, no. 10, pp. 9502–9517, Oct. 2024.

[6] M. Reiss-Mirzaei, M. Ghobaei-Arani, and L. Esmaeili, "A review on the edge caching mechanisms in the mobile edge computing: A social-aware perspective," *Internet Things*, vol. 22, Jul. 2023, Art. no. 100690.

[7] L. Chen et al., "Multi-MEC collaboration for VR video transmission: Architecture and cache algorithm design," *Comput. Netw.*, vol. 234, pp. 109864, Oct. 2023.

[8] H. Hu, W. Song, Q. Wang, R. Q. Hu, and H. Zhu, "Energy efficiency and delay tradeoff in an MEC-enabled mobile IoT network," *IEEE Internet Things J.*, vol. 9, no. 17, pp. 15942–15956, Sep. 2022.

[9] I.-H. Hou, T. Zhao, S. Wang, and K. Chan, "Asymptotically optimal algorithm for online reconfiguration of edge-clouds," in *Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2016, pp. 291–300.

[10] Y. Zhang, *Mobile Edge Computing*, 1st ed. cham, Switzerland: Springer, Oct. 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-83944-4_3

[11] M. Dehghan, L. Massoulie, D. Towsley, D. S. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," *IEEE/ACM Trans. Netw.*, vol. 27, no. 3, pp. 1013–1027, Jun. 2019.

[12] J. Yang, Z. Mao, Y. Yue, and K. V. Rashmi, "GL-Cache: Group-level learning for efficient and high-performance caching," in *Proc. 21st USENIX Conf. File Storage Technol.*, Santa Clara, USA, 2023, pp. 115–133.

[13] D. Xenakis, N. Passas, L. Merakos, and C. Verikoukis, "Mobility management for femtocells in LTE-advanced: Key aspects and survey of handover decision algorithms," *IEEE Commun. Surv. Tut.*, vol. 16, no. 1, pp. 64–91, First Quarter 2014.

[14] G. Hasslinger, M. Okhovatzadeh, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "An overview of analysis methods and evaluation results for caching strategies," *Comput. Netw.*, vol. 228, Jun. 2023, Art. no. 109583.

[15] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, "FIFO queues are all you need for cache eviction," in *Proc. 29th ACM Symp. Operating Syst. Princ.*, 2023, pp. 130–149.

[16] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Hoboken, NJ, USA: Prentice-Hall, 1973.

[17] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, Jan. 1971.

[18] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy," *ACM Trans. Storage*, vol. 13, no. 4, pp. 1–31, Nov. 2017.

[19] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, San Francisco, USA, Mar. 2003, pp. 115–130.

[20] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306, Jun. 1993.

[21] T. Johnson et al., "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. 20th VLDB Conf.*, San Francisco, USA, 1994, pp. 439–450.

[22] R. Karedla, J. Love, and B. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, Mar. 1994.

[23] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annu. Tech. Conf.*, Boston, USA, 2001, pp. 91–104.

[24] H. Gomaa, G. G. Messier, C. Williamson, and R. Davies, "Estimating instantaneous cache hit ratio using Markov chain analysis," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1472–1483, Oct. 2013.

[25] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, Renton, WA, 2018, pp. 389–403.

[26] K.-L. Wu, P. S. Yu, and J. L. Wolf, "Segment-based proxy caching of multimedia streams," in *Proc. 10th Int. Conf. World Wide Web*, New York, USA, 2001, pp. 36–44.

[27] K.-L. Wu, P. S. Yu, and J. L. Wolf, "Segmentation of multimedia streams for proxy caching," *IEEE Trans. Multimedia*, vol. 6, no. 5, pp. 770–780, Oct. 2004.

[28] L. Wang, S. Bayhan, and J. Kangasharju, "Optimal chunking and partial caching in information-centric networks," *Comput. Commun.*, vol. 61, pp. 48–57, May 2015.

[29] L. Maggi, L. Gkatzikis, G. Paschos, and J. Leguay, "Adapting caching to audience retention rate," *Comput. Commun.*, vol. 116, pp. 159–171, Jan. 2018.

[30] V. C. L. Narayana, S. Jain, and S. Moharir, "Caching partial files for content delivery," in *Proc. Nat. Conf. Commun.*, Bangalore, India, 2019, pp. 1–6.

[31] E. Friedlander and V. Aggarwal, "Generalization of lru cache replacement policy with applications to video streaming," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 4, no. 3, pp. 1–22, Aug. 2019.

[32] A. Ali-Eldin, C. Goel, M. Jha, B. Chen, K. Nahrstedt, and P. Shenoy, "CAVE: Caching 360° videos at the edge," in *Proc. 32nd ACM Workshop Netwo. Operating Syst. Support Digit. Audio Video*, New York, USA, 2022, pp. 50–56.

[33] R. Fagin, "Asymptotic miss ratios over independent references," *J. Comput. Syst. Sci.*, vol. 14, no. 2, pp. 222–250, Apr. 1977.

[34] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical web caching systems," in *Proc. 20th Annu. Joint Conf. IEEE Comput. Commun. Soc.*, Anchorage, USA, Apr. 2001, pp. 1416–1424.

[35] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for LRU cache performance," in *Proc. 24th Int. Teletraffic Congr.*, Krakow, Poland, 2012, pp. 1–8.

[36] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Comput. Netw.*, vol. 65, pp. 212–231, Jun. 2014.

[37] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *Proc. IEEE Conf. Comput. Commun.*, Toronto, Canada, 2014, pp. 2040–2048.

[38] N. Gast and B. Van Houdt, "Asymptotically exact TTL-approximations of the cache replacement algorithms LRU (m) and h-LRU," in *Proc. 28th Int. Teletraffic Congr.*, Germany, 2016, pp. 157–165.

[39] G. Hasslinger, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Analysis of the LRU cache startup phase and convergence time and error bounds on approximations by Fagin and Che," in *Proc. 20th Int. Symp. Model. Optim. Mobile Ad hoc Wireless Netw.*, Torino, Italy, Sep. 2022, pp. 254–261.

[40] H. Dai, B. Liu, H. Yuan, P. Crowley, and J. Lu, "Analysis of tandem PIT and CS with non-zero download delay," in *Proc. IEEE Conf. Comput. Commun.*, Atlanta, USA, May 2017, pp. 1–9.
[41] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, Boston, USA, 2017, pp. 483–498.
[42] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[43] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, USA, 2003, pp. 241–252.
[44] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.
[45] Y. T. Hou, J. Pan, B. Li, and S. S. Panwar, "On expiration-based hierarchical caching systems," *IEEE J. Sel. Areas Commun.*, vol. 22, no. 1, pp. 134–150, Jan. 2004.
[46] H. Gomaa, G. G. Messier, and R. Davies, "Hierarchical cache performance analysis under TTL-based consistency," *IEEE/ACM Trans. Netw.*, vol. 23, no. 4, pp. 1190–1201, Aug. 2015.
[47] J. Gao, S. Zhang, L. Zhao, and X. Shen, "The design of dynamic probabilistic caching with time-varying content popularity," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 1672–1684, Apr. 2021.
[48] Y. Li, H. Ma, L. Wang, S. Mao, and G. Wang, "Optimized content caching and user association for edge computing in densely deployed heterogeneous networks," *IEEE Trans. Mobile Comput.*, vol. 21, no. 6, pp. 2130–2142, Jun. 2022.
[49] K. Poularakis and L. Tassiulas, "Code, cache and deliver on the move: A novel caching paradigm in hyper-dense small-cell networks," *IEEE Trans. Mobile Comput.*, vol. 16, no. 3, pp. 675–687, Mar. 2017.
[50] R. W. Wolff, "Poisson arrivals see time averages," *Operations Res.*, vol. 30, no. 2, pp. 223–231, Apr. 1982.
[51] J. P. Schmidt and A. Siegel, "The spatial complexity of oblivious k-probe hash functions," *SIAM J. Comput.*, vol. 19, no. 5, pp. 775–786, 1990.
[52] J. E. Hopcroft, J. D. Ullman, and A. V. Aho, *Data Structures and Algorithms*. Boston, MA, USA: Addison-wesley, 1983.
[53] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, "SIEVE is simpler than LRU: An efficient turn-key eviction algorithm for web caches," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, Santa Clara, USA, Apr. 2024, pp. 1229–1246.
[54] J. Yang, "Libcachesim: A high-performance library for building cache simulators," 2024, Accessed: Nov. 4, 2024. [Online]. Available: https://github.com/1a1a11a/libCacheSim
[55] J. Yang, Y. Yue, and K. V. Rashmi, "A large-scale analysis of hundreds of in-memory key-value cache clusters at Twitter," *ACM Trans. Storage*, vol. 17, no. 3, pp. 1–35, Aug. 2021.
[56] R. Fagin and M. C. Easton, "The independence of miss ratio on page size," *J. ACM*, vol. 23, no. 1, pp. 128–146, Jan. 1976.
[57] M. C. Easton and R. Fagin, "Cold-start vs. warm-start miss ratios," *Commun. ACM*, vol. 21, no. 10, pp. 866–872, Oct. 1978.
[58] B. C. Arnold, *Pareto Distribution.*Hoboken, NJ, USA: Wiley, Sep. 2015.

Jinbei Zhang received the BS degree in electronic engineering from Xidian University, Xi'an, China, in 2010, and the PhD degree in electronic engineering from Shanghai Jiao Tong University, Shanghai, China, in 2016. From 2016 to 2018, he worked as a postdoc with the Chinese University of Hong Kong. Since 2018, he is an associate professor with Sun Yat-sen University, Shenzhen, China. His current research interests include network information theory and quantum networks.

Chunpeng Chen received the BS degree in communication engineering from Sun Yat-sen University, Shenzhen, China, in 2022. He is currently working toward the PhD degree with the School of Electronics and Communication Engineering, Sun Yat-sen University. His research interests include cache replacement algorithm and mobile edge computing.

Kechao Cai received the PhD degree from the Chinese University of Hong Kong, in 2019. He is currently an assistant professor with the School of Electronics and Communication Engineering, Sun Yat-sen University in China. His current research interests include distributed network protocol design and online learning algorithms for network systems.

John C. S. Lui (Fellow, IEEE) received the PhD degree in computer science from University of California, Los Angeles. He is currently the Choh-Ming Li chair professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His current research interests include communication networks, system security (e.g., cloud security, mobile security, etc.), network economics, network sciences, large-scale distributed systems and performance evaluation theory. He serves in the editorial board of *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Performance Evaluation and International Journal of Network Security*. He was the chairman of the CSE Department from 2005-2011. He received various departmental teaching awards and the CUHK vice-chancellors Exemplary Teaching Award. He is also a corecipient of the IFIP WG 7.3 Performance 2005 and IEEEIFIP NOMS 2006 Best Student Paper Awards. He is an elected member of the IFIP WG 7.3, fellow of the ACM, and Croucher senior research fellow. His personal interests include films and general reading.