Adaptive and Scalable Caching With Erasure Codes in Distributed Cloud-Edge Storage Systems

Kaiyang Liu, *Member, IEEE*, Jun Peng, *Senior Member, IEEE*, Jingrong Wang, *Graduate Student Member, IEEE*, Zhiwu Huang, *Member, IEEE*, and Jianping Pan, *Senior Member, IEEE*

Abstract—Erasure codes have been widely used to enhance data resiliency with low storage overheads. However, in geo-distributed cloud storage systems, erasure codes may incur high service latency as they require end users to access remote storage nodes to retrieve data. An elegant solution to achieving low latency is to deploy caching services at the edge servers close to end users. In this paper, we propose adaptive and scalable caching schemes to achieve low latency in the cloud-edge storage system. Based on the measured data popularity and network latencies in real time, an adaptive content replacement scheme is proposed to update caching decisions upon the arrival of requests. Theoretical analysis shows that the reduced data access latency of the replacement scheme is at least 50% of the maximum reducible latency. With the low computation complexity of our design, nearly no extra overheads will be introduced when handling intensive data flows. For further performance improvements without sacrificing its efficiency, an adaptive content adjustment scheme is presented to replace the subset of cached contents that incur the aforementioned performance loss. Driven by real-world data traces, extensive experiments based on Amazon Simple Storage Service demonstrate the effectiveness and efficiency of our design.

Index Terms—Cloud-edge storage systems, erasure codes, caching

1 Introduction

In the current era of big data, we have witnessed the explosive growth of workloads driven by the increasing demand for data-intensive applications, e.g., social networks, artificial intelligence, and Internet of Things (IoT). For example, IDC predicts that the amount of data generated by IoT devices will reach 73.1 zettabytes by 2025, growing from 18.3 zettabytes in 2019 [1]. Such applications require scalable, highly available, and cost-effective storage systems. Modern distributed cloud storage systems, such as Amazon Simple Storage Service (Amazon S3) [2], Hadoop Distributed File System (HDFS) [3], and Microsoft Azure [4], use data replication and erasure codes to improve data reliability and availability.

Erasure codes, e.g., (K, R) Reed-Solomon (RS) code, encode the data item into K data chunks and R parity chunks for redundancy. The coded chunks are placed at various storage nodes to achieve R-fault tolerance as the original data item can be reconstructed from any K out of $K + R$ chunks. Compared with traditional data replication schemes which entail a minimum of $2\times$ storage redundancy, erasure codes can significantly reduce storage overheads while retaining the same level of reliability [6], [7]. However, erasure codes may incur high access latency, especially in geo-distributed storage systems as end users need to contact remote storage nodes to reconstruct the data [8], [9]. As the needed K chunks are requested in parallel, the access latency is determined by the chunk placed at the farthest storage node. The latency dramatically affects user satisfaction. Amazon has reported that 100 ms of additional latency can decrease its revenue by 1% [10].

With the development of the geo-distributed cloud storage system, major content providers, e.g., Amazon, Akamai, and Google, deploy edge servers to achieve low latency [9]. End users issue requests to their nearest edge servers, which have cached a pool of popular data items. Compared with traditional caching services inside remote storage nodes or data centers [8], [11], [12], the resources of the edge server, e.g., computational capability and cache capacity, are typically limited [13]. Note that caching at the edge servers also has unique benefits: 1) the edge servers can be flexibly deployed

- *Kaiyang Liu is with the School of Computer Science and Engineering, Central South University, Changsha 410075, China and also with the Department of Computer Science, University of Victoria, Victoria, BC V8P 5C2, Canada. E-mail: liukaiyang@uvic.ca.*
- *Jun Peng is with the School of Computer Science and Engineering, Central South University, Changsha 410075, China. E-mail: pengj@csu.edu.cn.*
- *Jingrong Wang is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 1A1, Canada. E-mail: jr.wang@mail.utoronto.ca.*
- *Zhiwu Huang is with the School of Automation, Central South University, Changsha 410075, China. E-mail: hzw@csu.edu.cn.*
- *Jianping Pan is with the Department of Computer Science, University of Victoria, Victoria, BC V8P 5C2, Canada. E-mail: pan@uvic.ca.*

Manuscript received 4 Oct. 2021; revised 2 Mar. 2022; accepted 11 Apr. 2022. Date of publication 19 Apr. 2022; date of current version 7 June 2023.
This work was supported in part by the National Natural Science Foundation of China under Grant 61873353, in part by the China Postdoctoral Science Foundation under Grant 2019TQ0359, in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), in part by the Canada Foundation for Innovation (CFI), and in part by the British Columbia Knowledge Development Fund (BCKDF).
(Corresponding author: Jingrong Wang.)
Recommended for acceptance by Z. Wang.
Digital Object Identifier no. 10.1109/TCC.2022.3168662

at the network edges in an on-demand manner, and 2) the data access traffic and latency can be further reduced.

Without considering erasure codes, existing caching policies need to decide which data items should be cached (i.e., working "at the data item level") for cache hit ratio maximization [11], [14], [15], service latency minimization [12], or load balancing [16], [17]. However, the application of caching to coded storage systems faces practical challenges. Due to the presence of data and parity chunks, the caching scheme needs to decide 1) which data items to cache, and 2) how many chunks to cache for each selected data item? To serve end users across the globe, spreading the coded chunks of each data item across more storage nodes can reduce the latencies of geographically dispersed requests [9]. Therefore, the network latency of fetching different chunks varies as they are placed at geographically diverse sites. Since the data request latency is determined by the slowest chunk retrieval when they are requested in parallel, the data chunk with higher access latency should be cached first. However, preliminary experiment results in Section 3.2 show that caching more chunks may not proportionally reduce the overall data access latency. Traditional caching schemes at the data item level are not space-efficient to achieve the lowest latency [19].

As the storage locations of coded chunks are not identical, the caching performance in terms of latency reduction could be different for various data items. For a dynamic network scenario, the network conditions and user request rates may keep changing over time [5]. It is hard to obtain a precise analytical latency model for the coded storage systems [18]. More importantly, in the big data era, existing caching solutions, e.g., iterative optimization [8], [19], [20] and graph-based schemes [21], may face the challenges of long running time and large overheads to handle the increasing scale of datasets. How to adaptively and efficiently optimize the caching decisions for low latency in the coded storage system that spans multiple geographical sites is an important and challenging problem.

In this paper, we propose adaptive and scalable caching schemes that are specifically designed for the coded storage system. Based on the measured data popularity and network latencies in real time, online caching schemes are designed to determine which data chunks to cache upon the arrival of data requests. The computation complexity being sublinear to the size of the cache capacity ensures the scalability of our design to handle intensive data flows. The main contributions in this paper include:

- A novel adaptive content replacement scheme with adaptivity and scalability is designed for the distributed coded storage system. The worst-case performance guarantee of the proposed scheme is provided via theoretical analysis.
- We obtain the subset of cached contents that incur the performance loss in the replacement scheme. Without sacrificing the efficiency, an adaptive content adjustment scheme is developed, replacing this subset for further performance improvements.
- We deploy the experiment platform based on Amazon S3. Extensive experiment results demonstrate the high efficiency of the proposed caching schemes. Compared with traditional caching schemes that work at the data item level, caching at the data chunk level can reduce the average data access latency by up to 35.7%. Compared with the state-of-the-art caching scheme for the coded storage system, our design can reduce the computation overhead by up to 84.57% while only incurring a performance loss of 1.12%.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 presents the model of the distributed coded storage system and states the caching problem. In Section 4, the design of adaptive and efficient caching is provided along with theoretical analysis. In Section 5, the efficiency and performance of our design are evaluated and substantiated with extensive experiments. Section 6 concludes this paper and lists future work.

2 Related Work

Data Replication and Erasure Codes. The key challenge of distributed storage systems is to provide low-latency services. Recent research efforts have suggested that data replication can enhance data locality to reduce latency. Chowdhury *et al.* [22] proposed Sinbad, a data replica placement scheme that chose the storage nodes with lightly loaded links as the data write locations, reducing the end-to-end completion times of data-intensive tasks. DataBot [5] adopted reinforcement learning to optimize the data replica placement decisions in an online manner, reducing the data access latency in a dynamic network scenario. However, data replication inevitably incurs high bandwidth and storage overheads.

Erasure codes have been extensively investigated in distributed storage systems as they can provide space-optimal data redundancy. However, it is still an open problem to quantify the accurate service latency for coded storage systems [19]. Therefore, recent studies have attempted to analyze the latency bounds based on queuing theory [18], [19], [23]. These studies are under the assumption of a stable request arrival process and exponential service time distribution, which may not be suitable for a dynamic network scenario.

Furthermore, prior work also focused on the design of data request scheduling schemes to achieve load balancing in coded storage systems [6], [7], [24]. Then, the data access latency is reduced by the avoidance of congestion. These scheduling schemes are suitable for intra-data center storage systems as the network congestion dominates the overall data access latency. Instead, we consider reducing the data access latency through caching at the edge server in the geo-distributed coded storage system. To be more specific, this work minimizes the high latency of data retrieval from remote storage nodes to edge servers (which are deployed near end users) over Wide Area Networks (WAN).

Caching in Coded Storage Systems. In geo-distributed coded storage systems, caching for low latency has received significant attention in recent years. Aggarwal *et al.* [19] pointed out that caching partial data chunks had more scheduling flexibility when compared with caching the entire data items. Assuming the future data request rates

TABLE 1
Notations

Symbol	Definition
$\mathcal{N}$	Set of geo-distributed storage nodes, and $N = \|\mathcal{N}\|$
$\mathcal{M}$	Set of data items, and $M = \|\mathcal{M}\|$
K, R	Number of coded data and parity chunks
m_k, m_r	Coded data chunk and parity chunk of data item m
C	Cache capacity at the edge server
$\mathcal{T}$	Period of data services
Γ	Set of data requests in period $\mathcal{T}$
γ_m^t	Request to data item m at time t, $\gamma_m^t \in \Gamma$
λ_m^t	Number of cached chunks for data item m at time t
$f_m^t(\lambda_m^t)$	Discrete access latency function for data item m
r_m^t	Request rate for data item m from end users
Θ	Total amount of reduced latency
$l_{m_k}^t$	Average latency of accessing data chunk m_k from remote storage nodes
$\tau_{m,k}$	Reduced latency when k data chunks are cached for data item m
$\hat{\mathcal{M}}$	Set of cached data items
$\hat{\mathcal{M}}'$	Set of cache replacement candidates
$\mathcal{M}_i$	Unavailable data chunks when a server at node i fails

are known beforehand, a heuristic algorithm was proposed to reduce the latency with the performance no worse than caching the entire data item. Al-Abbasi *et al*. [25] designed a Time-To-Live Cache (TTLCache) policy to quantify and jointly optimize the mean and tail data access latency in coded storage systems. TTLCache can converge to a stationary point which may not be the global optimal solution. Halalai *et al*. [8] designed Agar, a dynamic programming-based caching scheme to achieve low latency in coded storage systems. Agar was a static policy that pre-computed a cache configuration for a certain time period without any worst-case performance guarantees. As the caching configuration is iteratively optimized for all data items, the scalability of Agar is limited. In [20], we were the first to investigate the optimal scheme to determine which coded chunks should be cached at the edge servers for low data access latency. Guided by the offline optimal scheme, an online near-optimal scheme was also designed with the approximation ratio $1 - \frac{2K-1}{C}$. However, the scalability of the online near-optimal scheme is limited as its computation complexity grows exponentially with the increase of K.

Unlike previous studies, our design mainly considers the adaptivity and scalability of caching schemes for the coded storage systems. Compared with the state-of-the-art online scheme in [20], the computation overhead of the proposed caching scheme is reduced by 84.57% while only incurring a performance loss of 1.12%.

3 System Model and Problem Statement

In this section, we introduce the model of the geo-distributed cloud storage system with erasure codes, and then discuss how to reduce the data access latency with caching services at the edge servers. The major notations used in this paper are summarized in Table 1.

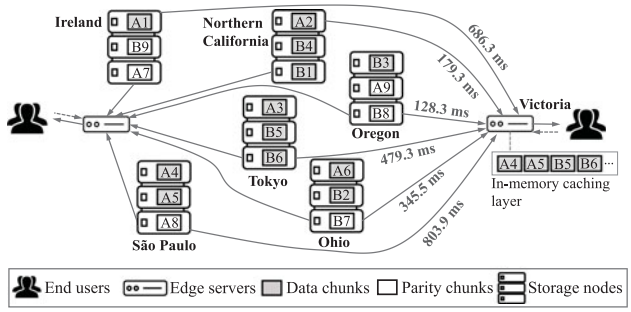


Fig. 1. An illustration of the geo-distributed storage system with erasure codes is shown. The coded storage system is deployed over $N = 6$ Amazon Web Services (AWS) regions. For low latency data access, edge servers are deployed to cache coded data chunks near end users across the globe. The average data access latencies from remote storage nodes to the edge server deployed at Victoria, Canada are labeled.

3.1 Geo-Distributed Storage System and Erasure Codes

Fig. 1 illustrates the model of the geo-distributed cloud storage system, which consists of a set of storage nodes $\mathcal{N}$ distributed across different geographical locations (with size $N = |\mathcal{N}|$). Each storage node could be a data center in practice, which may consist of multiple storage servers. Let $\mathcal{M}$ denote the set of data items stored in the system (with size $M = |\mathcal{M}|$). The data items could be files, tables, or blocks in practice. Similar to Hadoop [3] and Cassandra [27], all data items have a default block size. Reed-Solomon (RS) codes split each data item into K equal-sized fragments, i.e., data chunks, create linear combinations of these fragments to generate same-sized R parity chunks, and store them at different storage servers.[1] The storage locations of data and parity chunks are denoted by

$$\begin{cases} m_k \rightarrow i, k \in \{1, \ldots, K\}, i \in \mathcal{N}, \\ m_r \rightarrow j, r \in \{1, \ldots, R\}, j \in \mathcal{N}, \end{cases} \tag{1}$$

which means data chunk m_k and parity chunk m_r are stored at node i and j, respectively.[2] The coded chunks should not be placed at a single storage node since this will increase the data access latency of end users far from that node.

In the presence of storage server failures, the data availability can be ensured by decoding the original data item from any K out of $K + R$ chunks. The decoding with parity chunks will inherently incur considerable computation overheads to the storage system. In the distributed storage system with erasure codes, a read request is first served by obtaining K data chunks to reconstruct the original data item with low overheads [6]. The actions of fetching parity chunks and decoding for data reconstruction are defined as *degraded read*. The degraded read will be passively triggered 1) when the storage server storing the data chunks is momentarily unavailable for read requests, or 2) to restore a failed storage server. In this paper, the data write/update

1. Other erasure coding schemes, e.g., Local Reconstruction Codes (LRC) [28] with a low recovery cost, can also be applied in our solution.

2. To achieve low latency without increasing the storage costs, data replication at the remote storage nodes is not adopted for coded chunks in this paper. Our design is applicable to the scenario with data replication. The data request is served by fetching K data chunks from the nearest storage nodes. Section 5.3 evaluates the impact of data replication.

TABLE 2
The Deployment of Storage Nodes Over Six AWS Regions and the Average Data Access Latency (In Milliseconds) From Remote Storage Nodes to End Users at Three Different Locations

Storage node		1	2	3	4	5	6
Region		Tokyo	Ohio	Ireland	São Paulo	Oregon	Northern California
Average latency (ms)	**Victoria, CA**	479.3	345.5	686.3	803.9	128.3	179.3
	San Francisco, US	513.2	338.4	663.2	786.9	158.3	84.7
	Toronto, CA	794.7	129.0	631.5	705.5	302.6	355.7

process is not considered. This is because today many storage systems are append-only where the data items are immutable [6]. Data items with any changes are treated as separate objects with new timestamps.

Erasure codes may incur high data access latency, especially in a geo-distributed storage system. The requested chunks are obtained by accessing multiple remote storage nodes. The high latency impedes the extensive application of erasure codes. Existing storage systems, e.g., Windows Azure, adopt erasure codes to archive rarely accessed data [28].

3.2 Caching for Low Latency

Caching has been considered as a promising solution to achieve low latency services [8]. As shown in Fig. 1, multiple edge servers are deployed, each with an in-memory caching layer to cache popular data items near end users. Due to the scarcity of memory, the cache capacity C at the edge server is limited. This means that we may not cache all data items in the caching layer, i.e., $C \leq M \cdot K$. In general, the caching scheme works well for the scenario where the request patterns across data items are highly skewed [26]. Facebook and Microsoft have reported that the request frequency of the top 5% data items is seven times larger than that of the bottom 75% [29]. Intuitively, a fraction of data items with higher request frequency may benefit more from caching.

Traditional caching schemes usually cache full copies of data items [30]. However, in the coded storage system, caching at the data item level at the edge server may not achieve the full benefits of caching. We demonstrate this through experiments based on Amazon S3. As shown in Table 2, a coded storage system is deployed over $N = 6$ AWS regions, i.e., Tokyo, Ohio, Ireland, São Paulo, Oregon, and Northern California. Each AWS region creates three `buckets`, each of which denotes a storage server for remote data storage. The distributed storage system is populated with $M = 10,000$ data items. For the RS codes, we set $K = 6$ and $R = 3$. The coded data and parity chunks are with a default size of 1 MB [8]. For each data item, the coded nine chunks are evenly distributed among eighteen `buckets` to achieve load balancing. In particular, any chunks from the same data item are not placed at the same server to guarantee the R-fault tolerance. As noted in prior work [6], [26], the popularity of data items follows a Zipf distribution. Furthermore, three edge servers are deployed near the end users at various locations around the world. The edge server uses a thread pool to request data chunks in parallel. `Memcached` [31] module is adopted at the edge servers for data caching in RAM.

The data access latency includes the network latency from remote storage nodes to the edge server, the data reconstruction latency, and the network latency from the edge server to end users. This paper focuses on investigating how the edge server caches data chunks from remote storage servers to achieve low latency over WAN. For a given city, the edge server is deployed in close proximity to end users with low data access latency in the experiments [12]. Compared with the high network latency over WAN (in hundreds of milliseconds), the reconstruction latency with data chunks and the network latency from the edge server to end users are negligible. To serve more users that are scattered around this city, more edge servers can be flexibly deployed in an on-demand manner.

Table 2 shows the average latency of end users fetching data chunks from geo-distributed storage nodes. Experiment results confirm the positive correlation between physical distance and latency. For instance, the latency from the storage node in São Paulo to end users in Victoria, CA, is much higher than that in Oregon. This is because the propagation delay dominates and depends primarily on the physical distance of data transmission [32]. For data requests, the latency is determined by the slowest chunk retrieval among all chunks. As shown in Fig. 1, if data item B (including data chunk B1–B6) is requested from Victoria in parallel, the latency is about 479.3 ms as we need to fetch data chunks B5 and B6 from the farthest storage node in Tokyo.

Then, considering data caching at the edge server, we show the performance of latency reduction by gradually increasing the number of cached data chunks for a data item. In the experiments, only data chunks will be cached to avoid degraded read. Let $\mathcal{T}$ denote the period of data services. Let l_i^t denote the real-time network latency from storage node i to end users at time t, $i \in \mathcal{N}$, $t \in \mathcal{T}$. According to the storage locations in (1), the average latency of sending data chunk m_k is given by

$$l_{m_k}^t = l_i^t \cdot \mathbf{1}(m_k \rightarrow i), \tag{2}$$

where $\mathbf{1}(m_k \rightarrow i)$ indicates whether m_k is placed at node i or not, returning 1 if true or 0 otherwise, $k \in \{1, \ldots, K\}$. For ease of notation, let us relabel the data chunks according to the descending order of the data access latency, i.e., $l_{m_1}^t \geq \ldots \geq l_{m_K}^t$. The data chunks with higher access latencies will be cached first. In this way, the number of remote storage nodes that end users must access can be progressively reduced. Let λ_m^t denote the number of cached data chunks for data item m at time t, $\lambda_m^t \in \{0, \ldots, K\}$, $m \in \mathcal{M}$. The discrete latency function can be defined as follows:

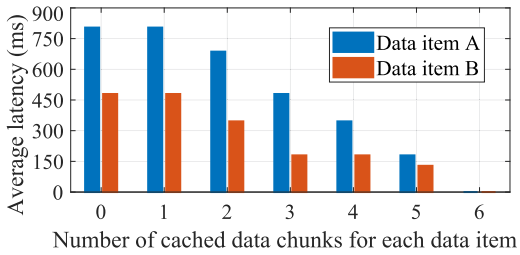


Fig. 2. Experiment results show the average access latency of caching a different number of data chunks at the edge server in Victoria. The relationship between the number of cached chunks and the reduced latency is nonlinear. The storage locations of data items A and B are shown in Fig. 1.

$$f_m^t(\lambda_m^t) = \begin{cases} l_{m_1}^t, \lambda_m^t = 0, \\ \quad \ldots \\ l_{m_k}^t, \lambda_m^t = k-1, \\ \quad \ldots \\ 0, \lambda_m^t = K. \end{cases} \tag{3}$$

Fig. 2 shows the average access latency of caching a different number of data chunks at the edge server in Victoria. We have the following two observations:

- The access latency function $f_m^t(\lambda_m^t)$ is monotonically decreasing but nonlinear to the number of cached data chunks. For a data item, caching more data chunks may not make the data access proportionally faster.
- The storage locations of chunks may be different for various data items, e.g., data items A and B in Fig. 1. For various data items, the access latency function could be different. For example, for data item A, the latency is reduced by 40.3% if three data chunks are cached. For data item B, three cached data chunks can reduce the latency by 62.6%.

As the values of data chunks to cache in terms of latency reduction are unequal, traditional caching policies at the data item level may not achieve the full benefits of caching.

3.3 Caching Problem Statement

Our objective is to minimize the data access latency over the service period by determining the number of cached data chunks for each data item:

$$\begin{aligned} \min_{\lambda_m^t \in \mathbb{N}, m \in \mathcal{M}} & \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} f_m^t(\lambda_m^t) \cdot r_m^t \\ \text{s.t.} \quad & 0 \le \lambda_m^t \le K, \\ & \sum_{m \in \mathcal{M}} \lambda_m^t = C, \end{aligned} \tag{4}$$

where r_m^t denotes the request rate at time t. Constraint $0 \le \lambda_m^t \le K$ ensures that the number of cached chunks for each data item is less than the number of coded data chunks. As $C \le M \cdot K$, $\sum_{m \in \mathcal{M}} \lambda_m^t = C$ ensures that the cache capacity is fully utilized for latency reduction. Then, the hardness of problem (4) is examined from the following aspects:

- Experiments show that $f_m^t(\lambda_m^t)$, e.g., the latency function of data item B, is both nonlinear and nonconvex. Problem (4) is an integer programming problem with non-convexity and nonlinearity. In general, complex combinatorial methods are needed for an efficient solution [33].
- In a dynamic scenario, the network conditions and user requests, i.e., $f_m^t(\lambda_m^t)$ and r_m^t, are time variant. It is a challenge to design adaptive solutions that can react quickly to real-time changes.

For a large-scale storage system with uncertainties, the caching scheme should be 1) highly efficient for a quick caching decision, and 2) flexible to change the caching decision in an online manner. Therefore, it is imperative to solve the caching problem efficiently and adaptively.

4 Adaptive and Scalable Caching Design

In this section, adaptive and scalable caching schemes are presented based on the measured request rates and data access latencies in real time. To begin with, we discuss why the formulated caching problem with erasure codes is not identical to the well-known Knapsack problem. Mature solutions with theoretical analysis to the Knapsack problem cannot be directly applied to the caching problem with erasure codes. Therefore, novel tailored schemes are needed for the distributed coded storage system. Detailed theoretical analysis for the tailored caching schemes is also a nontrivial task. Then, we consider the scenario without server failure, where only data chunks are fetched from the remote `buckets` or cached at the edge server. Furthermore, we extend the proposed caching schemes to the case of storage server failure.

4.1 Adaptive Content Replacement

Data Popularity. Exponentially Decayed Counter (EDC) [35], [36] tracks an approximate per-object request count over a time period, which is applied to predict the future data popularity information r_m^t in real time, $t \in \mathcal{T}$. Initially, r_m^t is initialized to 0. Whenever there is a read request for data item m, Δ_m is updated first, which indicates the amount of time since m was last requested. Please note that Δ is used by many successful heuristics, such as Least Recently Used (LRU) and its variants. Then, r_m^t is updated with

$$r_m^t = 1 + r_m^t \cdot 2^{-\frac{\Delta_m}{\alpha_r}}. \tag{5}$$

Compared with LRU which only considers the most recent data request, EDC explores the tradeoff between recency and frequency in data accesses. The parameter α_r determines the weight of recency versus frequency and it is empirically set to 2^9 according to [36]. EDC can accurately approximate the decay rate of data popularities, which has been widely used in block storage caching [36] and video popularity prediction [37]. Furthermore, EDC only needs $O(1)$ space to maintain the request popularity information for each data item.

Network Latency. The latency function $f_m^t(\lambda_m^t)$ is different for various data items and could be time-varying. Therefore, instead of formulating accurate mathematical latency models, the measured end-to-end latency is used to quantify the benefits of caching. Let l_i^t denote the real-time network latency from storage node i to end users at time t. Similar to the previous study [8], the Exponentially Weighted Moving Average (EWMA) method [34] is used to estimate the

average network latency of data requests. Specifically, after a data read operation, l_i^t is updated by

$$l_i^t = \alpha_l \cdot l_i^t + (1-\alpha_l) \cdot \iota_i, \tag{6}$$

where ι_i is the measured end-to-end latency of a data request, and α_l is the discount factor to reduce the impact of the previous requests. The advantage of EWMA is that it only needs $O(N)$ space for N storage nodes at each edge server to maintain the prediction. The long-tested techniques EDC and EWMA are used to estimate future data popularity and network latency information with low implementation overheads. Recent advances in future information prediction, e.g., Least Hit Density (LHD) [15] and Hyperbolic [38], could also be applied to our solution.

Let Γ denote the set of data requests in the service period $\mathcal{T}$. The caching decision is updated upon the arrival of each request γ_m^t, $\gamma_m^t \in \Gamma$. Based on the latest measurement of data access latency l_i^t and request rate r_m^t, a $(K+1)$-dimensional array is maintained for each data item

$$\{\tau_{m,0}, \tau_{m,1}, \ldots, \tau_{m,K}\} = \{0, (l_{m_1}^t - l_{m_2}^t) \cdot r_m^t, \ldots, (l_{m_1}^t - l_{m_k}^t) \cdot r_m^t, \ldots, (l_{m_1}^t - l_{m_K}^t) \cdot r_m^t, l_{m_1}^t \cdot r_m^t\}, \tag{7}$$

where $\tau_{m,k-1} = (l_{m_1}^t - l_{m_k}^t) \cdot r_m^t$ denotes the value of reduced latency when $k-1$ data chunks are cached. For example, if chunk m_1 and m_2 are cached, l_{m_3} becomes the bottleneck. Without caching, the latency reduction $\tau_{m,0} = 0$. If all K data chunks are cached, the latency is reduced to 0 with $\tau_{m,K} = l_{m_1}^t \cdot r_m^t$. Due to the monotonic decreasing nature of $f_m^t(\lambda_m^t)$, problem (4) is equivalent to maximizing the total amount of reduced latency

$$\begin{aligned} \max_{\lambda_m^t \in \mathbb{N}, m \in \mathcal{M}} \quad & \Theta(\lambda_m^t) = \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} \tau_{m,\lambda_m^t} \\ \text{s.t.} \quad & 0 \le \lambda_m^t \le K, \\ & \sum_{m \in \mathcal{M}} \lambda_m^t = C. \end{aligned} \tag{8}$$

Due to the presence of erasure codes, problem (8) does not belong to the well-studied 0-1 Knapsack problem, which requires each data item in its entirety to be cached. For a data item, each data chunk cannot be treated individually as the latency is progressively reduced with more cached chunks. Furthermore, problem (8) differs from Fractional Knapsack due to 1) the nonlinearity between the fraction (data chunk) and its value (latency reduction), and 2) the finite number of fractions for a data item. Therefore, mature solutions for Knapsack problems cannot be directly applied to problem (8). In this section, to obtain the number of cache data chunks λ_m^t in real time, an Adaptive Content Replacement (ACR) scheme is designed with a detailed theoretical analysis.

The pseudo code of our design is listed in Algorithm 1. Let $\hat{\mathcal{M}}$ denote the set of data items cached at the edge server. If the cache capacity is not fully utilized, i.e., $\sum_{n \in \hat{\mathcal{M}} \setminus \{m\}} \lambda_n^t \le C - K$, all K chunks of the requested data item m should be cached. In contrast, if $\sum_{n \in \hat{\mathcal{M}} \setminus \{m\}} \lambda_n^t > C - K$, we need to determine 1) whether data item m should be cached or not, 2) how many chunks for m should be cached, and 3) which data items in $\hat{\mathcal{M}}$ should be replaced? To solve this problem, let $\frac{\tau_{n,k}}{k}$ denote the unit valuation of latency reduction when k chunks of data item n are cached. Then, the data items $n \in \hat{\mathcal{M}}$ with the lowest unit valuations are added into subset $\hat{\mathcal{M}}'$. The data items in $\hat{\mathcal{M}}'$ will be replaced first by the requested data item m to maximize the amount of reduced latency. Besides, m is also added into $\hat{\mathcal{M}}'$. All data items in $\hat{\mathcal{M}}'$ are cache replacement candidates ($\forall \lambda_n^t \leftarrow 0$, $n \in \hat{\mathcal{M}}'$). The cached data items in $\hat{\mathcal{M}}$ are gradually added into $\hat{\mathcal{M}}'$ until the available cache capacity $C^{[a]} = C - \sum_{n \in \hat{\mathcal{M}}} \lambda_n^t \ge K$. This guarantees that all K data chunks of m have a chance to be cached. The expansion of $\hat{\mathcal{M}}'$ needs K iterations at most with $|\hat{\mathcal{M}}'| \le K+1$ and $C^{[a]} \le 2K-1$. Then, data chunks with the highest unit valuations, i.e., $\{n,k\} \leftarrow \arg\max_{n \in \hat{\mathcal{M}}'} \{\frac{\tau_{n,k}}{k}\}$, are selected to be cached. The selection process is repeated until the cache capacity is fully utilized, i.e., $\sum_{n \in \mathcal{M}} \lambda_n^t = C$. The theoretical analysis is provided as follows.

Algorithm 1. Adaptive Content Replacement

Input: Cache capacity C, number of coded data chunks K, data item set $\mathcal{M}$, valuation array τ, data request set Γ.
Output: Set of cached data items $\hat{\mathcal{M}}$, caching decision λ_n^t.

1: **for** Data request $\gamma_m^t \in \Gamma$, $t \in \mathcal{T}$ **do**
2: Update $\{\tau_{m,0}, \tau_{m,1}, \ldots, \tau_{m,K}\}$ based on (7);
3: **if** $\sum_{n \in \hat{\mathcal{M}} \setminus \{m\}} \lambda_n^t \le C - K$ and $\lambda_m^t < K$ **then**
4: $\lambda_m^t \leftarrow K$, add m to $\hat{\mathcal{M}}'$;
5: **else if** $\sum_{n \in \hat{\mathcal{M}} \setminus \{m\}} \lambda_n^t > C - K$ and $\lambda_m^t < K$ **then**
6: $\hat{\mathcal{M}}' \leftarrow \{m\}$, $\lambda_m^t \leftarrow 0$;
7: **repeat**
8: $n \leftarrow \arg\min_{n \in \hat{\mathcal{M}} \setminus \hat{\mathcal{M}}'} \{\frac{\tau_{n,k}}{k}\}$, $\lambda_n^t \leftarrow 0$, remove n from $\hat{\mathcal{M}}$, add n to $\hat{\mathcal{M}}'$;
9: **until** $C^{[a]} = C - \sum_{n \in \hat{\mathcal{M}}} \lambda_n^t \ge K$
10: **repeat**
11: Select $\{n,k\} \leftarrow \arg\max_{n \in \hat{\mathcal{M}}'} \{\frac{\tau_{n,k}}{k}\}$, $k \le C - \sum_{n \in \mathcal{M}} \lambda_n^t$;
12: $\lambda_n^t \leftarrow k$ and mark $\{n,k\}$ as selected;
13: **until** $\sum_{n \in \mathcal{M}} \lambda_n^t = C$
14: **end if**
15: Update $\hat{\mathcal{M}}$, insert n into $\hat{\mathcal{M}}$ if $\lambda_n^t > 0$, $\forall n \in \hat{\mathcal{M}}'$;[3]
16: **end for**

Theorem 1. *Algorithm 1 is $\frac{C-K+1}{2C-K+1}$-approximation, reaching more than $\frac{C-K+1}{2C-K+1}$ of the maximum reduced latency.*

Proof. Let $\{\hat{\mathcal{M}}, \hat{\mathcal{K}}\}$ denote the set of caching decisions induced by Algorithm 1. Let $\{\mathcal{M}^*, \mathcal{K}^*\}$ denote the set of optimal caching decisions. Then, we consider the following two cases:

1) $\forall m^* \in \mathcal{M}^* \cap \hat{\mathcal{M}}$: If $k^* \le k$, $\tau_{m^*,k^*} \le \tau_{m^*,k}$ holds as caching more chunks for a data item is always beneficial to reducing the access latency for that data item. Similarly, if $k < k^*$, $\tau_{m^*,k} \le \tau_{m^*,k^*}$ holds. However, for the

3. We use binary search for the insertion to ensure $\hat{\mathcal{M}}$ is maintained in ascending order. With size $|\hat{\mathcal{M}}| \le C$, the computation complexity of the binary search is $O(\log C)$. The sorted set $\hat{\mathcal{M}}$ can reduce the lookup cost of Algorithm 1 (Line 8). If set $\hat{\mathcal{M}}$ is not sorted, the insertion cost of $\hat{\mathcal{M}}'$ is $O(K)$ but with higher lookup cost $O(C)$. If set $\hat{\mathcal{M}}$ is sorted, the insertion cost goes up (from $O(K)$ to $O(K \cdot \log C)$) but the lookup cost reduces dramatically (from $O(C)$ to $O(K)$). In large-scale storage systems, the number of coded data chunks per data item K is much smaller than the cache capacity C, i.e., $K \ll C$. Therefore, our design can reduce the overall cost of insertion and lookup (from $O(K+C)$ to $O(K + K \cdot \log C)$).

caching space $k^* - k$, we can always find a data item $m' \in \hat{\mathcal{M}} \setminus \mathcal{M}^*$ with $\frac{\tau_{m^*,k^*}}{k^*} \leq \frac{\tau_{m',k'}}{k'}$. This is because if $\frac{\tau_{m^*,k^*}}{k^*} > \frac{\tau_{m',k'}}{k'}$ always holds, Algorithm 1 will update the caching decision of data item m^* from k to k^* for more latency reduction. As $\sum_{k^* \in \mathcal{K}^* \cap \hat{\mathcal{K}}} k^* \leq \sum_{k \in \hat{\mathcal{K}}} k = C$, we have

$$\sum\nolimits_{m^* \in \mathcal{M}^* \cap \hat{\mathcal{M}}} \tau_{m^*,k^*} \leq \sum\nolimits_{m \in \hat{\mathcal{M}}} \tau_{m,k}. \tag{9}$$

2) $\forall m^* \in \mathcal{M}^* \setminus \hat{\mathcal{M}}$: According to the value of $\frac{\tau_{m^*,k^*}}{k^*}$, set $\mathcal{M}^* \setminus \hat{\mathcal{M}}$ (with set size $|\mathcal{M}^* \setminus \hat{\mathcal{M}}| = q$) is sorted in descending order

$$\frac{\tau_{m_1^*,k_1^*}}{k_1^*} \geq \frac{\tau_{m_2^*,k_2^*}}{k_2^*} \geq \ldots \geq \frac{\tau_{m_q^*,k_q^*}}{k_q^*}. \tag{10}$$

Similarly, set $\hat{\mathcal{M}} \setminus \mathcal{M}^*$ (with set size $|\hat{\mathcal{M}} \setminus \mathcal{M}^*| = p$) is also sorted in descending order

$$\frac{\tau_{m_1,k_1}}{k_1} \geq \frac{\tau_{m_2,k_2}}{k_2} \geq \ldots \geq \frac{\tau_{m_p,k_p}}{k_p}. \tag{11}$$

Then, set $\hat{\mathcal{M}} \setminus \mathcal{M}^*$ is divided into two subsets $\hat{\mathcal{M}}_1$ and $\hat{\mathcal{M}}_2$

$$\begin{cases} \overbrace{\frac{\tau_{m_1,k_1}}{k_1} \geq \ldots \geq \frac{\tau_{m_{p'},k_{p'}}}{k_{p'}}}^{\hat{\mathcal{M}}_1} \geq \frac{\tau_{m_1^*,k_1^*}}{k_1^*}, \\ \frac{\tau_{m_1^*,k_1^*}}{k_1^*} \geq \underbrace{\frac{\tau_{m_{p'+1},k_{p'+1}}}{k_{p'+1}} \geq \ldots \geq \frac{\tau_{m_p,k_p}}{k_p}}_{\hat{\mathcal{M}}_2}, \end{cases} \tag{12}$$

Please note that $k_{p'+1} + \ldots + k_p < k_1^* \leq K$ always holds. This is because with Algorithm 1, if $k_{p'+1} + \ldots + k_p > k_1^*$, $\{m_1^*, k_1^*\}$ will be added to $\{\hat{\mathcal{M}}, \hat{\mathcal{K}}\}$. This contradicts with the fact that $m_1^* \in \mathcal{M}^* \setminus \hat{\mathcal{M}}$. Based on (9) and (12), we have

$$\begin{aligned} \frac{\Theta^*}{\Theta} &< 1 + \frac{\sum_{m^* \in \mathcal{M}^* \setminus \hat{\mathcal{M}}} \tau_{m^*,k^*}}{\sum_{m \in \hat{\mathcal{M}}_1} \tau_{m,k} + \sum_{m \in \hat{\mathcal{M}}_2} \tau_{m,k}} \\ &\leq 1 + \frac{\frac{\tau_{m_1^*,k_1^*}}{k_1^*} \cdot C}{\frac{\tau_{m_{p'},k_{p'}}}{k_{p'}} \cdot (C - K + 1)} \\ &\leq \frac{2C - K + 1}{C - K + 1}. \end{aligned} \tag{13}$$

The proof completes. □

Proposition 1. *For a data request, Algorithm 1 has the computation complexity of* $O(K^2 + K \cdot \log C)$.

Proof. For a data request, the $(K+1)$-dimensional array is updated with the computation complexity of $O(K)$ (Line 2). If $\sum_{n \in \hat{\mathcal{M}} \setminus \{m\}} \lambda_n^t \leq C - K$, the whole data item m is cached with the computation complexity of $O(1)$ (Line 4). Otherwise, the lookup process for subset $\hat{\mathcal{M}}'$ needs $K+1$ steps at most (Line 6–9). With $|\hat{\mathcal{M}}'| \leq K+1$, based on the radix sort algorithm, the valuation for all data items in $\hat{\mathcal{M}}'$ is sorted with the computation complexity of $O((K+1) \cdot K)$. As $\sum_{n \in \hat{\mathcal{M}}'} \lambda_n^t < 2K - 1$, the greedy selection needs $2K-1$ iterations at most to obtain the caching decisions (Line 10–13). Then, the update of $\hat{\mathcal{M}}$ needs $(K+1) \cdot \log C$ iterations at most for all uncached data items in $\hat{\mathcal{M}}'$ (Line 15). The computation complexity of Algorithm 1 is $O(K^2 + K \cdot \log C)$. □

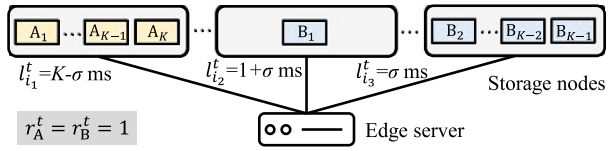


Fig. 3. An example of the distributed coded storage system with caching at the edge server is shown. Both data items A and B are coded into K data chunks and R parity chunks. According to our design in Section 3.1, the degraded read will be passively triggered during storage server failure. For simplicity, the coded parity chunks are not shown.

We use a simple example to demonstrate that the approximation ratio is a fairly tight bound. The storage locations of data items A and B, the average data access latencies, and the request popularities are provided in Fig. 3. We assume σ is an arbitrarily small positive number. The $(K+1)$-dimensional valuation arrays of latency reduction are

$$\begin{cases} \tau_A = \{0,\ 0,\ 0,\ \ldots.,\ 0,\ K - \sigma\}, \\ \tau_B = \{0,\ 1,\ 1,\ \ldots.,\ 1,\ 1 + \sigma\}. \end{cases} \tag{14}$$

The unit valuation of latency reduction $\frac{\tau_m}{k}, k = \{1, \ldots, K\}$, is

$$\begin{cases} \frac{\tau_A}{k} = \{0,\ 0,\ \ldots.,\ 0,\ 1 - \frac{\sigma}{K}\}, \\ \frac{\tau_B}{k} = \{1,\ \frac{1}{2},\ \ldots.,\ \frac{1}{K-1},\ \frac{1+\sigma}{K}\}. \end{cases} \tag{15}$$

At the high level, ACR prefers to choose data chunks with the highest unit valuations for caching to reduce the data access latency. Let cache capacity $C = K$. At the beginning, ACR selects $\lambda_B^t = 1$ (with $\frac{\tau_B}{1} = 1$). Then, ACR prefers to select $\lambda_A^t = K$ (with $\frac{\tau_A}{K} = 1 - \frac{\sigma}{K}$). However, the remaining cache capacity $K-1$ is not enough to accommodate the data chunks $\lambda_A^t = K$. So, $\lambda_B^t = 2$ (with $\frac{\tau_B}{2} = \frac{1}{2}$) is selected. The limited cache capacity incurs performance loss. We repeat the selection process until $\lambda_B^t = K$. ACR outputs caching decision $\{\lambda_A^t, \lambda_B^t\} = \{0, K\}$ with the reduced data access latency $\Theta = 1 + \sigma$. The optimal caching decision is $\{\lambda_A^{t*}, \lambda_B^{t*}\} = \{K, 0\}$ with the reduced data access latency $\Theta^* = K - \sigma$. We have $\frac{\Theta}{\Theta^*} = \frac{1+\sigma}{K-\sigma} > \frac{C-K+1}{2C-K+1}$ as $C = K$ in this case and σ is an arbitrarily small positive number. This means if $C = K$, high performance loss may be incurred.

In large-scale storage systems given $K \ll C$, Theorem 1 shows that the approximation ratio is about $\frac{1}{2}$. This indicates that in the worst case, the ACR scheme incurs up to a 50% performance loss in terms of reduced latency Θ. Guided by Algorithm 1, an Adaptive Content Adjustment (ACA) scheme is proposed for further performance improvements.

4.2 Adaptive Content Adjustment

For low computation complexity, Algorithm 1 prefers to choose data chunks with the highest unit valuations for caching. However, due to the limited size of cache capacity, the remaining capacity may not always be enough to accommodate the selected data chunks. As shown in (12), with lower unit valuations, data chunks in $\hat{\mathcal{M}}_2$ incur the performance loss. Therefore, for performance improvements, the cache contents in $\hat{\mathcal{M}}_2$ should be replaced by m_1^*. The pseudo

code of ACA is listed in Algorithm 2, which is based on the outputs of Algorithm 1.

Algorithm 2. Adaptive Content Adjustment

Input: Valuation array τ.
Output: Set of cached data items $\hat{\mathcal{M}}$, caching decision λ_n^t.
1: **for** Data request $\gamma_m^t \in \Gamma, t \in \mathcal{T}$ **do**
2: Invoke Algorithm 1 for cache replacement candidates $\hat{\mathcal{M}}'$ (with available capacity $C^{[\mathrm{a}]}$), cached data items $\hat{\mathcal{M}}$, and caching decisions λ_n^t;
3: Sort data items $n \in \hat{\mathcal{M}} \cap \hat{\mathcal{M}}'$ based on $\frac{\tau_{n,\lambda_n^t}}{\lambda_n^t}$ in ascending order;
4: **for** $k \in \{1, \ldots, K\}$ **do**
5: $\grave{\lambda}_n^t \leftarrow \lambda_n^t, \forall n \in \hat{\mathcal{M}}'$;
6: From left to right for $n \in \hat{\mathcal{M}} \cap \hat{\mathcal{M}}'$, $\grave{\lambda}_n^t$ is decremented until $C^{[\mathrm{a}]} - \sum_{n \in \hat{\mathcal{M}} \cap \hat{\mathcal{M}}'} \grave{\lambda}_n^t = k$, $\grave{\lambda}_n^t \geq 0$;
7: **for** $n \in \hat{\mathcal{M}}'$ **do**
8: Restore the value of $\grave{\lambda}_n^t$ as in Line 5 and 6;
9: $\grave{\lambda}_n^t \leftarrow \min\{K, \grave{\lambda}_n^t + k\}$, $V'_{n,k} \leftarrow \sum_{n \in \hat{\mathcal{M}}'} \grave{\lambda}_n^t$;
10: **end for**
11: **end for**
12: $\grave{\lambda}_n^t \leftarrow \arg\max\{V'_{n,k}\}$;
13: $\lambda_n^t \leftarrow \grave{\lambda}_n^t$ if $\max\{V'_{n,k}\} > \sum_{n \in \hat{\mathcal{M}}'} \tau_{n,\lambda_n^t}$, update $\hat{\mathcal{M}}$;
14: **end for**

To replace cache contents in set $\hat{\mathcal{M}}_2$, the selected data items among cache replacement candidates $n \in \hat{\mathcal{M}} \cap \hat{\mathcal{M}}'$ are sorted based on $\frac{\tau_{n,\lambda_n^t}}{\lambda_n^t}$ in ascending order first. Let $\grave{\lambda}_n^t$ denote the updated caching decisions, which is initialized as $\grave{\lambda}_n^t \leftarrow \lambda_n^t$, $\forall n \in \hat{\mathcal{M}}'$. Based on the sorted information, the cached chunks in $\hat{\mathcal{M}} \cap \hat{\mathcal{M}}'$ are gradually removed from the caching layer for replacement (Line 6). The data chunks with lower unit valuations will be removed first. As discussed in (12), with $k_{p'+1} + \ldots + k_p \leq K$, we gradually increase the number of removed chunks k from 1 to K (Line 4–11). Then, all cache replacement candidates $n \in \hat{\mathcal{M}}'$ are considered one by one to search for m_1^*. The cache decision is updated with $\grave{\lambda}_n^t \leftarrow \min\{K, \grave{\lambda}_n^t + k\}$ to fill the vacated caching space. The valuation of the updated cache decision is $V'_{n,k} \leftarrow \sum_{n \in \hat{\mathcal{M}}'} \grave{\lambda}_n^t$. We choose the cache decision with the highest valuation, i.e., $\grave{\lambda}_n^t \leftarrow \arg\max\{V'_{n,k}\}$. If $\max\{V'_{n,k}\} > \sum_{n \in \hat{\mathcal{M}}'} \tau_{n,\lambda_n^t}$, the cache decision yielded by Algorithm 1 is replaced by $\lambda_n^t \leftarrow \grave{\lambda}_n^t$. The set of cached data items $\hat{\mathcal{M}}$ is also updated accordingly. With a higher valuation than the output of Algorithm 1, the approximation ratio of Algorithm 2 is higher than $\frac{C-K+1}{2C-K+1}$.

Let us reuse the example in Fig. 3 to show the workflow of Algorithm 2. To search for better caching decisions, ACA gradually decreases the number of already cached chunks (Line 6) to make room for other uncached data chunks (Line 9). Based on the caching decision of ACR $\{\lambda_{\mathrm{A}}^t, \lambda_{\mathrm{B}}^t\} = \{0, K\}$, ACA considers the caching decisions of $\{1, K-1\}, \{2, K-2\}, \ldots, \{K-1, 1\}$, and $\{K, 0\}$. Among them, the caching decision $\{\lambda_{\mathrm{A}}^t, \lambda_{\mathrm{B}}^t\} = \{K, 0\}$ with the highest reduced latency is selected (Line 12–13). In this example, ACA obtains the optimal caching decision. Experiment results in Section 5 show that compared with Algorithm 1, Algorithm 2 can further reduce the data access latency by about 5% without sacrificing efficiency.

Proposition 2. *For a data request, Algorithm 2 requires an extra computation complexity of* $O(K^2)$.

Proof. With $|\hat{\mathcal{M}}'| \leq K+1$, the data items in $\hat{\mathcal{M}} \cap \hat{\mathcal{M}}'$ is sorted with the computation complexity of $O(K)$ (Line 3). Then, the calculation of the updated cache decisions $\grave{\lambda}_n^t$ needs $K \cdot 2 \cdot (K+1)$ iterations (Line 4–11). Based on the outputs of Algorithm 1, Algorithm 2 requires an extra computation complexity of $O(K^2)$. □

4.3 Caching Design Under Server Failure

In the distributed storage system, servers may experience downtime frequently. If a storage server at node i fails at time t, a set of data chunks (denoted by $\mathcal{M}_i$) becomes remotely unavailable. Recall that we need exactly K chunks to reconstruct a data item. If data chunk m_k is not cached beforehand, the degraded read is triggered to serve the data requests, $m_k \in \mathcal{M}_i$. In this case, the parity chunk m_r with the lowest data access latency will be fetched from node j to reconstruct the data item. In Algorithm 1 (or Algorithm 2), the unavailable data chunk m_k is replaced by parity chunk m_r, i.e., $m_k \leftarrow m_r$ and $l_{m_k}^t \leftarrow l_{m_r}^t$. Similar to (2), the average latency of sending m_r is given by

$$l_{m_r}^t = \min\{l_j^t \cdot \mathbf{1}(m_r \rightarrow j)\}. \tag{16}$$

When Algorithm 1 (or Algorithm 2) suggests caching m_r, the recovered m_k is directly added into the caching layer instead of m_r. As a result, the parity chunk m_r is not always required to reconstruct data item m. This means our design can reduce the decoding overheads of the subsequent data requests.

5 Experimental Evaluation

In this section, we implement the proposed cache schemes in Python, and then integrate them in the experiment platform deployed in Section 3.2. Next, extensive experiments are performed to evaluate the performance of our design.

5.1 Experimental Setup

As shown in Table 2, $N = 6$ storage nodes are deployed over six AWS regions. Each storage node creates three `buckets`, each of which represents a server for remote data storage. By default, the storage system is populated with $M = 10,000$ data items. The `zfec` [39] library is adopted to implement the RS codes. The numbers of data and parity chunks are set as $K = 6$ and $R = 3$. The coded nine chunks of each data item are with the same block size of 1 MB [8], which are uniformly distributed among eighteen `buckets` to achieve fault tolerance.

As shown in Table 2, three edge servers are deployed on personal computers in three different cities. The hardware features an Intel(R) Core(TM) i7-7700 HQ processor and 16 GB of memory. `Memcached` [31] module is used to cache data chunks in RAM. The assigned cache capacity is 1,000 MB, i.e., the maximum number of cached data chunks is $C = 1,000$. To request data chunks in parallel, a thread pool is created at each edge server. The time period of data services $\mathcal{T}$ is 1 hour. We rely on the MSR Cambridge Traces [40] as the workload of data requests, but not the data sizes as they are not collected from coded storage systems. We assume all data items are with the same size. MSR Cambridge Traces are production traces collected from Microsoft Research,

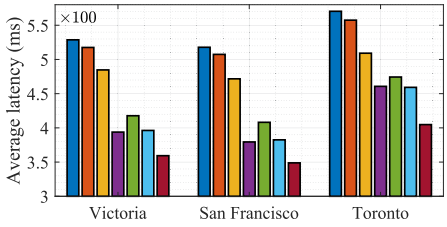


Fig. 4. Average data request latencies.

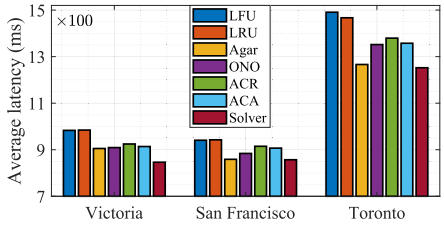


Fig. 5. 95th percentile tail latencies.

which have been widely used to evaluate the performance of the caching systems [12], [15].

Performance Baselines. To evaluate the performance of the proposed ACR and ACA schemes, five baselines are introduced for a fair performance comparison.

- LFU and LRU—The Least Frequently Used (LFU) and the Least Recently Used (LRU) policies are used for data eviction in the caching layer with the computation complexity of $O(1)$ [41]. LFU and LRU cache all K data chunks for each selected data item.
- Agar [8]—A dynamic programming-based scheme is designed to iteratively add data chunks with larger request rates and higher data access latencies in the caching layer with the computation complexity of $O(CKM)$.
- Online Near-Optimal (ONO) scheme [20]—This scheme updates the caching decision upon the arrival of each data request. Specifically, through iterative search on Diophantine equations, the feasible caching decisions are categorized into a set of cache partitions $\{x_1, \ldots, x_K\} \in \chi$ based on the number of cached chunks for each data item. Then, the online caching decision is obtained by applying the market clearing price [42] on all cache partitions (the caching partition with the highest latency reduction is selected). Theoretical analysis shows that the approximation ratio of the near-optimal scheme is $1 - \frac{2K-1}{C}$. The computation complexity is $O(K^2 \cdot P \cdot K!)$ in updating the caching decision of a data request, where P is the "potential energy" driving the pricing process

$$P = \sum\nolimits_{k=1}^{K} \underset{m \in \mathcal{M}}{\texttt{sum\_top}}\{\tau(m,k), x_k\}, \tag{17}$$

  and function $\underset{m \in \mathcal{M}}{\texttt{sum\_top}}\{\tau(m,k), x_k\}$ represents the sum of largest x_k elements.
- Solver—Assuming that the future data request rates and network condition information is available, the offline caching decisions are obtained by solving the integer programming problem with the optimization solver IBM ILOG CPLEX.

5.2 Experimental Results

In the beginning, the performance of seven schemes is compared in terms of the latency and hit ratio of data requests under default settings. LFU prefers to cache data items with higher request rates. LRU caches the recently requested data items by discarding the least recently used data items. As shown in Fig. 4, the average latencies of all data requests from three locations incurred by LFU and LRU are 538.9 ms and 527.5 ms, respectively. Besides, as shown in Fig. 6, with LFU and LRU, 28.7% and 30.2% of data chunk requests are served by cached data chunks. With the whole recently requested data items cached, LFU and LRU reduce the access latencies to 0 ms for 28.7% and 30.2% of data requests, respectively. Due to the limited cache capacity, the remaining parts of the data requests suffer from high access latencies. Therefore, as shown in Fig. 5, the 95th percentile tail latencies of LFU and LRU are 1,138.2 ms and 1,131.3 ms, respectively. LFU and LRU overlook the diversity of data chunk storage locations and the heterogeneity of latencies across different storage nodes. Caching the whole data item cannot enjoy the full benefits of caching for low latency.

Agar iteratively improves the existing caching configurations by considering new data chunks. Compared with LFU and LRU using whole data item caching, more data items enjoy the benefits of caching. The hit ratios of data requests from three edge servers are increased to 32.9%, 32.7%, and 33.5%, respectively. The average latencies of requests are reduced to 484.7 ms, 471.7 ms, and 509.1 ms, respectively. Furthermore, Agar prefers to evict low valuation data chunks which incur high access latencies from the caching layer. The 95th percentile tail latencies are reduced to 905.3 ms, 858.8 ms, and 1,266.2 ms, respectively.

The ONO scheme improves the caching configuration by applying the market clearing price on cache partitions. Compared with LFU and LRU using the whole data item caching, more data items enjoy the benefits of caching. The average hit ratio of data requests from three edge servers is increased to 37.0%, 37.1%, and 36.1%, respectively. The average latency of requests is reduced to 393.81 ms, 379.43 ms, and 460.68 ms, respectively. Furthermore, the ONO scheme prefers to evict low valuation data chunks that incur high access latencies from the caching layer. The average 95th percentile tail latency is reduced to 924.82 ms, 893.81 ms, and 1,380.77 ms, respectively.

The proposed ACR scheme prefers to cache the data chunks with the highest unit valuations. This means the contents in the caching layers are with higher data request rates and lower access latencies. The hit ratios of data requests from three edge servers are increased to 37.6%,

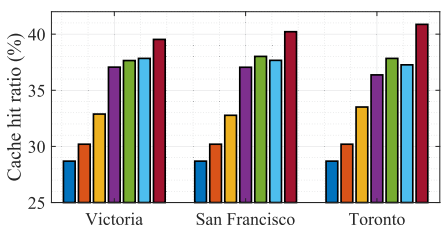


Fig. 6. Hit ratio of data chunk requests.

TABLE 3
Average Running Time of Various Schemes at Three Edge Servers

Scheme		LFU	LRU	Agar	ONO	ACR	ACA	Solver
Average running time	**Victoria, CA**	0.19 ms	0.16 ms	63.38 s	4.88 ms	1.67 ms	1.75 ms	820.18 s
	San Francisco, US	0.20 ms	0.17 ms	74.28 s	4.21 ms	1.63 ms	1.90 ms	718.43 s
	Toronto, CA	0.19 ms	0.17 ms	65.37 s	4.55 ms	1.46 ms	1.64 ms	750.39 s

38.0%, and 37.8%, respectively. The average access latencies of three edge servers are reduced to 417.7 ms, 408.1 ms, and 474.3 ms, respectively. Based on the outputs of ACR, the subset of cached contents which incur performance loss is updated with the proposed ACA scheme. As shown in Fig 4, the average latencies with ACA are further reduced to 396.2 ms, 382.5 ms, and 459.2 ms, respectively. Compared with the offline Solver, ACA incurs the performance loss of 9.3%, 8.8%, and 11.8%, respectively.

Then, the average running time of various schemes, which determine the efficiency of updating caching decisions, is compared under default settings.[4] In an offline scenario, the Solver scheme obtains near-optimal caching decisions by using the commercial solver. As shown in Table 3, the average running time of Solver is in tens of minutes. Agar periodically optimizes the caching configuration for all data items in the storage system, which needs tens of seconds for a round of optimization. The long running time implies that Agar cannot react quickly to real-time network changes. In contrast, LFU, LRU, and the proposed ACR and ACA schemes update the caching decision upon the arrival of each data request, without completely overriding the existing caching solution. By directly caching the recently requested data items, LFU and LRU achieve the minimum computation complexity with the average running time of 0.19 ms and 0.17 ms, respectively, for a data request. As a caching scheme working at the data chunk level, the ONO scheme optimizes the caching configuration by considering multiple cache partitions, which needs a longer running time of 4.44 ms to handle a data request. With the computation complexity sublinear to the cache capacity, ACR needs about 1.59 ms to finish. Compared with ACR, ACA reduces the average latency by about 5% with a little bit longer running time of 1.76 ms. With ACR and ACA, few extra delays will be introduced when handling the intensive data requests. Compared with ONO, the computation overhead of ACA is reduced by 60.36% while only incurring a performance loss of 0.32%.

5.3 Impact of Other Factors

The impact of other factors, i.e., cache capacity, number of coded data chunks, number of data replicas, server failure, concurrent applications, and prediction methods, is considered in the experiments. By default, we set the cache capacity $C = 1,000$ data chunks. The number of data items is set to $M = 10,000$. Without considering data replication, the numbers of coded data and parity chunks per data item are set to $K = 6$ and $R = 3$, respectively. For simplicity, the average latency represents the average latency of all data requests from three edge servers in the following parts of the paper.

Cache Capacity C. Fig. 7 illustrates the average latencies when the cache capacity C increases from 600 to 1,200 chunks. As more data requests enjoy the caching benefits, the average latencies with all seven caching schemes decrease. With the increase of C, the proposed schemes have more space for caching decision optimization. Compared with Agar, the percentage of reduced latency via the proposed ACA scheme is improved from 1.4% to 21.5%.

Then, the average running time of seven caching schemes is evaluated. As shown in Table 4, with the increase of cache capacity, the offline caching schemes, i.e., Agar and Solver, need more and more time for a round of optimization. In contrast, the online scheme updates the caching decision upon the arrival of each data request. According to our design in Algorithm 1 and 2, when a data request arrives, the caching decision will not be updated if the data item is already cached. Therefore, the average running time of ACR and ACA for each request decreases with the increase of cache capacity. This means the proposed online schemes are scalable solutions for a large-scale storage system.

Number of Coded Data Chunks K. We increase the size of data items from 4 MB to 10 MB. With the same size of coded chunks (1 MB), the number of coded data chunks K increases from 4 to 10. As coded chunks are uniformly distributed among eighteen `buckets`, more data chunks will be placed at the `buckets` with higher access latencies with the increase of K. Moreover, when the data item is coded into more data chunks, more requests are served by fetching data chunks from the remote `buckets`. The average latencies with all seven schemes increase accordingly. Fig. 8 shows that ACA always incurs lower latencies than ACR, Agar, LRU, and LFU. Compared with Agar, the percentage of reduced latency via ACA varies from 29.9% to 2.2% with the increase of K. Compared with LFU, the percentage of reduced latency via ACA varies from 35.7% to 13.3%. Furthermore, Table 4 shows that the average running time of ACA only increases from 1.18 ms to 2.23 ms. With the proposed online scheme,

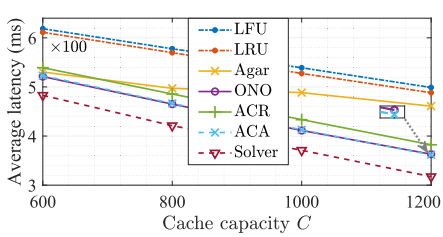


Fig. 7. Impact of cache capacity.

4. The average running time is the quantity of processor time taken by the caching schemes at the edge servers, which has been widely adopted as an indicator to show how CPU intensive a process or program is [20], [43].

TABLE 4
The Average Running Time of Caching Schemes Under Various Settings

C		600	800	1,000	1,200
Offline (s)	Agar	42.69	57.98	67.68	92.43
	Solver	431.36	621.82	763.0	1,042.05
Online (ms)	LFU	0.19			
	LRU	0.17			
	ONO	5.12	4.75	4.44	4.25
	ACR	1.92	1.79	1.59	1.43
	ACA	2.01	1.89	1.76	1.6

K		4	6	8	10
Offline (s)	Agar	45.87	67.68	96.21	116.60
	Solver	487.65	763.0	1,046.86	1,431.54
Online (ms)	LFU	0.19			
	LRU	0.17			
	ONO	3.10	4.44	9.14	14.45
	ACR	1.11	1.59	1.82	2.14
	ACA	1.18	1.76	2.08	2.23

few extra delays will be introduced to handle the intensive data requests. Compared with ONO, ACA can reduce the computation overhead by up to 84.57% while only incurring a performance loss of 1.12%.

Number of Data Replicas. In the distributed coded storage system, if data replication is considered, more than a single copy of data chunks will be created at remote `buckets`. With the increased number of data replicas, more data chunks will be placed at the storage nodes near end users. As shown in Fig. 9, since the faraway storage nodes are no longer the bottleneck, the average data access latency of ACA is considerably reduced from 412.6 ms to 130.8 ms. Compared with Agar, the percentage of reduced latency via ACA varies from 17.8% to 4.1% with the increased number of data replicas. Compared with ONO, the computation overhead of ACA is reduced by about 60% with a performance loss ranging from 0.32% to 1.64%.

Server Failure. The performance of our design is then evaluated in the presence of server failure. Although erasure codes can tolerate up to R simultaneous server failures, it has been reported that single server failure is responsible for 99.75% of all kinds of server failures [44]. Therefore, we

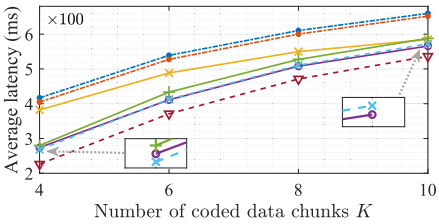


Fig. 8. Impact of number of coded data chunks.

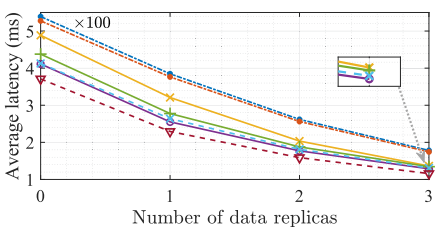


Fig. 9. Impact of data replicas.

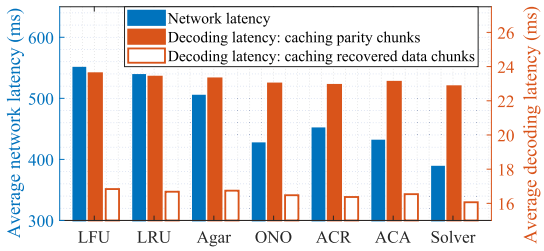


Fig. 10. Average data access latencies in the presence of server failure.

consider single server failure by terminating each storage server in turn. This experiment setting is identical to that in Section 5.2 except for the storage server failure. If the needed data chunks are not available on the remote servers or cached beforehand, degraded read will be triggered to serve data requests. Therefore, the data access latency includes both the network latency and the decoding latency.

Fig. 10 illustrates the average data access latencies with various schemes. We start by considering the performance of caching parity chunks if the needed data chunks are remotely unavailable under server failure. The seven caching schemes incur the average decoding latencies of about 23 ms. Then, let us consider the performance of caching the recovered data chunks to avoid unnecessary decoding overheads of the subsequent data requests. LFU, LRU, Agar, ONO, and the proposed ACR and ACA schemes incur low decoding latencies of 16.8 ms, 16.6 ms, 16.7 ms, 16.4 ms, 16.3 ms, and 16.5 ms, respectively. This means the computation overhead of the decoding process is reduced by about 28% if the recovered data chunks are cached.

Compared with LFU, LRU, Agar, and ACR, the ACA scheme reduces the overall average data access latency by 21.1%, 19.4%, 22.1%, 14.2%, and 4.3%, respectively. Furthermore, compared with ONO, ACA incurs a performance loss of 1.0% but can reduce the computation overhead by about 60%. Compared with the time-consuming Solver scheme, ACA incurs a performance loss of 9.0% but with much higher efficiency in the presence of server failure.

Concurrent Applications. As a general-purpose computing and storage paradigm, the edge system may need to handle a variety of applications, which may have different request patterns. Fig. 11 shows the time-varying data request rates of two MSR Cambridge Traces.[5] The Standard Deviations of Trace 1 and Trace 2 are 6.54 and 15.91 (in requests/second), respectively. Traces 1 and 2 represent two classes of applications with relatively steady and bursty request patterns, respectively.

Fig. 12 compares the performance of seven caching schemes by replaying the two traces separately. The average data access latencies of two traces are 588.66 ms, 581.15 ms, 524.12 ms, 488.48 ms, 508.17 ms, 492.10 ms, and 444.64 ms, respectively. Fig. 13 shows the performance by concurrently replaying the two traces at the edge servers. For a fair performance comparison, the cache capacity is increased to 2,000 chunks in this case. Due to the multiplexing gain, the average data access latencies of two traces decreased to 573.95 ms, 575.87 ms, 522.73 ms, 478.10 ms, 494.69 ms, 480.65 ms, and 439.51 ms, respectively.

5. Trace 1 is used in the above performance evaluation.

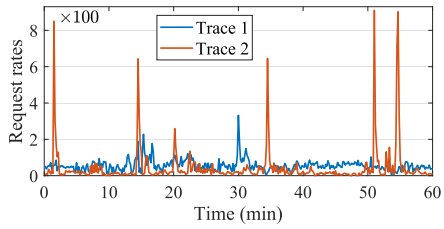


Fig. 11. Time-varying data request rates of two MSR Cambridge Traces [40].

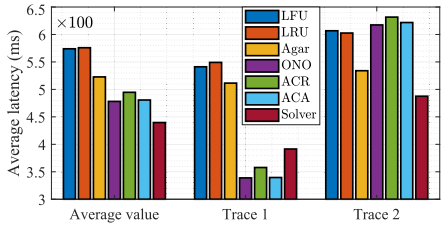


Fig. 13. Average data access latencies by concurrently replaying the two traces.

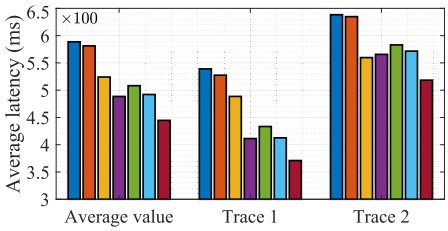


Fig. 12. Average data access latencies by separately replaying the two traces.

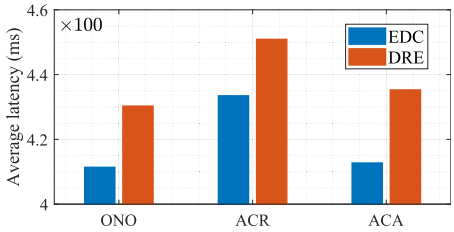


Fig. 14. Average data access latencies with different prediction methods.

As shown in Figs. 12 and 13, different caching schemes may work differently when concurrent applications compete for the limited cache capacity. For example, with the future data request rates known as the prior information, the offline Agar scheme assigns more cache capacity to Trace 2 for handling the bursty data requests. With concurrent caching services, the average latency of Trace 2 decreases from 559.74 ms to 534.01 ms while that of Trace 1 increases from 488.50 ms to 511.43 ms. The offline Solver scheme works similarly as Agar. In contrast, as an online scheme, ACA prefers to assign more cache capacity to the steadier trace. With concurrent caching services, the average latency of Trace 1 decreases from 412.64 ms to 339.57 ms while that of Trace 2 increases from 571.56 ms to 621.74 ms. Please note that the time-consuming Solver scheme incurs the lowest average latency of the two concurrent traces overall, although the online ONO, ACR, and ACA schemes incur lower latency than Solver for Trace 1.

Prediction Method. Beyond the EDC method [36] in Section 4.1, the Discounting Rate Estimator (DRE) [45] method is also introduced to construct the real-time request information for performance comparison. At the edge server, a counter is maintained for each data item, which increases with every data read request, and decreases periodically with a decay factor. The performance of the two prediction methods is compared when they are applied to three online caching schemes that need the real-time request information, i.e., ONO, ACR, and ACA. With EDC, the three caching schemes incur average data access latencies of 411.31 ms, 433.38 ms, and 412.64 ms, respectively. With DRE, the average data access latencies are 430.20 ms, 450.82 ms, and 435.19 ms, respectively. Fig. 14 indicates that the adopted EDC method can track the real-time request information more accurately for lower data access latency.

Summary and Insights. Overall, the evaluation results under various settings demonstrate that when compared with the caching schemes working at the data item level, the proposed ACR and ACA schemes can reduce the data access latency by up to 35.7%. More importantly, as shown in Table 4, the computation overheads of ACR and ACA are controlled well with the increasing scale of the storage system. Compared with the state-of-the-art ONO scheme that was designed for the coded storage system, our design can reduce the computation overhead by up to 84.57% while only incurring a performance loss of 1.12%. This indicates that ACR and ACA are scalable schemes, which could be applied to large-scale storage systems in the real world.

6 Conclusion and Future Work

In this paper, novel adaptive and efficient caching schemes were proposed to achieve low latency in the distributed cloud-edge storage system with erasure codes. Popular data chunks are cached at the edge servers near end users. As the distributed storage system spans multiple geographical sites, the measured end-to-end latencies were used to quantify the benefits of caching. Based on the measured data popularity and network latency information, an adaptive content replacement scheme with the approximation ratio of $\frac{C-K+1}{2C-K+1}$ was designed to determine which data chunks to cache. The computation complexity being sublinear to the size of the cache capacity ensured the scalability of our design. Guided by the theoretical analysis and outputs of the approximation scheme, we obtain the subset of cached contents which incurs the performance loss. Then, an adaptive content adjustment scheme was designed to update this subset for further performance improvements. Moreover, we also extended the proposed caching schemes to the case of storage server failure. Extensive experiments demonstrate the efficiency and effectiveness of our design.

In this paper, we considered the situations where data items are of the same or similar block size in the distributed coded storage system. In future work, we plan to investigate the caching problem with variable data item sizes. Considering the limited cache capacity of the edge servers, a size-aware cache admission scheme will be designed to ensure data items with small sizes have a higher priority to be admitted. This ensures more data items can enjoy the benefits of caching to achieve low latency.

REFERENCES

[1] IoT Growth Demands Rethink of Long-Term Storage Strategies, says IDC, 2020. [Online] Available: https://www.idc.com/getdoc.jsp?containerId=prAP46737220

[2] Amazon Simple Storage Service, 2020. [Online] Available: https://aws.amazon.com/s3/

[3] HDFS Architecture Guide, 2019. [Online] Available: https://hadoop.apache.org/

[4] Microsoft Azure, 2020. [Online] Available: https://azure.microsoft.com/en-us/

[5] K. Liu *et al*., "A learning-based data placement framework for low latency in data center networks," *IEEE Trans. Cloud Comput*., vol. 10, no. 1, pp. 146–157, Jan.–Mar. 2022.

[6] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang, "Latency reduction and load balancing in coded storage systems," in *Proc. ACM Symp. Cloud Comput*., 2017, pp. 365–377.

[7] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian, "EC-Store: Bridging the gap between storage and latency in distributed erasure coded systems," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst*., 2018, pp. 255–266.

[8] R. Halalai, P. Felber, A. Kermarrec, and F. Taïani, "Agar: A caching system for erasure-coded data," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst*., 2017, pp. 23–33.

[9] M. Uluyol, A. Huang, A. Goel, M. Chowdhury, and H.V. Madhyastha, "Near-optimal latency versus cost tradeoffs in geo-distributed storage," in *Proc. 17th USENIX Conf. Netw. Syst. Des. Implementation*, 2020, pp. 157–180.

[10] A. Singla, B. Chandrasekaran, P.B. Godfrey, and B. Maggs, "The Internet at the speed of light," in *Proc. 13th ACM Workshop Hot Topics Netw*., 2014, pp. 1–7.

[11] L.A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J*., vol. 5, no. 2, pp. 78–101, 1966.

[12] N. Atre, J. Sherry, W. Wang, and D.S. Berger, "Caching with delayed hits," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl., Technol., Architect.,Protoc. Comput. Commun*., 2020, pp. 495–513.

[13] S. ElAzzouni, F. Wu, N. Shroff, and E. Ekici, "Predictive caching at the wireless edge using near-zero caches," in *Proc. 21st Int. Symp. Theory, Algorithmic Found., Protoc. Des. Mobile Netw. Mobile Comput*., 2020, pp. 121–130.

[14] D.S. Berger, R. Sitaraman, and M. Harchol-Balter, "AdaptSize: Orchestrating the hot object memory cache in a CDN," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 483–498.

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

[16] X. Jin *et al*., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Oper. Syst. Princ*., 2017, pp. 121–136.

[17] Z. Liu *et al*., "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. 17th USENIX Conf. File Storage Technol*., 2019, pp. 143–157.

[18] Y. Xiang, T. Lan, V. Aggarwal, and Y.-F. Chen, "Optimizing differentiated latency in multi-tenant, erasure-coded storage," *IEEE Trans. Netw. Service Manag*., vol. 14, no. 1, pp. 204–216, Mar. 2017.

[19] V. Aggarwal, Y.F.R. Chen, T. Lan, and Y. Xiang, "Sprout: A functional caching approach to minimize service latency in erasure-coded storage," *IEEE/ACM Trans. Netw*., vol. 25, no. 6, pp. 3683–3694, Dec. 2017.

[20] K. Liu, J. Wang, J. Peng, and J. Pan, "Low-latency edge caching in distributed coded storage systems," *IEEE/ACM Trans. Netw*., to be published, doi: 10.1109/TNET.2021.3133215.

[21] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware, workload-adaptive SPARQL query caching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1777–1792.

[22] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl., Technol., Architect., Protoc. Comput. Commun*., 2013, pp. 231–242.

[23] A. Badita, P. Parag, and J.-F. Chamberland, "Latency analysis for distributed coded storage systems," *IEEE Trans. Inf. Theory*, vol. 65, no. 8, pp. 4683–4698, Aug. 2019.

[24] V. Aggarwal, J. Fan, and T. Lan, "Taming tail latency for erasure-coded, distributee storage systems," in *Proc. IEEE Conf. Comput. Commun*., 2017, pp. 1–9.

[25] A.O. Al-Abbasi and V. Aggarwal, "TTLCache: Taming latency in erasure-coded storage through TTL caching," *IEEE Trans. Netw. Service Manag*., vol. 17, no. 3, pp. 1582–1596, Sep. 2020.

[26] K.V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding," in *Proc. 12th USENIX Conf. Oper. Syst. Des. Implementation*, 2016, pp. 401–417.

[27] Cassandra, 2019. [Online] Available: http://cassandra.apache.org/

[28] C. Huang *et al*., "Erasure coding in Windows Azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf*., 2013, pp. 15–26.

[29] G. Ananthanarayanan *et al*., "PACMan: Coordinated memory caching for parallel jobs," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 267–280.

[30] B. Hu, Y. Chen, Z. Huang, N. A. Mehta, and J. Pan, "Intelligent caching algorithms in heterogeneous wireless networks with uncertainty," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst*., 2019, pp. 1549–1558.

[31] Memcached. 2019. [Online] Available: http://memcached.org/

[32] K.L. Bogdanov, W. Reda, G.Q. Maguire Jr, D. Kostić, and M. Canini, "Fast and accurate load balancing for geo-distributed storage systems," in *Proc. ACM Symp. Cloud Comput*., 2018, pp. 386–400.

[33] W. Wang, D. Niu, B. Liang, and B. Li, "Dynamic cloud instance acquisition via IaaS cloud brokerage," *IEEE Trans. Parallel Distrib. Syst*., vol. 26, no. 6, pp. 1580–1593, Jun. 2014.

[34] J.S. Hunter, "The exponentially weighted moving average," *J. Qual. Technol*., vol. 18, no. 4, pp. 203–210, 1986.

[35] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes, "Adaptive caching in big SQL using the HDFS cache," in *Proc. ACM Symp. Cloud Comput*., 2016, pp. 321–333.

[36] Z. Song, D.S. Berger, K. Li, and W. Lloyd, "Learning relaxed Belady for content distribution network caching," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2020, pp. 529–544.

[37] L. Tang, Q. Huang, A. Puntambekar, Y. Vigfusson, W. Lloyd, and K. Li, "Popularity prediction of Facebook videos for higher quality streaming," in *Proc. USENIX Conf. Annu. Tech. Conf*., 2017, pp. 111–123.

[38] A. Blankstein, S. Sen, and M.J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf*., 2017, pp. 499–511.

[39] zfec, 2013. [Online] Available: https://github.com/richardkiss/py-zfec

[40] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, 2008.

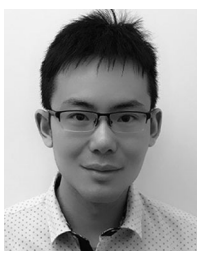
[41] G. Hasslinger, J. Heikkinen, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies," in *Proc. 16th Int. Symp. Model. Optim. Mobile, Ad Hoc, Wireless Netw*., 2018, pp. 1–6.

[42] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets*. Cambridge, U.K.: Cambridge Univ. Press, 2010.

[43] Y. Yu, J. Zhang, and K.B. Letaief, "Joint subcarrier and CPU time allocation for mobile edge computing," in *Proc. IEEE Glob. Commun. Conf*., 2016, pp. 1–6.

[44] O. Khan, R.C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. USENIX Conf. File Storage Technol*., 2012, pp. 20–33.

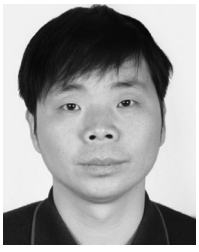
[45] M. Alizadeh *et al*., "Conga: Distributed congestion-aware load balancing for datacenters," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl., Technol., Architect., Protoc. Comput. Commun*., 2014, pp. 503–514.



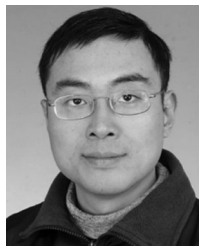
Kaiyang Liu (Member, IEEE) received the PhD degree from the School of Information Science and Engineering, Central South University, in 2019. During 2016–2018, he was a research assistant with the University of Victoria, Canada, with Prof. Jianping Pan. His current research interests include distributed cloud/edge computing and storage networks, data center networks, and distributed machine learning. One of his papers is one of three IEEE LCN 2018 Best Paper Award candidates. In 2020, he was awarded the IEEE Technical Committee on Cloud Computing (TCCLD) Outstanding PhD Thesis Award.

Jun Peng (Senior Member, IEEE) received the BS degree from Xiangtan University, Xiangtan, China, in 1987, the MSc degree from the National University of Defense Technology, Changsha, China, in 1990, and the PhD degree from Central South University, Changsha, China, in 2005. In April 1990, she joined Central South University. From 2006 to 2007, she was with the School of Electrical and Computer Science, University of Central Florida, USA, as a visiting scholar. She is a professor with the School of Computer Science and Engineering, Central South University, China. Her research interests include cooperative control, cloud computing, and wireless communications.

Jingrong Wang (Graduate Student Member, IEEE) received the bachelor's degree from the School of Electronic and Information Engineering, Beijing Jiaotong University, in 2017, and MSc degree in computer science from the University of Victoria, in 2019. She is currently working toward the PhD degree with the University of Toronto. Her research interests include wireless communications, mobile edge computing, and distributed machine learning.



Zhiwu Huang (Member, IEEE) received the BS degree in industrial automation from Xiangtan University, Xiangtan, China, in 1987, the MS degree in industrial automation from the Department of Automatic Control, University of Science and Technology Beijing, Beijing, China, in 1989, and the PhD degree in control theory and control engineering from Central South University, Changsha, China, in 2006. In October 1994, he joined Central South University. From 2008 to 2009, he was with the School of Computer Science and Electronic Engineering, University of Essex, U.K., as a visiting scholar. He is currently a professor with the School of Automation, Central South University, China. His research interests include fault diagnostic techniques and cooperative control.



Jianping Pan (Senior Member, IEEE) received the bachelor's and PhD degrees in computer science from Southeast University, Nanjing, Jiangsu, China, and he did his postdoctoral research with the University of Waterloo, Waterloo, Ontario, Canada. He is currently a professor of computer science with the University of Victoria, Victoria, British Columbia, Canada. He also worked with Fujitsu Labs and NTT Labs. His area of specialization is computer networks and distributed systems, and his current research interests include protocols for advanced networking, performance analysis of networked systems, and applied network security. He received the IEICE Best Paper Award, in 2009, the Telecommunications Advancement Foundation's Telesys Award, in 2010, the WCSP 2011 Best Paper Award, the IEEE Globecom 2011 Best Paper Award, the JSPS Invitation Fellowship, in 2012, the IEEE ICC 2013 Best Paper Award, and the NSERC DAS Award, in 2016, is a coauthor of one of three IEEE LCN 2018 Best Paper Award candidates, and has been serving on the technical program committees of major computer communications and networking conferences including IEEE INFOCOM, ICC, Globecom, WCNC and CCNC. He was the ad hoc and sensor networking symposium co-chair of IEEE Globecom 2012 and an associate editor of *IEEE Transactions on Vehicular Technology*. He is a senior member of the ACM.