Optimal Caching for Low Latency in Distributed Coded Storage Systems

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

***Abstract*— Erasure codes have been widely considered as a promising solution to enhance data reliability at low storage costs. However, in modern geo-distributed storage systems, erasure codes may incur high data access latency as they require data retrieval from multiple remote storage nodes. This hinders the extensive application of erasure codes to data-intensive applications. This paper proposes novel caching schemes to achieve low latency in distributed coded storage systems. Assuming that future data popularity and network latency information are available, an offline caching scheme is proposed to explore the optimal caching solution for low latency. The proposed scheme categorizes all feasible caching decisions into a set of cache partitions, and then obtains the optimal caching decision through market clearing price for each cache partition. Furthermore, guided by the optimal scheme, an online caching scheme is proposed according to the measured data popularity and network latency information in real time, without the need to completely override the existing caching decisions. Both theoretical analysis and experiment results demonstrate that the online scheme can approximate the offline optimal scheme well with dramatically reduced computation complexity.**

***Index Terms*— Distributed storage systems, erasure codes, optimal caching.**

I. Introduction

IN THE big data era, the world has witnessed the explosive growth of data-intensive applications. IDC predicts that the volume of global data will reach a staggering 175 Zettabytes by 2025 [1]. Modern distributed storage systems, e.g., Amazon Simple Storage Service (S3) [2], Google Cloud Storage [3], and Microsoft Azure [4], use two redundancy schemes, i.e., data replication and erasure codes, to enhance data reliability.

Manuscript received May 20, 2021; revised October 18, 2021; accepted November 29, 2021; approved by IEEE/ACM Transactions on Networking Editor S. Shakkottai. Date of publication December 20, 2021; date of current version June 16, 2022. 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.)*

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 V8W 2Y2, 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).

Jianping Pan is with the Department of Computer Science, University of Victoria, Victoria, BC V8W 2Y2, Canada (e-mail: pan@uvic.ca).

Digital Object Identifier 10.1109/TNET.2021.3133215

By creating full data copies at storage nodes near end users, data replication can reduce the data service latency with good fault tolerance performance [5], [6]. However, it suffers from high bandwidth and storage costs with the growing number of data replicas. With erasure codes, each data item is coded into data chunks and parity chunks. Compared with replication, erasure codes can lower the bandwidth and storage costs by an order of magnitude while with the same or better level of data reliability [7], [8]. 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 [9]. The slowest chunk retrieval dominates the data access latency. The high latency prevents the further application of erasure codes to data-intensive applications, limiting their use to rarely-accessed archive data [10].

As supplements to the geo-distributed storage system, major content providers, e.g., Akamai and Google, deploy frontend servers to achieve low latency [11]. End users issue requests to the nearest frontend servers, which have cached a pool of popular data items. Nevertheless, data caching faces critical challenges in the distributed coded storage system. As data items are coded into data chunks and parity chunks, the caching scheme should determine which chunks to cache for each data item. In fact, compared with caching the entire data item, caching a smaller number of chunks has more scheduling flexibility. To serve end users across the globe, the coded chunks of each data item are spread at more storage nodes to lower the latencies of geographically dispersed requests [11]. The latency of fetching different chunks varies as the coded chunks may be placed at geographically diverse nodes. Since the data request latency is determined by the slowest chunk retrieval, the data chunks with higher access latency are cached first. With a partial number of chunks cached, more data items can enjoy the benefits of caching for latency reduction considering the limited cache capacity of the frontend server. Traditional caching schemes at the data item level are not space efficient to achieve the lowest data access latency [13].

For a large-scale storage system, the goals of the caching schemes are 1) achieving the lowest data access latency, 2) highly efficient for a quick caching decision, and 3) flexible to change the caching decision in an online fashion. In this paper, we propose optimal offline and near-optimal online caching schemes that are specifically designed for distributed coded storage systems to achieve low latency. Preliminary experiment results in Sec. IV-A show that a positive correlation exists between the latency and the physical distance of data retrieval over the wide area network (WAN). For any two geographically diverse storage nodes, the latency gap of accessing the same data item keeps fairly stable. The average data access latency is used as the performance metric to quantify the benefits of caching. To explore the lowest data access latency, the offline scheme categorizes all feasible caching decisions into a set of cache partitions based on the number of cached chunks for each data item. Then, by applying the market clearing price on all cache partitions, the optimal data item assignment for cache partitions (or equivalently, the optimal caching decision), can be obtained with theoretical guarantees.

Although theoretically sound in design, the optimal scheme faces the challenges of long running time and large computation overheads when applied to a large-scale storage system. The optimal scheme motivates the design of a practical online caching scheme with low computation complexity. Based on the measured data popularity and network latencies in real time, the caching decision is updated upon the arrival of each request, without completely overriding the existing caching decisions. The theoretical analysis provides the worst-case performance guarantees of the online scheme. The main contributions in this paper include:

- A novel optimal caching scheme based on cache partitions and market clearing price is designed with performance guarantees, which can serve as the upper bound of performance gain on latency reduction. To the best of our knowledge, this is the first work that explores the optimal caching solution to achieve low latency in the distributed coded storage system.
- Guided by the optimal scheme, a near-optimal online caching scheme is proposed to reduce the computation complexity without sacrificing the performance of low data access latency.
- The proposed caching schemes are extended to the case of storage server failure.
- A prototype of the caching system is built based on Amazon S3. Extensive experiment results show that when compared with the optimal scheme, the online scheme only incurs a performance loss of about 2% while with greatly reduced computation overheads.

The rest of this paper is organized as follows. Sec. II summarizes the related work. Sec. III presents the model of the distributed coded storage system and states the caching problem. Sec. IV and V provide the design of the optimal and online caching schemes, respectively. Sec. VI evaluates the efficiency and performance of the caching schemes through extensive experiments. Sec. VII concludes this paper and lists future work.

II. Related Work

Caching at the data item level. Data caching has been considered as a promising solution to achieve low latency in distributed storage systems. Ma *et al.* [20] proposed a replacement scheme that cached a full copy of data items to reduce the storage and bandwidth costs while maintaining low latencies. Liu *et al.* [26] designed DistCache, a distributed caching scheme with provable load balancing performance for distributed storage systems. Song *et al.* [14] proposed a learning-based scheme to approximate Belady's optimal replacement algorithm with high efficiency [15]. However, all these previous studies focus on caching at the data item level. Due to the limited cache capacity, keeping a full copy of data items may not be space efficient to achieve the lowest data access latency with erasure codes.

Low latency in coded storage systems. 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 [13]. Therefore, recent studies have attempted to analyze the latency bounds based on queuing theory [13], [16]–[18]. These researches are under the assumption of stable request arrival process and exponential service time distribution, which may not be applicable to a dynamic network scenario.

Prior work also focused on the design of data placement and access schemes to achieve load balancing in coded storage systems [7], [8], [19]. In this way, the data access latency could be reduced by the avoidance of data access collision. These scheduling schemes are designed for intra-data center storage systems as the network congestion dominates the overall data access latency. Instead, we address the problem of optimizing the data access latency by considering caching on the frontend 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 end users over WAN, which is also different from traditional caching schemes in cellular networks [21] or cooperative caching in peer-to-peer networks [22].

Caching in coded storage systems. Compared with schemes that cache the entire data items, Aggarwal *et al.* [13] pointed out that caching partial data chunks had more scheduling flexibility. Then, a caching scheme based on augmenting erasure codes was designed to reduce the data access latency. Nevertheless, extra storage overheads were introduced with the augmented scheme. Halalai *et al.* [9] 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 period of time without any worst-case performance guarantees. Rashmi *et al.* [12] applied an online erasure coding scheme to data stored in the caching layer for latency reduction. However, this online erasure coding scheme introduced extra computation overheads when handling massive data requests. Different from previous studies, our proposed caching schemes leverage the measured end-to-end latency to quantify the benefits of caching. Furthermore, the proposed schemes only cache data chunks rather than parity chunks to avoid the decoding overheads of data requests.

III. System Model and Problem Statement

This section presents the architecture of the geo-distributed storage system with erasure codes and discusses how to reduce the access latency of data requests with caching services.

A. Geo-Distributed Storage System and Erasure Codes

As shown in Fig. 1, the geo-distributed storage system consists of a set of geographically dispersed storage nodes $\mathcal{N}$ (with size $N = |\mathcal{N}|$).[1] The set of data items stored in the system is denoted by $\mathcal{M}$ (with size $M = |\mathcal{M}|$). Similar to Hadoop [23] and Cassandra [24], all data items have a default block size. To achieve the target reliability with the maximum storage efficiency, the Reed-Solomon (RS) codes are adopted as the storage scheme.[2] With a linear mapping process, each

[1]The storage nodes could be data centers in practice. As shown in Fig. 1, each storage node consists of multiple storage servers.

[2]Other erasure codes, e.g., local reconstruction codes (LRC) [25], can also be applied in our solution.

TABLE I

THE DEPLOYMENT OF STORAGE NODES OVER SIX AMAZON WEB SERVICES (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

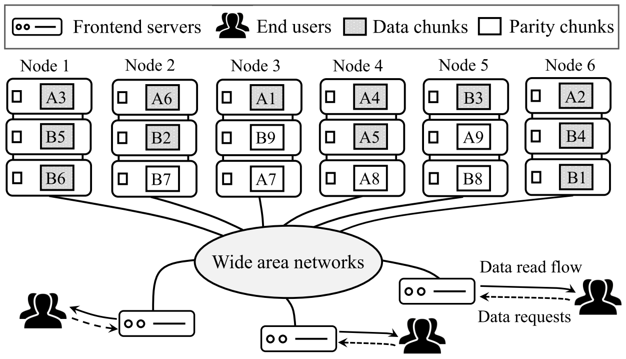


Fig. 1. An illustration of the distributed coded storage system with caching services is shown. Data items A and B are coded into $K = 6$ data chunks and $R = 3$ parity chunks.

data item is coded into equal-sized K data chunks and R parity chunks. All coded chunks are distributed among storage nodes for fault tolerance, which 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 represents chunk m_k and m_r are placed at node i and j, respectively.[3] Please note that the coded chunks are not placed at a single storage node since this will increase the data access latency of end users far from that node.

If the requested data is temporarily unavailable, the original data can be recovered by the decoding process from any K out of $K + R$ chunks. The decoding process with parity chunks will inherently incur considerable computation overheads to the storage system. Generally speaking, a read request is first served by obtaining K data chunks to reconstruct the original data with low overheads [7]. The action of parity chunk retrieval for decoding is defined as **degraded read**. The degraded read is passively triggered 1) when the storage server storing the data chunks is momentarily unavailable, or 2) during the recovery of server failure. Moreover, the data write/update process is not considered in this paper, since most storage systems are append-only where all data items are immutable. Instead, data with any updates will be considered as separate items with new timestamps [7].

Erasure codes may incur high data access latency in the geo-distributed storage system. The requested chunks are retrieved by accessing multiple remote storage nodes. The high latency impedes the extensive application of erasure codes to data-intensive applications. Therefore, it is imperative to reduce the data request latencies in the coded storage system.

[3]In this paper, data replication at the remote storage nodes is not considered to achieve low storage overheads. Our design is also applicable to the scenario of data replication. The data request is fulfilled by retrieving K data chunks from the closest storage nodes.

B. Caching at Frontend Servers for Low Latency

This paper adopts caching to achieve low latency data services. As illustrated in Fig. 1, multiple frontend servers are deployed to serve geographically dispersed end users. Each frontend server creates an in-memory caching layer to cache popular data items near end users. Instead of interacting directly with remote storage nodes, end users retrieve data from the frontend server. Let C denote the cache capacity of the frontend server. Due to the scarcity of memory, not all data chunks can be cached in the caching layer, i.e., $C \leq MK$.

With erasure codes, we may not need to cache all chunks of each data item to achieve the lowest data access latency. This can be demonstrated through preliminary experiments based on Amazon S3. As shown in Fig. 1, a prototype of the coded storage system is deployed over $N = 6$ Amazon Web Services (AWS) regions, i.e., Tokyo, Ohio, Ireland, São Paulo, Oregon, and Northern California. In each AWS region, three `buckets` are created. Each `bucket` represents a server for remote data storage. The storage system is populated with $M = 1,000$ data items. The numbers of data and parity chunks are set as $K = 6$ and $R = 3$. The default size of all chunks is 1 MB [9]. For each data item, the nine coded chunks are uniformly distributed among eighteen `buckets` to achieve load balancing. The coded chunks of each data item are not placed on the same server to guarantee the R-fault tolerance. As noted in prior work [7], [12], the popularity of data items follows a Zipf distribution.

Furthermore, three frontend servers are deployed near the end users at various locations, i.e., Victoria, Canada, San Francisco, United States, and Toronto, Canada. `Memcached` [27] module is adopted for data caching in RAM. The frontend server uses a thread pool to request data chunks in parallel. For each read request, the end user needs to obtain all six data chunks from remote `buckets` without caching on the frontend server. Table I shows the average data access latency from remote `buckets` to end users.[4] 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 the frontend server in Victoria, the latency is about 479.3 ms as we need to fetch data chunk B5 and B6 from the farthest storage node in Tokyo.

Then, let us consider the performance of latency reduction with caching services on the frontend server. In our design, only data chunks will be cached to avoid degraded read. Let L_i denote the average network latency of data access from storage node i to end users. According to the storage location

[4]The data access latency includes the network latency from remote storage nodes to the frontend server, the data reconstruction latency, and the network latency from the frontend server to end users. The frontend servers could be deployed in close proximity to end users with low data access latency [28]. Compared with the high network latency over WAN (in hundreds of milliseconds), the reconstruction latency with data chunks and the network latency from the frontend server to end users are negligible.

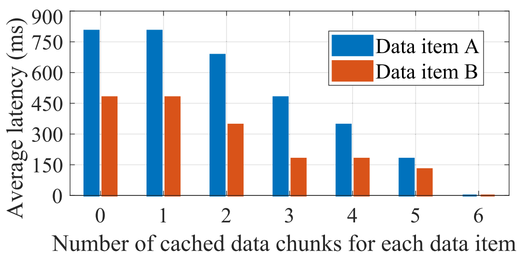


Fig. 2. Experiment results show the average access latency of caching different numbers of data chunks on the frontend server in Victoria. The relationship between the number of cached data chunks and the reduced latency is nonlinear. The storage locations of data items A and B are shown in Fig. 1.

information in (1), the average latency of sending data chunk m_k is given by

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

where $\mathbf{1}(m_k \rightarrow i)$ indicates whether data chunk m_k is placed at node i or not, returning 1 if true or 0 otherwise, $k \in \{1, \ldots, K\}$, $i \in \mathcal{N}$. For ease of notation, data chunks are re-labeled in the descending order of the data chunk access latency, i.e., $l_{m_1} \geq \ldots \geq l_{m_K}$. For data item m, data chunk with higher access latency is cached first. Then, the data access latency can be progressively decreased. Fig. 2 shows the average access latency of caching different numbers of data chunks on the frontend server in Victoria. Let λ_m denote the number of cached data chunks for data item m, $\lambda_m \in \{0, \ldots, K\}$, $m \in \mathcal{M}$. The discrete latency function can be defined as follows:

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

We have the following two observations:

- The access latency function $f_m(\lambda_m)$ is nonlinear.
- 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 $f_m(\lambda_m)$ could also be different due to the diverse storage locations. For instance, if three chunks are cached for data item A, the data access latency is reduced by 40.3%. For data item B, three cached data chunks can reduce the latency by 62.6%.

The observations show that the latency reduction varies from data item to data item. Traditional caching schemes at the data item level are not space efficient to achieve low latency.

C. Caching Problem Statement

To minimize the overall latency of data read requests on a frontend server, the number of cached chunks λ_m for each data item should be optimized as follows[5]:

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

[5]For data item m, λ_m data chunks placed at the farthest storage nodes, i.e., with the highest data access latencies, will be cached.

where r_m denotes the user request rate for data item m. Constraint $0 \leq \lambda_m \leq K$ ensures that the number of cached chunks for each data is no larger than the number of coded data chunks. Furthermore, $\sum_{m \in \mathcal{M}} \lambda_m \leq C$ ensures that the cache capacity constraint is not violated. Then, the hardness of the optimization problem (4) is examined as follows:

- Experiments demonstrate that $f_m(\lambda_m)$, e.g., the latency function of data item B in Fig. 2, could be both nonlinear and non-convex. Therefore, problem (4) is an integer programming problem with non-convexity and nonlinearity. At the fundamental level, the difficulty of solving non-convex nonlinear programming lies in the fact that the local optimal solutions may not necessarily be the global optimal solution. Relaxing the integer constraint $\lambda_m \in \mathbb{N}$ to continuous constraint $\lambda_m \in \mathbb{R}$, and then rounding the obtained results to the nearest integers is not a simple or always feasible solution. Generally speaking, complex combinatorial techniques are needed for an efficient solution [29].
- In a dynamic scenario, the network conditions and the data request rates may be time variant. It is a challenge to design online schemes that can react quickly to real-time changes.

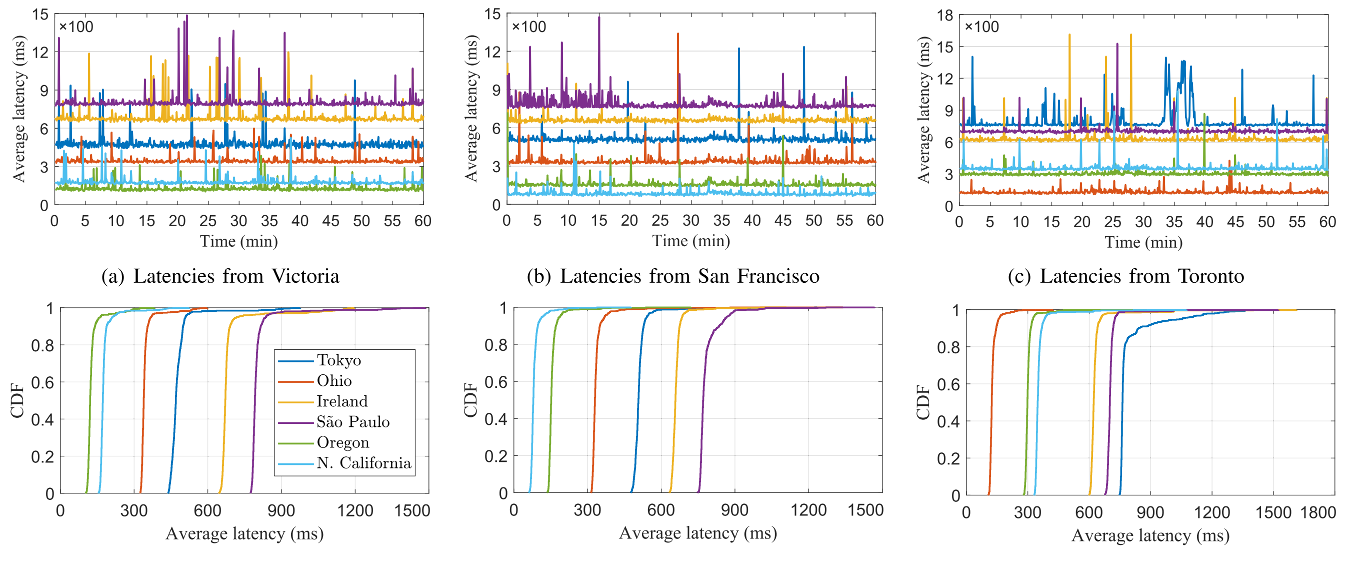
IV. Optimal Caching Schemes Design

In this section, the motivation and design overview are presented. Then, assuming that future data popularity and network condition information is available, an offline scheme is designed to find the optimal caching solution for low data access latency.

A. Motivation and Design Overview

As mentioned in Sec. III-B, the latency function $f_m(\lambda_m)$ can be different for various data items. Considering the diversity of chunk storage locations, it is complicated to design a mathematical latency model that is suitable for the entire storage system. Therefore, the end-to-end latency of data access is used to quantify the benefits of caching. Through experiments, we analyze the characteristics of data access latencies over WAN. Based on the deployed experiment platform, the access latencies of data chunk retrieval from remote `buckets` to end users are measured over an hour (from 15:00 to 16:00 on July 30, 2020). Fig. 3(a), (b), and (c) show that the data access latencies over WAN remain fairly stable in the long term as the propagation delay dominates and depends primarily on the physical distance of data transmission [30]. Experiment results confirm the positive correlation between physical distance and latency. For instance, the data access latency from São Paulo to end users in Victoria is always higher than that from Oregon.

Fig. 3(d), (e), and (f) demonstrate that the data access latencies are stable for most of the service time. For example, 89.58% of the data access latencies from Oregon to Victoria will be in the range of [100, 140] ms. Besides, 91.94% of the data access latencies from Ireland to Victoria will be in the range of [650, 700] ms. For two arbitrary storage nodes, the latency gap also keeps fairly stable in the long term. The average data access latency can be used to quantify the benefits of caching. In Sec. IV-B, assuming that future data popularity and network condition information are available, an offline scheme is designed to explore the optimal caching solution on latency reduction.



(a) Latencies from Victoria (b) Latencies from San Francisco (c) Latencies from Toronto

(d) CDF of latencies from Victoria (e) CDF of latencies from San Francisco (f) CDF of latencies from Toronto

Fig. 3. Experiment results show the data access latencies from storage nodes to end users over a period of one hour (from 15:00 to 16:00 on July 30, 2020).

B. Optimal Caching Scheme

Based on the latencies of data chunk access $\{l_{m_1}, \dots, l_{m_K}\}$, a $(K+1)$-dimensional array is maintained for each data item

$$\begin{aligned}\{\tau_{m,0}, \tau_{m,1}, \dots, \tau_{m,K}\}\\ = \{0, (l_{m_1} - l_{m_2}) \cdot r_m, \dots, (l_{m_1}\\ -l_{m_k}) \cdot r_m, \dots, (l_{m_1} - l_{m_K}) \cdot r_m, l_{m_1} \cdot r_m\}, \quad (5)\end{aligned}$$

where $\tau_{m,k-1} = (l_{m_1} - l_{m_k}) \cdot r_m$ represents the value of reduced latency when $k-1$ data chunks are cached. For example, if chunk m_1 and m_2 are cached, then l_{m_3} becomes the bottleneck. Clearly, $\tau_{m,0} = 0$ as no latency will be reduced without caching. When all K data chunks are cached, the maximum value of reduced latency is $\tau_{m,K} = l_{m_1} \cdot r_m$. Based on the reduced latency information, an $M \times (K+1)$ valuation array τ can be maintained for all data items. As shown in Fig. 2, $f_m(\lambda_m)$ is a monotonic decreasing function. Minimizing the overall data access latency in (4) is equivalent to maximizing the total amount of reduced latency:

$$\begin{aligned}\max_{\lambda_m \in \mathbb{N}, m \in \mathcal{M}} \Theta(\lambda_m) = \sum_{m \in \mathcal{M}} \tau_{m,\lambda_m}\\ \text{s.t.} \quad 0 \le \lambda_m \le K,\\ \sum_{m \in \mathcal{M}} \lambda_m = C. \quad (6)\end{aligned}$$

As $C \le MK$, $\sum_{m \in \mathcal{M}} \lambda_m = C$ ensures the cache capacity can be fully utilized for latency reduction. Problem (6) is hard to be solved. Intuitively, the optimal solution can be obtained through exhaustive search. However, for the $M \times (K+1)$ valuation array τ, the exhaustive search evaluates every possible caching solution with an unacceptable computation complexity of $O(2^{M \cdot (K+1)})$. To improve the solution efficiency without sacrificing the performance of low data access latency, we design a novel scheme based on cache partitions and market clearing price to find the optimal caching decision.

Algorithm 1 Iterative Search for Cache Partitions

```
Input: Cache capacity $C$, number of coded data chunks $K$,
    number of data items $M$.
Output: Set of cache partitions $\chi$.
Initialization: $x_1 \leftarrow C$, $\forall x_k \leftarrow 0$, $k \in \{2, \dots, K\}$, $\chi \leftarrow \emptyset$.
 1: while $\{x_1, \dots, x_K\} \notin \chi$ do
 2:   Add $\{x_1, \dots, x_K\}$ to $\chi$ if $\sum_{k=1}^{K} x_k \le M$;
 3:   $x_2 \leftarrow x_2 + 1$ if $x_2 < \hat{x}_2$ else $x_2 \leftarrow 0$;
 4:   for $k = 3$ to $K$ do
 5:     if $x_{k-1}$ is reset to $0$ then
 6:       $x_k \leftarrow x_k + 1$ if $x_k < \hat{x}_k$ else $x_k \leftarrow 0$;
 7:     end if
 8:     $x_1 = C - \sum_{k=2}^{K} k x_k$;
 9:   end for
10: end while
```

1) Cache Partitions: Let x_k denote the number of data items with k data chunks cached, i.e.,

$$x_k = \sum\nolimits_{m \in \mathcal{M}} \mathbf{1}(\lambda_m = k), \quad (7)$$

where $\mathbf{1}(\lambda_m = k)$ indicates whether k data chunks are cached for data item m or not. We define $\{x_1, x_2, \dots, x_K\}$ as a potential partition of caching decisions, $\forall x_k \in \mathbb{N}$. Based on the constraints in (6), we have the Diophantine equations as follows

$$\begin{cases} x_1 + 2x_2 + \dots + Kx_K = C, \\ x_1 + x_2 + \dots + x_K \le M. \end{cases} \quad (8)$$

All partitions of caching decisions χ can be derived from (8) through iterative search. The pseudo code of the iterative search is listed in Algorithm 1. Given the value of $\{x_{k+1}, \dots, x_K\}$, the maximum value that x_k can be assigned is

$$\hat{x}_k = \left\lfloor \frac{C - \sum_{n=k+1}^{K} n x_n}{k} \right\rfloor. \quad (9)$$

Initially, $\{x_1, x_2, \ldots, x_K\} = \{C, 0, \ldots, 0\}$ is a feasible solution if $\sum_{k=1}^{K} x_k = C \leq M$. We gradually increase the value of x_2 from 0. If $x_2 = \hat{x}_2$, x_2 is reset to 0 in the next step. In this way, the value of x_k can be iteratively determined, $k \in \{3, \ldots, K\}$. If $x_{k-1} = 0$, x_k is incremented by 1 next. If $x_k = \hat{x}_k$, x_k is also reset to 0 then. Based on the value of $\{x_2, \ldots, x_K\}$, x_1 is set to $C - \sum_{k=2}^{K} kx_k$. We repeat the above process until all cache partitions are included in χ. A simple example is used to demonstrate the iterative search process. Let $K = 3$ and $C = 5$. Algorithm 1 sequentially appends five cache partitions, i.e., $\{5, 0, 0\}$, $\{3, 1, 0\}$, $\{1, 2, 0\}$, $\{2, 0, 1\}$, and $\{0, 1, 1\}$. The theoretical analysis of the iterative search algorithm is provided as follows.

Proposition 1: The size of set χ is no larger than $\prod_{k=2}^{K}(\lfloor \frac{C}{k} \rfloor + 1)$.

Proof: According to (8), as $\forall x_k \in \mathbb{N}$, the maximum value of x_k is $\lfloor \frac{C}{k} \rfloor$. Therefore, x_k can be assigned up to $\lfloor \frac{C}{k} \rfloor + 1$ different values. As $\{x_2, \ldots, x_K\}$ cannot be assigned to the maximum values at the same time, the while loop in Algorithm 1 is performed for no more than $\prod_{k=2}^{K}(\lfloor \frac{C}{k} \rfloor + 1)$ times to generate a complete set of cache partitions χ. The set size $|\chi| \leq \prod_{k=2}^{K}(\lfloor \frac{C}{k} \rfloor + 1)$ also holds. □

Algorithm 2 Optimal Assignment for Cache Partitions

```
Input: Set of cache partitions $\chi$, valuation array $\tau$, market clearing price $p_m$.
Output: Caching decision $\lambda_m$.
Initialization: $\forall \lambda_m, \hat{\lambda}_m, p_m \leftarrow 0$.
1:  for Cache partition $\{x_1, \ldots, x_K\} \in \chi$ do
2:    $\mathcal{G} \leftarrow$ preferred_seller_graph($\tau, \{x_1, \ldots, x_K\}$);
3:    $\{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\} \leftarrow$ constricted_set($\mathcal{G}$);
4:    $\tau' \leftarrow \tau$;
5:    while $\{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\} \neq \emptyset$ do
6:      for $m \in \mathcal{M}^{[c]}$ do
7:        for $k \in \mathcal{K}^{[c]}$ do
8:          $V_k \leftarrow$ sum_top($\tau'(:, k), x_k$);
9:          $V_k^m \leftarrow$ sum_top($\tau'(:, k) \setminus \{\tau'_{m,k}\}, x_k$);
10:       end for
11:       $p_m \leftarrow p_m + \max\{1, \max\{V_k - V_k^m\}\}$;
12:       $\tau'(m, :) \leftarrow \tau(m, :) - p_m$;
13:     end for
14:     $\mathcal{G} \leftarrow$ preferred_seller_graph($\tau', \{x_1, \ldots, x_K\}$);
15:     $\{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\} \leftarrow$ constricted_set($\mathcal{G}$);
16:   end while
17:   $\hat{\lambda}_m \leftarrow k$ according to $\mathcal{G}$;
18:   $\forall \lambda_m \leftarrow \hat{\lambda}_m$ if $\sum_{m \in \mathcal{M}} \tau_{m,\lambda_m} < \sum_{m \in \mathcal{M}} \tau_{m,\hat{\lambda}_m}$;
19: end for
```

2) Optimal Assignment for Cache Partitions: Recall that each element x_k in a cache partition $\{x_1, \ldots, x_K\}$ represents that x_k data items are selected, each of which cached k chunks on the frontend server. For example, if data item m is assigned to x_k, the caching decision for m becomes $\lambda_m = k$. As shown in Fig. 4, the data items and cache partition can be treated as **sellers** and **buyers**, respectively. According to the valuation array τ, each buyer has a valuation for each data item. Thus, the optimal assignment can be considered as a market competing for data items with higher valuations. The pseudo code of the optimal assignment is listed in Algorithm 2.

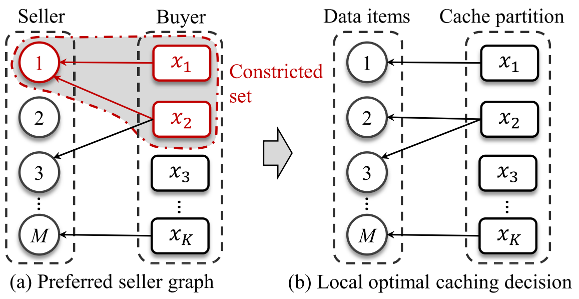


Fig. 4. An illustration of the data item assignment for data partition is shown.

As shown in Fig. 4, buyers may compete for a certain data item. The basic idea is to increase the price p_m of data item m until the competition is over. The price p_m is known as **market clearing price** [31]. With no competition, the local optimal caching decision $\hat{\lambda}_m$ can be obtained for a certain cache partition. The global optimal assignment is the one that has the maximum valuation among all cache partitions in χ.

Let $\tau(:, k)$ denote the k-th column of τ, which represents the reduced latencies of all data items when k data chunks of that data item are cached. To maximize the caching benefits, function `preferred_seller_graph` matches sellers and buyers with the largest x_k elements in $\tau(:, k)$. As shown in Fig. 4(a), a preferred seller graph $\mathcal{G}$ is constructed with function `preferred_seller_graph`. In $\mathcal{G}$, different buyers may compete for the same data item, while each data item can only be assigned to one buyer. Then, the constricted set $\{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\}$ is constructed with function `constricted_set`, where $\mathcal{M}^{[c]}$ denotes the set of competed data, and $\mathcal{K}^{[c]}$ represents the set of competing buyers. Then, we show how to set the market clearing price p_m for each data $m \in \mathcal{M}^{[c]}$. We initialize $p_m = 0$, $\forall m \in \mathcal{M}$. Then, the payoff array τ' can be initialized as the valuation array τ with $\tau' \leftarrow \tau$. Let V_k denote the total payoff of assigning data items (including the competed data item m) to buyer $x_k \in \mathcal{K}^{[c]}$, i.e.,

$$V_k = \texttt{sum\_top}(\tau'(:, k), x_k), \tag{10}$$

where function $\texttt{sum\_top}(\tau'(:, k), x_k)$ represents the sum of largest x_k elements in set $\tau'(:, k)$. Then, if data item m is not assigned to x_k, the total payoff is given by

$$V_k^m = \texttt{sum\_top}(\tau'(:, k) \setminus \{\tau'_{m,k}\}, x_k). \tag{11}$$

If $\max\{V_k - V_k^m\} > 0$ for all buyers in $\mathcal{K}^{[c]}$, p_m is incremented by $\max\{V_k - V_k^m\}$. If $\max\{V_k - V_k^m\} = 0$, p_m is incremented by the unit price 1.

$$p_m += \max\{1, \max\{V_k - V_k^m\}\}. \tag{12}$$

Remark 1: In [31], when multiple buyers are competing for a seller m, its price p_m is always increased by one unit in each round of pricing. Multiple buyers may still compete for m with the increased price. In contrast, the proposed pricing method in (12) ensures that m will be assigned to only one buyer $k = \arg\max\{V_k - V_k^m\}$ in each round of pricing, which improves the solution efficiency.

Then, the payoff of m for all buyers $\tau'(m, :)$ is updated as $\tau(m, :) - p_m$. The above process is repeated until the constricted set is empty. Then, the updated preferred seller graph with no competition is added to the existing assignment. If the local caching decision $\hat{\lambda}_m$ for the current cache partition yields a higher payoff than all previous ones, the global caching decision is updated with $\lambda_m \leftarrow \hat{\lambda}_m$. The theoretical analysis of the assignment algorithm is provided as follows.

Theorem 1: The pricing of data items in Algorithm 2 must come to an end with an empty constricted set, i.e., buyers will not compete for the same seller.

Proof: In Algorithm 2, the while loop (Line 5 – 16) ends if the constricted set is empty with a set of market-clearing prices; otherwise, the while loop continues with increased prices $p_m \geq 0$, $m \in \mathcal{M}$.

However, the prices cannot be increased forever without stop. This is because the "potential energy" which drives the pricing process is draining out as the algorithm runs. The potential of a seller is defined as the current price p_m she is charging. For any current set of prices, the potential of a buyer is the maximum payoff she can currently get from any seller, i.e., V_k in (10). Please note that these are the potential payoffs. The buyer and seller will actually get the payoffs if the constricted set is empty with no competition. Finally, the overall potential energy P is defined as the sum of the potential of all buyers and sellers.

$$P = \sum\nolimits_{k=1}^{K} V_k + \sum\nolimits_{m=1}^{M} p_m \tag{13}$$

Then, we prove that P has the following two properties:

1) The overall potential energy is never less than 0, i.e., $P \geq 0$. Let $\{m, x_k\}$ be an arbitrary assignment in the preferred seller graph $\mathcal{G}$. This means that seller m is assigned to buyer x_k to maximize the total amount of reduced latency. Then, we have the following two different cases:

- If $\{m, x_k\} \notin \{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\}$, no other buyers are competing for seller m. For $\{m, x_k\}$, the potential energy of buyer and seller is
$$\tau'_{m,k} + p_m = \tau_{m,k} \geq 0. \tag{14}$$
This means $P \geq 0$ holds.
- If $\{m, x_k\} \in \{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\}$, other buyers are competing for seller m. Let $x_{k'}$ be an arbitrary competitor, $\{m, x_{k'}\} \in \{\mathcal{M}^{[c]}, \mathcal{K}^{[c]}\}$.
$$\tau' = \begin{bmatrix} & & \vdots & & \\ \cdots & \boxed{\tau_{m,k} - p_m} & \cdots & \boxed{\tau_{m,k'} - p_m} & \cdots \\ & & \vdots & & \\ \cdots & \tau_{n,k} - p_n & \cdots & \tau_{n,k'} - p_n & \cdots \\ & & \vdots & & \end{bmatrix} \tag{15}$$
As shown in (15), for an unassigned data item n which is not in the preferred seller graph $\mathcal{G}$, we have
$$\tau_{m,k'} - p_m \geq \tau_{n,k'} - p_n. \tag{16}$$
For $\{m, x_k\}$, the potential energy is $\tau'_{m,k} + p_m = \tau_{m,k} \geq 0$. For $x_{k'}$ and n, the potential energy is
$$\tau'_{m,k} + p_n = \tau_{m,k'} - p_m + p_n \geq \tau_{n,k'} \geq 0. \tag{17}$$
As $x_1 + x_2 + \ldots + x_K \leq M$, we can always find a unique unassigned data item for each competitor with the potential energy no less than 0. For all other unassigned data items, we have $p_n \geq 0$, $n \in \mathcal{M}$. Then, based on (14) and (17), the overall potential energy $P \geq 0$ also holds.

2) The overall potential energy P decreases when the prices increase. Initially, we set $p_m = 0$, $\forall m \in \mathcal{M}$. It begins with all sellers having potential 0, and all buyers having a potential equal to the maximum valuation P_0.

$$P_0 = \sum\nolimits_{k=1}^{K} \texttt{sum\_top}(\tau(:,k), x_k) \tag{18}$$

Then, when the seller in the constricted set raises the price by at least one unit, the potential of the seller goes up by at least one unit. At the same time, the potential of each buyer in the constricted set goes down by at least one unit. Since more buyers are competing for the seller, the overall potential energy also goes down by at least one unit.

To sum up, the pricing scheme starts at a certain potential energy P_0, and it cannot drop below 0. So the pricing of data items must come to an end with P_0 steps at most. The constricted set is empty with a set of market-clearing prices. □

Theorem 2: Algorithm 2 yields the optimal caching decision on latency reduction.

Proof: Firstly, we prove that the optimal decision can be obtained for each cache partition. This is equivalent to proving that interchanging any two pairs of caching decisions cannot further increase the total valuations. Let m and m' denote two randomly selected data items in $\mathcal{G}$. With Algorithm 2, let k and k' denote their corresponding number of cached chunks. To verify optimality, we need to prove

$$\tau_{m,k} + \tau_{m',k'} \geq \tau_{m',k} + \tau_{m,k'}. \tag{19}$$

If k and k' are not in the constricted set, i.e., $\tau_{m,k} \geq \tau_{m',k}$ and $\tau_{m',k'} \geq \tau_{m,k'}$, we have $\tau_{m,k} + \tau_{m',k'} \geq \tau_{m',k} + \tau_{m,k'}$. If k and k' are in the constricted set of m in the previous while loop[6] and m is finally assigned to k, we have

$$V_k - V_k^m \geq V_{k'} - V_{k'}^m. \tag{20}$$

Besides, m' is finally assigned to k' with no competition, i.e., $\tau_{m',k}$ is not one of the largest x_k elements in set $\tau'(:,k)$. We have

$$\begin{cases} V_k - V_k^m \leq \tau_{m,k} - \tau_{m',k}, \\ V_{k'} - V_{k'}^m = \tau_{m,k'} - \tau_{m',k'}. \end{cases} \tag{21}$$

This means

$$\tau_{m,k} - \tau_{m',k} \geq \tau_{m,k'} - \tau_{m',k'}, \tag{22}$$

which concludes that the optimal caching decision is obtained for the cache partition. As all partitions in χ are considered, Algorithm 2 yields the global optimal caching decision. □

Proposition 2: The computation complexity of Algorithm 2 is no larger than $O(MKP_0 \prod_{k=2}^{K}(\lfloor \frac{C}{k} \rfloor + 1))$.

Proof: To obtain the preferred seller graph, all columns in τ are sorted via the radix sort algorithm. The sorting complexity is $O(MK)$ (Line 2). Then, all data items need to be considered to determine the constricted set with the complexity of $O(M)$ (Line 3). As all buyers may compete for a data item, the calculation of the market clearing price needs K steps at most (Line 6–12). Furthermore, the preferred seller graph and the constricted set are updated with the complexity of $O(M + MK)$ (Line 14–15). As discussed above, the while loop is performed for P_0 times at most. Furthermore, the data item assignment in Line 17 and 18 needs K steps at most. The optimal assignment for a cache partition needs $(MK + M + K)P_0 + MK + M + K$ steps at most. The computation complexity for a cache partition is $O(MKP_0)$. Considering all cache partitions in χ, the

[6]The case of data item m' can be proved in the same way.

computation complexity of Algorithm 2 is no larger than $O(MKP_0\prod_{k=2}^{K}(\lfloor\frac{C}{k}\rfloor+1))$. □

The computation complexity of Algorithm 2 is mainly determined by the total number of cache partitions $|\chi|$. Based on the experiment platform deployed in Sec. III-B, Table III in Sec. VI-B shows the number of cache partitions $|\chi|$ and the average running time (ART) of Algorithm 2 under different settings. The number of required iterations $|\chi|$ and the ART increase rapidly with the increase of cache capacity C and the number of coded data chunks K. This means that Algorithm 2 may incur a heavy computation burden for a large-scale storage system. Furthermore, the long running time implies that the optimal scheme cannot react quickly to real-time network changes. The network states may change before caching decisions can be updated. To sum up, the optimal scheme is an offline solution with the requirement of future data popularity and network condition information.

V. Online Caching Scheme Design

Guided by the optimal caching scheme in Sec. IV-B, an online caching scheme is proposed with no assumption about future data popularity and network condition information. Furthermore, we extend the proposed caching schemes to the case of storage server failure.

A. Online Caching Scheme

Let $\mathcal{T}$ denote the whole data service period. The online scheme updates the caching decision according to the measured data popularity r_m^t and network latencies L_i^t in real time, $t\in\mathcal{T}$. The valuation $\{\tau_{m,0},\tau_{m,1},\ldots,\tau_{m,K}\}$ is updated according to the latest measurement of data access latency L_i^t and request rate r_m^t.

Data Popularity: The Discounting Rate Estimator (DRE) [32] method is applied to construct the real-time request information r_m^t. On the frontend server, a counter is maintained for each data item, which increases with every data read, and decreases periodically with a decay factor α_r. The benefits of DRE are as follows: 1) it reacts quickly to the request rate changes, and 2) it only requires $O(1)$ space and update time to maintain the prediction for each counter.

Network Latency: Similar to [5], the Exponentially Weighted Moving Average (EWMA) method [33] is used to estimate the average network latency of data requests. Specifically, after a data read operation, L_i^t is updated by

$$L_i^t=\alpha_l\cdot L_i^t+(1-\alpha_l)\cdot\iota_i, \tag{23}$$

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 previous requests. The advantage of EWMA is that it only needs $O(1)$ space to maintain the prediction for each storage node.

The long-tested techniques DRE and EWMA are used to estimate the future data popularity and network latency information, which have been widely adopted in real-world applications with low implementation overheads. Recent advances in future information prediction, e.g., Least Hit Density (LHD) [34] and Learning Relaxed Belady (LRB) [14], could also be applied in our solution.

Let Γ denote the set of data requests in the service period $\mathcal{T}$. To ensure the adaptivity of our design, the caching decision is updated upon the arrival of each request γ_m^t, $\gamma_m^t\in\Gamma$, $t\in\mathcal{T}$. The challenge of designing an online caching scheme is how to reduce the computation complexity without sacrificing the performance of low data access latency. To improve the solution efficiency, a common practice is to choose a subset of replacement candidates from the caching layer, with no need to completely override the existing caching decisions. For example, Hyperbolic [35] and LRB [14] randomly sample cached contents to obtain replacement candidates, which may result in arbitrarily bad performance. In contrast, we select the cached contents with the lowest valuations per unit as the replacement candidates, with which the worst-case performance guarantee can be derived. Then, close-to-optimal caching decisions can be obtained by applying the proposed optimal scheme to the set of replacement candidates. The pseudo code of the online caching scheme is listed in Algorithm 3, which makes a good balance between the computation complexity and data access latency.

Algorithm 3 Online Caching Scheme

```
Input: Cache capacity $C$, number of coded data chunks $K$,
    number of data items $M$, valuation array $\tau$, set of data
    requests $\Gamma$ in period $\mathcal{T}$.
Output: Set of cached data items $\hat{\mathcal{M}}$, online caching decision
    $\lambda_m^t$, $m\in\mathcal{M}$.
Initialization: $\hat{\mathcal{M}}\leftarrow\emptyset$, $\forall\lambda_m^t\leftarrow 0$.
1:  for Data request $\gamma_m^t\in\Gamma$, $t\in\mathcal{T}$ do
2:    Update $\{\tau_{m,0},\tau_{m,1},\ldots,\tau_{m,K}\}$ according to (2) and (5);
3:    if $\sum_{n\in\mathcal{M}}\lambda_n^t\le C-K$ and $\lambda_m^t<K$ then
4:      $\lambda_m^t\leftarrow K$, add $m$ to $\hat{\mathcal{M}}$;
5:    else if $\sum_{n\in\mathcal{M}}\lambda_n^t>C-K$ and $\lambda_m^t<K$ then
6:      $\hat{\mathcal{M}}'\leftarrow\{m\}$;
7:      repeat
8:        $n\leftarrow\arg\min_{n\in\hat{\mathcal{M}}\setminus\hat{\mathcal{M}}'}\{\frac{\tau_{n,k}}{k}\}$, add $n$ to $\hat{\mathcal{M}}'$;
9:      until $K\le C-\sum_{n\in\hat{\mathcal{M}}}\lambda_n^t+\sum_{n'\in\hat{\mathcal{M}}'}\lambda_{n'}^t\le 2K-1$
10:     $\forall\lambda_{n'}^t\leftarrow 0$, $n'\in\hat{\mathcal{M}}'$;
11:     Invoke Algorithm 1 for cache partition set $\hat{\chi}$ based on
        the available cache capacity;
12:     Invoke Algorithm 2 to update the caching decisions
        $\lambda_{n'}^t$ based on $\hat{\mathcal{M}}'$ and $\hat{\chi}$, $n'\in\hat{\mathcal{M}}'$;
13:     Update $\hat{\mathcal{M}}$, remove $n$ from $\hat{\mathcal{M}}$ if $\lambda_n^t=0$, $\forall n\in\hat{\mathcal{M}}'$;
14:   end if
15: end for
```

Let $\hat{\mathcal{M}}$ denote the set of already cached data items. If the cache capacity is not fully utilized, i.e., $\sum_{n\in\hat{\mathcal{M}}}\lambda_n^t\le C-K$, all K data chunks of the requested data item m should be cached for latency reduction. In contrast, if $\sum_{n\in\hat{\mathcal{M}}}\lambda_n^t>C-K$, 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, the data items in $\hat{\mathcal{M}}$ with the lowest valuations per unit are added into subset $\hat{\mathcal{M}}'$. The data items in $\hat{\mathcal{M}}'$ are expected to be replaced first by the requested data item m to maximize the total amount of reduced latency. Furthermore, m is also added into $\hat{\mathcal{M}}'$. All data items in $\hat{\mathcal{M}}'$ are cache replacement candidates. The cached data items in $\hat{\mathcal{M}}$ are gradually added into $\hat{\mathcal{M}}'$ until the available cache capacity $C-\sum_{n\in\hat{\mathcal{M}}}\lambda_n^t+\sum_{n'\in\hat{\mathcal{M}}'}\lambda_{n'}^t\ge K$. This guarantees that all K data chunks of m have a chance to be cached. The expansion of $\hat{\mathcal{M}}'$ needs K steps at most with $|\hat{\mathcal{M}}'|\le K+1$

and $C - \sum_{n\in\hat{\mathcal{M}}} \lambda_n^t + \sum_{n'\in\hat{\mathcal{M}}'} \lambda_{n'}^t \leq 2K-1$. Based on the available cache capacity, Algorithm 1 is invoked to calculate the cache partition set $\hat{\chi}$. Then, based on subset $\hat{\mathcal{M}}'$ and $\hat{\chi}$, Algorithm 2 is invoked to update the caching decisions λ_n^t, $n \in \hat{\mathcal{M}}'$. The theoretical analysis of the online scheme is provided as follows.

Theorem 3: If the data request arrivals are stationary and follow a Zipf distribution, Algorithm 3 yields the worst-case approximation ratio of $1 - \frac{2K-1}{C}$.

Proof: In Algorithm 3, DRE can accurately track data popularities when data request arrivals are stationary and highly skewed [32]. The greedy selection of $\hat{\mathcal{M}}'$ (Line 8) may incur performance loss. Let $\lambda_m^t = k$ denote the caching decision obtained with Algorithm 3 for request γ_m^t, $0 \leq k \leq K$. Then, we consider the following two different cases:

1) $\sum_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t} \leq \tau_{m,K}$: Since Algorithm 2 is invoked, Algorithm 3 outputs the optimal decision for subset $\hat{\mathcal{M}}'$. As $\tau_{m,k} \leq \tau_{m,K}$, the obtained objective value Θ satisfies

$$\Theta \geq \sum\nolimits_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t} - \sum\nolimits_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t} + \tau_{m,K}. \quad (24)$$

The global optimal objective value satisfies

$$\Theta^* \leq \sum\nolimits_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t} + \tau_{m,K}. \quad (25)$$

Due to the greedy selection, $\frac{\tau_{n',\lambda_{n'}^t}}{\lambda_{n'}^t} \leq \frac{\tau_{n,\lambda_n^t}}{\lambda_n^t}$ holds, $\forall n' \in \hat{\mathcal{M}}'$, $\forall n \in \hat{\mathcal{M}} \setminus \hat{\mathcal{M}}'$. As $\sum_{n\in\hat{\mathcal{M}}'} \lambda_n^t \leq 2K-1$, we have

$$\frac{\sum_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t}}{C} \geq \frac{\sum_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t}}{2K-1}. \quad (26)$$

The worst-case performance bound is given by

$$\frac{\Theta}{\Theta^*} \geq \frac{C-2K+1}{C}. \quad (27)$$

2) $\sum_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t} > \tau_{m,K}$: In this case, we have

$$\begin{aligned}\Theta^* &< \sum\nolimits_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t} + \tau_{m,K} \\ &< \sum\nolimits_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t} + \sum\nolimits_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t}. \end{aligned} \quad (28)$$

As $\Theta \geq \sum_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t}$, we have

$$\frac{\Theta}{\Theta^*} > \frac{\sum_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t}}{\sum_{n\in\hat{\mathcal{M}}} \tau_{n,\lambda_n^t} + \sum_{n\in\hat{\mathcal{M}}'\setminus\{m\}} \tau_{n,\lambda_n^t}} > \frac{C}{C+2K-1}. \quad (29)$$

As $K \geq 1$, we have $\frac{C}{C+2K-1} > 1 - \frac{2K-1}{C}$. The proof completes. $\square$

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, Theorem 3 shows that Algorithm 3 can approximate the optimal solution well.

Remark 2: Our design can be applied to general time-varying data request distributions. Previous work shows that DRE is $(C + \left\lceil \frac{\log(1-\alpha_r)}{\log \alpha_r} \right\rceil - 1)$*-competitive, where* α_r *is the decay factor [36]. Therefore, Algorithm 3 yields the worst-case approximation ratio of* $(1 - \frac{2K-1}{C}) \cdot \frac{1}{C + \lceil \frac{\log(1-\alpha_r)}{\log \alpha_r} \rceil - 1}$ *upon the arrival of time-varying data requests.*

Proposition 3: For a data request, the computation complexity of Algorithm 3 is no larger than $O(K^2 P_0 K!)$.

TABLE II
THE MAXIMUM NUMBER OF CACHE PARTITIONS $\max|\hat{\chi}|$ WITH THE VARIATION OF NUMBER OF CODED DATA CHUNKS

K	**2**	**4**	**6**	**8**
$K!$	2	24	720	40,320
$\max\|\hat{\chi}\|$	2	9	37	127

Proof: For any data requests, we have $|\hat{\mathcal{M}}'| \leq K+1$ and $C - \sum_{n\in\hat{\mathcal{M}}} \lambda_n^t + \sum_{n'\in\hat{\mathcal{M}}'} \lambda_{n'}^t \leq 2K-1$. According to Proposition 1, the set size $|\hat{\chi}| < \prod_{k=2}^{K} (\lfloor \frac{2K-1}{k} \rfloor + 1)$ holds. As $\frac{2K-1}{k} < K-k+2$, $k \in \{2,\ldots,K\}$, we have $\lfloor \frac{2K-1}{k} \rfloor + 1 \leq K-k+2$. Therefore, $|\hat{\chi}| < K!$ holds. Furthermore, similar to the analysis in Proposition 2, the computation complexity for a cache partition is $O(K^2 P_0)$. The computation complexity of Algorithm 3 is no larger than $O(K^2 P_0 K!)$. $\square$

The computation complexity of Algorithm 3 is also determined by the number of cache partitions $|\hat{\chi}|$. As discussed in Proposition 1, the values of a cache partition cannot be assigned to the maximum values at the same time. Therefore, $K!$ is only the upper bound of $|\hat{\chi}|$ in theory. Table II shows the maximum number of cache partitions $\max|\hat{\chi}|$ generated by Algorithm 1 with the increase of K. The main observation is that $\max|\hat{\chi}|$ increases much slower than $K!$ in practice. Moreover, K is not set to a large number in the practical storage system [13], [19]. Therefore, Algorithm 3 has low computational complexity, which ensures that the online scheme can react quickly to real-time changes.

B. Caching With Server Failure

In practice, servers may experience downtime in the distributed storage system. In this subsection, the proposed caching schemes are extended to the case of storage server failure. Let $\mathcal{M}_i$ denote the set of remotely unavailable data chunks when a storage server at node i fails. Recall that we need exactly K chunks to reconstruct a data item. If data chunk $m_k \in \mathcal{M}_i$ is not cached beforehand, the degraded read is triggered to serve the data requests. The parity chunk m_r with the lowest data access latency will be fetched from node j for data reconstruction. 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

$$l_{m_r}^t = \min\{L_j^t \cdot \mathbf{1}(m_r \to j)\}. \quad (30)$$

When Algorithm 2 or 3 suggests caching m_r, the recovered data chunk m_k (instead of m_r) is directly added into the caching layer. In this way, the parity chunk m_r is not always required to reconstruct data item m. Compared with existing work which caches parity chunks [9], our design can reduce the decoding overheads of the subsequent data requests.

VI. EXPERIMENTAL EVALUATION

In this section, we build a prototype of the caching system in Python and then integrate it with the experiment platform deployed in Sec. III-B. Next, extensive experiments are performed to quantitatively evaluate the performance of the proposed optimal and online caching schemes.

Experimental Setup: We deploy eighteen `buckets` in $N = 6$ AWS regions. Each `bucket` denotes a remote storage server. The library `zfec` [37] is adopted to implement the RS codes. By default, we set the number of coded chunks $K = 6$ and $R = 3$. The coded chunks of $M = 1,000$ data items are

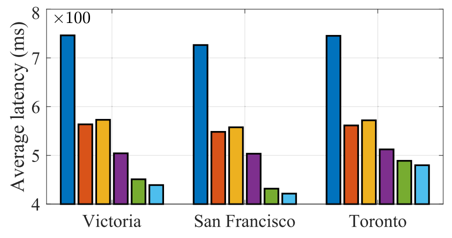


Fig. 5. Average data request latencies.

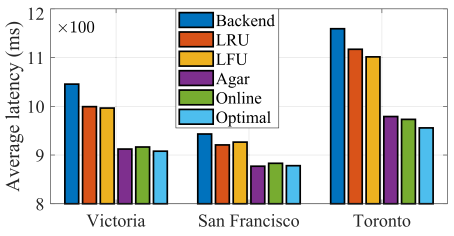


Fig. 6. 95th percentile tail latencies.

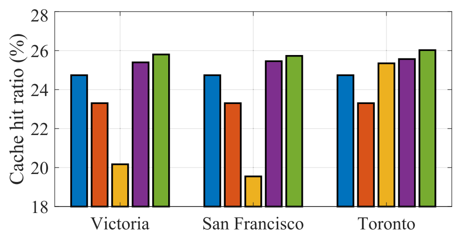


Fig. 7. Hit ratio of data chunk requests.

TABLE III
THE NUMBER OF CACHE PARTITIONS AND THE ART OF CACHING SCHEMES WITH THE VARIATION OF CACHE CAPACITY AND NUMBER OF CODED DATA CHUNKS

C		**60**	**80**	**100**	**120**
LRU	ART (ms)	0.05			
LFU	ART (ms)	0.05			
Agar	ART (ms)	544.48	661.37	800.02	960.97
Optimal	$\|\chi\|$	19,858	69,624	189,509	436,140
	ART (s)	391.64	2,239.67	7,863.99	23,746.93
Online	$\max\|\hat{\chi}\|$	37			
	ART (ms)	1.83	1.66	1.54	1.46

K		**2**	**4**	**6**	**8**
LRU	ART (ms)	0.05			
LFU	ART (ms)	0.05			
Agar	ART (ms)	421.74	628.31	800.02	948.22
Optimal	$\|\chi\|$	51	8,037	189,509	1,527,675
	ART (s)	2.04	666.27	7,863.99	49,662.91
Online	$\max\|\hat{\chi}\|$	2	9	37	127
	ART (ms)	0.25	0.62	1.54	4.53

with the same size 1 MB [9]. They are uniformly distributed at different buckets to achieve fault tolerance. As shown in Table I, three frontend servers are built on personal computers in different cities. The hardware features an Intel(R) Core(TM) i7-7700 HQ processor and 16 GB memory. The cache capacity of Memcached is set to 100 MB in RAM, i.e., the maximum number of cached data chunks is set to $C = 100$. The data service period $\mathcal{T}$ is set to 1 hour. Similar to the previous studies [7], [12], the popularity of data requests follows a Zipf distribution, which is common in many real-world data request distributions. The tail index of the Zipf distribution is set to 2.0 under the default settings, i.e., highly skewed.

Performance Baselines: For a fair performance comparison, four other schemes are adopted as performance baselines.

- Backend—All required K data chunks are directly fetched from the remote buckets with no caching. This scheme is adopted to quantify the benefits of caching.
- LRU and LFU—The Least Recently Used (LRU) and Least Frequently Used (LFU) caching policies are used to replace the contents in the caching layer with the computation complexity of $O(1)$ for data eviction [38]. For each selected data item, all K data chunks are cached.
- Agar [9]—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 $O(CKM)$.

A. Experimental Results

To begin with, the performances of six schemes, i.e., Backend, LRU, LFU, Agar, and the proposed optimal and online schemes, are compared under the default settings. As all requested data chunks are fetched from the remote buckets, Backend yields the highest average latency (746.27 ms, 726.43 ms, and 745.34 ms) and 95th percentile tail latencies (1,045.49 ms, 943.12 ms, and 1,159.16 ms) at three frontend servers.

LRU caches the recently requested data items by discarding the least recently used data items. LFU caches the data items with higher request rates. Compared with Backend, LRU and LFU reduce the average latencies of all data requests at three frontend servers by 24.6% and 23.2%, respectively. As illustrated in Fig. 7, 24.7% and 23.3% of requests are fulfilled by the cached data chunks with LRU and LFU. With the whole data item cached, LRU and LFU reduce the access latencies to 0 ms for 24.7% and 23.3% of data requests, respectively. However, as the cache capacity is limited, the remaining parts of the requests suffer from high access latencies. Compared with Backend, the 95th percentile tail latencies are only reduced by 3.5% and 3.9%, 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 achieve the lowest data access latency.

Agar iteratively improves the existing caching configurations by considering new data chunks. Each data item is assigned a weight, given by the number of data chunks to cache. Compared with LFU and LRU, more data items can enjoy the benefits of caching. The average latencies at three frontend servers are reduced to 504.21 ms, 503.42 ms, and 512.27 ms, respectively. Moreover, 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 912.24 ms, 877.02 ms, and 978.97 ms.

With an overall consideration of data request rates and access latencies, the proposed optimal scheme optimizes the number of cached data chunks for each data item, minimizing the average latencies to 439.06 ms, 421.52 ms, and 479.79 ms. Fig. 5 shows that the proposed online scheme approximates the optimal scheme well with a similar latency of 450.95 ms, 431.66 ms, and 488.84 ms. As shown in Table III, although raising the average latency by 2.3%, the online scheme greatly reduces the computation overheads. Furthermore, the proposed optimal and online schemes optimize the caching decisions by selecting the data chunks with higher valuations. This means the contents in the caching layers are with higher data

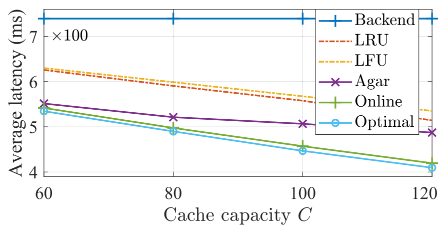


Fig. 8. Impact of cache capacity.

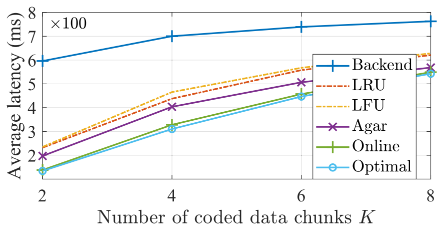


Fig. 9. Impact of number of coded data chunks.

TABLE IV

THE PERCENTAGE OF INCREASED DATA ACCESS LATENCY INCURRED BY THE ONLINE SCHEME, I.E., PERFORMANCE LOSS, WHEN COMPARED WITH THE OPTIMAL SCHEME

C	**60**	**80**	**100**	**120**
Performance loss (%)	1.3	1.6	2.3	2.4
K	**2**	**4**	**6**	**8**
Performance loss (%)	3.0	5.7	2.3	1.2
M	**500**	**1,000**	**2,000**	**5,000**
Performance loss (%)	6.7	2.3	2.0	2.2
Tail index	**Uniform**	**1.2**	**1.5**	**2.0**
Performance loss (%)	2.2	2.2	1.7	2.3

request rates and lower access latencies. The hit ratios of data requests from three frontend servers are 25.8%, 25.7%, and 26.0%, respectively. The 95th percentile tail latencies with the optimal scheme are reduced to 907.86 ms, 878.11 ms, and 955.65 ms. The online scheme incurs a similar 95th percentile tail latencies of 916.45 ms, 883.0 ms, and 973.13 ms.

B. Impact of Other Factors

In this section, the impacts of cache capacity, number of coded data chunks, number of data items, data popularity, server failure, and seller pricing, are considered for performance evaluation. For simplicity, the average latency represents the average latency of all data requests from three frontend servers in the following parts of the paper.

Cache Capacity: Fig. 8 illustrates the average latency when the cache capacity C increases from 60 to 120 chunks. With no data caching, the average latencies of Backend remain stable at 739.35 ms. With more data requests benefit from caching, the average latencies with all five 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 optimal scheme improves from 3.1% to 16.0%. As shown in Table IV, when compared with the optimal scheme, the online scheme only increases the average latency from 1.3% to 2.4% with the variation of cache capacity.

Then, the ART of five caching schemes is evaluated, which determines the efficiency of deploying a caching solution. As shown in Table III, by using simple heuristics, LRU and LFU only need 0.05 ms to update the caching decision. Agar periodically optimizes the caching configuration for all data items in the storage system, which needs hundreds of milliseconds for a round of optimization. With the increase of cache capacity, the number of cache partitions $|\chi|$ increases rapidly from 19,858 to 436,140. The ART of the optimal scheme increases from 391.64 s to 23,746.93 s. In contrast, the online scheme updates the caching decision upon the arrival of each data request. According to our design in Algorithm 3, the maximum number of cache partitions $|\hat{\chi}|$ is determined by the number of coded data chunks K. When a data request arrives, the caching decision will not be updated if the data item is already cached. Therefore, the ART of the online scheme for each request decreases from 1.83 ms to 1.46 ms with the increase of cache capacity. This means the online scheme is a scalable solution for a large-scale storage system.

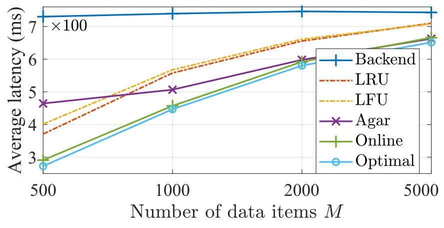


Fig. 10. Impact of number of data items.

Number of Coded Data Chunks: The size of data items is increased from 2 MB to 8 MB. With the same size of coded chunks (1 MB), the number of coded data chunks K increases from 2 to 8. As coded chunks are uniformly distributed among remote `buckets`, more data chunks will be placed at the `buckets` with higher access latencies with the increase of K. Therefore, as shown in Fig. 9, with $C = 100$ and $M = 1,000$, the average data access latency with Backend increases from 596.06 ms to 762.83 ms. 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 five caching schemes increase accordingly. Fig. 9 shows that the proposed optimal and online schemes always incur lower latencies than Agar, LRU, LFU, and Backend. Compared with Agar, the percentage of reduced latency via the online scheme varies from 29.9% to 3.4% with the increase of K. Furthermore, Table III shows that the ART of the online scheme only increases from 0.25 ms to 4.53 ms. With the online scheme, few extra delays will be introduced to handle the intensive data requests.

Number of Data Items: As shown in Fig. 10, with $C = 100$ and $K = 6$, the number of deployed data items M is increased from 500 to 5,000. The average data access latency with Backend remains basically the same. As the data popularity follows Zipf distribution, a small portion of data items get the majority of data requests. With the growing total number of data items, the number of data items with relatively higher request rates increases. Due to the limited cache capacity, more and more data requests are served by fetching data chunks from the remote servers. Therefore, the average latencies increase rapidly with LRU (from 370.93 ms to 710.07 ms), LFU (from 401.26 ms to 707.84 ms), Agar (from 464.45 ms to 662.62 ms), and the optimal (from 272.66 ms to 651.48 ms) and online (from 291.01 ms to 666.0 ms) schemes.

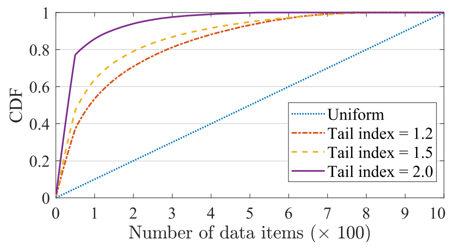


Fig. 11. CDF of data popularity.

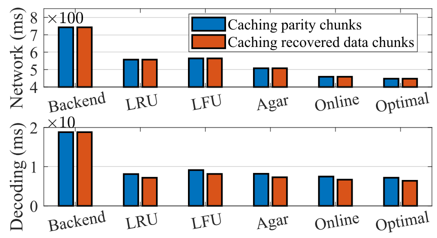


Fig. 13. Impact of server failure.

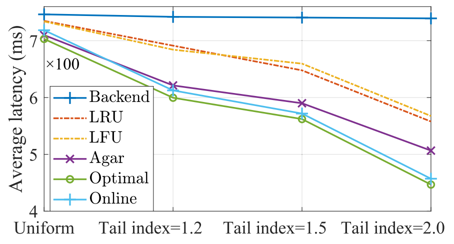


Fig. 12. Impact of data popularity.

TABLE V

PERFORMANCE COMPARISON WITH DIFFERENT PRICING SCHEMES WHEN APPLIED TO THE PROPOSED ONLINE CACHING SCHEME

Pricing method		Our design	[31]
Average latency (ms)	**Victoria**	450.95	450.95
	San Francisco	431.66	431.66
	Toronto	488.84	488.84
Average pricing rounds	**Victoria**	1.13	431.12
	San Francisco	1.45	488.26
	Toronto	1.29	409.82
ART (ms)	**Victoria**	1.39	388.02
	San Francisco	1.78	412.75
	Toronto	1.44	340.04

Data Popularity: Fig. 11 illustrates the CDF of the data popularity using uniform and Zipf distributions. As shown in Fig. 12, all six schemes incur similar data access latencies when the data popularity follows a uniform distribution. When all data items have the same popularity, the caching valuation is only determined by the storage locations of the data items. With a similar caching valuation for different data items, the benefits of caching are not significant when the cache capacity is limited. Then, with the increase of the tail index from 1.2 to 2.0, the skew of the data popularity becomes higher and higher. A fraction of data items with higher request frequencies can benefit more from caching. When the tail index is set to 2.0, compared with Backend, LRU, LFU, and Agar, the optimal scheme reduces the average latency by 39.6%, 21.3%, 19.9%, and 11.8%, respectively.

Furthermore, the online scheme can approximate the optimal scheme well. With a uniform distribution of data popularity, the optimal scheme and the online scheme incur the average data access latencies of 718.52 ms and 703.12 ms, respectively. The performance loss is 15.4 ms. With a higher skewness of data popularity, the greedy selection of cache replacement candidates can approximate the optimal solution better. With the tail index 2.0, the optimal scheme and the online scheme incur the average data access latencies of 457.15 ms and 446.79 ms, respectively. The performance loss is reduced to 10.36 ms. Table IV demonstrates that when compared with the optimal scheme, the online scheme only increases the average latency by about 2% under different settings of data popularity.

Server Failure: Then, we evaluate the performance of the proposed optimal and online schemes when server failure happens. Please note that erasure codes can tolerate up to R simultaneous server failures. Recent research indicated that single server failure is responsible for 99.75% of all kinds of server failures [39]. Therefore, single server failure is considered in this paper by terminating each storage server in turn. The experiment setting is identical to that in Sec. VI-A except for the storage server failure. If the needed data chunks are not available on the remote servers or cached in the caching layer, degraded read will be triggered to serve the data requests. In this case, the data access latency contains two parts, i.e., the network latency and the decoding latency.

Fig. 13 illustrates the average data access latencies with various schemes. Without caching services, Backend incurs the average network latency of 742.81 ms and the average decoding latency of 18.82 ms. To begin with, we consider the performance of caching parity chunks if the needed data chunks are remotely unavailable under server failure. Compared with Backend, LFU, LRU, Agar, and the proposed optimal and online schemes reduce the average decoding latencies by over 50%.

Then, let us consider the performance of caching the recovered data chunks to avoid unnecessary decoding overheads of the subsequent data requests. As shown in Fig. 13, compared with caching parity chunks, the average decoding latencies of five caching schemes can be reduced by about 11%. Compared with Backend, LRU, LFU, and Agar, the optimal scheme reduces the overall average data access latency by 40.4%, 20.5%, 19.7%, and 12.0%, respectively. Furthermore, compared with the optimal scheme (with the average network latency of 447.69 ms and decoding latency of 6.4 ms), the online scheme (with the average network latency of 458.87 ms and the decoding latency of 6.69 ms) incurs a performance loss of 2.5% in the presence of server failure.

Seller Pricing: We compare the performance of the proposed pricing method in (12) and the pricing method in [31] when applied to the online caching scheme. The pricing method in [31] has relatively low efficiency by raising the price of the competed seller with only one unit in each round. In fact, due to the prohibitively high computation overheads, the pricing scheme in [31] cannot be applied to the offline optimal scheme. In contrast, our proposed pricing method ensures the competed seller will be assigned to only one buyer in each round of pricing, which improves the solution efficiency. As shown in Table V, our proposed pricing method is about 300× faster than the classic pricing scheme.

VII. Conclusion and Future Work

In this paper, novel caching schemes were proposed to achieve low latency in the distributed coded storage system. To reduce the data access latency, frontend servers, each with an in-memory caching layer, were deployed to cache coded data chunks near end users. Experiments based on Amazon S3 confirmed the positive correlation between the latency and the physical distance of data retrieval over the WAN. As the distributed storage system spans multiple geographical sites, the average data access latency was used to quantify the benefits of caching. With the assumption of future data popularity and network latency information, an optimal caching scheme was proposed to obtain the lower bound of data access latency. Guided by the optimal scheme, we further designed an online caching scheme based on the measured data popularity and network latencies in real time. Extensive experiments demonstrated that the online scheme approximates the optimal scheme well and significantly reduces the computation complexity.

In future work, we plan to further reduce the computation complexity of the online scheme incurred by $|\hat{\chi}|$, which increases rapidly with the increased number of coded data chunks K. Intuitively, we do not need to consider all cache partitions in $\hat{\chi}$ to obtain near-optimal caching decisions. We plan to analyze the structure of $\hat{\chi}$ and propose a sampling scheme to select a subset of cache partitions that can contribute to the lowest data access latency.

References

[1] D. Reinsel, J. Gantz, and J. Rydning. (2018). *The Digitization of the World From Edge to Core*. Available: https://www.seagate.com/em/en/our-story/data-age-2025/

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

[3] (2020). *Google Cloud Storage*. [Online]. Available: https://cloud.google.com/storage/

[4] (2020). *Microsoft Azure*. [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.*, early access, Sep. 12, 2019, doi: 10.1109/TCC.2019.2940953.

[6] M. Annamalai *et al.*, "Sharding the shards: Managing datastore locality at scale with Akkio," in *Proc. OSDI*, 2018, pp. 445–460.

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

[8] 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. ICDCS*, Jul. 2018, pp. 255–266.

[9] R. Halalai, P. Felber, A. Kermarrec, and F. Taïani, "Agar: A caching system for erasure-coded data," in *Proc. ICDCS*, 2017, pp. 23–33.

[10] Y. Li, J. Zhou, W. Wang, and Y. Chen, "RE-Store: Reliable and efficient KV-store with erasure coding and replication," in *Proc. CLUSTER*, Jun. 2019, pp. 1–12.

[11] M. Uluyol, A. Huang, A. Goel, M. Chowdhury, and H. V. Madhyastha, "Near-optimal latency versus cost tradeoffs in geo-distributed storage," in *Proc. NSDI*, 2020, pp. 157–180.

[12] 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. OSDI*, 2016, pp. 401–417.

[13] 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.

[14] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed Belady for content distribution network caching," in *Proc. NSDI*, 2020, pp. 529–544.

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

[16] K. Lee, N. B. Shah, L. Huang, and K. Ramchandran, "The MDS queue: Analysing the latency performance of erasure codes," *IEEE Trans. Inf. Theory*, vol. 63, no. 5, pp. 2822–2842, May 2017.

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

[18] A. Badita, P. Parag, and J.-F. Chamberland, "Latency analysis for distributed coded storage systems," *IEEE Trans. Inf. Theory*, vol. 65, no. 6, pp. 4683–4698, Apr. 2019.

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

[20] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee, "An ensemble of replication and erasure codes for cloud file systems," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1276–1284.

[21] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "FemtoCaching: Wireless content delivery through distributed caching helpers," *IEEE Trans. Inf. Theory*, vol. 59, no. 12, pp. 8402–8413, Dec. 2013.

[22] S. Ioannidis and E. Yeh, "Adaptive caching networks with optimality guarantees," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 113–124, Jun. 2016.

[23] (2019). *HDFS Architecture Guide*. [Online]. Available: https://hadoop.apache.org/

[24] (2019). *Cassandra*. [Online]. Available: http://cassandra.apache.org/

[25] C. Huang *et al.*, "Erasure coding in Windows Azure storage," in *Proc. ATC*, 2012, pp. 15–26.

[26] Z. Liu *et al.*, "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. FAST*, 2019, pp. 143–157.

[27] (2019). *Memcached*. [Online]. Available: http://memcached.org/

[28] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *Proc. SIGCOMM*, 2020, pp. 495–513.

[29] K. Liu, J. Peng, B. Yu, W. Liu, Z. Huang, and J. Pan, "An instance reservation framework for cost effective services in geo-distributed data centers," *IEEE Trans. Services Comput.*, vol. 14, no. 2, pp. 356–370, Mar. 2021.

[30] 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. SoCC*, 2018, pp. 386–400.

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

[32] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. SIGCOMM*, 2014, pp. 503–514.

[33] H. J. Stuart, "The exponentially weighted moving average," *J. Qual. Technol.*, vol. 18, no. 4, pp. 203–210, 1986.

[34] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving hit rate by maximizing hit density," in *Proc. NSDI*, 2018, pp. 1–14.

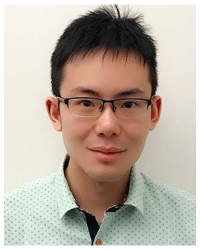
[35] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *Proc. ATC*, 2017, pp. 499–511.

[36] E. Cohen, H. Kaplan, and U. Zwick, "Competitive analysis of the LRFU paging algorithm," *Algorithmica*, vol. 33, no. 4, pp. 511–516, Aug. 2002.

[37] (2013). *Zfec*. [Online]. Available: https://github.com/richardkiss/py-zfec

[38] 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. WiOpt*, 2018, pp. 1–6.

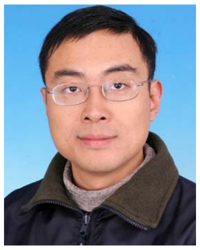
[39] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. FAST*, 2012, pp. 20–33.



Kaiyang Liu (Member, IEEE) received the Ph.D. degree from the School of Information Science and Engineering, Central South University, in 2019. From 2016 to 2018, he was a Research Assistant at the University of Victoria, Canada, with Prof. Jianping Pan. His current research areas include networked systems, distributed systems, and cloud/edge computing, with a special focus on the analysis and optimization of data-intensive services. 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 Ph.D. Thesis Award.

Jun Peng (Senior Member, IEEE) received the B.S. degree from Xiangtan University, Xiangtan, China, in 1987, the M.Sc. degree from the National University of Defense Technology, Changsha, China, in 1990, and the Ph.D. degree from Central South University, Changsha, 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. 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 the M.Sc. degree in computer science from the University of Victoria in 2019. She is currently pursuing the Ph.D. degree with the University of Toronto. Her research interests cover wireless communications, mobile edge computing, and distributed machine learning.



Jianping Pan (Senior Member, IEEE) received the bachelor's and Ph.D. degrees in computer science from Southeast University, Nanjing, Jiangsu, China. He did his post-doctoral research at the University of Waterloo, Waterloo, ON, Canada. He worked at Fujitsu Labs and NTT Labs. He is currently a Professor of computer science at the University of Victoria, Victoria, BC, Canada. 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 is a Senior Member of the ACM. 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, and is a coauthor of one of three IEEE LCN 2018 Best Paper Award candidates. He 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.