Advanced Elastic Reed–Solomon Codes for Erasure-Coded Key–Value Stores

Junmei Chen, Zongpeng Li, *Senior Member, IEEE*, Ruiting Zhou, *Member, IEEE*, Lina Su, and Ne Wang

***Abstract*—Erasure coding is a storage-efficient redundancy scheme for modern key–value (KV) stores, storing stripes of data and parity chunks in multiple nodes. To accommodate the highly skewed and time-varying nature of the workload, KV stores require erasure code that dynamically optimizes its parameters, known as *redundancy converting*. Stretched Reed–Solomon (SRS) and elastic Reed–Solomon (ERS) codes represent promising candidates for meeting such requirements. However, both SRS and ERS are limited to RS $(d, r) \rightarrow$ RS (d', r') converting, where $d' > d, r' = r$, failing to fully meet actual needs. This work presents an advanced ERS code (AERS code), which builds upon flexible encoding matrices and placement strategies, serving different types of redundancy converting, and minimizing converting traffic. We further prove that the AERS code is an optimal redundancy converting solution that achieves the theoretical lower bound on data traffic during redundancy converting while guaranteeing node-level fault tolerance. We evaluate the AERS code through both mathematical analysis and experiments. In the *mise-en-scène* of its state-of-the-art alternatives, AERS stands out by reducing network traffic up to 50%–85.7% while accelerating redundancy converting.**

***Index Terms*—Access skew, elastic Reed–Solomon (ERS) code, erasure code, key–value (KV) stores, redundancy converting, time varying.**

I. Introduction

KEY–VALUE (KV) stores are customarily deployed in enterprise applications, e.g., big data analytics, online sales, and social networks [1], [2], [3]. Well-designed KV stores offer excellent performance and scalability while maintaining consistency across store schemes [2], [3]. Servers in KV stores face frequent daily failures caused by software errors, power outages, and maintenance interruptions. In a Facebook cluster of 3000 servers, 50 failures per day are expected to render data unavailable [4]. To ensure the systems' reliability in a storage-efficient manner, modern KV stores increasingly adopt erasure coding instead of replication to provide data protection, which can achieve higher reliability under the same level of redundancy [5], [6]. A systematic Reed–Solomon (RS) code [8] is an erasure code that encodes d original data chunks into $d + r$ coded chunks, including d original data chunks and r redundant chunks (called parity chunks). All original data chunks can be successfully reconstructed by any d of the $d + r$ available chunks, known as the maximum distance separable (MDS) property.

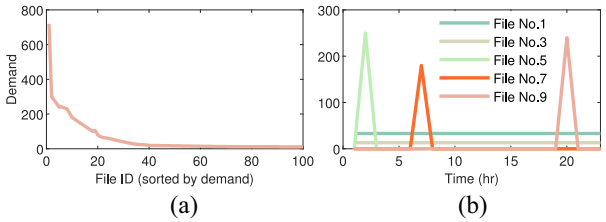


Fig. 1. Demand skewness of files in a Facebook workload trace [7]. (a) Demand of top 100 files. (b) Demand over time.

A trace analysis collected from Facebook's Memcached deployment shows that the demand for caching workloads exhibits a high skew [7], [9], [10], with a small fraction of data chunks accessed frequently, and the rest rarely accessed, as shown in Fig. 1(a). Furthermore, the access requirements of storage workloads exhibit temporal variations [7] [11], as shown in Fig. 1(b). Thus, to accommodate resiliency requirements, the erasure code in KV store systems must support *redundancy converting*, i.e., dynamically adjusting the erasure code's parameters to balance performance, storage overhead, and reliability.

Existing studies of erasure-coded redundancy converting are mainly concentrated on trading off storage and reconstruction overhead, as well as reducing I/O operations [24], [25], [26], [27], [28], [29], [30], [31], [32]. They incur substantial redundancy converting traffic (i.e., chunks transmitted over the network in redundancy converting operations), making converting performance unsatisfactory. Besides, they are not suitable for KV stores since they fail to ensure strong consistency. Stretched RS (SRS) [37] and elastic RS (ERS) [23] codes can simultaneously meet the elastic requirements of KV stores and ensure consistency. Nonetheless, both SRS and ERS consider only the conversion from hot data to cold data, i.e., from RS $(d, r) \rightarrow$ RS(d', r') with $d' > d, r' = r$, where d' and r' are the number of data and parity chunks in post-converting stripes, respectively. In fact, there are practical

Manuscript received 14 February 2023; revised 3 July 2023; accepted 25 July 2023. Date of publication 28 July 2023; date of current version 24 January 2024. This work was supported in part by the Huawei Project under Grant FA2019071041, and in part by China Telecom under Grant 20222930033. *(Corresponding author: Zongpeng Li.)*

Junmei Chen, Lina Su, and Ne Wang are with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: chenjm@whu.edu.cn; lina.su@whu.edu.cn; ne.wang@whu.edu.cn).

Zongpeng Li is with the School of Computer Science, Wuhan University, Wuhan 430072, China, and also with the School of Automation, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail: zongpeng@hdu.edu.cn).

Ruiting Zhou is with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China (e-mail: ruitingzhou@whu.edu.cn).

Digital Object Identifier 10.1109/JIOT.2023.3299574

cases of converting from cold data to hot data. For example, when searching for past information on hot search objects (such as COVID-19, popular stars, or Olympic games), this past information will convert from cold data into hot data, requiring redundancy converting from RS $(d, r) \rightarrow \text{RS}(d', r')$, where $d' < d$. This is also reflected in the access trend of File No. 9 in Fig. 1(b), which started with almost zero access and witnessed a substantial increase at 20 o'clock. We propose advanced ERS code (AERS code), a new redundancy converting mechanism, including an adaptive code construction phase and a redundancy converting phase. We aim to minimize converting traffic while supporting the variety of redundancy converting, guaranteeing consistency and node-level fault tolerance. The core of the AERS code is to maximize the number of superimposed data chunks (defined in Section II), so that old parity chunks (before converting) can be reused when updating new parity chunks (after converting). The two types of parity chunks share the same encoding operations on the superimposed data chunks. We further prove that the AERS code is an optimal redundancy converting scheme, which can reach the lower bound on converting traffic. To summarize, our contributions include the following.

1) We propose an advanced ERS code, referred to as an AERS code. By carefully constructing a series of RS codes with flexible parameters, the AERS code reduces the number of I/O operations and network traffic incurred by converting. Specifically, when converting from hot data to cold data, i.e., converting RS (d, r) into RS (d', r'), where $(d' + r'/d') < (d + r/d)$, we jointly design the extended encoding matrix and corresponding data placement strategy. When converting from cold data to hot data, i.e., converting RS (d, r) into RS (d', r'), where $(d' + r'/d') > (d + r/d)$, we jointly design encoding submatrices and corresponding data placement strategy. Then redundancy converting is performed on those basis. We further show that AERS is a general design for a variety of representative erasure codes.
2) We further prove that the AERS code is an optimal redundancy converting solution, reaching the lower bound on the number of data chunks transmitted during redundancy converting in both hot to cold converting and vice versa.
3) We conduct several sets of mathematical analyses and experiments to verify the efficiency of the AERS code. We implemented a KV store prototype for AERS code on Memcached [12]. The prototype supports all basic KV operations, such as PUT, GET, and UPDATE, while supporting different types of redundancy converting. The results of mathematical analysis and experiment show that, compared to the SRS code, the AERS code significantly slashes the latency of redundancy converting, and can save up to 50%–85.7% on the number of chunks transmitted during various redundancy converting.

The remainder of this article is organized as follows. Section II presents background and motivation. Section III describes the detailed design of the AERS code. Section IV provides the theoretical analysis. Section V contains evaluations. Section VI goes through related work. Section VII concludes this article.

II. Background and Motivation

We first introduce the fundamental concept of erasure coding, redundancy converting, and consistency in redundancy converting (Section II-A), and then illustrate our motivation through examples (Section II-B).

A. Background

Erasure Coding in KV Stores: In this work, we concentrate on RS codes [8], a well-known family of erasure codes that have been used in contemporary KV stores [13], [14], [15], [16], [17], [18]. In RS codes, the encoding matrix is multiplied with the data chunk vector (i.e., the column vector of d data chunks) to compute the parity chunk vector (i.e., the column vector of r parity chunks), over a Galois field GF(2^{ω}). The encoding matrix can be constructed from the Vandermonde matrix [19] and ω is the size of the encoding chunk (in bits). For example, when $(d, r) = (2, 2)$ and $\omega = 4$, the parity vector can be expressed as

$$\begin{bmatrix} P_0 \\ P_1 \end{bmatrix} = G_{2\times 2} \times \begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 8 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \end{bmatrix}.$$

There are two types of encoding methods for objects in KV stores, namely, per-object encoding [14], [20] and cross-object encoding [13], [16]. The former divides an object into d data chunks and then encodes them, while the latter stores multiple objects in one data chunk and then collects d data chunks to encode them together. We focus on the former, especially for workloads with large-size objects, which can show better load balancing and I/O performance.

Redundancy Converting: The essence of redundancy converting is to tailor the encoding parameters d and r of existing erasure-coded objects to adapt to dynamic access characteristics and reliability requirements. We design a redundancy converting scheme for general parameters d and r. By increasing d or decreasing r, the storage redundancy can be reduced, thereby improving the overall storage efficiency. It is also possible to reduce d or increase r to ensure the reliability of frequently accessed data.

Changes in encoding parameters will result in updating the encoding matrix and the corresponding parity chunks, bringing additional I/O cost and communication overhead to redundancy converting. Minimizing these two types of overhead in redundancy converting is a nontrivial task. In this work, we focus on reducing the I/O cost and communication overhead in redundancy converting.

Consistency: During converting, we update the parity chunks in the servers where the parity chunks are stored. We must ensure that all parity chunks are updated consistently. Specifically, the proxy sends the new parity chunks first. The servers receive them and store the new parity chunks in their transient buffer. Next, the servers provide feedback on the success of receiving and buffering the parity chunks, to the proxy. If the proxy receives acknowledgment from all servers

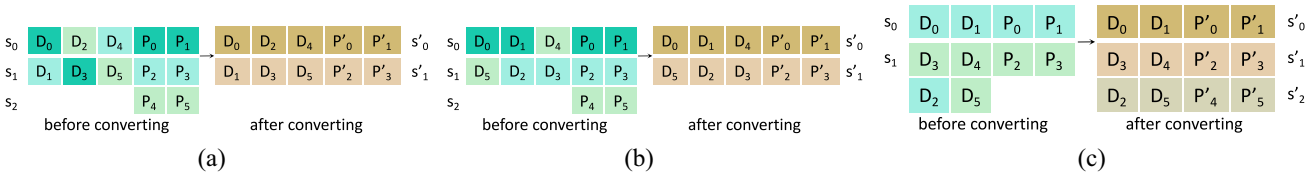


Fig. 2. Examples of redundancy converting by different codes. Chunks of the same color belong to the same stripe. (a) Convert RS $(2, 2)$ into RS $(3, 2)$ by SRS. (b) Convert RS $(2, 2)$ into RS $(3, 2)$ by ERS (AERS). (c) Convert RS $(3, 2)$ into RS $(2, 2)$ by AERS.

that cache the new parity chunks, it informs all servers to store them; otherwise, it informs all servers to discard the buffered parity chunks. We can take advantage of piggybacking to reduce the communication overhead of consistency by 50% [13].

B. Motivation

During redundancy converting, different encoding matrices and data placement strategies result in different numbers of I/O operations and network traffic. We describe those issues via examples of redundancy converting, as shown in Fig. 2.

Fig. 2(a) describes the redundancy converting process from RS $(2, 2)$ to RS $(3, 2)$ by the SRS code, where $d = 2, d' = 3, r = r' = 2$. Here, we will show the encoding process of the preconverting stripe s_1 and the post-converting stripe s'_1, respectively. The encoding process of stripe s_1 is multiplying the (old) encoding matrix $G_{2\times2}$ with two data chunks $\{D_1, D_4\}$, and the encoding process of stripe s'_1 is multiplying the (new) encoding matrix $G_{2\times3}$ with three data chunks $\{D_1, D_3, D_5\}$

$$\begin{bmatrix} P_2 \\ P_3 \end{bmatrix} = G_{2\times2} \times \begin{bmatrix} D_1 \\ D_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} D_1 \\ D_4 \end{bmatrix}$$
$$\begin{bmatrix} P'_2 \\ P'_3 \end{bmatrix} = G_{2\times3} \times \begin{bmatrix} D_1 \\ D_3 \\ D_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} D_1 \\ D_3 \\ D_5 \end{bmatrix}.$$

Since $P_2 = D_1 + D_4$, $P_3 = D_1 + 2D_4$, $P'_2 = D_1 + D_3 + D_5$, and $P'_3 = D_1 + 2D_3 + 3D_5$, so in order to update P_2, P_3 into P'_2, P'_3, we have to retrieve D_3, D_4, and D_5. Here, P_2 and P_3 are the parity chunks of preconverting stripes, P'_2 and P'_3 are the parity chunks of post-converting stripes. Similarly, updating P_0, P_1 into P'_0, P'_1 requires us to retrieve D_2, D_3, and D_4. Therefore, when converting from RS $(2, 2)$ to RS $(3, 2)$ using the SRS code, D_2, D_3, D_4, and D_5 need to be accessed for parity chunk updates.

Fig. 2(b) describes the redundancy converting process from $(2, 2)$ to $(3, 2)$ by the ERS (AERS) code, where $d = 2, d' = 3, r = r' = 2$, and $d' > d$. The ERS (AERS) code can generate more superimposed data chunks by using an extended encoding matrix $G_{r\times d'}$ and a specific data placement strategy, thereby reducing parity chunk updates during the converting. For example, the encoding process of stripe s_1 is multiplying the (old) encoding matrix $G_{2\times3}$ with two data chunks $\{D_2, D_3\}$, and the encoding process of stripe s'_1 is multiplying the (new) encoding matrix $G_{2\times3}$ with three data chunks $\{D_5, D_2, D_3\}$

$$\begin{bmatrix} P_2 \\ P_3 \end{bmatrix} = G_{2\times3} \times \begin{bmatrix} 0 \\ D_2 \\ D_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 \\ D_2 \\ D_3 \end{bmatrix}$$
$$\begin{bmatrix} P'_2 \\ P'_3 \end{bmatrix} = G_{2\times3} \times \begin{bmatrix} D_5 \\ D_2 \\ D_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} D_5 \\ D_2 \\ D_3 \end{bmatrix}.$$

Since $P_2 = D_2+D_3$, $P_3 = 2D_2+3D_3$, $P'_2 = D_5+D_2+D_3$, and $P'_3 = D_5 + 2D_2 + 3D_3$, we can compute $P'_2 = P_2 + D_5$ and $P'_3 = P_3 + D_5$. So in order to update P_2, P_3 into P'_2, P'_3, only the nonsuperimposed data chunk D_5 needs to be accessed. Similarly, updating P_0, P_1 into P'_0, P'_1 only requires to retrieve D_4. So, when converting from RS(2, 2) to RS(3, 2) using the ERS (AERS) code, only the nonsuperimposed data chunks D_4, D_5 need to be retrieved for parity chunks updates. Compared with the SRS code, ERS (AERS) reduces half number of data chunks transmitted during parity updating.

Through a more reasonable design of the encoding matrix and data placement strategy, AERS can further reduce the I/O operations and network traffic required for parity chunk updates under different types of redundancy converting, not just in the case of $d' > d, r' = r$. Fig. 2(c) is an example of the AERS code, where $d = 3, d' = 2, r' = r = 2$, and $d' < d$. We adopt several submatrices of size $r \times d'$ and a specific data placement policy for encoding. When d' cannot divide d, dummy chunks are added. For example, the encoding process of stripe s_1 is multiplying two (old) encoding submatrices $G_{2\times2}$ with three data chunks $\{D_3, D_4, D_5\}$, and the encoding process of stripe s'_1 is multiplying the (new) encoding matrix $G_{2\times2}$ with two data chunks $\{D_3, D_4\}$

$$\begin{bmatrix} P_2 \\ P_3 \end{bmatrix} = \mathbf{G}_{2\times2} \times \begin{bmatrix} D_3 \\ D_4 \end{bmatrix} + \mathbf{G}_{2\times2} \times \begin{bmatrix} 0 \\ D_5 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} D_3 \\ D_4 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 0 \\ D_5 \end{bmatrix}$$
$$\begin{bmatrix} P'_2 \\ P'_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} D_3 \\ D_4 \end{bmatrix}.$$

Since $P_2 = D_3+D_4+D_5$, $P_3 = D_3+2D_4+4D_5$, $P'_2 = D_3+D_4$, and $P'_3 = D_3 + 2D_4$, we can compute $P'_2 = P_2 + D_5$ and $P'_3 = P_3+4D_5$. So in order to update P_2, P_3 into P'_2, P'_3, only the nonsuperimposed data chunk D_5 needs to be accessed. Similarly, updating P_0, P_1 into P'_0, P'_1 only requires to retrieve D_2. Besides, $P'_4 = D_2 + D_5$ and $P'_5 = 3D_2 + 4D_5$, i.e., there is no new data transfer. So, when converting from RS(3, 2) to RS(2, 2) through the AERS code, only the nonsuperimposed data chunks D_2, D_5 need to be retrieved for parity chunks updates, which is the same as the ERS (AERS) code.

TABLE I
SUMMARY OF NOTATIONS

d, d'	number of data chunks in pre/post-converting stripes
r, r'	number of parity chunks in pre/post-converting stripes
α, α'	number of pre-converting stripes that can fit entirely in a post-converting stripe when $d' > d$ or $d' < d$
β, β'	number of pre-converting stripes that cannot fit entirely in a post-converting stripe when $d' > d$ or $d' < d$
$\bar{d}_i$	number of data chunks of the i pre-converting stripe are placed into a post-converting stripe
q	quotient of d' divided by d
q'	quotient of d divided by d'
γ	remainder of d' divided by d
γ'	remainder of d divided by d'
l	the least common multiple of d and d'
$gcd(d, d')$	the greatest common divisor of d and d'
a	the larger of d and d'
b	the smaller of d and d'
$\frac{l}{d}$	number of pre-converting stripes
$\frac{l}{d'}$	number of post-converting stripes

Summary: The motivating examples above show that the judicious design of the encoding matrix and layout of data chunks can increase the number of superimposed data chunks. If the position of a data chunk has not changed before and after converting, and its coding coefficients remain intact, we call it a superimposed data chunk; otherwise, we call it a non-superimposed data chunk. The core of the AERS code is to add or delete a small number of nonsuperimposed data chunks on the basis of the preconverting parity chunks when updating the post-converting parity chunks, which greatly reduces the I/O operation and network traffic. Besides, both SRS and ERS codes only consider the case of $d' > d, r' = r$, and they do not discuss whether the placement strategy is optimal. Our proposed AERS code can realize more types of redundancy converting, making it more practical. We also prove that AERS codes are the optimal redundancy converting solution that can reach a lower bound on the number of data chunks transmitted during redundancy converting, which is achieved by co-designing encoding matrix and data placement strategy.

III. ADVANCED ERS CODE

We propose the AERS code, which allows efficient redundancy converting under a broad spectrum of coding parameters. We elaborate the overview of the AERS code (Section III-A) and present the details of code construction, including data placement strategies and encoding matrices design (Section III-B). We further propose a redundancy converting solution based on the above code construction (Section III-C). Finally, we extend the code construction and redundancy converting mechanism to make it applicable to other erasure codes, such as LRC code [21] (Section III-D).

A. Overview

When using the AERS(d, r, d', r') code to convert hot into cold data, i.e., when $(d' + r'/d') < (d + r/d)$, we can maintain high storage efficiency and satisfactory performance for write-intensive I/O. Conversely, when converting cold to hot data, the reconstruction bandwidth can be reduced with minimal side effects on application I/Os. Table I lists relative important notations.

Both SRS and ERS codes only consider the converting of one group parameter, that is, (d, r, d', r'), where $d' > d, r' = r$, which is essentially converting hot data into cold data. In fact, this type of converting has the following five kinds of situations.

1) *Case* 1*:* Converting RS(d, r) into RS(d', r'), where $d' > d, r' = r$ (this case is same as the ERS code).
2) *Case* 2*:* Converting RS(d, r) into RS(d', r'), where $d' > d, r' > r$ and $[(d' + r')/d'] < [(d + r)/d]$.
3) *Case* 3*:* Converting RS(d, r) into RS(d', r'), where $d' > d, r' < r$.
4) *Case* 4*:* Converting RS(d, r) into RS(d', r'), where $d' = d, r' < r$.
5) *Case* 5*:* Converting RS(d, r) into RS(d', r'), where $d' < d, r' < r$ and $[(d' + r')/d'] < [(d + r)/d]$.

Besides hot-to-cold conversions, the case of cold data changing to hot data is also getting common. For example, the past deeds of some hot search stars, and the past related information of hot news. The AERS code considers not only the above five cases but also the following five cases of cold-to-hot conversion.

6) *Case* 6*:* Converting RS(d, r) into RS(d', r'), where $d' < d, r' = r$.
7) *Case* 7*:* Converting RS(d, r) into RS(d', r'), where $d' < d, r' < r$ and $[(d' + r')/d'] > [(d + r)/d]$.
8) *Case* 8*:* Converting RS(d, r) into RS(d', r'), where $d' < d, r' > r$.
9) *Case* 9*:* Converting RS(d, r) into RS(d', r'), where $d' = d, r' > r$.
10) *Case* 10*:* Converting RS(d, r) into RS(d', r'), where $d' > d, r' > r$ and $[(d' + r')/d'] > [(d + r)/d]$.

The values of r and r' only determine the addition or deletion of parity chunks and do not affect the layout of data chunks. When $r' > r$, we only need to read the corresponding data chunks to generate new parity chunks, those parts of I/O operations and network traffic are inevitable; when $r' < r$, we only need to delete $r' - r$ parity chunks after converting, without generating new I/O operations and network traffic. Therefore, the key of this work will focus on the converting of the parameters d and d'. Note that, when $d' = d$, the number of data chunks remains unchanged, and we only need to add or delete $|r' - r|$ parity chunks. So, we only consider the case of $d' > d$ and $d' < d$.

B. General Construction

In traditional systems, chunks and nodes are often mapped one by one, but AERS code, like SRS and ERS, stores d data chunks of RS code in d' node, breaking this mapping. First, AERS divides an object into l data chunks, denoted as $D_0, \ldots, D_{l-1}$, where l is the least common multiple (LCM) of d and d', i.e., $l = \text{lcm}(d, d')$. Then, it arranges the l data chunks into d logical columns, encoding every d data chunks in l to compute r parity chunks. Finally, AERS distributes l data chunks over d' nodes such that each node stores exactly (l/d') chunks, and distributes parity chunks over c nodes, where $c = \max(r, r')$. Since data chunks are physically distributed over d' nodes, the AERS code does not require data chunk

relocations when objects are converted from RS(d, r) into RS(d', r').

A reasonable data placement strategy can make more data chunks in the same position of the preconverting and post-converting stripe, which helps to reduce the communication overhead during converting. An adequately designed coding matrix can make the coding coefficients of the data chunks the same in the preconverting and post-converting stripe, which is also the key to reducing the communication overhead during converting. Next, we will introduce the data placement strategy and encoding strategy in detail.

1) Round-Robin-Based Data Layout: We first present the overall idea of the placement strategy, then describe the algorithm in detail, and finally demonstrate algorithm examples.

Overall Idea: The core idea behind data placement strategies is to maximize the amount of superimposed data chunks in the stripes before and after converting. To this end, we layout l data chunks into a 2-D array of size $(l/d') \times d'$, where a row in the array maps to a post-converting stripe. The maximum number of superimposed data chunks between the preconverting and the post-converting stripe is $\hat{d}$, where $\hat{d} = \min(d, d')$. If the number of superimposed data between the preconverting and the post-converting stripe is $\hat{d}$, we call it a same-row stripe; otherwise, we call it a cross-row stripe. Therefore, how to maximize the number of stripes with superimposed data chunks $\hat{d}$ is the key to minimizing conversion data traffic.

When $d' > d$, a row in the array can hold at most $\lfloor (d'/d) \rfloor$ same-row preconverting stripes, and a cross-row preconverting stripe containing γ data chunks. Since there are (l/d') rows in total, we have $\alpha = \lfloor (d'/d) \rfloor \times (l/d')$ same-row preconverting stripes in total, such that each of them is placed consecutively in d nodes in the same row (i.e., the number of superimposed data chunks is d), and there are $\beta = (l/d) - \alpha$ across-row stripes, such that the maximum number of superimposed data chunks between them and one post-converting stripe is at most γ, where $\gamma = d' \bmod d$.

When $d' < d$, because there are (l/d) preconverting stripes, each of them needs to be placed in at least $\lfloor (d/d') \rfloor$ rows, we can place $\alpha' = \lfloor (d/d') \rfloor \times (l/d)$ rows so that each of them contains consecutive d' data chunks of the preconverting stripe (i.e., the number of superimposed data chunks is d'). Since each preconverting stripe has been filled with $d' \times \lfloor (d/d') \rfloor$ data chunks, there are γ' data chunks remaining for each preconverting stripe, where $\gamma' = d \bmod d'$. Therefore, the maximum number of superimposed data chunks between the remaining γ' data chunks in a preconverting and a post-converting stripe is at most γ', and all remaining data chunks from the preconverting stripes will be placed in $\beta' = (l/d') - \alpha'$ rows (i.e., β' across-row stripes).

We can tolerate any single node failure by storing up to r chunks of arbitrary stripes in a node, since we can always retrieve at least d surviving chunks of the same stripe from other available nodes (except the failed one). In order to guarantee node-level fault tolerance, we place preconverting stripes into the array in a round-robin manner, which can ensure that the d data chunks are distributed into d nodes when $d' > d$, and γ' data chunks are distributed into γ' nodes when $d' < d$. Besides, we should make sure that $\lfloor (d/d') \rfloor + 1 \leq r$ when $d' < d$, so as to guarantee the node-level fault tolerance.

Algorithm 1 Round-Robin Data Placement Algorithm

1: **if** $d' > d$ **then**
2: **for** the i-th $(0 \leq i \leq \beta - 1)$ row **do**
3: Put $\lfloor \frac{d'}{d} \rfloor$ same-row stripes in it, with starting node ID $(i \times (d - \gamma)) \bmod d'$
4: Put γ data chunks of one cross-row stripe in the remaining γ empty positions.
5: **end for**
6: **for** the i-th $(\beta \leq i \leq \frac{l}{d'} - 1)$ row **do**
7: Put $\lfloor \frac{d'}{d} \rfloor$ same-row stripes in it, with starting node ID $((i - \beta + 1) \times \gamma) \bmod d'$
8: Put $d - \gamma$ data chunks of β cross-row stripes sequentially in the empty positions of each row.
9: **end for**
10: **end if**
11: **if** $d' < d$ **then**
12: **for** the i-th $(0 \leq i \leq \alpha' - 1)$ row **do**
13: Put each pre-converting stripes in $\lfloor \frac{d}{d'} \rfloor$ consecutive rows, with starting node ID 0.
14: **end for**
15: **for** $j = \alpha'$ to $\frac{l}{d'} - 1$ **do**
16: Put remaining γ' data chunks of each pre-converting stripe into the remaining β' post-converting stripes in the same way as for $d' > d$.
17: **end for**
18: **end if**

Algorithm Details: Algorithm 1 describes the placement strategy. Lines 1–10 show the placement strategy when $d' > d$. In order to ensure that the d data chunks are distributed into d nodes to guarantee node-level fault tolerance, we place the preconverting stripes into the array in a round-robin manner. Specifically, for the first β rows, we first put $\lfloor (d'/d) \rfloor$ same-row stripes in it, with starting node ID $i \times (d - \gamma) \bmod d'$, and then put γ data chunks of one cross-row stripe in the remaining γ empty positions (lines 2–5). For the remaining $(l/d') - \beta$ rows, we first put $\lfloor (d'/d) \rfloor$ same-row stripes in it, with starting node ID $((i - \beta + 1) \times \gamma) \bmod d'$, and then put $d - \gamma$ data chunks of β cross-row stripes sequentially in the vacant positions (lines 6–9). If there are no $d \times \lfloor (d'/d) \rfloor$ nodes after a starting node, we return to the first one on the current row and continue selecting positions backward until $d \times \lfloor (d'/d) \rfloor$ nodes are selected. Lines 11–18 show the placement strategy when $d' < d$. Specifically, for the first α' rows, we put each preconverting stripe in $\lfloor (d/d') \rfloor$ consecutive same-row stripes, with starting node ID 0 (lines 12–14). For the rest $(l/d') - \alpha'$ rows, we put remaining γ' data chunks of each preconverting stripe into the remaining post-converting stripes in the same way as for $d' > d$ (lines 15–17).

Example of Algorithm 1: We present an example of placement strategy for AERS (d, r, d', r') code in Fig. 3. Fig. 3 is a placement example of AERS$(4, 2, 5, 2)$ code, where $d' > d$. Since $\lfloor (d'/d) \rfloor = 1, \alpha = 4, \beta = 1$, and $\gamma = 1$, one same-row stripe can be placed in each of the first four rows of the array, and the respective starting node IDs are row numbers. Therefore, the number of superimposed data chunks between each preconverting stripe and the post-converting

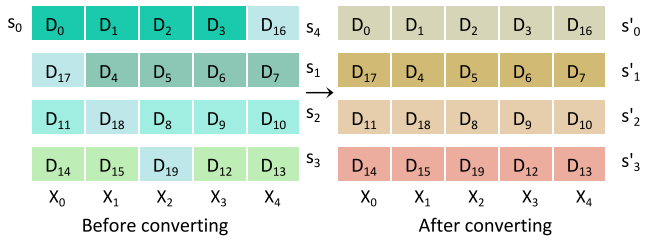


Fig. 3. Placement example for converting RS(4, 2) into RS(5, 2) by AERS(ERS). The data and parity chunks of the same color form a stripe.

stripe (i.e., a row) is maximized to $d = 4$, and there are four same-row stripes. The remaining one cross-row stripe is placed in the remaining one vacancy in each row in turn, so that the number of superimposed data chunks between it and the post-converting stripe is $\gamma = 1$. Note that the first data chunk of the third preconverting stripe is placed in the third node, and there are only two positions after that, which is smaller than $d \times \lfloor (d/d') \rfloor = 4 \times 1 = 4$. Therefore, it is necessary to go back to the first position of the row to place the remaining data chunk D_{11}. Similarly, it is necessary to return to the first position of the fourth row to place the data chunks D_{14} and D_{15} of the fourth preconverting stripe.

2) Encoding Strategy: We first illustrate the overall idea of the encoding strategy based on the placement strategy given above, then describe the algorithm details and finally discuss an example.

Overall Idea: We assume that l data chunks have been placed on d' nodes according to the data placement strategy shown in Algorithm 1. In physical storage, all data chunks form a 2-D array with the size of $(l/d') \times d'$.

When $d' > d$, we apply the extended matrix $G_{r\times d'}$ to encode each of the (l/d) preconverting stripes, each of which includes $d \times \lfloor (d'/d) \rfloor$ data chunks and γ zero chunks. The encoding matrix $G_{r\times d'}$ is still constructed from the Vandermonde matrix [22]. Note that zero chunks do not incur additional computational overhead.

When $d' < d$, we encode each of the (l/d) preconverting stripes using at least $\lfloor (d/d') \rfloor$ encoding submatrices of size $r \times d'$. When d cannot divide d', $d' - \gamma'$ zero chunks are added. For example, two zero chunks can be added to the AERS(6, 3, 4, 3) code. Then, parity generation for stripe s_i is reformulated

$$\begin{aligned}\mathbf{P}_i &= \sum_{q=0}^{\left\lfloor \frac{d}{d'} \right\rfloor} \mathbf{G}_{i,q,r\times d'} \times \mathbf{D}_{i,q} \\ &= \mathbf{p}_{i,0} + \mathbf{p}_{i,1} + \cdots + \mathbf{p}_{i,q-1} + \mathbf{p}_{i,q}\end{aligned}$$

where $\mathbf{G}_{i,q,r\times d'}$ refers to the qth encoding submatrix of size $r \times d'$ of the ith stripe, $\mathbf{D}_{i,q}$ is the qth data chunks set of size $d' \times 1$ in the ith stripe, and $\mathbf{p}_{i,q}$ represents the intermediary parity chunks.

Algorithm Details: We apply matrix $G_{r\times d'}$ for encoding (line 1). Lines 2–5 use the node ID to represent the id of the data chunk stored in it, to ensure that the coefficients of superimposed data are the same. For example, in Fig. 4, D_0, D_1, D_2, D_3, and D_4 are stored on X_0, X_1, X_2, X_3, and X_0, so the node ids of D_0, D_1, D_2, D_3, and D_4 are 0, 1, 2, 3, and 0, respectively.

Algorithm 2 Encoding Method of Parity Chunks

1: Select $G_{r\times d'}$ for encoding
2: **for** each stripe $s_i (0 \le i \le \frac{l}{d} - 1)$ **do**
3: **for** each data chunk $D_j (0 \le j \le d - 1)$ **do**
4: Set its ID in the data vector as its node id
5: **end for**
6: **if** $d' > d$ **then**
7: Add $d' - \gamma$ dummy chunks into the remaining $d' - d$ positions in the data vector to constitute d' chunks
8: $\mathbf{P}_i = \mathbf{G}_{i,r\times d'} \times \mathbf{D}_i$
9: **end if**
10: **if** $d' < d$ **then**
11: Add $d' - \gamma'$ dummy chunks into the remaining $d' - \gamma'$ positions in the data vector to constitute d' chunks
12: $\mathbf{P}_i = \sum_{q=0}^{\lfloor \frac{d}{d'} \rfloor} \mathbf{G}_{i,q,r\times d'} \times \mathbf{D}_{i,q} = \mathbf{p}_{i,0} + \cdots + \mathbf{p}_{i,q-1} + \mathbf{p}_{i,q}$
13: **end if**
14: **end for**

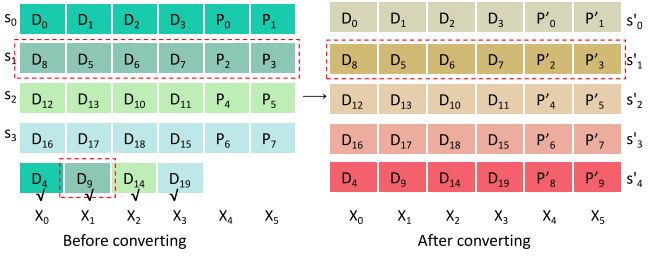


Fig. 4. Encoding example of AERS(5, 2, 4, 2) code. The chunks with black check marks indicate the nonsuperimposed data chunks.

Lines 6–9 of Algorithm 2 describe the detailed procedure of encoding the preconverting stripes with an extended encoding matrix in the case of $d' > d$. There are already $d \times \lfloor (d'/d) \rfloor$ data chunks in the preconverting stripe s_i, and on this basis, γ empty chunks are added to form d' data chunks, and then encode the d' chunks by $G_{r\times d'}$ (line 8).

Lines 10–13 of Algorithm 2 describe the detailed procedure of encoding the preconverting stripes with several submatrices in the case of $d' < d$. We use $\lfloor (d/d') \rfloor + 1$ $G_{r\times d'}$ to encode intermediary parity chunks, then add them together to form the final parity chunks (line 12). As described in the above, when d cannot divide d', $d' - \gamma'$ zero chunks are appended (line 11).

Example of Algorithm 2: Fig. 4 shows the encoding procedure of the AERS(5, 2, 4, 2) code. Since $r = r' = 2, d' = 4, d' - \gamma' = 3$, the size of the encoding submatrix $\mathbf{G}$ is 2×4, and three zero chunk needs to be added. Let us also take the encoding procedure of stripe s_1 as an example. The node ids of the five data chunks D_5, D_6, D_7, D_8, and D_9 of s_1 are 1, 2, 3, 0, and 1, respectively. The encoding procedure is shown as follows:

$$\begin{bmatrix} P_2 \\ P_3 \end{bmatrix} = \mathbf{G}_{2\times 4} \times \begin{bmatrix} D_8 \\ D_5 \\ D_6 \\ D_7 \end{bmatrix} + \mathbf{G}_{2\times 4} \times \begin{bmatrix} 0 \\ D_9 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 \end{bmatrix} \times \begin{bmatrix} D_8 \\ D_5 \\ D_6 \\ D_7 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 & 1 \\ 13 & 16 & 20 & 23 \end{bmatrix} \times \begin{bmatrix} 0 \\ D_9 \\ 0 \\ 0 \end{bmatrix}.$$

C. Redundancy Converting Strategy

Overall Idea: Redundancy converting changes the layout of stripes, which in turn leads to data chunk relocation and parity update. As the data chunks of AERS(d, r, d', r') are stored in d' nodes, converting RS(d, r) into RS(d', r') incurs no data chunk relocation. Our key idea is that the update of the new parity chunks (after converting) can reuse the old parity chunks (before converting), as both types of parity chunks often share the same encoding operations for the superimposed data chunks. Based on this insight, we now explore the converting procedure using the code structure proposed above. Specifically, when applying AERS(d, r, d', r') to convert hot data to cold data, where $(d' + r'/d') < (d + r/d)$, we perform redundancy converting based on the extended encoding matrix and the corresponding data placement strategy, and when applying AERS(d, r, d', r') to convert cold data to hot data, i.e., $(d' + r'/d') > (d + r/d)$, we perform redundancy converting based on encoding submatrices and another data placement strategy. Let $\mathbf{P}_i (0 \le i \le (l/d) - 1)$ denote the (old) parity chunks vector for stripe s_i, where $\mathbf{P}_i = [P_{i\times r}, \ldots, P_{i\times r+r-1}]^T$; $\mathbf{P}'_i (0 \le i \le (l/d') - 1)$ denote the (new) parity chunks vector for $s_{i'}$, where $\mathbf{P}'_i = [P'_{i\times r}, \ldots, P'_{i\times r+r-1}]^T$. For example, for AERS$(5, 2, 4, 2)$, $\mathbf{P}_0 = [P_0, P_1]^T$, and $\mathbf{P}'_0 = [P'_0, P'_1]^T$.

Algorithm Details: Algorithm 3 describes the procedure of redundancy converting. Lines 1–3 initialize all new parity chunks $\mathbf{P}'_j$ with a value of 0. Lines 4–11 are the procedure of redundancy converting when $d' > d$. Lines 5–10 provide the converting details of each preconverting stripe. We first select the $\lfloor (d'/d) \rfloor$ same-row preconverting stripes $s_i (i \in [0, \lfloor (d'/d) \rfloor - 1])$ with d superimposed data chunks with the post-converting stripe s'_j, and updates $\mathbf{P}'_j$ with s_i's parity chunks $\mathbf{P}_i (i \in [0, \lfloor (d'/d) \rfloor - 1])$ (line 6). Then select those data chunks that belong to s'_j but not s_i, and further update $\mathbf{P}'_j$ (line 8). Note that, $\mathcal{D}_x$ are the nonsuperimposed data chunks sets, and the number of data chunks in each set is $x (x \in \{d-\gamma, \gamma\})$. Specifically, for the first β rows, $x = d - \gamma$, and for the remaining $\lfloor (l/d') \rfloor - \beta$ rows, $x = \gamma$.

Lines 12–32 are the procedure of redundancy converting when $d' < d$ and $\lceil (d/d') \rceil \le r$. Each preconverting stripe s_i is placed in at least $\lceil (d/d') \rceil$ rows. So, there are $\alpha' = (l/d) \times \lfloor (d/d') \rfloor$ same-row post-converting stripes, where their parity chunks are $\mathbf{P}'_{i\times q}, \mathbf{P}'_{i\times q+1}, \cdots, \mathbf{P}'_{i\times q+q-1} (0 \le i \le (l/d) - 1, q = \lfloor (d'/d) \rfloor)$, and $\beta = (l/d') - \alpha'$ cross-row post-converting stripes, where their parity chunks are $\mathbf{P}'_{\alpha'}, \mathbf{P}'_{\alpha+1}, \ldots,$ and $\mathbf{P}'_{l/d'-1}$. Set $d = qd' + \gamma' = d' + (q-1)d' + \gamma'$. When $q - 1 = 0$, then $d = d' + \gamma'$. For same-row post-converting stripes, if $\mathbf{P}'_{i\times q}$ is directly computed, d' data chunks need to be transmitted, and if $\mathbf{P}'_{i\times q}$ is updated with the parity chunk $\mathbf{P}_i$ of s_i, $\gamma' (0 \le \gamma \le d')$ data chunks need to be transmitted. Therefore, in order to reduce the transmission of data chunks, we choose to update $\mathbf{P}'_{i\times q}$ with $\mathbf{P}_i$ (line 15); for cross-row post-converting stripes, we directly read all data chunks in the current stripe to compute new parity chunks (lines 17–19). Note that this part of the data chunks has already been read when the parity chunks of the same-row post-converting stripes are updated, so no new data chunks need to be transmitted. When $1 \le q - 1 \le r - 2$, then $d = d' + (q-1)d' + \gamma'$. For same-row post-converting stripes, if $\mathbf{P}'_{i\times q}, \mathbf{P}'_{i\times q+1}, \ldots,$ and $\mathbf{P}'_{i\times q+q-1}$ are directly computed, d' data chunks need to be transmitted, respectively. If $\mathbf{P}'_{i\times q}, \mathbf{P}'_{i\times q+1}, \ldots,$ and $\mathbf{P}'_{i\times q+q-1}$ are updated, respectively, by the parity chunks $\mathbf{P}_i$ of s_i, then we need to transmit $(q-1) \times d' + \gamma', (q-2) \times d' + \gamma', \ldots,$ and γ' data chunks, respectively. Therefore, in order to minimize data chunk transfer, we choose to directly read the corresponding data chunks to compute $\mathbf{P}'_{i\times q}, \mathbf{P}'_{i\times q+1}, \ldots,$ and $\mathbf{P}'_{i\times q+q-2}$ (lines 22–24), and update $\mathbf{P}'_{i\times q+q-1}$ with $\mathbf{P}_i$ (lines 25 and 26). For cross-row post-converting stripes, the operation is the same as when $q - 1 = 0$ (lines 28–30).

Example of Algorithm 3 ($d' > d$): Fig. 5 shows the procedure of converting RS$(4, 2)$ code to RS$(5, 2)$ using AERS$(4, 2, 5, 2)$ code. Let us take the update of $\mathbf{P}'_1$ as an example. Since there are the most superimposed data chunks between the preconverting stripes s_1 and post-converting s'_1,

Algorithm 3 Procedure of Redundancy Converting

1: **for** $j = 0$ to $\frac{l}{d'} - 1$ **do**
2: Initialize $\mathbf{P}'_j$ to be zero vector
3: **end for**
4: **if** $d' > d$ **then**
5: **for** each pre-converting stripe $s_i (0 \le i \le \frac{l}{d} - 1)$ **do**
6: Set $\mathbf{p}'_j = \mathbf{p}'_j + \mathbf{p}_i$
7: **for** non-superimposed data chunk sets $\mathcal{D}_x, x \in \{d-\gamma, \gamma\}$ **do**
8: Set $\mathbf{p}'_j = \mathbf{p}'_j + \mathbf{G}_{r\times d'} \times \mathcal{D}_x$
9: **end for**
10: **end for**
11: **end if**
12: **if** $d' < d$ **then**
13: **if** $q - 1 = 0$ **then**
14: **for** each post-converting stripe $s_j (0 \le i \le \alpha' - 1)$ **do**
15: Set $\mathbf{p}'_j = \mathbf{p}_i + \mathbf{G}_{i,q,r\times d'} \times \mathbf{D}_{i,q}$
16: **end for**
17: **for** each post-converting stripe $s_j (\alpha' \le j \le \frac{l}{d'} - 1)$ **do**
18: Set $\mathbf{p}'_j = \mathbf{G}_{j-d',q,r\times d'} \times \mathcal{D}_{d'}$
19: **end for**
20: **else**
21: **for** each pre-converting stripe $s_i (0 \le i \le \frac{l}{d} - 1)$ **do**
22: **for** each $y, 0 \le y \le q - 2$ **do**
23: Set $\mathbf{p}'_{i\times q+y} = \mathbf{G}_{i,y,r\times d'} \times \mathbf{D}_{i,y}$
24: **end for**
25: Set $\mathbf{p}'_{i\times q+q-1} = \mathbf{p}_i + \sum_{y=0}^{q-2} \mathbf{p}'_{i\times q+y}$
26: Set $\mathbf{p}'_{i\times q+q-1} = \mathbf{p}'_{i\times q+q-1} + \mathbf{G}_{i,q,r\times d'} \times \mathbf{D}_{i,q}$
27: **end for**
28: **for** each post-converting stripe $s_j (\alpha' \le j \le \frac{l}{d'} - 1)$ **do**
29: Set $\mathbf{p}'_j = \mathbf{G}_{j-qd',q,r\times d'} \times \mathcal{D}_{d'}$
30: **end for**
31: **end if**
32: **end if**

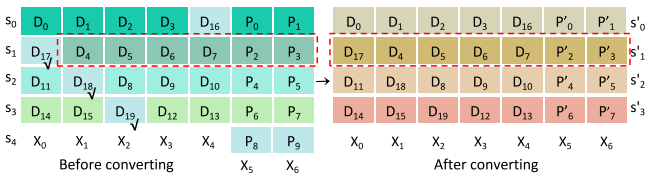


Fig. 5. Redundancy converting example for AERS(4, 2, 5, 2) code.

first update $\mathbf{P}'_1$ with the parity chunks $\mathbf{P}_1$ of s_1, then, since D_{17} belongs to s'_1 but not to s_1, continue to use D_{17} to update $\mathbf{P}'_1$. Thus, in the end, the value of $\mathbf{P}'_1$ is

$$\mathbf{P}'_1 = \begin{bmatrix} P'_2 \\ P'_3 \end{bmatrix} = \begin{bmatrix} P_2 \\ P_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 14 \end{bmatrix} \times D_{17}.$$

D. Extension

Although AERS mainly focuses on RS(d, r) codes, we show that it can be readily extended to other representative codes such as LRC [21]. The LRC(d, r, r_l) adds a set of local parity chunks r_l, which are generated by dividing the data chunks into a set of groups and then performing bit-wise XOR on the data chunks in each group. When repairing a failure chunk, it only needs to read the parity chunk and the surviving data chunks in the group, which can reduce the repair cost compared with RS. When the data chunks are updated due to redundancy converting, both the corresponding local parity chunks and the global parity chunks need to be updated. Interestingly, no matter which type of parity chunk of LRC is updated, the update method is the same as that of the parity chunks update of the RS code, which is equivalent to two sets of redundancy converting with AERS.

IV. Theoretical Analysis

Theorem 1 (Redundancy Converting Optimality): The AERS code is an optimal redundancy converting mechanism, which can reach the lower bound on the number of data chunks transmitted during redundancy converting. Specifically, when $d' > d$, the lower bound is $T_{\text{data}}(d' > d) = \beta \times (d - \gamma)$, otherwise, the lower bound is $T_{\text{data}}(d' < d) = (l/d) \times (d - d')$.

Proof: See Appendix A. ■

Corollary 1: When $d' > d$, the lower bound on the number of data chunks transmitted during redundancy converting is $\mathcal{T}_{\text{data}}(d' > d) = \beta \times (d - \gamma)$. Its minimum value is 0, and the maximum value is $\gamma \times (d - \gamma)$. When $d' < d$, the lower bound is $\mathcal{T}_{\text{data}}(d' < d) = (l/d) \times (d - d')$. Its minimum value is $d - d'$, and the maximum value is $d' \times (d - d')$.

Proof: See Appendix B. ■

Corollary 2: When the values of d and d' are interchanged and $r = r'$, we deduced that $\mathcal{T}_{\text{data}}(d' > d) \leq \mathcal{T}_{\text{data}}(d' < d)$ and $\mathcal{T}_{\text{data}}(d' = qd) < \mathcal{T}_{\text{data}}(\gcd(d' > d) \neq 1) < \mathcal{T}_{\text{data}}(\gcd(d' > d) = 1) \leq \mathcal{T}_{\text{data}}(d = q'd') < \mathcal{T}_{\text{data}}(\gcd(d' < d) \neq 1) < \mathcal{T}_{\text{data}}(\gcd(d' < d) = 1)$.

Proof: See Appendix C. ■

Theorem 2 (Node-Level Fault Tolerance): Our scheme can guarantee node-level fault tolerance when $d' > d$, or when $d' < d$ and $\lfloor (d/d') \rfloor + 1 \leq r$.

Proof: Wu et al. [23] have proved the single-node fault tolerance when $d' > d$, and guarantees that the d data chunks of any preconverting stripe are stored on d nodes, so a single node only stores one data chunk of a single stripe. We further prove that single-node fault tolerance can also be guaranteed when $d' < d$ and $\lfloor (d/d') \rfloor + 1 \leq r$. Obviously, each of the same-row stripes is distributed on d' nodes, and each node stores $\lfloor (d/d') \rfloor < r$ data chunks of a single stripe, so it meets the fault tolerance requirements. And because the remaining γ' data chunks of each preconverting stripe are placed in the manner when $d' > d$, so the remaining data chunks of the preconverting stripe are only placed in each node once. To sum up, a single node stores at most $\lfloor (d/d') \rfloor + 1 \leq r$ data chunks of a single stripe. So, even if a node fails, we can still recover the data chunks in it using the d available data chunks of the remaining nodes. So it can be concluded that our scheme can guarantee the node-level fault tolerance. ■

V. Performance Evaluation

We conduct numerical analysis and a series of empirical studies to demonstrate the performance gain in redundancy converting of AERS code over SRS code. To compare with AERS, we simultaneously analyze and implement SRS under various redundancy converting cases.

A. Numerical Analysis

We count the amount of traffic transmitted for parity chunk updates when using SRS and AERS codes to convert RS(d, r) to RS(d', r'), respectively. Then analyze its trend as the parameter changes.

SRS Code: For the SRS code, since the RS code with different parameters uses encoding matrices of different dimensions, the probability that the data chunks and their encoding coefficients in the post-converting stripe are the same as preconverting is almost zero. Therefore, we compute $(l/d') \times r$ new parity chunks by directly reading and transmitting all data chunks, which requires to transmit l chunks in total, i.e., $T_{\text{total}}(SRS) = l$.

AERS Code: The AERS code generates new parity chunks by reading and transmitting old parity and nonsuperimposed data chunks. We counted the number of old parity chunks and nonsuperimposed data chunks that need to be read and transmitted under various redundancy converting cases (defined in Section III-A), as shown in Table II.

Analysis of Actual Parameters: Here, parameters applied in the actual systems are taken into account, e.g., $(d, r) = (10, 4)$ in Facebook HDFS [24], $(12, 4)$ in Microsoft Azure [21]. Furthermore, we consider parameters with small d, r, d' and r', for instance, $(d, r, d', r') = (2, 1, 3, 1)$. We replace (d, r, d', r') with (d, r, d') for analyze the case of $r = r'$.

Fig. 6 shows the statistical results of data chunk traffic during redundancy converting for 15 different sets of encoding parameters. According to the statistical results in Table II, the data chunk traffic has nothing to do with r, so (d, r, d') is further abbreviated as (d, d') in Fig. 6. An important observation is that the judicious selection of AERS parameters before and after redundancy converting substantially reduces the amount

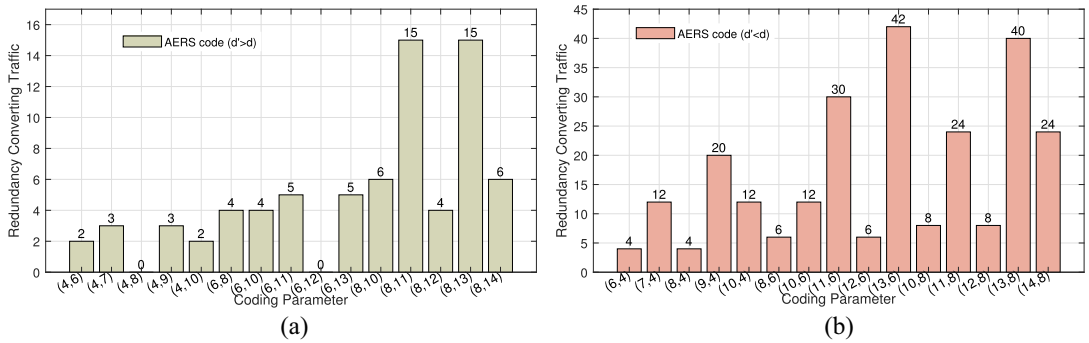


Fig. 6. Numerical results of the number of data chunks transmitted by AERS code under redundancy converting. (a) Convert RS(d, r) into RS(d', r') code, where $d' > d$. (b) Convert RS(d, r) into RS(d', r') code, where $d' < d$.

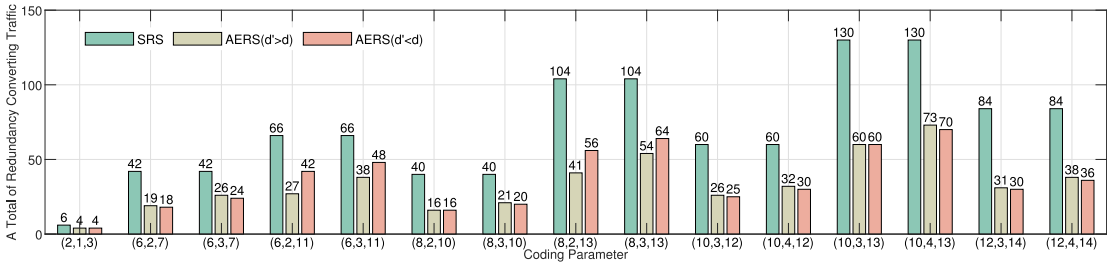


Fig. 7. Numerical results on the total number of chunks transmitted by various erasure codes under redundancy converting.

TABLE II
NUMBER OF CHUNKS THAT NEED TO BE READ AND TRANSMITTED UNDER VARIOUS CONVERTING CASES

	Setting	SRS	AERS
hot→ cold	Case 1	l	$r \times \frac{l}{d} + (\frac{l}{d} - \lfloor \frac{d'}{d} \rfloor \times \frac{l}{d'}) \times (d - r)$
	Case 2	l	l
	Case 3	l	$r' \times \frac{l}{d} + (\frac{l}{d} - \lfloor \frac{d'}{d} \rfloor \times \frac{l}{d'}) \times (d - r)$
	Case 4	0	0
	Case 5	l	$r' \times \frac{l}{d} + (\frac{l}{d} - \lfloor \frac{d'}{d} \rfloor \times \frac{l}{d'}) \times (d - r)$
cold→ hot	Case 6	l	$r \times \frac{l}{d} + \frac{l}{d} \times (d - d')$
	Case 7	l	$r' \times \frac{l}{d} + \frac{l}{d} \times (d - d')$
	Case 8	l	l
	Case 9	l	l
	Case 10	l	l

of redundancy converting traffic. In particular, the AERS code is more suitable for the parameters with even numbers, especially for $d' = qd$, or $d = q'd'$. For example, the number of traffic is 0 and 18 when converting from RS(6, 3) to RS(12, 3) and from RS(12, 3) to RS(6, 3), respectively. But the number of traffic is 15 and 126 when converting from RS(6, 3) to RS(13, 3) and from RS(13, 3) to RS(6, 3), respectively. Compared with the latter, the former saves 100% and 85.7% of the data chunk traffic, respectively. In the above-mentioned practical application scenarios, d are all set to even numbers, such as 6, 8, and 10, which indicates that the AERS code meets the actual requirements.

Fig. 7 shows the statistical results of total chunk traffic during redundancy converting, for 15 different sets of encoding parameters. From Fig. 7, we can observe that:

1) The AERS code transmits significantly fewer chunks than the SRS code during redundancy converting. For example, for $(d, r, d') = (12, 3, 14)$, the AERS code achieves 63.1% saving over SRS code, and 64.3% for $(d, r, d') = (14, 3, 12)$.
2) In some cases, AERS($d' > d$) has the same number of chunks read as AERS($d' < d$). Such as $(d, r, d') = (8, 2, 10)$ for AERS($d' > d$), and $(d, r, d') = (10, 2, 8)$ for AERS($d' < d$).
3) With other parameters unchanged, the number of chunks read and transmitted by AERS($d' > d$) and AERS($d' < d$) increases linearly with r, while the number of chunks read and transmitted by SRS (i.e., l) remains unchanged. Thus, the smaller r, the greater the improvement of AERS over SRS. e.g., AERS(10, 3, 13) and AERS(13, 3, 10) reduce the number of chunks read and transmitted of SRS by 53.8%, while AERS(10, 4, 13) and AERS(13, 4, 10) reduce the number of chunks read and transmitted of SRS by 43.8% and 46.2%.

Analysis of General Parameters: We now consider more parameters and see how AERS behaves under general parameters. We set $2 \leq d, d' \leq 14$, $1 \leq r, r' \leq 4$ and $r < d, r' < d'$, and there are a total of 244, 145, 311, 58, 250, 244, 81, 58, 58, and 145 sets of parameters in Cases 1–10, respectively. Note that simulations carried out using an odd value of r and r' are for proven concept.

Table III compares SRS and AERS in terms of reliability, storage cost, repair cost, and converting cost, and then discusses the changing trend of various performances before and after redundancy converting. "=" indicates that AERS and SRS perform the same after converting. "<=" means that AERS has less converting overhead than SRS. "−, ↑, ↓"

TABLE III
COMPARISONS OF AERS AND SRS UNDER VARIOUS REDUNDANCY CONVERTING CASES

	Case	reliability	storage-cost	repair-cost	converting-cost
hot→ cold	Case 1	$=,-$	$=,\downarrow$	$=,\uparrow$	$<$
	Case 2	$=,\uparrow$	$=,\downarrow$	$=,\uparrow$	$=$
	Case 3	$=,\downarrow$	$=,\downarrow$	$=,\uparrow$	$<$
	Case 4	$=,\downarrow$	$=,\downarrow$	$=,-$	$=$
	Case 5	$=,\downarrow$	$=,\downarrow$	$=,\downarrow$	$<$
cold→ hot	Case 6	$=,-$	$=,\uparrow$	$=,\downarrow$	$<$
	Case 7	$=,\downarrow$	$=,\uparrow$	$=,\downarrow$	$<$
	Case 8	$=,\uparrow$	$=,\uparrow$	$=,\downarrow$	$=$
	Case 9	$=,\uparrow$	$=,\uparrow$	$=,-$	$=$
	Case 10	$=,\uparrow$	$=,\uparrow$	$=,\uparrow$	$=$

indicate that the performance of AERS and SRS does not change, increases and decreases before and after converting, respectively. For example, "$=,\uparrow$" means that AERS and SRS have the same performance after converting, and the performance is increased compared to before converting. It can be seen from Table III that:

1) When redundancy converting is performed without changing the reliability (i.e., $r' = r$), the converting cost of AERS is smaller than that of SRS codes, such as Cases 1 and 6.
2) When redundancy converting is performed under increasing reliability (i.e., $r' > r$), the converting cost of AERS and SRS codes is equal, and both are l, such as Cases 2, 8, 9, and 10.
3) When redundancy converting is performed under the premise of reducing reliability (i.e., $r' < r$), the converting cost of AERS is less than or equal to that of SRS code, such as Cases 3, 4, 5, and 7.
   a) In general, when converting hot data into cold data, the storage cost of the system will decrease, while the repair cost will increase, such as in Cases 1, 2, 3, and 4. On the contrary, when converting cold data into hot data, the storage cost of the system will increase, and the repair cost will decrease, such as in Cases 6, 7, 8, and 9. However, there are also the following exceptions.
   b) The storage and repair costs in Case 5 are reduced simultaneously, which is at the expense of reliability.
   c) The storage and repair costs in Case 10 increase simultaneously but increases reliability.

Table IV compares ten cases of AERS and SRS from three aspects: 1) average ratio, the average ratio of the number of read and transmission chunks reduced by AERS compared to SRS; 2) the optimal ratio, the maximum ratio of the number of read and transmission chunks reduced by AERS compared to SRS; and 3) the best parameters, the parameters corresponding to the best ratio.

In all cases, AERS is better than or equal to SRS. Specifically, compared to SRS, AERS in Cases 1, 3, 5, 6, and 7 reduces the average number of chunks read and transmitted by 50.6%, 64.2%, 54.7%, 38.8%, and 28.9%, respectively, and at $(d, r, d', r') = \{(7, 1, 14, 1), (13, 1, 14, 1)\}$, $\{(7, 2, 14, 1), (7, 3, 14, 1), (7, 4, 14, 1), (13, 2, 14, 1), (13, 3, 14, 1), (13, 4, 14, 1)\}$, $\{(14, 4, 13, 1), (14, 3, 13, 1), (14, 2, 13, 1)\}$, $\{(14, 1, 13, 1)\}$ up to 85.7% and 50% at $(d, r, d', r') = \{(14, 4, 10, 3), (14, 3, 9, 2)\}$. In the remaining Cases 2, 8, 9, and 10, both AERS and SRS need to read l data chunks to generate new parity blocks, so both AERS and SRS read and transmit l data chunks. Furthermore, in Case 4, since $d' = d, r' < r$, both AERS and SRS only need to discard old $r - r'$ parity chunks, so no chunks need to be read and transmitted.

TABLE IV
COMPARISONS OF AERS AND SRS UNDER GENERAL PARAMETER ANALYSIS

	Case	average ratio	best ratio	best parameter
hot→ cold	Case 1	50.6%	85.7%	$(7, 1, 14, 1)(13, 1, 14, 1)$
	Case 2	0	0	
	Case 3	64.2%	85.7%	$(7, 2, 14, 1)(7, 3, 14, 1)$ $(7, 4, 14, 1)(13, 2, 14, 1)$ $(13, 3, 14, 1)(13, 4, 14, 1)$
	Case 4	0	0	
	Case 5	54.7%	85.7%	$(14, 4, 13, 1)(14, 3, 13, 1)$ $(14, 2, 13, 1)$
cold→ hot	Case 6	38.8%	85.7%	$(14, 1, 13, 1)$
	Case 7	28.9%	50%	$(14, 4, 10, 3)(14, 3, 9, 2)$
	Case 8	0	0	
	Case 9	0	0	
	Case 10	0	0	

B. Testbed Experiments

Setup: The AERS KV store prototype in our scheme adopts a proxy-based storage structure [23], which includes multiple clients and servers, as well as a proxy. The proxy is the entry point for clients to access objects stored in the servers, which implements the basic read and write operations, as well as the redundancy converting processes based on libmemcached-1.0.18. For comparison, we also implemented SRS code under various converting cases (defined in Section III-A) on the proxy. The AERS KV store prototype is deployed on a local cluster with ten nodes, each of which contains an Ubuntu 16.04 LTS with Intel Core i5-10210U, and 20-GB SCSI disk storage. The clients are deployed in one of the nodes and the servers are deployed in the remaining nodes.

Methodology: Unless otherwise specified, the following experiments follow the default settings. We set the object size to be 1 kB–4 MB and the network bandwidth to 10 Gb/s. Besides, we set AERS and SRS code parameters $(d, r, d', r') = \{(2, 1, 3, 1), (2, 1, 5, 2), (3, 2, 4, 1), (3, 2, 3, 1), (3, 2, 2, 1), (3, 1, 2, 1), (8, 3, 4, 2), (4, 1, 3, 2), (3, 1, 3, 2), (7, 1, 8, 2)\}$ in ten redundancy converting cases, respectively. On this basis, we measure the normal read/write time, recovery time, and redundancy converting time of an object when it is converted from RS(d, r) code into RS(d', r') code.

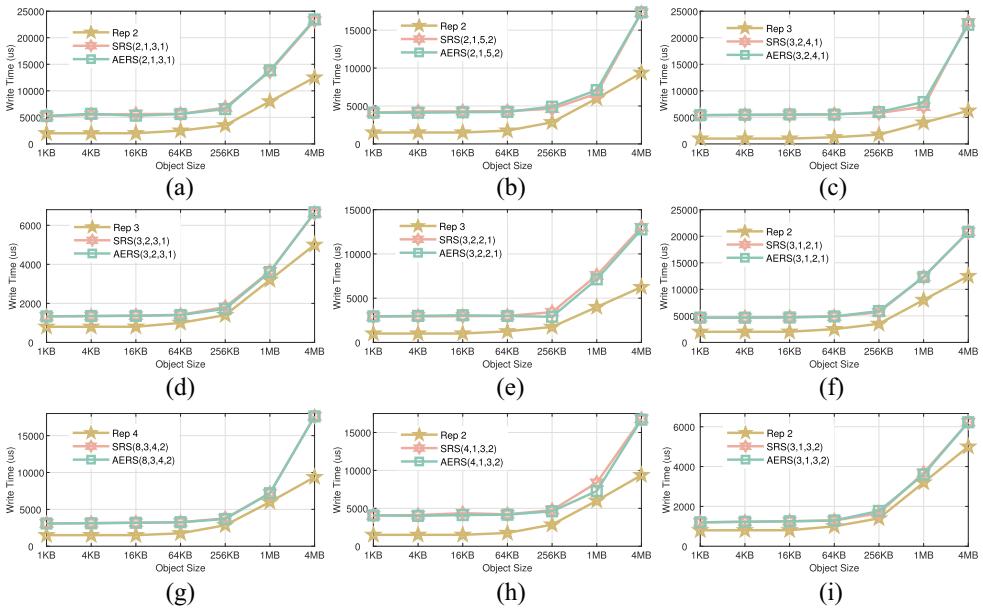


Fig. 8. Normal write time under different object sizes and coding parameters. (a) RS(2, 1) → RS(3, 1). (b) RS(2, 1) → RS(5, 2). (c) RS(3, 2) → RS(4, 1). (d) RS(3, 2) → RS(3, 1). (e) RS(3, 2) → RS(2, 1). (f) RS(3, 1) → RS(2, 1). (g) RS(8, 3) → RS(4, 2). (h) RS(4, 1) → RS(3, 2). (i) RS(3, 1) → RS(3, 2).

Combinations of multiple sets of settings were considered during the measurement to more comprehensively evaluate the performance gain of the AERS code. Note that, for the sake of brevity, we give the results of normal read time under the above 10 cases, and the results of normal write time, repair time, and redundancy converting time under only some of those cases.

Experiment 1 (Normal I/Os Latency With Different Object Sizes and Coding Parameters): We first evaluate the normal write and read performance of the AERS KV store prototype with SRS, as well as the default Memcached (denoted by Rep). To make Rep to tolerate the same number of failures as SRS(d, r, d', r') and AERS(d, r, d', r') codes, we set the number of copies of Rep as $r + 1$.

Figs. 8 and 9 show the evaluation results of the write and read time of Rep, SRS, and AERS codes under different object sizes and coding parameters. From Fig. 8, it can be seen that the write time of Rep, SRS, and AERS codes increases with the increase of object size. SRS and AERS codes have similar (slightly higher) write latency, and both higher than that of Rep. The write latency of SRS and AERS is higher than that of Rep because SRS and AERS codes have more connection overhead. Specifically, SRS and AERS need to connect to $d'+r$ servers and initiate $l + r \times (l/d)$ write requests to them, while Rep only needs to create two requests to two servers.

From Fig. 9, the read time of SRS and AERS codes and Rep also increases with the increase of object size, and the read latency of the SRS and AERS codes is also higher than that of Rep (e.g., with object size ≥ 1 MB).

Experiment 2 (Repair Latency at Various Object Sizes and Coding Parameters): Fig. 10 shows that the repair time of SRS and AERS codes and Rep under various cases also increases with the increase of object size. The repair delay of SRS and AERS codes is similar and higher than that of Rep. The reason

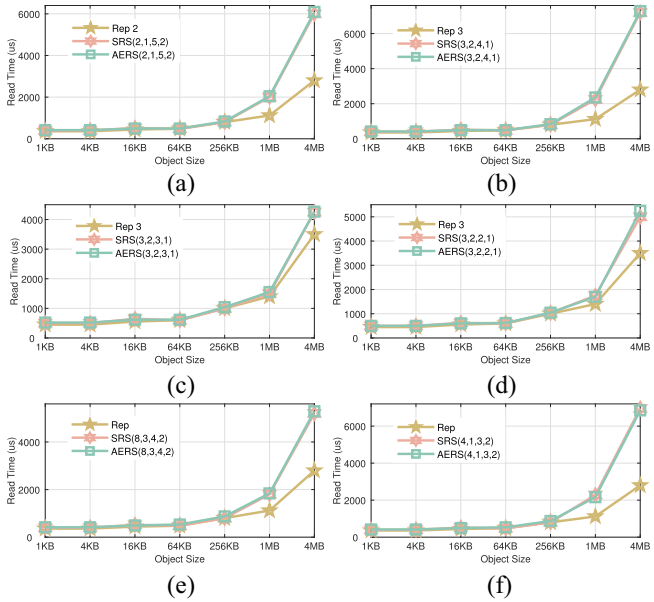


Fig. 9. Normal read time under different object sizes and coding parameters. (a) RS(2, 1) → RS(5, 2). (b) RS(3, 2) → RS(4, 1). (c) RS(3, 2) → RS(3, 1). (d) RS(3, 2) → RS(2, 1). (e) RS(8, 3) → RS(4, 2). (f) RS(4, 1) → RS(3, 2).

is that SRS and AERS need to connect to d (d') servers and initiate d read requests to them, while Rep only needs to create a request to one server.

Experiment 3 (Converting Latency at Various Object Sizes and Redundancy Converting Cases): We then evaluate the redundancy converting time from RS(d, r) to RS(d', r') for the first nine converting cases at different object sizes. From Fig. 11, it can be seen that the redundancy converting time of various redundancy converting cases increases with the increase of object size. The redundancy converting

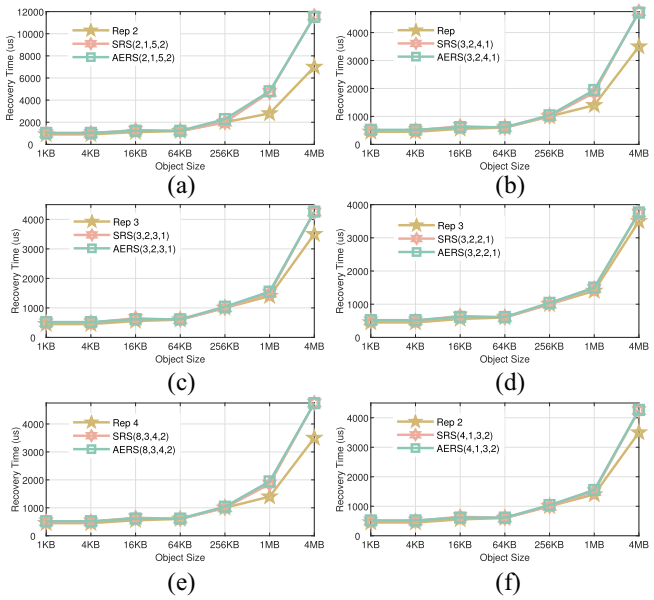


Fig. 10. Recovery time under different object sizes and coding parameters. (a) RS(2, 1) → RS(5, 2). (b) RS(3, 2) → RS(4, 1). (c) RS(3, 2) → RS(3, 1). (d) RS(3, 2) → RS(2, 1). (e) RS(8, 3) → RS(4, 2). (f) RS(4, 1) → RS(3, 2).

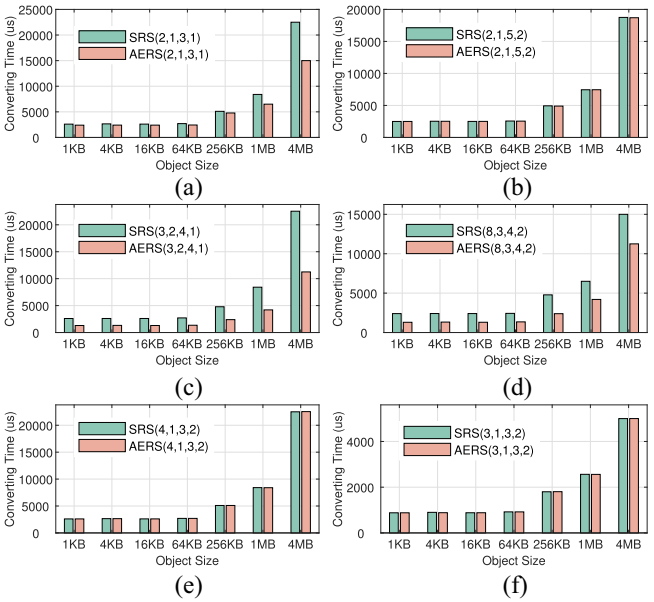


Fig. 11. Converting time under different object sizes and coding parameters. (a) RS(2, 1) → RS(3, 1). (b) RS(2, 1) → RS(5, 2). (c) RS(3, 2) → RS(4, 1). (d) RS(8, 3) → RS(4, 2). (e) RS(4, 1) → RS(3, 2). (f) RS(3, 1) → RS(3, 2).

performance of AERS(d, r, d', r') is better than or similar to that of SRS(d, r, d', r'). The reason is that the AERS code not only reduces the number of chunks that need to be read and transmitted and also the number of servers that need to be connected for converting is less. For example, in Fig. 11(a), AERS(2, 1, 3, 1) and SRS(2, 1, 3, 1) need to read and transmit 4 and 6 chunks and connect 2 and 4 servers, respectively. Note that Case 1 (AERS(2, 1, 3, 1)), Case 5 (AERS(3, 2, 2, 1)), and Case 6 (AERS(3, 1, 2, 1)) need to transmit the same number of chunks and connect to the same number of servers, so to save space, the textual and graphical representations of converting Cases 5 and 6 are omitted.

Experiment 4 (Generality): We use the LRC(d, r, r_l) code to verify the generality of AERS(d, r, d', r'). To show the difference, we renamed SRS(d, r, d', r') to SLRC$(d, r, r_l, d', r', r'_l)$, AERS$(d, r, d', r')$ to AELRC$(d, r, r_l, d', r', r'_l)$. we set code parameters $(d, r, r_l, d', r', r'_l) = \{(4, 1, 2, 6, 1, 2), (4, 1, 2, 6, 2, 2), (4, 2, 2, 6, 1, 2), (4, 2, 2, 4, 1, 2), (6, 2, 2, 4, 1, 1), (6, 1, 2, 4, 1, 2), (6, 2, 2, 4, 1, 2), (6, 1, 2, 4, 2, 2), (4, 1, 2, 4, 2, 2)\}$ in nine redundancy converting cases, respectively.

From Fig. 12, it also can be seen that the redundancy converting performance of AELRC$(d, r, r_l, d', r', r'_l)$ is also better than or similar to that of SLRC$(d, r, r_l, d', r', r'_l)$, and the redundancy converting time of various redundancy converting cases increases with the increase of object size. The reason is similar to converting RS(d, r) into RS(d', r'), so it will not be repeated. Note that Case 1 (AELRC(4, 1, 2, 6, 1, 2)) and Case 3 (AELRC(4, 2, 2, 6, 1, 2)), Case 2 (AELRC(4, 1, 2, 6, 2, 2)), Case 8(AELRC(6, 1, 2, 4, 2, 2)) and Case 9 (AELRC(4, 1, 2, 4, 2, 2)), Case 5 (AELRC(6, 2, 2, 4, 1, 1)), Case 6 (AELRC(6, 1, 2, 4, 1, 2)) and Case 7 (AELRC(6, 2, 2, 4, 1, 2)) need to transmit the same number of chunks and connect to the same number of server, so to save space, the textual and graphical representations of converting Cases 3, 6, 7, 8, and 9 are omitted.

Experiment 5 (Converting Latency Under More Coding Parameters in Cases 1 and 6): Next, We evaluate the redundancy converting time of SRS and AERS under different encoding parameters in Cases 1 and 6, where $r = r'$. Here, we set three groups of (d, r, d') parameter for SRS and AERS$(d' > d)$, which are $(2, 1, 3)$, $(4, 1, 5)$, and $(5, 1, 6)$, and set $(3, 1, 2)$, $(5, 1, 4)$, and $(6, 1, 5)$ for AERS$(d' < d)$. Additionally, we also set the two object sizes to 1 and 4 MB.

From Fig. 13, it can be seen that compared with SRS code, AERS greatly reduces the converting time, and the converting time of AERS$(d' > d)$ and AERS$(d' < d)$ are relatively close. According to the coding matrix and placement strategy designed in this work, when AERS is set to the above-mentioned groups of parameters, the amount of chunks transmitted by AERS$(d' > d)$ and AERS$(d' < d)$ conversion are the same, and are greatly lower than that of SRS code, verifying that AERS can significantly reduce converting time. For example, when convert RS(4, 1) into RS(5, 1), AERS can reduce converting time by 45% compared to SRS at object size with 4 MB.

Experiment 6 (Converting Latency Under Various Network Bandwidth): Then, we evaluate the converting performance under different bandwidths. Our test platform configures the bandwidth to be 1 and 10 Gb/s, respectively. We consider $(d, r, d') = \{(4, 1, 5), (5, 1, 6), (5, 1, 4), (6, 1, 5)\}$, and $r' = r = 1$, the object size is 4 MB. Fig. 14 shows the results.

We can see that AERS consistently outperforms SRS. For example, under 1-Gb/s network, for parameters $(d, r, d') = (4, 1, 5)$, AERS reduces the converting time by 46.9% compared to SRS, and for parameters $(d, r, d') = (6, 1, 5)$, AERS decreased by 44%. Besides, in the case of bandwidth constrained, i.e., bandwidth is 1 Gb/s, the improvement of

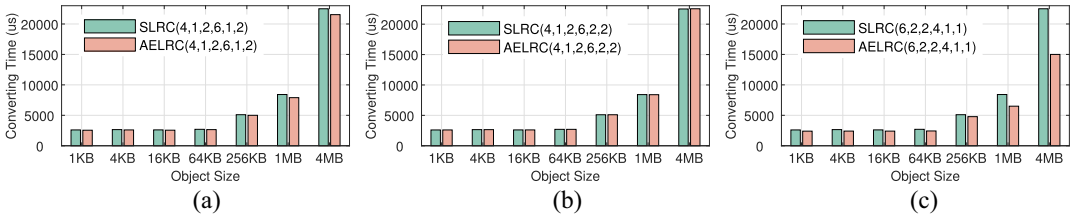


Fig. 12. Converting time from LRC(d, r, r_l) into LRC(d', r', r'_l) under different object sizes and coding parameters. (a) LRC$(4, 1, 2) \rightarrow$ LRC$(6, 1, 2)$. (b) LRC$(4, 1, 2) \rightarrow$ LRC$(6, 2, 2)$. (c) LRC$(6, 2, 2) \rightarrow$ LRC$(4, 1, 1)$.

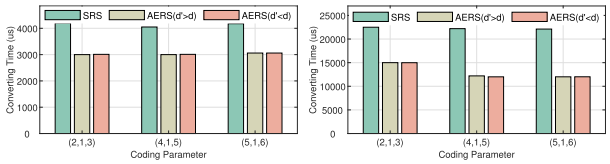


Fig. 13. Converting time under more coding parameters in Cases 1 and 6. (a) Object size 1 MB. (b) Object size 4 MB.

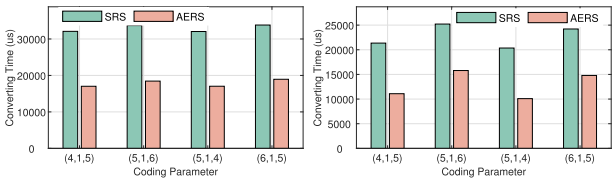


Fig. 14. Converting time under different network bandwidth. (a) Network bandwidth 1 Gb/s. (b) Network bandwidth 10 Gb/s.

AERS over SRS is larger. For example, for $(d, r, d') = (5, 1, 6)$, AERS reduces the converting time by 37.4% compared to SRS at 10 Gb/s while reducing 45.2% at 1 Gb/s.

VI. Related Work

Redundancy Converting: There have been several studies applying erasure codes with redundancy converting properties to different storage systems. Zebra [25], [26], LRC-HH [27], TEA [28], POST [29], Graftage [30], and Cascade [31] study the redundancy converting for distributed storage systems (DSSs), while EC-fusion [32] and STREAMDFP [33] study the redundancy converting for cloud storage systems (CSSs). All of those schemes employed different erasure codes or replication mechanisms to store popular and unpopular data, respectively, achieving a good balance between repair overhead and storage overhead. Repair-Scaling [34] studies the redundancy converting for the clustered system. In this work, we pay close attention to redundancy converting schemes in in-memory KV stores with high elastic requirements [35].

In particular, prior studies [13], [36], [37], [23] address the elasticity of erasure-coded in-memory KV stores. Chen et al. [13] and Xu et al. [36] employed replication and erasure coding separately for in-memory data sets to guarantee the reliability of popular and nonpopular data, and provide redundancy converting from replication to erasure coding. All of them depend on the reconstruction of the stripes, which leads to expensive parity chunk updates. Shen and Lee [38] considered the cross-rack update traffic in the data center. We utilize the joint design of the encoding matrix and placement strategy, which greatly reduces update traffic of the redundancy converting.

Wu et al. [23] and Taranov et al. [37] tradeoff storage overhead and access performance by studying data placement strategies. ERS [23] allows new parity chunks to be updated from old parity chunks plus a small number of nonsuperimposed data chunks, thus reducing I/O caused by parity updating. All the above studies do not consider the converting from cold to hot data. Our work is motivated by the scenario in ERS and focuses on reducing network traffic during redundancy converting with different types of code parameters, not just when $d' > d, r' = r$. We also further present a theoretical analysis on redundancy converting traffic.

Erasure Coding in KV Stores: Cocytus [13] is an in-memory KV store that utilizes replication and erasure codes, respectively, to ensure key and value reliability, thereby improving storage efficiency. Lai et al. [14] used KV separation in cloud storage, where keys and metadata are stored in a replicated LSM-tree, and values are stored separately in erasure coding to achieve low-redundancy fault tolerance. Shen et al. [15] designed an efficient memory caching scheme that can reduce disk input and output (I/O) latency, thereby improving the performance of erasure-coded storage systems. Cheng et al. [16] proposed a new erasure coding model that does not generate parity updates during scaling. Qin et al. [17] provided a high-performance, write-efficient, and update-friendly erasure coding scheme for KV-SSD. Besides, [20] considers load balancing when repairing, and [18] and [39] consider small-object encoding. Our work focuses on the tradeoff between storage overhead and access performance for large objects.

VII. Conclusion

This work presented the AERS code, extended from the original ERS code. The AERS code eliminates data chunk relocation, while minimizing I/O operations and data transmission caused by redundancy converting under a spectrum of parameters, by co-designing the encoding matrices and data placement strategies. We evaluated the AERS code through mathematical analysis and experiments. The results show that the AERS code significantly reduces network traffic during redundancy converting and the latency of redundancy converting, compared to the state-of-the-art alternatives. The

current work is oriented toward flat store systems, and cross-cluster bandwidth is not considered. We will consider clustered store systems in the future, where cross-cluster traffic is the bottleneck.

APPENDIX A
PROOF OF THEOREM 1

When $d' > d$, it is assumed that $\bar{d}_i$ $(1 \le \bar{d}_i \le d)$ data chunks of the i preconverting stripe are placed into a post-converting stripe and there are (l/d) preconverting stripes, then when updating a new parity chunk, $d - \bar{d}_i$ data chunks are to be transmitted. So a total number of $\mathcal{T}_{\text{data}}(d' > d) = \sum_{i=1}^{(l/d)} (d - \bar{d}_i)$ data chunks need to be transmitted. Due to $1 \le \bar{d}_i \le d$, so it can be seen from the formula that when $\bar{d}_i = d$, the transmission amount can be smaller. Therefore, we assume that there are α preconverting stripes where d data chunks are placed in one post-converting stripe, and the remaining $(l/d) - \alpha$ stripes are only can place $\bar{d}$ $(1 \le \bar{d} \le \gamma)$ in one post-converting stripe. Then, the formula can be rewritten as $\mathcal{T}_{\text{data}}(d' > d) = \sum_{i=1}^{\alpha}(d - d) + \sum_{i=1}^{\beta}(d - \bar{d}_i)$, where $\alpha + \beta = (l/d)$. Obviously, when $\bar{d}_i = \gamma$, the transmission amount is the smallest. So the formula can be further rewritten as $\mathcal{T}_{\text{data}}(d' > d) \ge \sum_{i=1}^{\beta}(d - \gamma) = \beta \times (d - \gamma)$. Similarly, when $d' < d$, it is assumed that $\bar{d}_i$ $(1 \le \bar{d}_i \le d')$ data chunks of the i preconverting stripes are placed into a post-converting stripe and there are (l/d) stripes, then when updating a new parity chunk, $d - \bar{d}_i$ data chunks are to be transmitted. So a total number of $\mathcal{T}_{\text{data}}(d' < d) = \sum_{i=1}^{(l/d)}(d - \bar{d}_i)$ data chunks need to be transmitted. Due to $1 \le \bar{d}_i \le d'$, so it can be seen from the formula that when $\bar{d}_i = d'$, the transmission amount is minimized. So the formula can be further rewritten as $\mathcal{T}_{\text{data}}(d' < d) \ge \sum_{i=1}^{(l/d)}(d - d') = (l/d) \times (d - d')$.

It can be seen from our Algorithms 1 and 3 that the nonsuperimposed data chunks only exist in the β cross-row stripes when $d' > d$, so the numbers of nonsuperimposed data chunks that need to be retrieved to update the parity chunks are $\beta \times (d - \gamma)$ when $d' > d$. When $d' < d$ and $q - 1 = 0$, it can be seen from Algorithm 3 that γ' data chunks of each preconverting stripe need to be read, so a total of $(l/d) \times \gamma' = (l/d) \times (d - qd') = (l/d) \times (d - d')$ data chunks need to be transmitted. When $d' < d$ and $q - 1 \ne 0$, it can be seen from Algorithm 3 that in the q after-converting stripes corresponding to each preconverting stripe, the parity chunks of $q - 2$ stripes are calculated by rereading the corresponding data chunks, and the parity chunks of one stripe only need to read γ' data chunks, and the last stripe does not need to read any new data chunks. So a total of $\sum_{i=0}^{(l/d)-1}\sum_{y=0}^{q-2}\mathbf{D}_{i,y} + \sum_{i=0}^{(l/d)-1}\mathbf{D}_{i,q} = (l/d) \times (q-1) \times d' + (l/d) \times \gamma' = (l/d) \times (d - d')$ data chunks need to be transmitted. So, when $d' < d$, the number of data chunks to transmit is $(l/d) \times (d - d')$.

APPENDIX B
PROOF OF COROLLARY 1

As we can known from Theorem 1 that $\beta = (l/d) - \alpha$. When $d' > d$, one row in the array can hold at most $\lfloor (d'/d) \rfloor$ preconverting stripes, such that each of them is placed consecutively in d nodes in the same row. Since there are (l/d') rows in total, so $\alpha = (l/d') \times \lfloor (d'/d) \rfloor$. Let $d' = qd + \gamma$, so $q = \lfloor (d'/d) \rfloor$, then $\beta = (l/d) - q \times (l/d') = (ld' - lqd/dd') = (l(d' - qd)/dd') = (l\gamma/dd')$. According to the relationship between the greatest common divisor and the LCM, i.e., $d \times d' = l \times \gcd(d, d')$, where $\gcd(d, d')$ is the greatest common divisor of d and d', the formula can be further rewritten as $\beta = (l\gamma/dd') = (\gamma/\gcd(d, d'))$. According to the property of the greatest common divisor, β can be readily calculated as follows:

$$\beta = \frac{\gamma}{\gcd(d, d')} = \begin{cases} \gamma, & \gcd(d, d') = 1 \\ 0, & d' = qd \\ (0, \gamma), & \text{others.} \end{cases}$$

It can be seen from the above equation that $\beta \in [0, \gamma]$, and when $d' = qd$, i.e., when d' is divisible by d, the β value is the smallest, which is 0; and when $\gcd(d', d) = 1$, the β value is the largest, which is γ. So, when $d' > d$, the minimum value of $\mathcal{T}_{\text{data}}(d' > d)$ is $\mathcal{T}_{\text{data}}(d' > d) = \beta \times (d - \gamma) = 0$, where $\beta = 0$, and the maximum value of $\mathcal{T}_{\text{data}}(d' > d)$ is $\mathcal{T}_{\text{data}}(d' > d) = \beta \times (d - \gamma) = \gamma \times (d - \gamma)$, where $\beta = \gamma$.

When $d' < d$, let $d = q'd' + \gamma'$. According to the relationship between the greatest common divisor and the LCM, i.e., $d \times d' = l \times \gcd(d, d')$, so $(l/d) = ([d \times d']/[d \times \gcd(d, d')]) = (d'/\gcd(d, d'))$. Similarly, according to the property of the greatest common divisor, (l/d) can be readily calculated as follows:

$$\frac{l}{d} = \frac{d'}{\gcd(d, d')} = \begin{cases} d', & \gcd(d, d') = 1 \\ 1, & d = q'd' \\ (1, d'), & \text{others} \end{cases}.$$

It can be seen from the above equation that $(l/d) \in [1, d']$, and when $d = q'd'$, i.e., when d is divisible by d', the (l/d) value is the smallest, which is 1; and when $\gcd(d', d) = 1$, the (l/d) value is the largest, which is d'. So, when $d' < d$, the minimum value of $\mathcal{T}_{\text{data}}(d' < d)$ is $\mathcal{T}_{\text{data}}(d' < d) = (l/d) \times (d - d') = d - d'$, where $(l/d) = 1$, and the maximum value of $\mathcal{T}_{\text{data}}(d' < d)$ is $\mathcal{T}_{\text{data}}(d' < d) = (l/d) \times (d - d') = d' \times (d - d')$, where $(l/d) = d'$.

APPENDIX C
PROOF OF COROLLARY 2

Set variables $a = \max(d, d') = qd + \gamma$, $b = \min(d, d')$, then $\mathcal{T}_{\text{data}}(d' < d) - \mathcal{T}_{\text{data}}(d' > d) = (d'/\gcd(d, d')) \times (d - d') - (\gamma/\gcd(d, d')) \times (d - \gamma) = ((q - 1)b^2 + \gamma^2/\gcd(d, d')) \ge 0$, so $\mathcal{T}_{\text{data}}(d' > d) \le \mathcal{T}_{\text{data}}(d' > d)$ (When $q = 1, \gamma = 0$, $\mathcal{T}_{\text{data}}(d' > d) = \mathcal{T}_{\text{data}}(d' < d)$). Besides, from Corollary 1, we know that $\mathcal{T}_{\text{data}}(d' = qd) = 0$, $\mathcal{T}_{\text{data}}\big(\gcd(d' > d) \ne 1\big) \in \big(0, \gamma \times (d - \gamma)\big)$, $\mathcal{T}_{\text{data}}\big(\gcd(d' > d) = 1\big) = \gamma \times (d - \gamma)$, so $\mathcal{T}_{\text{data}}(d' = qd) < \mathcal{T}_{\text{data}}(\gcd(d' < d) \ne 1) < \mathcal{T}_{\text{data}}(\gcd(d' > d) = 1)$; We can also know from Corollary 1 that $\mathcal{T}_{\text{data}}(d = q'd') = d - d'$, $\mathcal{T}_{\text{data}}(\gcd(d' < d) \ne 1) \in \big(d - d', d' \times (d - d')\big)$, $\mathcal{T}_{\text{data}}\big(\gcd(d' < d) = 1\big) = d' \times (d - d')$, so, $\mathcal{T}_{\text{data}}(d = q'd') < \mathcal{T}_{\text{data}}(\gcd(d' < d) \ne 1) < \mathcal{T}_{\text{data}}(\gcd(d' < d) = 1)$. Thus, we can conclude that $\mathcal{T}_{\text{data}}(d' = qd) < \mathcal{T}_{\text{data}}(\gcd(d' > d) \ne 1) < \mathcal{T}_{\text{data}}(\gcd(d' > d) = 1) \le \mathcal{T}_{\text{data}}(d = q'd') < \mathcal{T}_{\text{data}}(\gcd(d' < d) \ne 1) < \mathcal{T}_{\text{data}}(\gcd(d' < d) = 1)$.

REFERENCES

[1] R. Hecht and S. Jablonski, "NoSQL evaluation: A use case oriented survey," in *Proc. Int. Conf. Cloud Service Comput.*, 2011, pp. 336–341.

[2] L. Cheng et al., "LogECMem: Coupling erasure-coded in-memory key-value stores with parity logging," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–15.

[3] R. Pitchumani and Y.-S. Kee, "Hybrid data reliability for emerging {Key-Value} storage devices," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 309–322.

[4] M. Sathiamoorthy et al., "XORing elephants: Novel erasure codes for big data," 2013, *arXiv:1301.3791*.

[5] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–337.

[6] W. Haddock, P. V. Bangalore, M. L. Curry, and A. Skjellum, "High performance erasure coding for very large stripe sizes," in *Proc. Spring Simulat. Conf. (SpringSim)*, 2019, pp. 1–12.

[7] "Workloads repository." 2012. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

[8] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.

[9] G. Ananthanarayanan et al., "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 287–300.

[10] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of Facebook photo caching," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 167–181.

[11] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.

[12] "What is Memcached?" 2018. [Online]. Available: https://memcached.org/

[13] H. Chen et al., "Efficient and available in-memory KV-store with hybrid erasure coding and replication," *ACM Trans. Storage*, vol. 13, no. 3, pp. 1–30, 2017.

[14] C. Lai et al., "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol. (MSST)*, 2015, pp. 1–14.

[15] J. Shen, Y. Li, G. Sheng, Y. Zhou, and X. Wang, "Efficient memory caching for erasure coding based key-value storage systems," in *Proc. CCF Conf. Big Data*, 2018, pp. 512–539.

[16] L. Cheng, Y. Hu, and P. P. Lee, "Coupling decentralized key-value stores with erasure coding," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 377–389.

[17] M. Qin, A. N. Reddy, P. V. Gratz, R. Pitchumani, and Y. S. Ki, "KVRAID: High performance, write efficient, update friendly erasure coding scheme for KV-SSDs," in *Proc. 14th ACM Int. Conf. Syst. Storage*, 2021, pp. 1–12.

[18] U. Maheshwari, "{StripeFinder}: Erasure coding of small objects over {Key-Value} storage devices (an uphill battle)," in *Proc. 12th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2020.

[19] J. S. Plank, "A tutorial on reed–solomon coding for fault-tolerance in raid-like systems," *Softw. Pract. Exp.*, vol. 27, no. 9, pp. 995–1012, 1997.

[20] K. 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 Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 401–417.

[21] C. Huang et al., "Erasure coding in windows azure storage," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 15–26.

[22] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. Fast*, vol. 9, 2009, pp. 253–265.

[23] S. Wu, Z. Shen, and P. P. Lee, "Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed–Solomon codes," in *Proc. Int. Symp. Rel. Distrib. Syst. (SRDS)*, 2020, pp. 246–255.

[24] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. 5th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2013, p. 8.

[25] J. Li and B. Li, "Zebra: Demand-aware erasure coding for distributed storage systems," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*, 2016, pp. 1–10.

[26] J. Li and B. Li, "Demand-aware erasure coding for distributed storage systems," *IEEE Trans. Cloud Comput.*, vol. 9, no. 2, pp. 532–545, Apr.–Jun. 2021.

[27] Z. Wang, H. Wang, A. Shao, and D. Wang, "An adaptive erasure-coded storage scheme with an efficient code-switching algorithm," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, pp. 1–11.

[28] B. Xu, J. Huang, Q. Cao, and X. Qin, "TEA: A traffic-efficient erasure-coded archival scheme for in-memory stores," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.

[29] T. Cao et al., "A popularity-aware reconstruction technique in erasure-coded storage systems," *J. Parallel Distrib. Comput.*, vol. 146, pp. 122–138, Dec. 2020.

[30] J. Rui, Q. Huang, and Z. Wang, "Graftage coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 67, no. 4, pp. 2192–2205, Apr. 2021.

[31] M. Elyasi and S. Mohajer, "Cascade codes for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 66, no. 12, pp. 7490–7527, Dec. 2020.

[32] H. Qiu et al., "EC-fusion: An efficient hybrid erasure coding framework to improve both application and recovery performance in cloud storage systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2020, pp. 191–201.

[33] S. Han, P. P. Lee, Z. Shen, C. He, Y. Liu, and T. Huang, "Toward adaptive disk failure prediction via stream mining," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2020, pp. 628–638.

[34] S. Wu, Z. Shen, and P. P. Lee, "On the optimal repair-scaling trade-off in locally repairable codes," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 2155–2164.

[35] R. Nishtala et al., "Scaling memcache at Facebook," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 385–398.

[36] B. Xu, J. Huang, X. Qin, and Q. Cao, "Traffic-aware erasure-coded archival schemes for in-memory stores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2938–2953, Dec. 2020.

[37] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–14.

[38] Z. Shen and P. P. C. Lee, "Cross-rack-aware updates in erasure-coded data centers: Design and evaluation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2315–2328, Oct. 2020.

[39] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory KV storage," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 1–12.

Junmei Chen received the M.S. degree from the School of Computer Science and Engineering/School of Software, Guangxi Normal University, Guilin, China, in 2017. She is currently pursuing the Ph.D. degree with the School of Computer Science, Wuhan University, Wuhan, China.

Her current research interests include computer networks, error correction code in communication system, and erasure code in storage system.

Zongpeng Li (Senior Member, IEEE) received the B.E. degree in computer science from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree from the University of Toronto, Toronto, ON, Canada, in 2005.

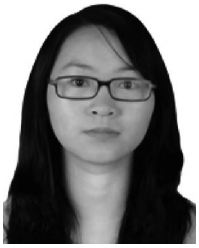
His research interests are in computer networks and cloud computing.

Dr. Li was named an Edward S.Rogers Sr. Scholar, in 2004, won the Alberta Ingenuity New Faculty Award, in 2007, and was nominated for the Alfred P. Sloan Research Fellow in 2007. He coauthored papers that received the Best Paper Awards at PAM 2008, HotPOST 2012, ACM e-Energy 2016, ACM TURC 2019, and BigCom 2020. He received the Outstanding Young Computer Science Researcher Prize from the Canadian Association of Computer Science, and the Research Excellence Award from the Faculty of Science, University of Calgary.

Ruiting Zhou (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, University of Calgary, Calgary, AB, Canada, in 2018.

She has been an Associate Professor with the School of Cyber Science and Engineering, Wuhan University, Wuhan, China, since June 2018. She has published research papers in top-tier computer science conferences and journals, including IEEE INFOCOM, ACM MobiHoc, IEEE ICDCS, ICPP, IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, and IEEE TRANSACTIONS ON MOBILE COMPUTING. Her research interests include cloud computing, machine learning, and mobile network optimization.

Dr. Zhou serves as the TPC Chair for INFOCOM workshop-ICCN2019-2022 and BIGCOM 2022. She also serves as a reviewer for journals and international conferences, such as IEEE/ACM IWQoS, IEEE GLOBECOM, IEEE MSN, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, IEEE/ACM TRANSACTIONS ON NETWORKING, and IEEE TRANSACTIONS ON MOBILE COMPUTING.



Lina Su received the M.S. degree from the School of Computer Science, Chongqing University of Posts and Telecommunications, Chongqing, China, in 2011. She is currently pursuing the Ph.D. degree with the School of Computer Science, Wuhan University, Wuhan, China.

Her research interests include online optimization, online learning, and federated learning.

Ne Wang received the B.E. degree from the School of Computer Science and Technology, Wuhan University of Technology, Wuhan, China, in 2016. She is currently pursuing the Ph.D. degree with the School of Computer Science, Wuhan University, Wuhan.

Her research interests include edge computing, distributed machine learning optimization, and online scheduling.