Elastic Reed-Solomon Codes for Efficient Redundancy Transitioning in Distributed Key-Value Stores

Si Wu, Zhirong Shen, Patrick P. C. Lee, Zhiwei Bai, Yinlong Xu

Abstract—Modern distributed key-value (KV) stores increasingly adopt erasure coding to reliably store data. To adapt to the changing demands on access performance and reliability requirements, distributed KV stores perform *redundancy transitioning* by tuning the redundancy schemes with different coding parameters. However, redundancy transitioning incurs extensive network I/Os, which impair the performance of distributed KV stores. We propose a new family of erasure codes, called *Elastic Reed-Solomon (ERS)* codes, whose primary goal is to mitigate network I/Os in redundancy transitioning. ERS codes eliminate data block relocation, while limiting network I/Os for parity block updates via the new co-design of encoding matrix construction and data placement. ERS codes achieve such gains in both forward and backward transitioning scenarios. We realize ERS codes in a distributed KV store prototype based on Memcached, and show via testbed experiments in both local and cloud environments that ERS codes significantly reduce the latency of redundancy transitioning compared with state-of-the-arts.

Index Terms—Erasure codes, Redundancy transitioning, Key-value stores.

1 Introduction

Distributed key-value (KV) stores improve scalability and access performance of object storage compared to traditional relational databases. To provide reliability guarantees against frequent failures [18], modern distributed KV stores increasingly adopt *erasure coding* to provide low-cost data redundancy [3], [13], [14], [33], [47]. Compared with replication, erasure coding significantly reduces the amount of redundancy to attain the same degree of fault tolerance [38]. Among many erasure coding constructions, Reed-Solomon (RS) codes [35] are one popular family of erasure codes that minimize the storage overhead for reliability guarantees. At a high level, RS codes encode k data blocks into additional m redundant blocks, called *parity blocks*, such that the k data blocks can be reconstructed from any k out of $k+m$ available data and parity blocks.

To adapt to the elastic demands on access efficiency and fault tolerance, it is desirable for erasure-coded KV stores to support *redundancy transitioning*, which dynamically adjusts the coding parameters k and m to balance performance, storage overhead, and reliability. We motivate that redundancy transitioning is critical for modern KV stores for two reasons.

- **Adaptation to workload changes.** Real-world storage workloads exhibit highly skewed patterns of popularity [9], [21], in which a small fraction of hot data is frequently accessed, while the remaining large fraction of cold data is rarely accessed. Also, the access patterns of storage workloads are time-varying [44]. Fixing the coding parameters makes KV stores inflexible to achieve both high performance and low storage overhead. Given that erasure coding poses a design trade-off between performance and storage efficiency [16], practical KV stores should incorporate multiple redundancy schemes, such that hot objects are encoded with high-redundancy erasure codes for better performance, while cold objects are encoded with low-redundancy erasure codes for better storage efficiency. For example, Ceph adopts RS codes and allows the coding parameters to be tunable [3], so that objects can be stored with different coding parameters to balance between performance and space efficiency.
- **Adaptation to reliability requirements.** Disk reliability changes throughout the entire disk lifetime, so data centers can dynamically switch across different redundancy schemes to balance between storage overhead and fault tolerance [22], [23]. Also, the reliability importance varies across data types, in which the loss of important data may imply costly recovery [36]. Such important data may be protected by erasure codes with higher redundancy.

However, realizing efficient redundancy transitioning in a distributed environment is a non-trivial task, mainly because the redundancy transitioning process often incurs data block relocation and parity block updates, both of which incur substantial network I/Os that lead to extensive amounts of traffic being transferred over the network. Specifically, traditional erasure codes often map blocks to a fixed set of nodes, such that the number of data blocks for an object is equal to the number of nodes that store the object data.

- *S. Wu, Z. Bai, and Y. Xu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China (Email: siwu5938@ustc.edu.cn,baizhiwei@mail.ustc.edu.cn,ylxu@ustc.edu.cn).*
- *Z. Shen is with the College of Informatics, Xiamen University, Xiamen, China (Email: zhirong.shen2601@gmail.com).*
- *P. P. C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (Email: pclee@cse.cuhk.edu.hk).*

- *An earlier conference version of the paper appeared in [41]. In this extended version, we address backward transitioning from RS(k', m) to RS(k, m) in ERS codes, where $k < k'$. We also re-run existing experiments on both a larger-scale local cluster and Alibaba Cloud, and add evaluation results on backward transitioning.*

- *Corresponding author: Patrick P. C. Lee.*

If redundancy transitioning changes the number of data blocks (i.e., k), then some data blocks have to be relocated to different nodes, thereby incurring extra I/Os. Parity block updates further aggravate I/Os: since the layout of data blocks has changed, the parity blocks need to be updated accordingly with additional I/Os, including the retrieval of all data blocks for recomputing the new parity blocks and the writes of the new parity blocks to nodes.

In this paper, we propose a new family of RS codes, called *Elastic Reed-Solomon (ERS)* codes, so as to enable efficient redundancy transitioning for erasure-coded distributed KV stores. ERS codes build on the decoupling of block-to-node mappings [36] by distributing data blocks into an extended number of nodes, so as to completely eliminate data block relocation. Furthermore, our key insight is that the computation of the new parity blocks (after transitioning) can reuse the old parity blocks (before transitioning), as both types of parity blocks often share the same encoding operations for some *overlapping* data blocks (defined in Section 2.3). Based on this insight, we propose a novel co-design of encoding matrix construction and data placement for ERS codes to increase the number of such overlapping data blocks. This allows the new parity blocks to be computed from largely the old parity blocks plus a small number of non-overlapping data blocks, thereby mitigating the network I/Os due to parity block updates. ERS codes require limited network I/Os in both forward and backward transitioning scenarios (defined in Section 2.2). Note that ERS codes preserve the storage optimality of RS codes (i.e., the minimum redundancy overhead for reliability guarantees).

We implement a distributed KV store prototype that realizes ERS codes based on Memcached [6]. Our prototype supports all basic KV operations (e.g., PUT, GET, UPDATE, etc.), while enabling redundancy transitioning. Experiments in both a local cluster and Alibaba Cloud [1] show that ERS codes reduce the transitioning latency of the state-of-the-art stretched RS codes [36] by up to 65.8% and 60.9% in forward transitioning and backward transitioning, respectively.

The source code of our prototype of ERS codes is available at **http://adslab.cse.cuhk.edu.hk/software/ers**.

2 Background and Motivation

We present the background of erasure coding in distributed KV stores (Section 2.1). We formulate the redundancy transitioning problem and state its challenges (Section 2.2). We motivate our solutions to the challenges (Section 2.3).

2.1 Erasure Coding in Distributed KV Stores

Distributed KV stores disperse objects (in the form of KV pairs) across a cluster of nodes. By using fast mapping algorithms, they achieve high-performance and scalable data accesses. For example, Dynamo [15] and Memcached [6] are two well-known distributed KV stores using *consistent hashing* [24] for object mapping. Consistent hashing organizes nodes into a *hash ring*. It deterministically maps each object to a node by hashing the object's key to a location in the hash ring, such that the object is stored in the nearest node in the clockwise direction of the hash ring.

Failures are prevalent in distributed KV stores [18], so it is essential for distributed KV stores to guarantee fault tolerance by introducing redundancy. *Erasure coding* incurs much less storage redundancy for the same degree of fault tolerance compared to replication [38]. In this work, we focus on Reed-Solomon (RS) codes [35], a popular family of erasure codes that are widely used in modern KV stores [3], [13], [25], [33], [47]. We construct RS codes, denoted by RS(k, m), with two configurable parameters k and m. RS(k, m) takes k *data blocks* (denoted by $D_0, D_1, \cdots, D_{k-1}$) as input for encoding, and generates m *parity blocks* (denoted by $P_0, P_1, \cdots, P_{m-1}$), such that any k out of the $k+m$ data and parity blocks suffice to reconstruct the original k data blocks. The $k+m$ data and parity blocks that are encoded together collectively form a *stripe*. In practice, a KV store comprises multiple stripes that are encoded independently. It distributes each stripe of $k+m$ blocks across $k+m$ nodes (denoted by $X_0, X_1, \cdots, X_{k+m-1}$) to tolerate any m node failures.

Mathematically, the encoding process of RS codes can be specified by an $m \times k$ *encoding matrix* (denoted by $\mathbf{G}_{m\times k}$), constructed by the Vandermonde matrix [31]. Given a *data vector* (i.e., a column vector of k data blocks), RS codes multiply $\mathbf{G}_{m\times k}$ by the data vector to compute the *parity vector* (i.e., a column vector of m parity blocks) over the Galois Field GF(2^ω), where ω is the size of a coding unit (in bits). For example, when $(k, m) = (2, 2)$ and $\omega = 4$, the matrix-vector product representation is:

$$\begin{bmatrix} P_0 \\ P_1 \end{bmatrix} = \mathbf{G}_{2\times 2} \times \begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 8 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \end{bmatrix}. \quad (1)$$

There are two approaches of applying erasure coding to objects in KV stores, namely *per-object coding* [3], [25], [33], which divides each object into k data blocks for encoding, and *cross-object coding* [13], [14], [47], which stores multiple objects within a data block and collects every k data blocks for encoding. In this work, we mainly consider per-object coding, which exhibits better load balancing and I/O performance [33]. We target the workloads with large-size objects (e.g., in cloud storage), in which each object can be divided into multiple data blocks for encoding. For example, EC-Cache focuses on the object size of at least 1 MB [33].

2.2 Redundancy Transitioning

Problem definition. *Redundancy transitioning* focuses on changing the coding parameters k and m of existing erasure-coded objects, so as to adapt to the varying access characteristics and reliability demands (Section 1). In this work, we pay special attention to the transitioning that changes k only, while the number of tolerable failures m remains unchanged. We call the conversion *forward transitioning* when transitioning from RS(k, m) to RS(k', m) (where $k < k'$), and *backward transitioning* when transitioning from RS(k', m) to RS(k, m), such that RS(k', m) is derived from RS(k, m) by forward transitioning. By increasing k in forward transitioning, we can reduce the storage redundancy and increase the overall storage efficiency; on the other hand, by decreasing k in backward transitioning, we can enhance the access performance and reliability of the objects. Thus, both forward transitioning and backward transitioning are essential in distributed KV stores. In the following, we first focus on forward transitioning and state the challenges in redundancy transitioning, and show how our ERS codes address the

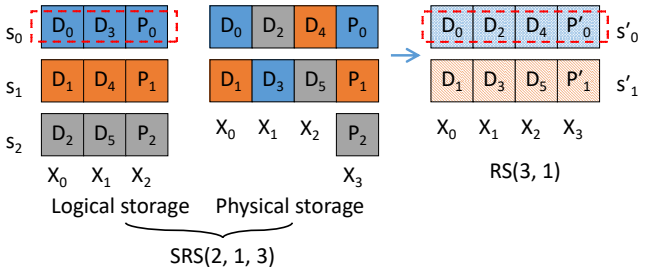


Figure 1. Example of SRS(2,1,3), which eliminates data block relocation in the transitioning from RS(2,1) to RS(3,1). The data and parity blocks of the same color constitute a stripe.

challenges (Sections 2 and 3). Later, we address backward transitioning in ERS codes (Section 4).

Redundancy transitioning inevitably changes the encoding layout of a stripe (e.g., in forward transitioning, the number of data blocks changes from k to k'), so we need to update both the encoding matrix and the corresponding parity blocks. There are two key I/O operations, namely *data block relocation*, which relocates existing data blocks to form a new stripe, and *parity block updates*, in which the parity blocks are updated based on the new encoding matrix $\mathbf{G}_{m\times k'}$. Both operations incur network I/Os in distributed KV stores and hence data traffic being transferred over the network (Section 1). In this work, we focus on mitigating the network I/Os in redundancy transitioning.

To eliminate data block relocation during redundancy transitioning, Ring [36] proposes *Stretched Reed-Solomon (SRS)* codes (denoted by SRS(k, m, k')). SRS codes differentiate *logical storage* from *physical storage*: the former refers to the $k+m$ logical columns that store the RS(k, m) stripes, while the latter refers to the $k'+m$ physical nodes that actually store the stripes. Then, the block-to-node mapping and data placement operate on physical storage. The idea of SRS codes is to store the k data blocks of RS(k, m) in $k' > k$ nodes (i.e., relaxing the tight coupling of the same block-to-node mappings). SRS(k, m, k') operates on a group of multiple stripes. It first computes the least common multiple (LCM) of k and k', denoted by $l = \text{lcm}(k, k')$. It distributes l data blocks into k columns (in logical storage) in column-major order. It encodes every k data blocks into m parity blocks via RS(k, m). It finally stores the l data blocks evenly over k' nodes (in physical storage), while keeping the parity blocks in m nodes. As the data blocks of SRS(k, m, k') are now stored in k' nodes, transitioning from RS(k, m) to RS(k', m) has no data block relocation.

To illustrate, Figure 1 shows an example of SRS(2,1,3). Let s_i ($i \geq 0$) be a *pre-transitioning stripe* (before transitioning), and let s'_i ($i \geq 0$) be a *post-transitioning stripe* (after transitioning). Before transitioning, there are $\frac{l}{k} = 3$ pre-transitioning stripes (i.e., s_0, s_1, and s_2) for RS(2,1). After transitioning, there are $\frac{l}{k'} = 2$ post-transitioning stripes (i.e., s'_0 and s'_1) for RS(3,1). We can see that the data block distribution for RS(3,1) is preserved, so data block relocation is eliminated. However, the parity blocks need to be updated accordingly (i.e., P'_0 and P'_1).

Challenges. While SRS codes eliminate data block relocation, it is still challenging to realize efficient redundancy transitioning due to the expensive parity block updates. The reasons are two-fold.

Challenge 1: The encoding matrices before and after transitioning substantially differ. We elaborate this issue via an example of transitioning from RS(4,3) to RS(5,3).

We first show the encoding process of the pre-transitioning stripe, in which the encoding matrix $\mathbf{G}_{3\times 4}$ is multiplied by the four data blocks $\{D_0, D_1, D_2, D_3\}$:

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \mathbf{G}_{3\times 4} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 15 & 2 & 14 \\ 1 & 12 & 8 & 5 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix}. \quad (2)$$

We next show the encoding process of the post-transitioning stripe to generate $\{P'_0, P'_1, P'_2\}$, formed by multiplying the matrix $\mathbf{G}_{3\times 5}$ by $\{D_0, D_1, D_2, D_3, D_4\}$:

$$\begin{bmatrix} P'_0 \\ P'_1 \\ P'_2 \end{bmatrix} = \mathbf{G}_{3\times 5} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 & 14 \\ 1 & 10 & 4 & 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix}. \quad (3)$$

Since $P_0 = D_0 + D_1 + D_2 + D_3$ and $P'_0 = D_0 + D_1 + D_2 + D_3 + D_4$ (in Galois Field arithmetic), we can simply retrieve D_4 to update P_0 into P'_0. However, $P_1 = D_0 + 15D_1 + 2D_2 + 14D_3$ and $P'_1 = D_0 + 3D_1 + 6D_2 + 10D_3 + 14D_4$, so in order to update P_1 into P'_1, we have to retrieve D_1, D_2, D_3, and D_4. Similarly, updating P_2 into P'_2 also needs to retrieve D_1, D_2, D_3, and D_4. Thus, in order to transition from RS$(4, 3)$ to RS$(5, 3)$, we need to access D_1, D_2, D_3, and D_4 (i.e., four data blocks) for parity block updates.

Challenge 2: The placement of data blocks also determines the number of data blocks to be read during redundancy transitioning. For example, in Figure 1, we can only find one common data block D_0 in s_0 and s'_0, as well as one common data block D_1 in s_1 and s'_1. If we use P_0 and P_1 to generate P'_0 and P'_1, respectively, then $P'_0 = P_0 + D_2 + D_3 + D_4$, and $P'_1 = P_1 + D_3 + D_4 + D_5$. Thus, we need to retrieve four data blocks (i.e., D_2, D_3, D_4, and D_5) from other nodes to generate the new parity blocks. The network I/O cost of parity block updates is higher for larger coding parameters (e.g., from RS$(4, 3)$ to RS$(5, 3)$), where we have to retrieve more data blocks.

2.3 Motivation

Definition. Our major goal is to mitigate the network I/Os for parity block updates in redundancy transitioning, by limiting the number of data blocks to be retrieved. We call a data block an *overlapping data block* if its coefficients encoded into the old parity blocks of the pre-transitioning stripe are the same as its coefficients encoded into the new parity blocks of the post-transitioning stripe; otherwise, we call it a *non-overlapping data block*.

Main insight. Our main insight is that during redundancy transitioning, we do not need to retrieve the overlapping data blocks, as the old and new parity blocks share the same encoding operations for the overlapping data blocks. We only need to access the non-overlapping data blocks as their encoding operations differ in the old and new parity blocks. Our idea is to increase the number of overlapping data blocks (or equivalently, decrease the number of non-overlapping data blocks to be retrieved during redundancy transitioning). We motivate our solutions via the following examples.

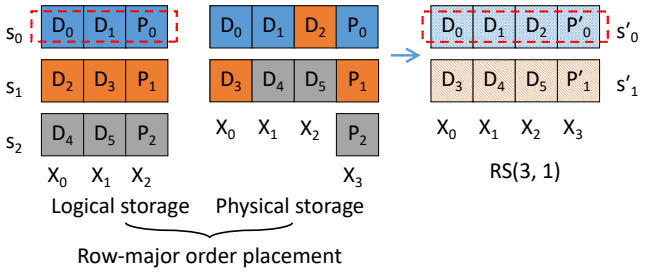


Figure 2. Example of transitioning from RS(2,1) to RS(3,1) with the row-major order data placement. The blocks of the same color form a stripe.

Motivation 1 (Using an enlarged encoding matrix). Instead of directly using the encoding matrices of RS codes for the pre-transitioning and post-transitioning stripes, we adopt a large-sized encoding matrix before transitioning and add dummy blocks for encoding. For example, in transitioning from RS(4,3) to RS(5,3), the pre-transitioning encoding process can be denoted by:

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \mathbf{G}_{3\times 5} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 & 14 \\ 1 & 10 & 4 & 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ 0 \end{bmatrix}. \quad (4)$$

The key difference between the conventional and new encoding mechanisms is that the conventional approach employs an $m \times k$ encoding matrix $\mathbf{G}_{m\times k}$ to encode k data blocks, while we exploit an $m \times k'$ encoding matrix $\mathbf{G}_{m\times k'}$ to encode k' blocks (with k data blocks and $k'-k$ dummy blocks). For example, the conventional approach uses $\mathbf{G}_{3\times 4}$ to encode D_0-D_3 (Equation (2)). We now adopt $\mathbf{G}_{3\times 5}$ to encode D_0-D_3, and one dummy block (Equation (4)).

Recall that in the transitioning from RS$(4,3)$ to RS$(5,3)$ using the conventional encoding matrix (i.e., Equation (2)), there is only one overlapping data block D_0 in the pre-transitioning and post-transitioning stripes. Now, if we use an enlarged matrix (i.e., Equation (4)), there are four overlapping data blocks, i.e., D_0-D_3. To update P_1 into P_1', we now simply retrieve the only non-overlapping data block D_4. The transitioning between P_2 and P_2' is similar.

Motivation 2 (Producing more overlapping data blocks). Our insight is that even though we adopt an enlarged encoding matrix, the conventional rigid data placement in column-major order still leads to a limited number of overlapping data blocks in both the pre-transitioning and post-transitioning stripes. Thus, we aim to allow more overlapping data blocks via a new data placement strategy. For example, in Figure 2, we distribute the data blocks logically in k nodes and physically in k' nodes, both in row-major order. In Figure 2, there are two overlapping data blocks D_0 and D_1 in s_0 and s_0', as well as two overlapping data blocks D_4 and D_5 in s_2 and s_1'. We can compute $P_0' = P_0 + D_2$ and $P_1' = P_2 + D_3 = P_2 + P_1 + D_2$. Thus, we need to access only D_2 for parity block updates. Note that in general, the row-major order does not necessarily imply efficient data placement, and we still need to carefully design a data placement strategy to increase the number of overlapping data blocks.

Summary. The above examples suggest that we can explore a new co-design of encoding matrix construction and data placement, so as to increase the number of overlapping data blocks. This allows the new parity blocks to be recomputed from largely the old parity blocks plus a small number of non-overlapping data blocks that need to be retrieved. This mitigates the network I/Os of parity block updates.

3 Elastic Reed-Solomon Codes

We present Elastic Reed-Solomon (ERS) codes, a new family of erasure codes for efficient redundancy transitioning. ERS codes exploit both new encoding matrix construction and new data placement to increase the number of overlapping data blocks, so as to mitigate the network I/Os for parity block updates. We first present an overview of ERS codes (Section 3.1). We then present the design details of the encoding matrix of ERS codes to increase the number of overlapping data blocks (Section 3.2). Next, we present our data placement strategy based on our new encoding matrix to further increase the number of overlapping data blocks (Section 3.3). Furthermore, we discuss how to extend ERS codes to handle objects with small sizes (Section 3.4). In this section, we focus on forward transitioning, while we study backward transitioning in Section 4.

3.1 Design Overview

ERS codes (denoted by ERS(k, m, k')) build on the decoupling of block-to-node mappings in SRS codes [36] by storing the k data blocks of RS codes in k' nodes, where $k < k'$. Similar to SRS, ERS first computes the LCM of k and k', i.e., $l = \text{lcm}(k, k')$, and divides an object into l data blocks denoted by $D_0, D_1, \cdots, D_{l-1}$. It then arranges the l data blocks into k logical columns, and encodes every k data blocks with the same logical offset to calculate m parity blocks. It finally distributes the l data blocks over k' nodes such that each node stores exactly $\frac{l}{k'}$ blocks, and distributes the parity blocks on m nodes. For a group of $\frac{l}{k}$ stripes, the parity blocks are put on the same m nodes, while for different groups of stripes, we put the parity blocks on different nodes to balance the storage and I/O overhead for parity updates. Like SRS codes, as the data blocks are distributed over k' nodes, ERS codes also eliminate data block relocation when objects are transitioned from RS(k, m) to RS(k', m).

However, ERS coding differs from SRS coding in the following aspects. First, ERS coding logically distributes l data blocks into k nodes in *row-major order*, and hence introduces a considerable number of overlapping data blocks for redundancy transitioning (e.g., Figure 2). In addition, ERS coding encodes every k data blocks using a novel encoding matrix, and physically distributes l data blocks over k' nodes according to a novel data placement strategy, so as to increase the number of overlapping data blocks. As a result, ERS coding requires only a small number of non-overlapping data blocks and reduces network I/Os for parity block updates.

Figure 2 shows an example of ERS$(2, 1, 3)$ with the row-major order data placement. As $k = 2$ and $k' = 3$, we can deduce that $l = \text{lcm}(2, 3) = 6$. We start by sequentially arranging $l = 6$ data blocks (i.e., D_0-D_5) into $k = 2$ logical columns in row-major order. Every $k = 2$ data blocks with the same logical offset (i.e., same color in the figure) are encoded into $m = 1$ parity block in logical storage. The $l = 6$

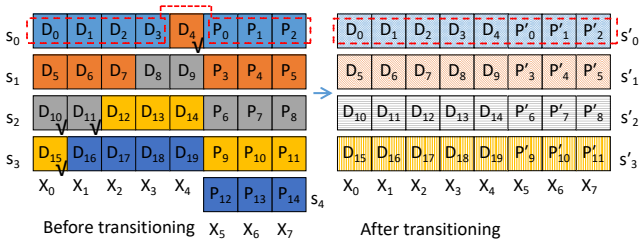


Figure 3. Row-major order data placement for $(k, m, k') = (4, 3, 5)$. The blocks of the same color constitute a stripe. The data blocks with black check marks indicate the non-overlapping data blocks of the pre-transitioning stripes.

data blocks are then physically stretched across $k' = 3$ nodes also in row-major order in physical storage. We can see that the distribution of data blocks for RS$(3, 1)$ is preserved, so data block relocation is eliminated in the transitioning from RS$(2, 1)$ to RS$(3, 1)$. We can also show that $P'_0 = P_0 + D_2$, and $P'_1 = P_2 + D_3 = P_2 + P_1 + D_2$. Thus, we only need to access D_2 for parity block updates in redundancy transitioning.

3.2 Encoding Matrix Design

Overall idea. We assume that the sequential data blocks are placed on k' nodes based on row-major order by default as shown in Figure 3. In physical storage, all l data blocks form a $\frac{l}{k'} \times k'$ array that is composed of the data blocks from $\frac{l}{k}$ pre-transitioning stripes. For each pre-transitioning stripe, we exploit an $m \times k'$ encoding matrix (i.e., $\mathbf{G}_{m\times k'}$) to encode the k' blocks, which comprise k data blocks and $k'-k$ additional dummy blocks; here, the dummy blocks can be zero blocks. The new encoding matrix $\mathbf{G}_{m\times k'}$ is still constructed by the Vandermonde matrix [31]. The key design is exploiting the same encoding matrix (i.e., $\mathbf{G}_{m\times k'}$) before and after transitioning, without depending on specific coefficients inside the matrix. Thus, other matrix constructions, such as the Cauchy matrix in Cauchy Reed-Solomon codes [31], [32], are also applicable. Note that the dummy blocks are not involved in the encoding operations, so they do not incur extra computational overhead or memory space. Our approach of utilizing an enlarged encoding matrix is analogous to the *shortening* scheme [30]; here, we design an enlarged encoding matrix for redundancy transitioning.

Algorithm details. We call the enlarged encoding matrix (i.e., $\mathbf{G}_{m\times k'}$) an *ERS encoding matrix*. If a block is stored in a node X_i, then we say the *node id* of this block is i. For example, in Figure 3, D_0 and D_1 are stored in X_0 and X_1, so the node ids of D_0 and D_1 are 0 and 1, respectively. Algorithm 1 presents the detailed procedure to encode the pre-transitioning stripes utilizing an ERS encoding matrix. Specifically, we use $\mathbf{G}_{m\times k'}$ for encoding (Line 1). For each data block in a pre-transitioning stripe s_i, we set its id in the data vector as its node id in physical storage (Lines 3-5). There are k data blocks in s_i and we add extra $k'-k$ dummy blocks to form k' blocks (Line 6), and then encode the k' blocks (Line 7).

For example, we show the encoding process of s_1 in ERS$(4, 3, 5)$. The ERS encoding matrix we use is $\mathbf{G}_{3\times 5}$. The node ids of the four data blocks D_4, D_5, D_6, and D_7 of s_1 are 4, 0, 1, and 2, respectively (Figure 3). With one extra dummy block, the encoding process is shown as follows.

Algorithm 1 Encoding using an ERS encoding matrix

1: Select $\mathbf{G}_{m\times k'}$ for encoding
2: **for** each stripe s_i $(0 \le i \le \frac{l}{k} - 1)$ **do**
3: **for** each data block D_j $(0 \le j \le k-1)$ **do**
4: Set its id in the data vector as its node id
5: **end for**
6: Add $k'-k$ dummy blocks into the remaining $k'-k$ positions in the data vector to constitute k' blocks
7: Encode the k' blocks
8: **end for**

$$\begin{bmatrix} P_3 \\ P_4 \\ P_5 \end{bmatrix} = \mathbf{G}_{3\times 5} \times \begin{bmatrix} D_5 \\ D_6 \\ D_7 \\ 0 \\ D_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 & 14 \\ 1 & 10 & 4 & 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_5 \\ D_6 \\ D_7 \\ 0 \\ D_4 \end{bmatrix}. \quad (5)$$

Analysis. For each stripe, multiplying the first row of $\mathbf{G}_{m\times k'}$ by the data vector incurs $k-1$ additions in Galois Field, while multiplying each of the remaining rows of $\mathbf{G}_{m\times k'}$ by the data vector needs $k-1$ multiplications and $k-1$ additions, both in Galois Field. Thus, the encoding of each stripe costs $(m-1)(k-1)$ multiplications and $m(k-1)$ additions. This implies that using the ERS encoding matrix and adding dummy blocks do not increase the number of multiplications and additions (e.g., Equation (5)), so we do not add additional computational cost. Also, as the dummy blocks do not participate in calculation, the system does not allocate memory space to store them (i.e., they do not consume additional memory space).

We now show that the MDS property is preserved using the ERS encoding matrix. In each stripe, if any m blocks are lost, we can reconstruct the original data blocks by the k remaining blocks and $k'-k$ dummy blocks. Thus, the MDS property is preserved.

Forward transitioning process. As we employ an ERS encoding matrix for encoding (i.e., Algorithm 1), we can efficiently utilize the old parity blocks for parity block updates. We now elaborate the forward transitioning process (from RS(k, m) to RS(k', m)) using an ERS encoding matrix. Let $\mathbf{p}_i$ $(0 \le i \le \frac{l}{k} - 1)$ be the column vector composed of the old parity blocks of s_i, i.e., $\mathbf{p}_i = [P_{i\times m}, P_{i\times m+1}, \cdots, P_{i\times m+m-1}]^T$, and $\mathbf{p}'_i$ $(0 \le i \le \frac{l}{k'} - 1)$ be the column vector composed of the new parity blocks of s'_i, i.e., $\mathbf{p}'_i = [P'_{i\times m}, P'_{i\times m+1}, \cdots, P'_{i\times m+m-1}]^T$. Let $\mathbf{g}_j$ $(0 \le j \le k'-1)$ be the j-th column of the ERS encoding matrix $\mathbf{G}_{m\times k'}$. For example, for ERS$(4, 3, 5)$, $\mathbf{p}_0 = [P_0, P_1, P_2]^T$, $\mathbf{p}'_0 = [P'_0, P'_1, P'_2]^T$ and $\mathbf{g}_4 = [1, 14, 11]^T$.

Algorithm 2 shows the forward transitioning process. We first initialize all new parity blocks (Lines 1-3). For a pre-transitioning stripe s_i, if it shares the most overlapping data blocks with a post-transitioning stripe s'_j, then the old parity blocks in $\mathbf{p}_i$ are used to generate the new parity blocks in $\mathbf{p}'_j$ (Lines 5-7). We next retrieve only the non-overlapping data blocks of s_i, i.e., the data blocks that are encoded into s_i but not s'_j (Line 8). A non-overlapping data block is encoded into the old parity blocks in $\mathbf{p}_i$ but not the new parity blocks in $\mathbf{p}'_j$, so we need to use it to update the new parity blocks in $\mathbf{p}'_j$. For each non-overlapping data block D_x, we find its node id y (Line 10), so $\mathbf{g}_y$ represents the coding coefficients of D_x encoded into the parity blocks. We then use D_x and

Algorithm 2 Forward transitioning from RS(k, m) to RS(k', m)

1: **for** $j = 0$ to $\frac{l}{k'} - 1$ **do**
2: Initialize $\mathbf{p}'_j$ to be zero vector
3: **end for**
4: **for** each pre-transitioning stripe s_i $(0 \leq i \leq \frac{l}{k} - 1)$ **do**
5: Find the post-transitioning stripe s'_j, such that s_i shares the most overlapping data blocks with s'_j
6: `// p_i is used to generate p'_j`
7: Set $\mathbf{p}'_j = \mathbf{p}'_j + \mathbf{p}_i$
8: Retrieve the non-overlapping data blocks of s_i, i.e., the data blocks encoded in s_i but not s'_j
9: **for** each non-overlapping data block D_x **do**
10: $y \leftarrow$ node id of D_x
11: `// Use D_x to update p'_j`
12: Set $\mathbf{p}'_j = \mathbf{p}'_j + \mathbf{g}_y \times D_x$
13: $z \leftarrow$ id of post-transitioning stripe that includes D_x
14: `// Use D_x to update p'_z`
15: Set $\mathbf{p}'_z = \mathbf{p}'_z + \mathbf{g}_y \times D_x$
16: **end for**
17: **end for**

its coefficient vector $\mathbf{g}_y$ to update the new parity blocks in $\mathbf{p}'_j$ (Lines 11-12). Furthermore, such a non-overlapping data block D_x will fall into another post-transitioning stripe s'_z (different from s'_j) (Line 13), and D_x will be encoded into the new parity blocks in $\mathbf{p}'_z$. Thus, we finally use D_x to update $\mathbf{p}'_z$ (Lines 14-15).

For example, in Figure 3, in transitioning from RS$(4, 3)$ to RS$(5, 3)$, we show the update mechanisms for $\mathbf{p}'_0$ and $\mathbf{p}'_1$. Since s_0 shares the maximum number of overlapping data blocks with s'_0, the old parity blocks in $\mathbf{p}_0$ are used to generate the new parity blocks in $\mathbf{p}'_0$. There is no non-overlapping data block in s_0. We can also find that the non-overlapping data block D_4 of s_1 will fall into s'_0. We then use D_4 and its coefficient vector $\mathbf{g}_4$ (the node id of D_4 is 4) to update $\mathbf{p}'_0$.

$$\begin{bmatrix} P'_0 \\ P'_1 \\ P'_2 \end{bmatrix} = \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 14 \\ 11 \end{bmatrix} \times D_4. \quad (6)$$

Since both s_1 and s_2 share the most overlapping data blocks with s'_1, both $\mathbf{p}_1$ and $\mathbf{p}_2$ are added into $\mathbf{p}'_1$. The non-overlapping data blocks of s_1, and s_2 are D_4 (with node id 4), and D_{10} and D_{11} (with node ids 0 and 1), respectively. We then use D_4 (and $\mathbf{g}_4$), D_{10} (and $\mathbf{g}_0$), and D_{11} (and $\mathbf{g}_1$) to update $\mathbf{p}'_1$.

$$\begin{bmatrix} P'_3 \\ P'_4 \\ P'_5 \end{bmatrix} = \begin{bmatrix} P_3 \\ P_4 \\ P_5 \end{bmatrix} + \begin{bmatrix} 1 \\ 14 \\ 11 \end{bmatrix} \times D_4 + \begin{bmatrix} P_6 \\ P_7 \\ P_8 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 10 \end{bmatrix} \times \begin{bmatrix} D_{10} \\ D_{11} \end{bmatrix}. \quad (7)$$

Recall that the main overhead in redundancy transitioning is to read the non-overlapping data blocks of the pre-transitioning stripes across the network (Line 8). We show how using an ERS encoding matrix reduces the number of non-overlapping data blocks and hence mitigates network I/Os. For example, in Figure 3, in the transitioning from RS$(4, 3)$ to RS$(5, 3)$ using the ERS encoding matrix, we only need to retrieve four non-overlapping data blocks (i.e., D_4, D_{10}, D_{11}, and D_{15}) for parity block updates. However, in SRS$(4, 3, 5)$, there are very few overlapping data blocks, so we need to access nearly all data blocks.

3.3 Data Placement Design

In Section 3.2, we utilize an ERS encoding matrix to increase the number of overlapping data blocks under the row-major order data placement. However, the row-major order does not always imply efficient data placement. In this subsection, we design efficient data placement to further increase the number of overlapping data blocks based on an ERS encoding matrix.

Overall idea. We attempt to put the data blocks of $\frac{l}{k}$ pre-transitioning stripes into a $\frac{l}{k'} \times k'$ array, such that the number of overlapping data blocks of the pre-transitioning stripes is maximized. As the number of data blocks in a pre-transitioning stripe is k, we can deduce that the maximum number of overlapping data blocks between a pre-transitioning stripe and a post-transitioning stripe is k. Note that a row in the array maps to a post-transitioning stripe. Thus, if we place a pre-transitioning stripe entirely in one row, then the number of overlapping data blocks between this pre-transitioning stripe and a post-transitioning stripe is maximized to k. To this end, we first place the maximum number of pre-transitioning stripes such that each of them is entirely put in one row (i.e., the number of overlapping data blocks is k). As a row can accommodate at most $\lfloor \frac{k'}{k} \rfloor$ pre-transitioning stripes and there are $\frac{l}{k'}$ rows, we can place $\alpha = \lfloor \frac{k'}{k} \rfloor \times \frac{l}{k'}$ stripes such that each of them is entirely put in one row (i.e., the number of overlapping data blocks is k).

Since there remain $r = k' \mod k$ empty positions in each row after filling $\lfloor \frac{k'}{k} \rfloor$ pre-transitioning stripes, the maximum number of overlapping data blocks between each of the remaining $\beta = \frac{l}{k} - \alpha$ pre-transitioning stripes and a post-transitioning stripe (i.e., a row) is at most r. We then place r data blocks of each remaining stripe in the empty positions of a row, so as to make the number of overlapping data blocks of this stripe equal to r, and also place $k - r$ data blocks in other rows.

To place each pre-transitioning stripe, we must make sure that the k data blocks are distributed into k nodes to maintain node-level fault tolerance.

Algorithm details. We call our data placement *ERS-aware data placement*. We call a pre-transitioning stripe a *same-row stripe* if it is entirely put in one row; otherwise, we call it a *cross-row stripe*. Algorithm 3 shows the design of the ERS-aware data placement. In each of the first β rows, we put $\lfloor \frac{k'}{k} \rfloor$ same-row stripes in it, and the starting node id of the stripes is $(i \times (k - r)) \mod k'$ $(0 \leq i \leq \beta - 1)$ (Lines 2-4). In each of the remaining $\frac{l}{k'} - \beta$ rows, we also put $\lfloor \frac{k'}{k} \rfloor$ same-row stripes in it, and the starting node id of the stripes is $((i - \beta + 1) \times r) \mod k'$ $(\beta \leq i \leq \frac{l}{k'} - 1)$ (Lines 5-7). In total, we put α same-row stripes (with number of overlapping data blocks of k). Note that the same-row stripes have different starting node ids in different rows, which is to guarantee node-level fault tolerance. For the i-th $(0 \leq i \leq \beta - 1)$ cross-row stripe, we place r data blocks in the empty positions of the i-th row such that the number of overlapping data blocks of it is r (Line 10), and $k - r$ data blocks on other rows (Line 11).

Figure 4 shows the ERS-aware data placement for $(k, m, k') = (4, 3, 5)$. In the first row (i.e., $\beta = 1$), we put $\lfloor \frac{k'}{k} \rfloor = 1$ same-row stripe in it, and the starting node id of the stripe is 0 (Line 3). In each of the remaining three rows (i.e., $\frac{l}{k'} - \beta = 3$), we also put $\lfloor \frac{k'}{k} \rfloor = 1$ same-row stripe in it,

Algorithm 3 ERS-aware data placement

1: `// Put` α `same-row stripes`
2: **for** the i-th $(0 \leq i \leq \beta - 1)$ row **do**
3: Put $\lfloor \frac{k'}{k} \rfloor$ same-row stripes in it, with starting node id $(i \times (k-r)) \mod k'$
4: **end for**
5: **for** the i-th $(\beta \leq i \leq \frac{l}{k'} - 1)$ row **do**
6: Put $\lfloor \frac{k'}{k} \rfloor$ same-row stripes in it, with starting node id $((i-\beta+1) \times r) \mod k'$
7: **end for**
8: `// Locate the remaining` β `cross-row stripes`
9: **for** the i-th $(0 \leq i \leq \beta - 1)$ cross-row stripe **do**
10: Put r data blocks in the r empty positions of the i-th row
11: Put $k-r$ data blocks sequentially in the empty positions of the β-th to $(\frac{l}{k'} - 1)$-th rows
12: **end for**

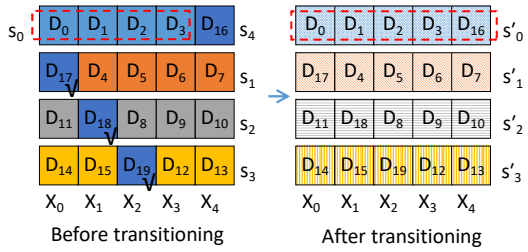


Figure 4. ERS-aware data placement for $(k, m, k') = (4, 3, 5)$. We omit the placement of the parity blocks as it does not affect the number of overlapping data blocks. The data blocks of the same color constitute a stripe. Note that s_0-s_3 are $\alpha = 4$ same-row stripes, and s_4 is $\beta = 1$ cross-row stripe. The data blocks with black check marks indicate the non-overlapping data blocks of the pre-transitioning stripes.

and the starting node ids of the stripes in the second, third, and fourth rows are $r = 1$, $2r = 2$, and $3r = 3$, respectively (Line 6). In total, we put $\alpha = 4$ same-row stripes, and the number of overlapping data blocks between each such pre-transitioning stripe and a post-transitioning stripe (i.e., a row) is maximized to $k = 4$.

The remaining $\beta = 1$ cross-row stripe has $r = 1$ data block in the first row and $k - r = 3$ data blocks in the second to fourth rows, such that the number of overlapping data blocks between it and a post-transitioning stripe (i.e., a row) is $r = 1$.

Analysis. In ERS-aware data placement, the placement of each same-row stripe or cross-row stripe is determined (Line 3, Line 6, and Lines 10-11). The calculation of the placement of each stripe costs $O(1)$ time. This means that we do not introduce additional computational cost under ERS-aware data placement.

We now analyze how ERS-aware data placement improves network I/O efficiency. We maximize the number of overlapping data blocks of the first α same-row stripes to be k, and then maximize the number of overlapping data blocks of the remaining β cross-row stripes to be r, subject to the condition that the number of overlapping data blocks of the α same-row stripes is maximized. The non-overlapping data blocks now only exist in the β cross-row stripes, and the number of non-overlapping data blocks retrieved for parity block updates is thus $(\frac{l}{k} - \lfloor \frac{k'}{k} \rfloor \times \frac{l}{k'}) \times (k - r)$.

For example, in the forward transitioning from RS$(4, 3)$ to RS$(5, 3)$ using ERS-aware data placement, we need three non-overlapping data blocks (i.e., D_{17}, D_{18}, and D_{19}) for parity block updates (Figure 4). Recall that the row-major order data placement needs four data blocks (Section 3.2). Thus, ERS-aware data placement decreases the number of non-overlapping data blocks to further save network I/Os.

Fault tolerance guarantee. Each of the α same-row stripes is spread over k nodes, so it follows the fault tolerance requirement. We can readily calculate that the node ids of the k data blocks of the first cross-row stripe are $\lfloor \frac{k'}{k} \rfloor \times k, \lfloor \frac{k'}{k} \rfloor \times k + 1, \cdots, k' - 1$ and $0, 1, \cdots, k - r - 1$. By induction, the node ids of the $(i+1)$-th cross-row stripe are right rotated by $k - r$ based on the i-th cross-row stripe. Thus, each cross-row stripe is sequentially spread across k different nodes.

3.4 Discussion

Handling small objects. Some workloads have objects with small sizes (e.g., tens of bytes [10], [12]). We can extend our ERS codes to store such small-sized objects. Specifically, we can organize the objects with the same demands into blocks, and collect l blocks for encoding. For example, the objects that are stored on the same device groups have the same reliability demands [22], [23], so we can store them into ERS codes. When the demands change, redundancy transitioning in ERS codes still incurs limited network I/Os.

Applicability of ERS codes on KV stores. ERS codes eliminate data block relocation, and require limited network I/Os for parity block updates, so they are suitable for KV stores that need high elasticity [2], [29]. In contrast, traditional redundancy transitioning schemes still require substantial network I/Os for both data block relocation and parity block updates [20], [50], so they are not applicable to KV stores. On the other hand, ERS codes can also be applied to distributed storage systems under dynamic workloads [44] or with varied reliability requirements [22], [23].

Data updates. When a data block is updated, the parity blocks in the same stripe will be updated accordingly [13], [14], [47]. Data deletion is treated the same as updating a data block into zero bytes. The redundancy level of the data keeps unchanged when data updates happen.

4 Backward Transitioning in ERS Codes

In this section, we address backward transitioning from RS(k', m) to RS(k, m) in ERS codes, where $k < k'$. Note that RS(k', m) is derived from RS(k, m) by forward transitioning.

4.1 New Challenges And Motivation

Differences from forward transitioning. In forward transitioning, there are $m \times \frac{l}{k}$ old parity blocks, and we need to generate $m \times \frac{l}{k'}$ new parity blocks. There are $\frac{l}{k'}$ update equations (i.e., Equations (6) and (7)), and each equation is used to generate m new parity blocks. Thus, we can use the old parity blocks (and the update equations) to generate all new parity blocks (Section 3).

In backward transitioning, there are now $m \times \frac{l}{k'}$ old parity blocks (i.e., the new parity blocks in forward transitioning), and we need to generate $m \times \frac{l}{k}$ new parity blocks (i.e., the old parity blocks in forward transitioning). We can utilize the reverse of the update equations in forward transitioning to generate the new parity blocks. However, the total number of new parity blocks (i.e., $m \times \frac{l}{k}$) is larger than the number

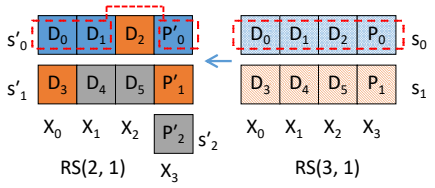


Figure 5. Backward transitioning in ERS$(k, m, k') = (2, 1, 3)$ with row-major order data placement. The data and parity blocks of the same color constitute a stripe.

of new parity blocks (i.e., $m \times \frac{l}{k'}$) that can be generated by the *reverse equations* (i.e., the reverse of Equations (6) and (7)). Thus, in order to successfully compute all new parity blocks, we need to first recalculate $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks based on their corresponding encoding data blocks. We then use the old parity blocks to generate the remaining $m \times \frac{l}{k'}$ new parity blocks based on the reverse equations.

New challenges. There are many possible choices for the set of $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks that need to be recalculated from their encoding data blocks. We show that the choice affects the network I/O cost for parity block updates.

For example, in Figure 5, we show the backward transitioning from RS$(3, 1)$ to RS$(2, 1)$ in ERS$(2, 1, 3)$ with the row-major order data placement. Note that data block relocation is also eliminated in backward transitioning. There are two old parity blocks (i.e., P_0 and P_1) and we need to compute three new parity blocks (i.e., P'_0, P'_1, and P'_2). Taking the reverse of the update equations in forward transitioning gives the following equations: $P'_0 = P_0 + D_2$ and $P'_2 + P'_1 = P_1 + D_2$. Thus, we need to recalculate one new parity block and generate two other new parity blocks via these equations.

If we recalculate P'_2, then $P'_2 = D_4 + D_5$, $P'_0 = P_0 + D_2$, and $P'_1 = P_1 + D_2 + P'_2 = P_1 + D_2 + D_4 + D_5$. In total, we need to access three data blocks D_2, D_4, and D_5 for parity block updates. For larger coding parameters, a naïve selection of the set of new parity blocks to be recalculated can result in substantial network I/O overhead for parity block updates.

Motivation. In the above example, there are two overlapping data blocks between s_1 and s'_2, but there is only one overlapping data block between s_1 and s'_1. If we recalculate P'_2 (or equivalently, using P_1 to update P'_1), then the network I/O cost is high because there are limited overlapping data blocks between s_1 and s'_1.

Alternatively, we can recalculate P'_1 first (or equivalently, using P_1 to update P'_2). Then, $P'_1 = D_2 + D_3$, $P'_0 = P_0 + D_2$, and $P'_2 = P_1 + D_2 + P'_1 = P_1 + D_3$. Thus, the number of data blocks accessed for parity block updates is reduced to two (i.e., D_2 and D_3). The reason is that there are more overlapping data blocks between s_1 and s'_2, and they can be exploited to save the network I/Os for parity block updates.

The above example motivates us to carefully choose the new parity blocks to be recalculated, such that the remaining new parity blocks can be updated from the old parity blocks with more overlapping data blocks. This saves the network I/Os for parity block updates in backward transitioning.

4.2 Algorithm Design

Main idea. Note that data block relocation is also eliminated when transitioning from RS(k', m) to RS(k, m) in backward transitioning. As RS(k', m) is derived from RS(k, m) in forward transitioning, we can record the $\frac{l}{k'}$ update equations in forward transitioning. Reversing the update equations produces $\frac{l}{k'}$ equations that can be used to update some new parity blocks in backward transitioning. For parity block updates, we choose $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks to be recalculated from their encoding data blocks, such that the other $m \times \frac{l}{k'}$ new parity blocks can be updated from the old parity blocks and the reverse equations with the maximum number of overlapping data blocks. Thus, the key requirement of backward transitioning is to select the proper order of new parity blocks to update, so as to minimize the network I/Os for parity block updates.

Algorithm 4 Backward transitioning from RS(k', m) to RS(k, m)

```
1:  // Obtain reverse equations
2:  Record the l/k' update equations of forward transitioning
3:  for i = 0 to l/k' − 1 do
4:      Reverse the i-th update equation
5:  end for
6:  // Recalculate m × (l/k − l/k') new parity blocks
7:  for i = 0 to l/k' − 1 do
8:      if there are more than one post-transitioning stripe
        involved in the i-th reverse equation then
9:          Find the post-transitioning stripe s'_j, such that s_i
            shares the most overlapping data blocks with s'_j
10:         for each remaining post-transitioning stripe s'_z do
11:             Recalculate p'_z from its encoding data blocks
12:         end for
13:     end if
14: end for
15: // Update m × l/k' remaining new parity blocks
16: for i = 0 to l/k' − 1 do
17:     Use p_i and the i-th reverse equation to update p'_j, where
        s_i shares the most overlapping data blocks with s'_j
18: end for
```

Algorithm details. Algorithm 4 shows the detailed approach for backward transitioning in ERS codes. First, we record all $\frac{l}{k'}$ update equations in forward transitioning (Line 2), and obtain $\frac{l}{k'}$ reverse equations that will be utilized in backward transitioning (Lines 3-5). We next recalculate $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks from their encoding data blocks. For the i-th reverse equation, if there are more than one post-transitioning stripe involved (i.e., the old parity blocks in $\mathbf{p}_i$ are encoded into the new parity blocks from more than one post-transitioning stripe), then we select the post-transitioning stripe s'_j such that s_i shares the maximum number of overlapping data blocks with s'_j (Line 9). For a remaining post-transitioning stripe s'_z that is involved in the i-th reverse equation, we recalculate its new parity blocks (i.e., $\mathbf{p}'_z$) via the encoding data blocks (Lines 10-12). Lastly, we use the old parity blocks and reverse equations to update the remaining $m \times \frac{l}{k'}$ new parity blocks. Specifically, based on the i-th reverse equation, we use $\mathbf{p}_i$ to update $\mathbf{p}'_j$, where s_i shares the maximum number of overlapping data blocks with s'_j (Lines 16-18). In summary, we select the order of new parity blocks to update by first determining $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks to be recalculated (Lines 8-13), followed by updating the remaining $m \times \frac{l}{k'}$ new parity blocks from the old parity blocks (Lines 16-18).

We now exemplify the backward transitioning from

RS$(5,3)$ to RS$(4,3)$ (i.e., $(k,m,k')=(4,3,5)$).

In SRS$(4,3,5)$, since it generally has a limited number of overlapping data blocks, we need to recalculate almost all new parity blocks by accessing almost all data blocks.

In ERS$(4,3,5)$ with the row-major order data placement (i.e., Figure 3), we show the update mechanisms for $\mathbf{p}'_1$ and $\mathbf{p}'_2$. Note that $\mathbf{p}_1$ is encoded into both $\mathbf{p}'_1$ and $\mathbf{p}'_2$ (by Equation (7)). Since s_1 shares the most overlapping data blocks with s'_1, the new parity blocks of s'_2 (i.e., $\mathbf{p}'_2$) should be recalculated. Afterwards, we use the reverse equation and $\mathbf{p}_1$ to update $\mathbf{p}'_1$, which results in

$$\begin{bmatrix} P'_3 \\ P'_4 \\ P'_5 \end{bmatrix} = \begin{bmatrix} P_3 \\ P_4 \\ P_5 \end{bmatrix} + \begin{bmatrix} 1 \\ 14 \\ 11 \end{bmatrix} \times D_4 + \begin{bmatrix} 1 & 1 \\ 10 & 14 \\ 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_8 \\ D_9 \end{bmatrix}. \quad (8)$$

Overall, we need to access six data blocks (i.e., D_4, D_8, D_9, D_{10}, D_{11}, and D_{15}) for parity block updates.

In ERS$(4,3,5)$ with ERS-aware data placement (i.e., Figure 4), we show the update mechanisms of $\mathbf{p}'_0$ and $\mathbf{p}'_4$. We can calculate that $\mathbf{p}_0$ is encoded into both $\mathbf{p}'_0$ and $\mathbf{p}'_4$. By Algorithm 4, $\mathbf{p}'_4$ is recalculated using its encoding data blocks, while $\mathbf{p}'_0$ is updated from $\mathbf{p}_0$. Overall, four data blocks D_{16}, D_{17}, D_{18}, and D_{19} are retrieved for parity block updates.

As Algorithm 4 exploits the overlapping data blocks to save the network I/Os for parity block updates and ERS has more overlapping data blocks than SRS, ERS shows higher network I/O efficiency than SRS in backward transitioning.

Analysis. The main overhead in backward transitioning lies in two parts. The first part is to read the data blocks to recalculate $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks (Lines 10-12). The second part is to read the non-overlapping data blocks to update the remaining $m \times \frac{l}{k'}$ new parity blocks (Lines 16-18). Algorithm 4 aims to recalculate a set of $m \times (\frac{l}{k_l} - \frac{l}{k'})$ new parity blocks, such that the remaining $m \times \frac{l}{k'}$ new parity blocks are updated from the old parity blocks with the maximum number of overlapping data blocks. In this way, we save the network I/Os for backward transitioning.

We now analyze the network I/O efficiency of Algorithm 4. For example, in ERS$(4,3,5)$ with the row-major order data placement, backward transitioning using Algorithm 4 needs to access six data blocks. However, if we recalculate $\mathbf{p}'_1$ and use $\mathbf{p}_1$ to update $\mathbf{p}'_2$, then the number of data blocks accessed increases to seven.

In ERS$(4,3,5)$ with ERS-aware data placement, backward transitioning using Algorithm 4 needs to access four data blocks. However, if we recalculate $\mathbf{p}'_0$ and use $\mathbf{p}_0$ to update $\mathbf{p}'_4$, then the number of data blocks accessed increases to seven. Thus, Algorithm 4 effectively saves the network I/Os.

In particular, in ERS codes with ERS-aware data placement (Section 3.3), each pre-transitioning stripe shares the most overlapping data blocks with one same-row post-transitioning stripe. There are $k'-k$ non-overlapping data blocks in each pre-transitioning stripe, and a total of $\frac{l}{k'} \times (k'-k) = l \times (1-\frac{k}{k'})$ non-overlapping data blocks. We can see that recalculating the set of $m \times (\frac{l}{k} - \frac{l}{k'})$ new parity blocks and updating the remaining $m \times \frac{l}{k'}$ new parity blocks need to access exactly these non-overlapping data blocks. Thus, the number of data blocks accessed for parity block updates is $l \times (1-\frac{k}{k'})$.

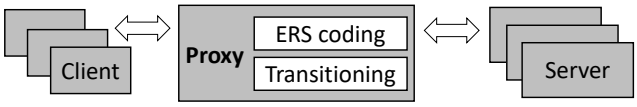


Figure 6. Architecture of the distributed KV store prototype for ERS.

5 System Design and Implementation

We prototype a distributed KV store that realizes ERS codes. We present the architecture (Section 5.1) and metadata management of our prototype (Section 5.2). We discuss how to maintain consistency during redundancy transitioning (Section 5.3). We finally show the implementation details of our prototype atop Libmemcached [5] (Section 5.4).

5.1 Architecture

Figure 6 shows the architecture of our distributed KV store prototype, which mainly comprises multiple clients, multiple servers, and a proxy. The clients interact with the foreground user applications, while the servers store the object data. The proxy acts as an interface for the clients to access the objects in the servers. It implements multiple redundancy strategies (including RS codes [35], SRS codes [36], and ERS codes). It also realizes the basic I/O operations (e.g., PUT, GET, UPDATE, etc.) and the redundancy transitioning processes. In particular, the transitioning processes are coordinated by the proxy in that the proxy downloads the old parity blocks and a subset of data blocks, recomputes the new parity blocks, and finally uploads the new parity blocks. To avoid the proxy being the single-point-of-failure, we can deploy multiple proxies for fault tolerance. It is worth mentioning that the proxy-based design can be seen in other cloud storage systems (e.g., BlueSky [37] and OpenStack [7]).

When first storing objects, users can specify the coding parameters k, m, and k'. Then the objects are stored using ERS(k,m,k'). If the reliability requirements change, users can trigger the executions of redundancy transitioning to update the redundancy schemes from RS(k,m) into RS(k',m), and from RS(k',m) back to RS(k,m).

5.2 Metadata Management

For each object, the distributed KV store prototype divides and encodes it into l data blocks and $m \times \frac{l}{k}$ parity blocks, each of which is stored as a new KV pair. The metadata (e.g., the key length, the value length) of each new KV pair is maintained by each server.

For each set of coding parameters k, m, and k', the proxy maintains two lists: (i) a list of keys of the objects under transitioning and (ii) a list of keys of the objects that have been transitioned into RS(k',m). Also, the proxy avoids the I/O requests to the objects that are currently under transitioning. It ensures that the I/O requests are processed based on RS(k',m) when the objects are transitioned from RS(k,m) to RS(k',m); similarly, the I/O requests are processed based on RS(k,m) when the objects are transitioned from RS(k',m) back to RS(k,m).

We show that the metadata storage overhead is negligible in the proxy side. We focus on objects with regular-size keys (e.g., around 20 bytes [14]) and large-size values (e.g., 1 MB in our evaluation). For a distributed KV store with 100 GB data

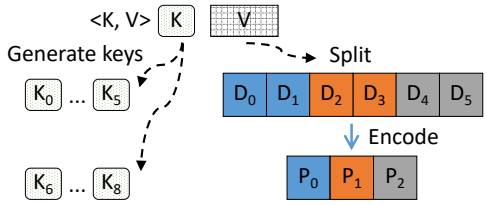


Figure 7. Implementation of ERS$(2, 1, 3)$ on a KV pair. The original KV pair is transitioned into 9 new KV pairs (e.g., $\langle K_0, D_0\rangle$, $\langle K_6, P_0\rangle$).

volume, the total key size accounts for around 2 MB. Even if we record all keys into the two lists, it only costs a few Megabytes of memory in the proxy side. Thus, the metadata storage is not a bottleneck.

5.3 Consistency

We discuss one open issue in our prototype, i.e., maintaining consistency during redundancy transitioning. During transitioning, we need to update the parity blocks in the servers that store the parity blocks. We must guarantee that all parity blocks are consistently and successfully updated. A solution to maintain consistency is to incorporate the two-phase commit protocol [11] into the update process. Specifically, in the first phase, the proxy sends the new parity blocks, and the servers store the new parity blocks in their temporarily allocated buffers and respond acknowledgments to indicate whether the parity blocks have been successfully received and buffered. In the second phase, if the proxy receives the acknowledgements from all servers that buffer the new parity blocks, it notifies all servers to commit and store the new parity blocks; otherwise, it notifies all servers to discard the buffered parity blocks. To reduce the communication overhead of the two-phase commit protocol, we can leverage the piggybacking approach to reduce two rounds of communication into one round [13].

5.4 Implementation

We implement ERS codes atop Libmemcached 1.0.18 [5] that acts as the proxy, by adding about 4,800 SLoC. We deploy multiple Memcached servers [6] for object storage. We leverage the Jerasure Library [32] to realize ERS codes. To show the improvements of ERS codes over SRS codes, we also implement SRS codes into Libmemcached.

Implementation of ERS coding. We take ERS$(2, 1, 3)$ as an example to show the detailed implementation of ERS codes (Figure 7). For each KV pair, we separate the key from the value. We then generate 9 keys (by adding unique suffixes to the original key), among which there are 6 keys (i.e., $K_0, K_1, \cdots, K_5$) for the data blocks and 3 keys (i.e., K_6, K_7, K_8) for the parity blocks. Next, we split the value to obtain 6 data blocks, and encode them to get 3 parity blocks. Thus, we transition the original KV pair into 9 new KV pairs (e.g., $\langle K_0, D_0\rangle$, $\langle K_6, P_0\rangle$).

Distribution of data and parity blocks. To distribute the data and parity blocks, we first locate a server by applying consistent hashing [5] on the original key (i.e., K). Starting from this server, we select $l + m$ successive servers along the clockwise direction on the hash ring. Finally, we distribute the data and parity blocks to the $l + m$ servers according to the specific data placement policy. For example, in Figure 8, we show the distribution of ERS$(2, 1, 3)$ with the row-major order data placement. We locate $Server_4$ by hashing K, then select $Server_4$, $Server_0$, $Server_1$, and $Server_2$, and finally distribute D_0 and D_3, D_1 and D_4, D_2 and D_5, and P_0-P_2 to the four servers, respectively.

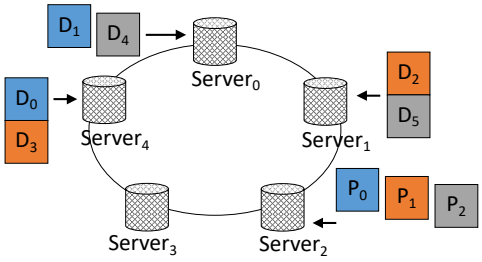


Figure 8. Distribution of ERS$(2, 1, 3)$ with row-major order data placement. D_0, D_3; D_1, D_4; D_2, D_5; and P_0, P_1, P_2 are stored on four successive servers on the hash ring.

6 Evaluation

We present evaluation results on ERS codes using our distributed KV store prototype in three aspects: numerical analysis, local cluster experiments, and cloud experiments. We show the performance gain in redundancy transitioning of ERS codes over SRS codes, the state-of-the-art erasure codes for redundancy transitioning used by Ring [36].

6.1 Numerical Analysis

We analyze the number of data and parity blocks read for parity block updates when an object is transitioned from RS(k, m) to RS(k', m), or from RS(k', m) to RS(k, m).

SRS. In both forward transitioning from RS(k, m) to RS(k', m) and backward transitioning from RS(k', m) to RS(k, m), if we exploit the old parity blocks to generate the new parity blocks, then we have to read all old parity blocks and almost all data blocks. To save the storage I/Os, we resort to reading all data blocks and calculating the new parity blocks directly without using any old parity block. Hence, the number of blocks read is l.

ERS. We consider two variants of ERS: (i) *ERS-1*, the ERS codes using the ERS encoding matrix and row-major order data placement, and (ii) *ERS-2*, the ERS codes with the ERS encoding matrix and ERS-aware data placement; ERS-2 is our complete design. In forward transitioning, we read the old parity blocks and the non-overlapping data blocks of the pre-transitioning stripes to generate the new parity blocks. In particular, the number of blocks read of ERS-2 is $m \times \frac{l}{k} + (\frac{l}{k} - \lfloor \frac{k'}{k} \rfloor \times \frac{l}{k'}) \times (k - r)$, where $r = k' \mod k$. In backward transitioning, we read part of the data blocks to recalculate a subset of new parity blocks, and read the old parity blocks to update the remaining new parity blocks. The number of blocks accessed of ERS-2 is $m \times \frac{l}{k} + l \times (1 - \frac{k}{k'})$.

Analysis of representative parameters. We consider parameters with small k, m, and k', such as $(k, m, k') = (2, 1, 3)$, and $(k, m, k') = (4, 1, 5)$. We also consider parameters that are practically deployed, for example $(k, m) = (6, 3)$ (used by Google ColossusFS [4]), $(k, m) = (8, 3)$ (used by Yahoo Object Store [8]), $(k, m) = (10, 4)$ (used by Facebook HDFS [34]), and $(k, m) = (12, 4)$ (used by Microsoft Azure [19]).

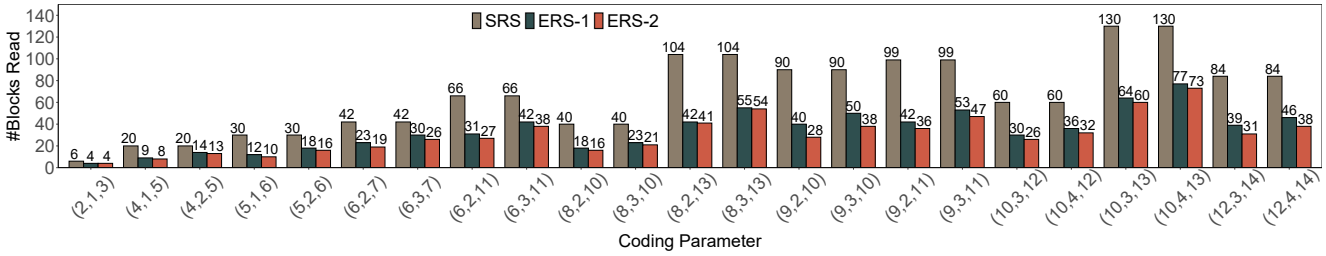


Figure 9. Numerical results of number of blocks read for parity block updates when transitioning from RS(k, m) to RS(k', m).

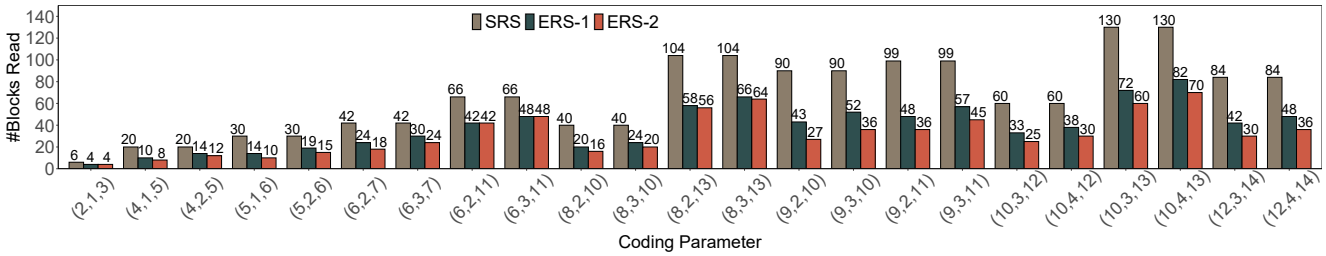


Figure 10. Numerical results of number of blocks read for parity block updates when transitioning from RS(k', m) back to RS(k, m).

Figure 9 shows the results of forward transitioning from RS(k, m) to RS(k', m) for 23 sets of coding parameters. We summarize the observations as follows.

- ERS-1 significantly outperforms SRS in terms of the number of blocks read, while ERS-2 further reduces the I/Os of ERS-1. For example, for $(k, m, k') = (6, 2, 7)$, ERS-1 reduces the number of blocks read of SRS by 45.2%, while ERS-2 further reduces the number of blocks read of ERS-1 by 17.4%.
- In some cases (e.g., $(k, m, k') = (2, 1, 3)$), ERS-1 has the same number of blocks read as ERS-2.
- When m increases, the number of blocks read of SRS (i.e., l) stays unchanged, while those of both ERS-1 and ERS-2 increase. Therefore, ERS-1 and ERS-2 have better improvements over SRS with smaller m. For example, for $(k, m, k') = (12, 3, 14)$, ERS-1 (ERS-2) saves the number of blocks read of SRS by 53.6% (63.1%), while for $(k, m, k') = (12, 4, 14)$, ERS-1 (ERS-2) saves the read I/Os of SRS by 45.2% (54.8%). Note that $m = 3, 4$ are enough for data protection in practical deployment.

Figure 10 shows the results of backward transitioning from RS(k', m) to RS(k, m). We can see that ERS-1 outperforms SRS in terms of the read I/Os, while ERS-2 further outperforms ERS-1, also in backward transitioning. For example, for $(k, m, k') = (4, 1, 5)$, ERS-1 reduces the number of blocks read of SRS by 50.0%, while ERS-2 further reduces the number of blocks read of ERS-1 by 20.0%.

Analysis of general parameters. We now consider more parameters and see how ERS-1 and ERS-2 behave under general parameters. We set $3 \leq k' \leq 14, 2 \leq k \leq k' - 1, 1 \leq m \leq 4$ and $m < k$, and there are 244 sets of parameters.

Table 1 compares SRS, ERS-1, and ERS-2 in four aspects: (i) the number of improved sets (i.e., the number of parameters where ERS-1 (resp. ERS-2) outperforms SRS (resp. ERS-1)); (ii) average ratio (i.e., the average reduction ratio of the number of blocks read of ERS-1 (resp. ERS-2) over SRS (resp. ERS-1)); (iii) best ratio (i.e., the maximum reduction ratio of the number of blocks read of ERS-1 (resp. ERS-2) over SRS (resp. ERS-1)); and (iv) best parameters, the parameters (k, m, k') corresponding to the best ratio.

Table 1
Parametric analysis of SRS, ERS-1, and ERS-2.

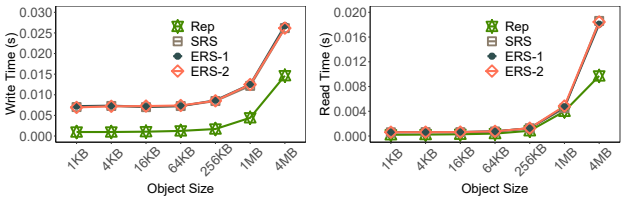
	# improved	average ratio	best ratio	best parameters
ERS-1 over SRS	221	46.2%	73.8%	(6, 1, 14)
ERS-2 over ERS-1	123	14.8%	53.6%	(13, 1, 14)
ERS-1 over SRS (back)	222	36.2%	65.9%	(13, 1, 14)
ERS-2 over ERS-1 (back)	121	20.2%	58.1%	(13, 1, 14)

In forward transitioning from RS(k, m) to RS(k', m), there are 221 sets of parameters where ERS-1 outperforms SRS, and ERS-1 reduces the number of blocks read of SRS by 46.2% on average, and up to 73.8% under $(k, m, k') = (6, 1, 14)$. Note that in the remaining 23 sets of parameters, k' is divisible by k, and both SRS and ERS codes only require the old parity blocks to generate the new parity blocks, so SRS equals ERS-1 in the number of blocks read. There are 123 sets of parameters where ERS-2 further outperforms ERS-1. ERS-2 can save the I/Os of ERS-1 by 14.8% on average, and up to 53.6% under $(k, m, k') = (13, 1, 14)$. In the remaining 98 cases, ERS-1 equals ERS-2 in the number of blocks read, as the row-major order data placement (in ERS-1) already produces a considerable number of overlapping data blocks.

In backward transitioning from RS(k', m) to RS(k, m), ERS-1 outperforms SRS under 222 sets of parameters. On average, ERS-1 reduces the I/Os of SRS by 36.2%. For $(k, m, k') = (13, 1, 14)$, ERS-1 can save the I/Os of SRS by up to 65.9%. Also, ERS-2 outperforms ERS-1 under 121 sets of parameters. ERS-2 reduces the I/Os of ERS-1 by 20.2% on average, and up to 58.1% for $(k, m, k') = (13, 1, 14)$.

6.2 Local Cluster Experiments

Setup. We deploy the distributed KV store prototype on a local cluster comprising 18 physical nodes, each of which runs CentOS version 7.9.2009 with 2 dodeca-core 2.20 GHz Intel(R) Xeon(R) E5-2650 v4 CPUs, 64 GB RAM, and a Seagate



(a) Normal write time (b) Normal read time

Figure 11. Exp#1: Normal write/read time.

ST1000NM0023 7200 RPM 1 TB SATA hard disk. Each node has 10 Gbps of network bandwidth. We deploy the proxy in one node, and the servers in the remaining nodes.

Methodology. We assume the following default configurations. We adopt transitioning between RS$(4, 1)$ and RS$(5, 1)$. We consider various object sizes from 1 KB to 4 MB. We set the network bandwidth as 10 Gbps. We vary different settings in our experiments. We measure the normal read and write time and the transitioning time of an object. The results of each experiment are averaged over ten runs.

Experiment 1 (Normal read/write latency under different object sizes). We first evaluate the normal I/O performance of the distributed KV store prototype and the vanilla Memcached (denoted by Rep), which uses replication for fault tolerance. We consider $(k, m, k') = (4, 1, 5)$ for SRS/ERS-1/ERS-2. We set the replication factor of Rep as two, so that Rep can tolerate the same number of failures as SRS/ERS-1/ERS-2. We evaluate the write time and read time under different object sizes. Figure 11 shows the results.

From Figure 11(a), the write time increases with a larger object size. According to the theoretical analysis, both SRS codes and ERS codes divide an object into l data blocks, and encode them to produce $m \times \frac{l}{k}$ parity blocks. Thus, SRS codes should have similar write latency to ERS codes. From Figure 11(a), the experimental results comply with the theoretical analysis. Also, SRS codes and ERS codes have higher write latency than Rep. The reasons are two-fold. First, SRS and ERS codes generate $l + m \times \frac{l}{k}$ requests for writing, while Rep only requires two requests. Second, SRS and ERS codes connect to $k' + m$ servers when sending the requests, while Rep only connects to two servers. Thus, SRS and ERS codes have more connection overhead.

From Figure 11(b), the read time also increases with a larger object size. The read latency of SRS/ERS-1/ERS-2 is also higher than that of Rep (e.g., object size $\geq$ 1 MB).

Experiment 2 (Transitioning latency under different object sizes). We evaluate the transitioning time under different object sizes. We consider transitioning between RS$(4, 1)$ and RS$(5, 1)$. Figure 12 shows the results (the error bars show the maximum and minimum results across ten runs).

Forward transitioning: From Figure 12(a), we can see that the transitioning time increases with a larger object size, and ERS-1 and ERS-2 constantly outperform SRS. Note that ERS codes not only reduce the transitioning I/Os, but also connect to fewer servers during transitioning (e.g., SRS connects to six servers while ERS-1 connects to only four servers). For example, ERS-1 reduces the forward transitioning time of SRS from 8.7% to 31.3%, while ERS-2 reduces the forward transitioning time of SRS from 10.8% to 38.7%, across all object sizes. We can also see that the improvements of ERS-1 and ERS-2 over SRS become more prominent with larger object sizes, since the network now dominates the overall performance. ERS-2 has smaller transitioning time than ERS-1, which is consistent with the numerical results.

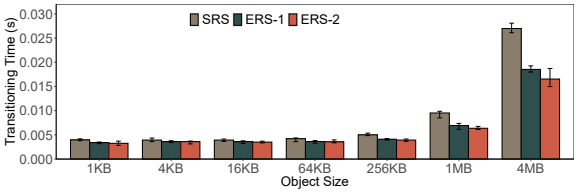
Backward transitioning: From Figure 12(b), ERS-1 reduces the backward transitioning time of SRS from 4.7% to 15.2%, while ERS-2 reduces the backward transitioning time of SRS from 10.8% to 26.5%, across all object sizes. This validates the efficiency of ERS codes in backward transitioning under different object sizes.

Experiment 3 (Transitioning latency under different coding parameters). We next evaluate the transitioning time under different coding parameters. We consider six sets of (k, m, k'), i.e., $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$. We consider two object sizes: 1 MB and 4 MB. Figure 13 shows the results (the error bars show the maximum and minimum results across ten runs).

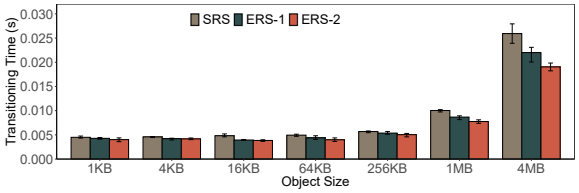
Forward transitioning: From Figure 13(a), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 27.4% and 33.2%, 27.6% and 31.2%, 41.7% and 49.6%, 36.7% and 47.3%, 48.4% and 54.9%, and 43.8% and 51.2%, under the object size of 1 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. From Figure 13(b), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 31.3% and 38.7%, 27.4% and 39.4%, 51.2% and 56.0%, 44.3% and 61.4%, 53.8% and 61.2%, and 55.4% and 65.8%, under the object size of 4 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. We can see that ERS-1 greatly reduces the transitioning time of SRS due to the effect of ERS encoding matrix. ERS-2 can further lower the transitioning time of ERS-1 via designing ERS-aware data placement with more overlapping data blocks. Moreover, ERS codes achieve larger gains with a larger object size under larger coding parameters, and the actual improvement ratios are comparable to the theoretical improvement ratios. For example, ERS-2 reduces the read I/Os of SRS by 68.9% and 63.1%, under the parameters $(9, 2, 10)$ and $(12, 3, 14)$, respectively (Figure 9). In our experiments, ERS-2 reduces the transitioning time of SRS by 61.4% and 65.8%, with the object size of 4 MB for $(9, 2, 10)$ and $(12, 3, 14)$, respectively.

Backward transitioning: From Figure 13(c), ERS-1 and ERS-2 save the backward transitioning time of SRS by 13.5% and 22.7%, 12.4% and 23.1%, 34.1% and 42.1%, 30.9% and 44.5%, 36.4% and 52.0%, and 36.1% and 48.9%, under the object size of 1 MB for parameters $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. From Figure 13(d), ERS-1 and ERS-2 save the backward transitioning time of SRS by 15.2% and 26.5%, 14.5% and 35.7%, 38.8% and 50.4%, 36.5% and 59.4%, 41.4% and 56.1%, and 45.0% and 60.9%, under the object size of 4 MB for parameters $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. Thus, ERS-1 and ERS-2 greatly improve the backward transitioning efficiency as we can exploit more overlapping data blocks for parity block updates (Algorithm 4). Also, ERS-2 shows notable improvement over ERS-1.

Experiment 4 (Transitioning latency under different bandwidth). We now evaluate the transitioning performance under different bandwidth. We configure the bandwidth to be 10 Gbps, 1 Gbps, 500 Mbps, and 200 Mbps, respectively. We consider $(k, m, k') = (4, 1, 5)$ and $(k, m, k') = (5, 1, 6)$ with the object size of 1 MB. Figure 14 shows the results.

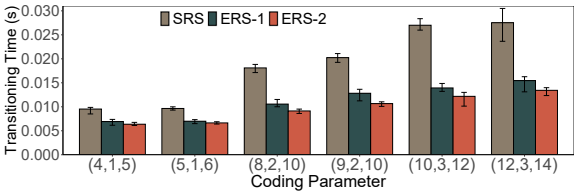


(a) Forward transitioning

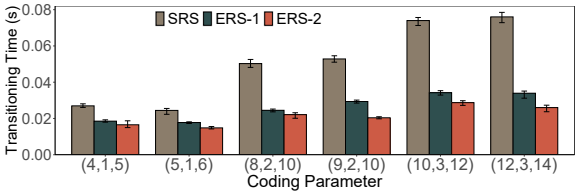


(b) Backward transitioning

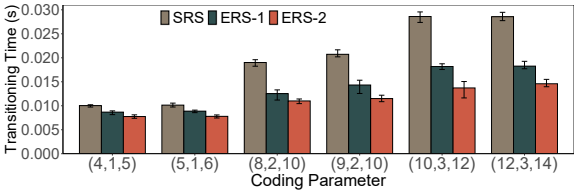
Figure 12. Exp#2: Transitioning time under different object sizes.



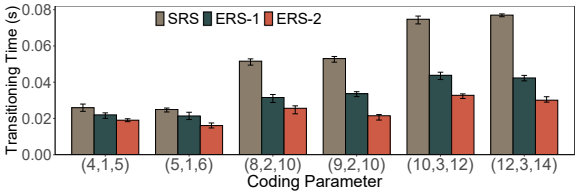
(a) Forward transitioning, object size 1 MB



(b) Forward transitioning, object size 4 MB



(c) Backward transitioning, object size 1 MB



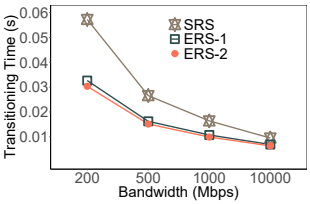
(d) Backward transitioning, object size 4 MB

Figure 13. Exp#3: Transitioning time under different coding parameters.

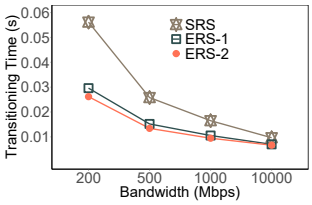
Forward transitioning: From Figure 14(a), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 43.0% and 47.0%, 38.9% and 43.0%, 34.5% and 39.6%, and 27.4% and 33.2%, for parameter $(k, m, k') = (4, 1, 5)$ under the bandwidth of 200 Mbps, 500 Mbps, 1 Gbps, and 10 Gbps, respectively. From Figure 14(b), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 47.3% and 53.4%, 41.1% and 48.0%, 36.1% and 42.7%, and 27.6% and 31.2%, for parameter $(k, m, k') = (5, 1, 6)$ under different bandwidth. We can see that the improvements of ERS-1 and ERS-2 over SRS are greater with more limited bandwidth.

Backward transitioning: From Figure 14(c), ERS-1 and ERS-2 reduce the backward transitioning time of SRS by 36.5% and 44.5%, 29.9% and 39.0%, 23.9% and 34.0%, and 13.5% and 22.7%, for parameter $(k, m, k') = (4, 1, 5)$ when the bandwidth is 200 Mbps, 500 Mbps, 1 Gbps, and 10 Gbps, respectively. From Figure 14(d), ERS-1 and ERS-2 reduce the backward transitioning time of SRS by 38.8% and 51.0%, 32.1% and 44.8%, 26.1% and 41.9%, and 12.4% and 23.1%, for parameter $(k, m, k') = (5, 1, 6)$ under different bandwidth. This again validates that ERS-1 and ERS-2 have larger performance gains with more limited network bandwidth.

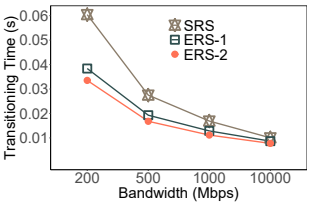
Experiment 5 (Impact of selecting the order of parity blocks to update in backward transitioning). We now evaluate the effectiveness of Algorithm 4, whose key novelty is to select the proper order of new parity blocks to update. We compare ERS-2 with order selection to ERS-2 without order selection in backward transitioning. We consider six sets of (k, m, k'), i.e., $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, and two object sizes, i.e., 1 MB and 4 MB. Under each (k, m, k'), there is one reverse Equation with two post-transitioning stripes (labeled by s'_j and s'_z) involved. If ERS-2 with order selection recalculates $\mathbf{p}'_j$ and



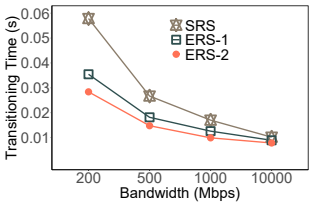
(a) Forward, $(4, 1, 5)$, 1 MB



(b) Forward, $(5, 1, 6)$, 1 MB



(c) Backward, $(4, 1, 5)$, 1 MB

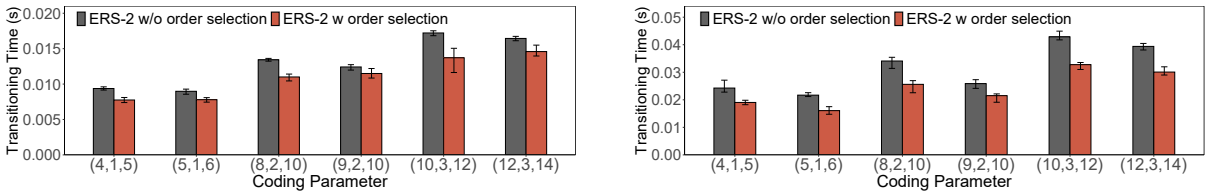


(d) Backward, $(5, 1, 6)$, 1 MB

Figure 14. Exp#4: Transitioning time under different network bandwidth.

updates $\mathbf{p}'_z$, then we enforce ERS-2 without order selection to recalculate $\mathbf{p}'_z$ and update $\mathbf{p}'_j$. Figure 15 shows the results (the error bars show the maximum and minimum results across ten runs).

From Figure 15(a), ERS-2 with order selection reduces the backward transitioning time of ERS-2 without order selection by 17.3%, 13.2%, 18.0%, 7.4%, 20.3%, and 11.3%, under the object size of 1 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. From Figure 15(b), ERS-2 with order selection reduces the backward transitioning time of ERS-2 without order selection by 21.5%, 26.0%, 24.9%, 16.9%, 23.6%, and 23.7%, under the object size of 4 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. This verifies that Algorithm 4 can be executed with small amount of network I/Os, by



(a) Backward transitioning, object size 1 MB

(b) Backward transitioning, object size 4 MB

Figure 15. Exp#5: Impact of selecting the order of new parity blocks to update in backward transitioning.

selecting the proper order of new parity blocks to update in backward transitioning.

6.3 Cloud Experiments

Setup. We conduct experiments on Alibaba Cloud (ECS) [1] to evaluate ERS codes in a cloud environment. We deploy our ERS KV store prototype on 18 instances in different Zones in Beijing China. Specifically, we deploy one ecs.g6.8xlarge instance in Zone K to act as the proxy. This instance is equipped with 32 vCPU and 128 GiB memory and runs CentOS 7.9 64-bit. We deploy 17 ecs.r6e.xlarge instances in Zones I, J, and L to act as the servers. Each of such instances is equipped with 4 vCPU and 32 GiB memory and runs CentOS 7.9 64-bit.

Experiment 6 (Transitioning latency under different object sizes in cloud setting). We repeat Experiment 2 in cloud setting. Figure 16 shows the results (the error bars show the maximum and minimum results across ten runs).

From Figure 16(a), ERS-1 reduces the forward transitioning time of SRS from 18.9% to 48.4%, while ERS-2 reduces the forward transitioning time of SRS from 20.6% to 52.3%, across all object sizes. From Figure 16(b), ERS-1 reduces the backward transitioning time of SRS from 10.9% to 41.5%, while ERS-2 reduces the backward transitioning time of SRS from 11.2% to 49.7%, across all object sizes. The improvements in cloud setting are more significant than in local setting, since the cloud has less available bandwidth than the local setting.

Experiment 7 (Transitioning latency under different coding parameters in cloud setting). Figure 17 repeats Experiment 3 in cloud setting (the error bars show the maximum and minimum results across ten runs).

From Figure 17(a), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 43.5% and 48.5%, 46.1% and 47.7%, 36.2% and 37.5%, 29.3% and 31.1%, 31.8% and 34.2%, and 26.0% and 29.8%, under the object size of 1 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. From Figure 17(b), ERS-1 and ERS-2 reduce the forward transitioning time of SRS by 48.4% and 52.3%, 51.5% and 57.5%, 38.7% and 48.6%, 39.5% and 51.4%, 51.3% and 57.1%, and 43.7% and 52.9%, under the object size of 4 MB for $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively.

From Figure 17(c), ERS-1 and ERS-2 save the backward transitioning time of SRS by 38.4% and 45.0%, 42.7% and 47.6%, 32.0% and 33.8%, 29.4% and 31.2%, 28.0% and 30.9%, and 24.2% and 26.1%, under the object size of 1 MB for parameters $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively. From Figure 17(d), ERS-1 and ERS-2 save the backward transitioning time of SRS by 41.5% and 49.7%, 44.7% and 55.9%, 38.0% and 47.7%, 40.3% and 48.7%, 42.5% and 45.6%, and 21.1% and 40.7%, under the object size of 4 MB for parameters $(4, 1, 5)$, $(5, 1, 6)$, $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$, respectively.

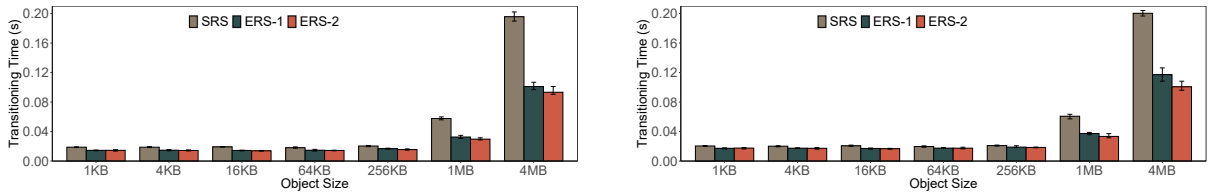
Under small parameters (e.g., $(4, 1, 5)$ and $(5, 1, 6)$), the improvements in cloud setting are greater than in local setting. However, under large parameters (e.g., $(8, 2, 10)$, $(9, 2, 10)$, $(10, 3, 12)$, and $(12, 3, 14)$), the improvements in cloud setting are not as significant as in local setting. The reason is that in cloud setting, redundancy transitioning under large parameters is more likely to be bottlenecked by stragglers, which may offset the gains brought by reduced I/Os in ERS codes.

7 Related Work

Redundancy transitioning. Several studies address efficient redundancy transitioning for different storage architectures. AutoRAID [39], DiskReduce [17], and EAR [26] study the transitioning from replication to RAID or erasure coding. Some studies propose efficient data redistribution approaches for RAID [40], [48], [50] and erasure-coded distributed storage systems [20], [42], [43], [44], [49]. In this work, we specifically focus on redundancy transitioning for KV objects in in-memory KV stores, which pose high elasticity demands in real-world deployment [2], [29].

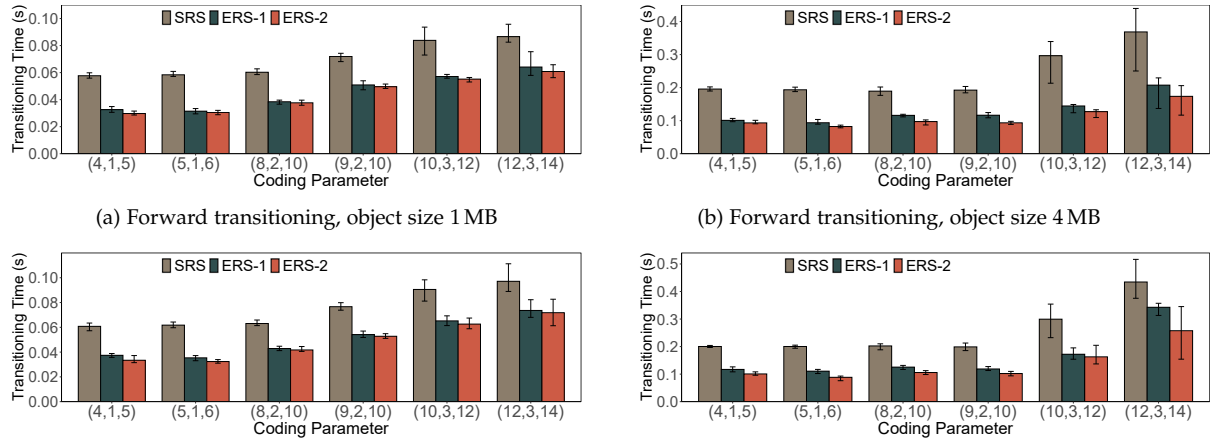
Some studies [27], [28], [46] apply stripe merging for redundancy transitioning in erasure-coded storage. They focus on limited setting that changes from (k, m) to (xk, m), while our work considers a more general setting that transitions between (k, m) and (k', m).

Erasure coding in KV stores. Erasure coding has been extensively studied in modern KV stores for low-cost fault tolerance [3], [25], data availability [13], [47], and tail latency mitigation [33]. In particular, prior studies [14], [36], [45], [51] address the elasticity of erasure-coded in-memory KV stores. PaRS [51] adjusts the replication factor of the data blocks that have varying popularity, yet it requires data block relocation and incurs expensive parity block updates. TEA [45] realizes the transitioning from replication to erasure coding in in-memory stores. ECHash [14] avoids parity block updates via a new fragmented erasure coding model with node additions or removals, while keeping coding parameters unchanged; in contrast, our work addresses the change of coding parameters. The closest work to ours is Ring [36], which also addresses redundancy transitioning for KV objects. In contrast to Ring, our work puts specific emphasis on mitigating I/Os during redundancy transitioning via a co-design of encoding matrix and data placement.



(a) Forward transitioning (b) Backward transitioning

Figure 16. Exp#6: Transitioning time under different object sizes in cloud setting.



(a) Forward transitioning, object size 1 MB (b) Forward transitioning, object size 4 MB

(c) Backward transitioning, object size 1 MB (d) Backward transitioning, object size 4 MB

Figure 17. Exp#7: Transitioning time under different coding parameters in cloud setting.

8 Conclusion

We study how to enable efficient redundancy transitioning with small network I/Os in erasure-coded distributed KV stores. We propose a new class of erasure codes, ERS codes, to mitigate network I/Os for redundancy transitioning. ERS codes eliminate data block relocation and reduce network I/Os for parity block updates via a co-design of the ERS encoding matrix and ERS-aware data placement. We show that ERS codes can be applied in both forward transitioning and backward transitioning cases. We implement a distributed KV store that realizes ERS codes atop Memcached to allow redundancy transitioning. Evaluation with numerical studies, local cluster, and cloud experiments validates the efficiency of ERS codes in redundancy transitioning.

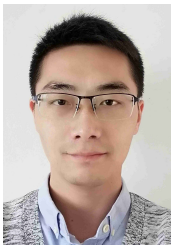
Acknowledgements

The work is supported by NSFC (62202440, 61832011), the Anhui Provincial Natural Science Foundation under Grant 2208085QF189, and Research Grants Council of HKSAR (AoE/P-404/18).

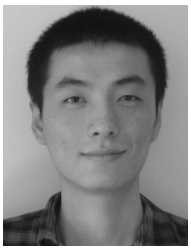
References

[1] Alibaba Cloud Elastic Compute Service (ECS). https://www.alibabacloud.com/product/ecs.
[2] Amazon ElastiCache. https://docs.aws.amazon.com/elasticache.
[3] Erasure code - Ceph documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/.
[4] Google Colossus File System. https://cloud.google.com/files/storage_architecture_and_challenges.pdf.
[5] Libmemcached. https://libmemcached.org/libMemcached.html.
[6] Memcached. https://memcached.org.
[7] Openstack. https://openstack.org.
[8] Yahoo Cloud Object Store. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at.
[9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in Mapreduce clusters. In *Proc. of ACM EuroSys*, 2011.
[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS Performance Evaluation Review*, 2012.
[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.
[12] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and bench-marking RocksDB key-value workloads at Facebook. In *Proc. of USENIX FAST*, 2020.
[13] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Trans. on Storage (TOS)*, 13(3):25, 2017.
[14] L. Cheng, Y. Hu, and P. P. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SoCC*, 2019.
[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOPS*, 2007.
[16] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.
[17] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Proc. of ACM PDSW*, 2009.
[18] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.
[19] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.
[20] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie. Scale-RS: An efficient scaling scheme for RS-coded storage clusters. *IEEE Trans. on Parallel and Distributed Systems*, 26(6):1704–1717, 2014.
[21] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proc. of ACM SOSP*, 2013.

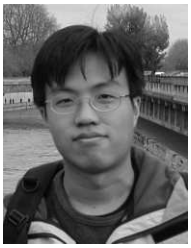
[22] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. Rashmi, and G. R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *Proc. of USENIX OSDI*, 2020.
[23] S. Kadekodi, K. Rashmi, and G. R. Ganger. Cluster storage systems gotta have HeART: Improving storage efficiency by exploiting disk-reliability heterogeneity. In *Proc. of USENIX FAST*, 2019.
[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.
[25] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, 2015.
[26] R. Li, Y. Hu, and P. P. Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Trans. on Parallel and Distributed Systems*, 28(9):2500–2513, 2017.
[27] F. Maturana, C. Mukka, and K. Rashmi. Access-optimal linear MDS Convertible Codes for all parameters. In *Proc. of IEEE ISIT*, 2020.
[28] F. Maturana and K. Rashmi. Convertible Codes: New class of codes for efficient conversion of coded data in distributed storage. In *Innovations in Theoretical Computer Science (ITCS)*, 2020.
[29] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.
[30] J. Plank and C. Huang. Tutorial: Erasure coding for storage applications. In *Slides presented at USENIX FAST*, 2013.
[31] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.
[32] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical report, University of Tennessee, Tech. Rep. CS-08-627, 2008.
[33] 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. of USENIX OSDI*, 2016.
[34] K. 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. of USENIX HotStorage*, 2013.
[35] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
[36] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, 2018.
[37] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proc. of USENIX FAST*, 2012.
[38] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.
[39] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108–136, 1996.
[40] C. Wu, X. He, J. Han, H. Tan, and C. Xie. SDM: A stripe-based data migration scheme to improve the scalability of RAID-6. In *Proc. of IEEE Cluster*, 2012.
[41] 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. of IEEE SRDS*, 2020.
[42] S. Wu, Z. Shen, and P. P. Lee. On the optimal repair-scaling trade-off in Locally Repairable Codes. In *Proc. of IEEE INFOCOM*, 2020.
[43] S. Wu, Y. Xu, Y. Li, and Z. Yang. I/O-Efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Trans. on Parallel and Distributed Systems*, 27(9):2639–2652, 2016.
[44] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in HDFS. In *Proc. of USENIX FAST*, 2015.
[45] B. Xu, J. Huang, Q. Cao, and X. Qin. TEA: A traffic-efficient erasure-coded archival scheme for in-memory stores. In *Proc. of ACM ICPP*, 2019.
[46] Q. Yao, Y. Hu, L. Cheng, P. P. Lee, D. Feng, W. Wang, and W. Chen. Stripemerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *Proc. of IEEE ICDCS*, 2021.
[47] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, 2017.
[48] G. Zhang, W. Zheng, and K. Li. Rethinking RAID-5 data layout for better scalability. *IEEE Trans. on Computers*, 63(11):2816–2828, 2014.
[49] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou. Toward optimal storage scaling via network coding: From theory to practice. In *Proc. of IEEE INFOCOM*, 2018.
[50] W. Zheng and G. Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proc. of USENIX FAST*, 2011.
[51] P. Zhou, J. Huang, X. Qin, and C. Xie. PaRS: A popularity-aware redundancy scheme for in-memory stores. *IEEE Trans. on Computers*, 68(4):556–569, 2018.



Si Wu received the B.Eng. and Ph.D. degrees in Computer Science from University of Science and Technology of China in 2011 and 2016, respectively. He is now an associate researcher in School of Computer Science and Technology, University of Science and Technology of China. His research interests include storage reliability and distributed storage.



Zhirong Shen received the B.Eng. degree from University of Electronic Science and Technology of China in 2010, and the Ph.D. degree in Computer Science from Tsinghua University in 2016. He is now an associate professor at Xiamen University. His research interests include storage reliability and storage security. He is a member of the IEEE.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now a professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security.

Zhiwei Bai received Bachelor's degree in Computer Science and Technology in Xidian University in 2021. He is now an postgraduate student in School of Computer Science and Technology, University of Science and Technology of China. His research interests include erasure coding and key-value stores.

Yinlong Xu received the Bachelor's degree in mathematics from Peking University in 1983, and the master and PhD degrees in computer science from University of Science and Technology of China (USTC) in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology at USTC. Currently, he is leading a group of research students in doing some networking and high performance computing research. His research interests include network coding, wireless network, combinatorial optimization, design and analysis of parallel algorithm, parallel programming tools, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences in 2006.