Enabling I/O-Efficient Redundancy Transitioning in Erasure-Coded KV Stores via Elastic Reed-Solomon Codes

Si Wu[†], Zhirong Shen[‡], and Patrick P. C. Lee[†]
[†]The Chinese University of Hong Kong [‡]Xiamen University

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

I. Introduction

Key-value (KV) stores improve scalability and access performance of object storage compared to traditional relational databases. To provide reliability guarantees against frequent failures [13], modern KV stores increasingly adopt *erasure coding* to provide low-cost data redundancy [2], [9], [10], [18], [24], [36]. Compared to replication, erasure coding significantly reduces the amount of redundancy to attain the same degree of fault tolerance [29]. Among many erasure coding constructions, Reed-Solomon (RS) codes [26] 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 [8], [16], 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 [34]. 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 [11], 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.
- **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 [17]. Also, the reliability importance varies across data types, in which the loss of important data may imply costly recovery [27]. Such important data may be protected by erasure codes with higher redundancy.

However, realizing I/O-efficient redundancy transitioning 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 I/O costs. 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. 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 newly computed 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 I/O-efficient redundancy transitioning for erasure-coded KV stores. ERS codes build on the decoupling of block-to-node mappings [27] 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 II-C). 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 I/Os due to parity updates. Note that ERS codes preserve the storage-optimality of the original RS codes.

We implement a KV store prototype that realizes ERS codes based on Memcached [5]. Our prototype supports all basic KV

operations (e.g., PUT, GET, UPDATE, etc.), while enabling redundancy transitioning. Our prototype experiments suggest that ERS codes reduce the latency of redundancy transitioning by up to 55.6% compared to the state-of-the-art stretched RS codes in [27], which incur high parity update overhead.

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

II. Background and Motivation

We present the background of erasure coding (Section II-A). We define the redundancy transitioning problem and state its challenges (Section II-B). We further motivate via examples our solutions to the challenges (Section II-C).

A. Erasure Coding in KV Stores

Erasure coding incurs much less storage redundancy for the same degree of fault tolerance compared to replication [29]. In this work, we focus on Reed-Solomon (RS) codes [26], a popular family of erasure codes that have been widely studied in modern KV stores [2], [9], [18], [24], [36]. We construct RS codes, denoted by $\mathrm{RS}(k,m)$, with two configurable parameters k and m. $\mathrm{RS}(k,m)$ takes k *data blocks* (denoted by $D_0,\cdots,D_{k-1}$) as input for encoding, and generates m *parity blocks* (denoted by $P_0,\cdots,P_{m-1}$), 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*. A practical KV store comprises multiple stripes that are encoded independently. It distributes each stripe of $k+m$ blocks across $k+m$ nodes (denoted by $X_0,...,X_{k+m-1}$) to tolerate any m node failures.

Mathematically, the encoding process of RS codes can be specified via an $m\times k$ *encoding matrix* (denoted by $\mathbf{G}_{m\times k}$), constructed by the Vandermonde matrix [22]. Given a *data vector* (i.e., a column vector of k data blocks), RS codes multiply the encoding matrix $\mathbf{G}_{m\times k}$ by the data vector to compute the *parity vector* (i.e., a column vector of m parity blocks) over the Galois Field $\mathrm{GF}(2^\omega)$, 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* [2], [18], [24], which divides each object into k data blocks for encoding, and *cross-object coding* [9], [10], [36], 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 [24]. 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.

B. 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 I). In this work, we pay special attention to the transitioning from $\mathrm{RS}(k,m)$ to $\mathrm{RS}(k',m)$, where $k<k'$ (i.e., the number of tolerable failures m remains unchanged). By increasing k, we can reduce the storage redundancy and increase the overall storage efficiency. We pose the transitioning for general coding parameters k and m as future work.

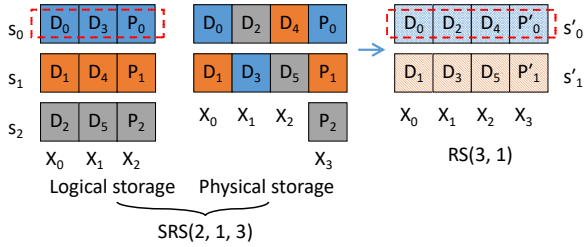


Fig. 1. Example of SRS(2,1,3), which eliminates data block relocation in the transitioning from RS(2,1) to RS(3,1). The data and parity blocks of the same color constitute a stripe.

Redundancy transitioning inevitably changes the encoding layout of a stripe (in our case, 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 additional I/O costs in KV stores (Section I). In this work, we focus on mitigating the I/O costs in redundancy transitioning.

To eliminate data block relocation during redundancy transitioning, Ring [27] proposes *Stretched Reed-Solomon (SRS)* codes (denoted by $\mathrm{SRS}(k,m,k')$), whose idea is to store the k data blocks of $\mathrm{RS}(k,m)$ in $k'>k$ nodes (i.e., relaxing the tight coupling of the same block-to-node mappings). $\mathrm{SRS}(k,m,k')$ operates on a group of multiple stripes. It first computes the least common multiple (LCM) of k and k', denoted by $l=\mathrm{lcm}(k,k')$. 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 $\mathrm{RS}(k,m)$. 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 $\mathrm{SRS}(k,m,k')$ are now stored in k' nodes, transitioning from $\mathrm{RS}(k,m)$ to $\mathrm{RS}(k',m)$ 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 I/O-efficient redundancy transitioning due to the expensive parity block updates. The reasons are two-fold.

Challenge 1: The encoding matrices before and after redundancy 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 $\mathbf{G}_{3\times4}$ is multiplied by the four data blocks $\{D_0, D_1, \cdots, D_3\}$:

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \mathbf{G}_{3\times4} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 15 & 2 & 14 \\ 1 & 12 & 8 & 5 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix}. \quad (2)$$

We next show the encoding process of the post-transitioning stripe to generate $\{P'_0, P'_1, P'_2\}$, formed by multiplying the matrix $\mathbf{G}_{3\times5}$ by $\{D_0, D_1, \cdots, D_4\}$:

$$\begin{bmatrix} P'_0 \\ P'_1 \\ P'_2 \end{bmatrix} = \mathbf{G}_{3\times5} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 & 14 \\ 1 & 10 & 4 & 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix}. \quad (3)$$

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 $\mathrm{RS}(4,3)$ to $\mathrm{RS}(5,3)$, we need to access D_1, D_2, D_3 and D_4 (i.e., a total of 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 I/O overhead of parity block updates is higher for larger coding parameters (e.g., from $\mathrm{RS}(4,3)$ to $\mathrm{RS}(5,3)$), where we have to retrieve more data blocks.

C. Motivation

Our major goal is to mitigate the 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*. 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.

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 redundancy transitioning and add dummy blocks for encoding. For example, in order to transition from RS(4,3) to RS(5,3), the pre-transitioning encoding process can be denoted by:

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \mathbf{G}_{3\times5} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 10 & 14 \\ 1 & 10 & 4 & 15 & 11 \end{bmatrix} \times \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ 0 \end{bmatrix}. \quad (4)$$

The key difference between the conventional and new encoding mechanisms is that the conventional approach employs an $m \times k$ dimensional encoding matrix $\mathbf{G}_{m\times k}$ to encode the k data blocks, while we exploit an $m \times k'$ dimensional encoding matrix $\mathbf{G}_{m\times k'}$ to encode the k' data blocks (with k data blocks and $k'-k$ dummy blocks). For example, the conventional approach uses $\mathbf{G}_{3\times4}$ to encode D_0, D_1, D_2 and D_3 (see Equation (2)). We now adopt $\mathbf{G}_{3\times5}$ to encode D_0, D_1, D_2, D_3 and one dummy block (Equation (4)).

Recall that in the transitioning from $\mathrm{RS}(4,3)$ to $\mathrm{RS}(5,3)$ 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_1, D_2, and 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 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 motivating 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 I/Os of parity updates.

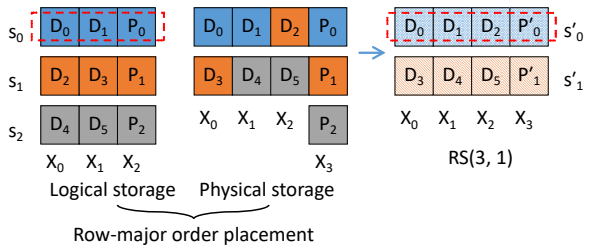


Fig. 2. Example of transitioning from RS(2,1) to RS(3,1) under the placement in row-major order. The blocks of the same color constitute a stripe.

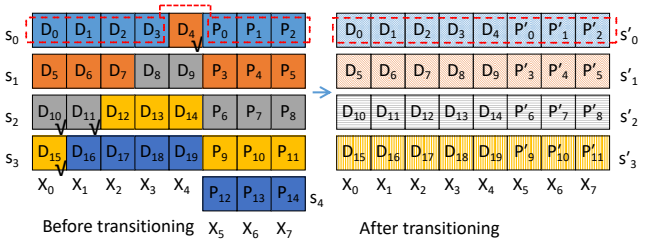


Fig. 3. Row-major order placement for $(k,m,k') = (4,3,5)$. The data and parity blocks of the same color constitute a stripe. The data blocks with black check marks indicate the non-overlapping data blocks.

III. Elastic Reed-Solomon Codes

We present Elastic Reed-Solomon (ERS) codes, a new family of erasure codes for I/O-efficient redundancy transitioning. ERS codes exploit both new encoding matrix construction and data placement to increase the number of overlapping data blocks, so as to mitigate the I/Os for parity updates. We first present an overview of ERS codes (Section III-A). We then present the design details of the encoding matrix of ERS codes to increase the number of overlapping data blocks (Section III-B). Finally, we present our data placement strategy based on our new encoding matrix to further increase the number of overlapping data blocks (Section III-C).

A. Design Overview

ERS codes (denoted by ERS(k,m,k')) build on the decoupling of block-to-node mappings in SRS codes [27] by storing the k data blocks of RS codes in k' nodes, where $k \leq k'$. 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 $D_0, \cdots, D_{l-1}$. 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. Note that 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 I/O overhead for parity updates. Like SRS codes, as the data blocks are now distributed over k' nodes, ERS codes also do not require 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 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 can reduce the I/Os for parity block updates.

Figure 2 shows an example of ERS$(2,1,3)$ with the row-major order 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., $D_0, \cdots, D_5$) 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 $l = 6$ 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 require to retrieve D_2 to update the parity blocks in redundancy transitioning.

B. 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 $\frac{l}{k'} \times k'$ dimensional array that is composed of the data blocks from $\frac{l}{k}$ pre-transitioning stripes. For each pre-transitioning stripe, we exploit an $m \times k'$ dimensional 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 [22]. Note that the dummy blocks are not involved in the encoding operations, so they do not incur extra computational overhead. Our approach of utilizing an enlarged matrix is analogous to the *shortening* scheme [21], which is carefully tailored for redundancy transitioning.

Algorithm details. If a block is stored on 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 on 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 a larger encoding matrix. To be specific, 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 the physical storage (Lines 3-5). There are k data blocks in s_i and we add extra $k' - k$ dummy blocks to constitute k' blocks (Line 6), and then encode the k' blocks (Line 7).

Example. For ERS$(4,3,5)$, we show the encoding process of s_1. The encoding matrix we use for encoding is $\mathbf{G}_{3 \times 5}$. The node ids of the four data blocks D_4, D_5, D_6, D_7 of s_1 are 4, 0,

Algorithm 1 The encoding method

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**

1, 2, respectively (Figure 3). With one extra dummy block, the encoding process is shown as follows.

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

Redundancy transitioning process. We now explore the transitioning process using the designed 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., $\mathbf{p}_i = [P_{i\times m}, \cdots, P_{i\times m+m-1}]^T$, 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., $\mathbf{p}'_i = [P'_{i\times m}, \cdots, P'_{i\times m+m-1}]^T$. Let $\mathbf{g}_j$ $(0 \le j \le k'-1)$ be the j-th column of the 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 $\mathbf{g}_4 = [1, 14, 11]^T$. The transitioning method is illustrated in Algorithm 2. 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 in 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 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 the new parity blocks in $\mathbf{p}'_z$ (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 $\mathbf{p}'_0$.

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

Algorithm 2 The transitioning method

1: **for** $i = 0$ to $\frac{l}{k'}-1$ **do**
2: Initialize $\mathbf{p}'_i$ to be zero vector
3: **end for**
4: **for** each pre-transitioning stripe s_i $(0 \le i \le \frac{l}{k}-1)$ **do**
5: Find the post-transitioning stripe s'_j, such that s_i shares the most overlapping data blocks with s'_j
6: // $\mathbf{p}_i$ is used to generate $\mathbf{p}'_j$
7: Set $\mathbf{p}'_j = \mathbf{p}'_j + \mathbf{p}_i$
8: Retrieve the non-overlapping data blocks of s_i
9: **for** each non-overlapping data block D_x **do**
10: $y \leftarrow$ node id of D_x
11: // Use D_x to update $\mathbf{p}'_j$
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 $\mathbf{p}'_z$
15: Set $\mathbf{p}'_z = \mathbf{p}'_z + \mathbf{g}_y \times D_x$
16: **end for**
17: **end for**

Since s_1 and s_2 both 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 $\mathbf{p}'_1$.

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

As we use the enlarged encoding matrix, only a small number of non-overlapping data blocks are needed for parity updates. For example in Figure 3, in transitioning from RS$(4,3)$ to RS$(5,3)$ using the designed matrix, we only need to retrieve four non-overlapping data blocks, i.e., D_4, D_{10}, D_{11} and D_{15} for parity updates. However, with SRS$(4,3,5)$, the transitioning exhibits very few overlapping data blocks, so we need to access nearly all data blocks.

C. Data Placement Design

In Section III-B, we utilize an enlarged 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 the placement strategy to further increase the number of overlapping data blocks based on the enlarged matrix.

Overall idea. We attempt to put the data blocks of $\frac{l}{k}$ pre-transitioning stripes into a $\frac{l}{k'} \times k'$ dimensional 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

Algorithm 3 The placement policy

1: // Put α same-row stripes
2: **for** the i-th $(0 \le i \le \beta - 1)$ 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 $(\beta \le i \le \frac{l}{k'} - 1)$ 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 $(0 \le i \le \beta - 1)$ 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**

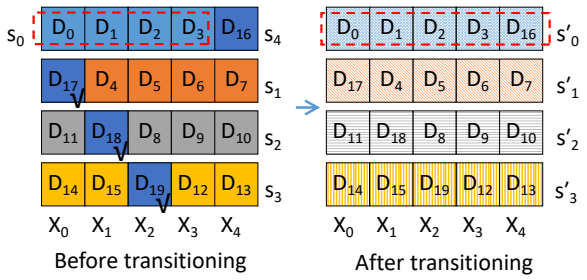


Fig. 4. Placement illustration for $(k, m, k') = (4, 3, 5)$. We omit the placement of the parity blocks as it does not affect the number of overlapping data blocks. The data blocks of the same color constitute a stripe. Note that s_0, s_1, s_2, s_3 are $\alpha = 4$ same-row stripes, and s_4 is $\beta = 1$ cross-row stripe. The data blocks with black check marks indicate the non-overlapping data blocks.

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., with number of overlapping data blocks of 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., with number of overlapping data blocks of 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 to make the number of overlapping data blocks of this stripe be r, and $k - r$ data blocks on other rows.

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

Algorithm details. 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 placement strategy. 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 $(i \times (k - r)) \mod k'$ $(0 \le i \le \beta - 1)$ (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 $((i - \beta + 1) \times r) \mod k'$ $(\beta \le i \le \frac{l}{k'} - 1)$ (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 $(0 \le i \le \beta - 1)$ 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).

Example. We show in Figure 4 the designed 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, 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 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. 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 updates is thus $(\frac{l}{k} - \lfloor \frac{k'}{k} \rfloor \times \frac{l}{k'}) \times (k - r)$.

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

Note that in theory, the optimal placements should be the ones that maximize the sum of the number of overlapping data blocks of all stripes. We pose it as a future work to discuss whether our placements are optimal or not.

Proof of fault tolerance. We now prove that our placement strategy preserves node-level fault tolerance:

(i) It is obvious that each of the α same-row stripes is spread over k nodes, thus following the fault tolerance requirement.

(ii) The first cross-row stripe has r data blocks distributed in the empty positions of the first row (with node ids of $\lfloor \frac{k'}{k} \rfloor \times k, \cdots, k' - 1$), and $k - r$ data blocks distributed in other rows (with node ids of $0, \cdots, k - r - 1$). Therefore, the k data blocks of the first cross-row stripe are sequentially distributed into k different nodes.

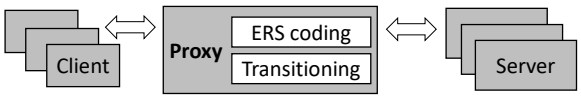


Fig. 5. Architecture of the ERS KV store prototype.

(iii) The second cross-row stripe has r data blocks distributed in the second row (with node ids of $(\lfloor \frac{k'}{k} \rfloor \times k + k - r) \bmod k', \cdots, k-r-1$), and $k-r$ data blocks distributed in different rows (with node ids of $k-r, \cdots, (2(k-r)-1) \bmod k'$). Thus, the node ids of the second cross-row stripe are right rotated by $k-r$ based on the first stripe, and so the second cross-row stripe is also spread across k different nodes.

(iv) By induction, we can conclude that 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.

IV. System Design and Implementation

We prototype a KV store that realizes ERS codes. We present the architecture of our prototype (Section IV-A). We show the metadata management of the prototype (Section IV-B). We next discuss how to maintain consistency during redundancy transitioning (Section IV-C). We finally show our implementation of the prototype atop Libmemcached (Section IV-D).

A. Architecture

Figure 5 shows the architecture of the ERS 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. The proxy implements multiple redundancy strategies (e.g., RS codes, ERS codes), and 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 backup. It is also noteworthy that the proxy-based design can be seen in other cloud storage systems (e.g., Bluesky [28], OpenStack [6]).

When first storing objects, users can specify the coding parameters k, m and k', and then the objects are stored using $\text{ERS}(k,m,k')$. If the reliability requirements change, users can call the transitioning functions to update the redundancy schemes from $\text{RS}(k,m)$ into $\text{RS}(k',m)$.

B. Metadata Management

For each object, the ERS 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. Also, the proxy avoids the I/O requests to the objects that are currently under transitioning, and ensures that the I/O requests to the transitioned objects are processed based on $\text{RS}(k',m)$.

C. 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 into the update process. 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 [9].

D. Implementation

We implement ERS codes atop Libmemcached 1.0.18 [4] that acts as the proxy, by adding about 3,800 SLoC. We also deploy multiple Memcached servers [5] for object storage. We leverage the Jerasure Library [23] to realize ERS codes. To show the improvements of ERS codes over SRS codes, we also implement SRS codes into Libmemcached.

V. Evaluation

We present evaluation results of the ERS KV store prototype. We show via numerical analysis and testbed experiments the performance gain in redundancy transitioning of ERS (i.e., ERS codes with the row-major order placement and the designed matrix) and ERS+ (i.e., ERS codes with the designed matrix and the designed placement) over SRS (i.e., SRS codes).

A. Numerical Analysis

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

SRS. For SRS, 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 and ERS+. For ERS and ERS+, we read the old parity blocks and the non-overlapping data blocks to generate the

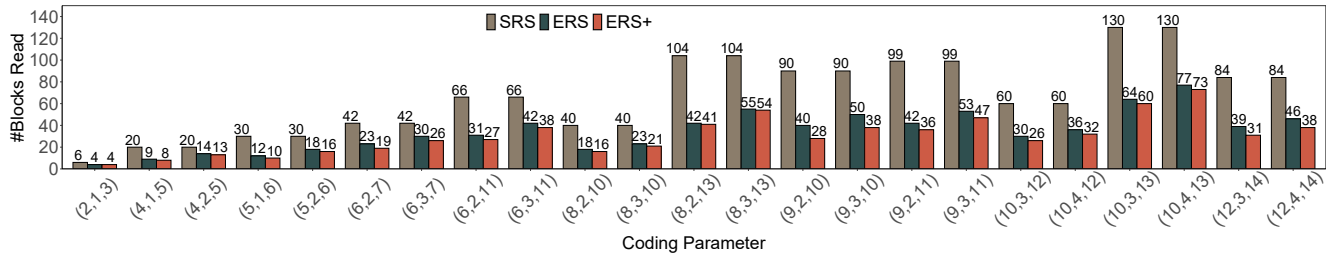


Fig. 6. Numerical results of the number of blocks read for parity block updates.

TABLE I
COMPARISONS OF SRS, ERS, AND ERS+ UNDER PARAMETRIC ANALYSIS.

	number improve	average ratio	best ratio	best parameter
ERS over SRS	221	46.2%	73.8%	(6, 1, 14)
ERS+ over ERS	123	14.8%	53.6%	(13, 1, 14)

new parity blocks. In particular, the number of blocks read of ERS+ is $m \times \frac{l}{k} + (\frac{l}{k} - \lfloor \frac{k'}{k} \rfloor \times \frac{l}{k'}) \times (k-r)$, where $r = k' \mod k$.

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 deployed in practical systems, for example $(k,m) = (6,3)$ (used by Google ColossusFS [3]), $(k,m) = (8,3)$ (used by Yahoo Object Store [7]), $(k,m) = (10,4)$ (used by Facebook HDFS [25]), and $(k,m) = (12,4)$ (used by Microsoft Azure [14]).

Figure 6 shows the results for 23 sets of coding parameters. We summarize the observations as follows.

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

Analysis of general parameters. We now consider more parameters and see how ERS (ERS+) behaves under general parameters. We set $3 \le k' \le 14, 2 \le k \le k'-1, 1 \le m \le 4$ and $m < k$, and there are a total of 244 sets of parameters.

Table I compares SRS, ERS and ERS+ in four aspects: i) number improve, the number of parameters where ERS (ERS+) outperforms SRS (ERS), ii) average ratio, the average reduction ratio of the number of blocks read of ERS (ERS+) over SRS (ERS), iii) best ratio, the maximum reduction ratio of the number of blocks read of ERS (ERS+) over SRS (ERS), and iv) best parameter, the parameters corresponding to the best ratio. There are 221 sets of parameters where ERS outperforms SRS, and ERS 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 only require the old parity blocks to generate the new parity blocks, so SRS equals ERS in the number of blocks read. There are 123 sets of parameters where ERS+ further outperforms ERS. ERS+ can save the I/Os of ERS by 14.8% on average, and up to 53.6% under $(k,m,k') = (13,1,14)$.

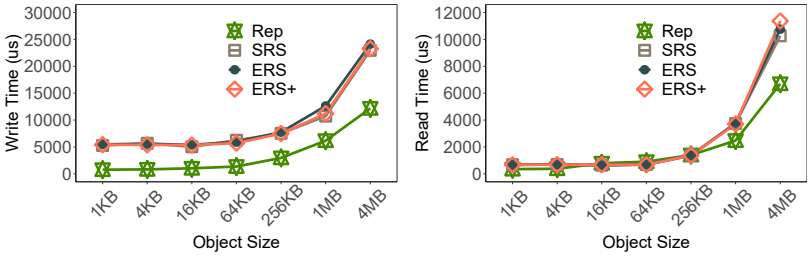
B. Testbed Experiments

Setup. We deploy the ERS KV store prototype on a local cluster which comprises 8 physical nodes, each of which runs Ubuntu 16.04.5 LTS with a quad-core 3.40 GHz Intel Core i5-3570, 16 GB RAM, and a Seagate ST1000DM003 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 from RS$(2,1)$ to RS$(3,1)$. 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 ERS KV store prototype and the vanilla Memcached (denoted by Rep). We consider $(k,m,k') = (2,1,3)$ for SRS/ERS/ERS+. We set the replication factor of Rep as two such that Rep can tolerate the same number of failures as SRS/ERS/ERS+. We evaluate the write time and read time under different object sizes. Figure 7 shows the results.

From Figure 7(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 7(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 generate $l + m \times \frac{l}{k}$ requests for writing, while Rep only requires two requests. Second, SRS and ERS connect to $k' + m$



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

Fig. 7. Exp#1: Normal write/read time.

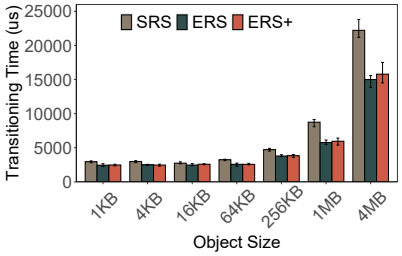
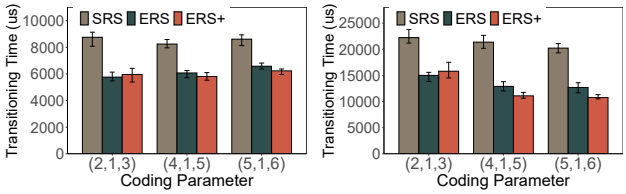


Fig. 8. Exp#2: Transitioning time with different object sizes.



(a) Object size 1 MB (b) Object size 4 MB

Fig. 9. Exp#3: Transitioning time under different coding parameters.

servers when sending the requests, while Rep only connects to two servers. Thus, SRS codes and ERS codes have more connection overhead.

From Figure 7(b), the read time also increases with a larger object size. The read latency of SRS/ERS/ERS+ is also higher than that of Rep (e.g., with 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 from RS$(2,1)$ to RS$(3,1)$. Figure 8 shows the results (the error bars show the maximum and minimum results across ten runs).

We can see that the transitioning time increases with a larger object size, and ERS and ERS+ 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 four servers while ERS connects to only two servers). For example, ERS reduces the transitioning time of SRS from 9.9% to 34.2%, while ERS+ reduces the transitioning time of SRS from 5.7% to 31.9%, across all object sizes. We can also see that the improvements of ERS and ERS+ over SRS become more prominent with larger object sizes, since the network now dominates the overall performance. ERS and ERS+ have similar transitioning time, which is consistent with the numerical results.

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

From Figure 9(a), ERS and ERS+ reduce the transitioning time of SRS by 34.2% and 31.9%, 26.3% and 29.5%, and 23.6% and 27.6%, for $(2,1,3)$, $(4,1,5)$, and $(5,1,6)$, respectively. From Figure 9(b), ERS and ERS+ reduce the transitioning time of SRS by 32.5% and 28.9%, 39.6% and 48.1%, and 37.2% and 46.6%, for $(2,1,3)$, $(4,1,5)$, and $(5,1,6)$, respectively. We can see that ERS greatly reduces the transitioning time of SRS due to the effect of the designed encoding matrix. ERS+ can further lower the transitioning time of ERS via designing data placement with more overlapping data blocks (e.g., $(k,m,k') = (4,1,5)$ and $(k,m,k') = (5,1,6)$).

TABLE II
EXP#4: TRANSITIONING TIME (μS) UNDER LIMITED BANDWIDTH.

Setting	SRS	ERS	ERS+
$(4,1,5)$, 1Gbps	64084	36532	34079
$(4,1,5)$, 10Gbps	21353	12896	11087
$(5,1,6)$, 1Gbps	67352	33916	29897
$(5,1,6)$, 10Gbps	20215	12685	10789

Experiment 4 (Transitioning latency under limited network bandwidth). We now evaluate the transitioning performance under limited bandwidth. We configure the bandwidth to 1 Gbps in our testbed. We consider $(k,m,k') = (4,1,5)$ and $(k,m,k') = (5,1,6)$ with object size of 4 MB. Table II shows the results.

We can see that ERS consistently outperforms SRS, while ERS+ further outperforms ERS. For example, under 1 Gbps network, ERS and ERS+ reduce the transitioning time of SRS by 43.0% and 46.8% for parameter $(k,m,k') = (4,1,5)$, and ERS and ERS+ reduce the transitioning time of SRS by 49.6% and 55.6% for parameter $(k,m,k') = (5,1,6)$. Also, the improvements of ERS and ERS+ over SRS are greater with more limited bandwidth. For example, ERS+ reduces the transitioning time of SRS by 46.6% under 10 Gbps network, and 55.6% under 1 Gbps network, for $(k,m,k') = (5,1,6)$.

VI. RELATED WORK

Redundancy transitioning. Several studies address efficient redundancy transitioning for different storage architectures. AutoRAID [30], DiskReduce [12], and EAR [19] study the transitioning from replication to RAID or erasure coding. Some studies propose efficient data redistribution approaches for RAID [31], [37], [39] and erasure-coded distributed storage systems [15], [32]–[34], [38]. 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 [1], [20].

Erasure coding in KV stores. Erasure coding has been extensively studied in modern KV stores for low-cost fault

tolerance [2], [18], data availability [9], [36], and tail latency mitigation [24]. In particular, prior studies [10], [27], [35], [40] address the elasticity of erasure-coded in-memory KV stores. PaRS [40] 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 [35] realizes the transitioning from replication to erasure coding in in-memory stores. ECHash [10] 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 [27], 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 construction and data placement.

VII. Conclusion

We study how to enable I/O-efficient redundancy transitioning in erasure-coded KV stores. We propose a new class of erasure codes, ERS codes, to mitigate I/O costs for redundancy transitioning. ERS codes eliminate data block relocation, and reduce I/Os for parity block updates via the co-design of encoding matrix construction and placement strategy. We implement a KV store that realizes ERS codes atop Memcached to allow redundancy transitioning. Both numerical studies and testbed experiments validate the efficiency of ERS codes in redundancy transitioning.

Acknowledgements

The work was supported in part by Research Grants Council of Hong Kong (GRF 14216316) and CCF-Tencent Open Fund WeBank Special Fund. The corresponding author is Zhirong Shen.

References

[1] Amazon ElastiCache. https://docs.aws.amazon.com/elasticache.

[2] Erasure code - Ceph documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[3] Google Colossus File System. https://cloud.google.com/files/storage_architecture_and_challenges.pdf.

[4] Libmemcached. https://libmemcached.org/libMemcached.html.

[5] Memcached. https://memcached.org.

[6] Openstack. https://openstack.org.

[7] Yahoo Cloud Object Store. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at.

[8] 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.

[9] 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.

[10] L. Cheng, Y. Hu, and P. P. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SoCC*, 2019.

[11] 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.

[12] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Proc. of ACM PDSW*, 2009.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] J. Plank and C. Huang. Tutorial: Erasure coding for storage applications. In *Slides presented at USENIX FAST*, 2013.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, 2018.

[28] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proc. of USENIX FAST*, 2012.

[29] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in HDFS. In *Proc. of USENIX FAST*, 2015.

[35] 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.

[36] 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.

[37] G. Zhang, W. Zheng, and K. Li. Rethinking RAID-5 data layout for better scalability. *IEEE Trans. on Computers*, 63(11):2816–2828, 2014.

[38] 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.

[39] W. Zheng and G. Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proc. of USENIX FAST*, 2011.

[40] 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.