Optimal Data Placement for Stripe Merging in Locally Repairable Codes

Si Wu[†], Qingpeng Du[†], Patrick P. C. Lee[‡], Yongkun Li[†], Yinlong Xu[†]
[†]University of Science and Technology of China [‡]The Chinese University of Hong Kong

***Abstract*—Erasure coding is a storage-efficient redundancy scheme for modern clustered storage systems by storing stripes of data and parity blocks across the nodes of multiple clusters; in particular, locally repairable codes (LRC) continue to be one popular family of practical erasure codes that achieve high repair efficiency. To efficiently adapt to the dynamic requirements of access efficiency and reliability, storage systems often perform redundancy transitioning by tuning erasure coding parameters. In this paper, we apply a stripe merging approach for redundancy transitioning of LRC in clustered storage systems, by merging multiple LRC stripes to form a large LRC stripe with low storage redundancy. We show that the random placement of multiple LRC stripes that are being merged can lead to high cross-cluster transitioning bandwidth. To this end, we design an optimal data placement scheme that provably minimizes the cross-cluster traffic for stripe merging, by judiciously placing the blocks to be merged in the same cluster while maintaining the repair efficiency of LRC. We prototype and implement our optimal data placement scheme on a local cluster. Our evaluation shows that it significantly reduces the transitioning time by up to 43.2% compared to the baseline.**

I. INTRODUCTION

Due to the exponential growth of data volume [3], modern storage systems continue to grow in size and scale [14], [24], [26]. The large scale of storage systems leads to prevalent failures [9]. To reliably store the tremendous amount of data in a storage-efficient way, modern storage systems increasingly adopt *erasure coding* to provide data redundancy [1], [2], [9], [14], [24], [29], [43]. Compared to replication, erasure coding greatly saves the amount of redundancy to achieve the same degree of data reliability [36]. At a high level, erasure coding encodes several original uncoded *data blocks* to produce additional redundant coded blocks, called *parity blocks*, such that all original data blocks can be successfully reconstructed by a sufficient number of data and parity blocks.

Among numerous erasure coding constructions, *locally repairable codes (LRC)* [14], [18] are one popular family of repair-friendly erasure codes. An LRC can be configured by three parameters (k,l,g). It partitions k data blocks into l small-size groups, and generates a *local parity block* for each group, such that the repair can be performed within each small-size group. It further generates g *global parity blocks* for high fault tolerance. The $k+l+g$ data and parity blocks that are encoded together are called a *stripe*. Large-scale storage systems usually store a number of stripes, which are encoded independently. Due to its repair efficiency and high reliability, LRC is widely deployed in enterprise data centers [14], [29].

In response to the varying access characteristics [12], [43], [44] and dynamic reliability requirements [16], [34], modern storage systems often perform frequent *redundancy transitioning* operations on erasure-coded data. By redundancy transitioning, we mean the change of coding parameters k, l, and g. We motivate that redundancy transitioning is crucial in the following scenarios.

- **Adaptation to skewed and dynamic workloads.** Workloads in real storage systems are very skewed and dynamic [43]. The skewness means that a small portion of hot data is frequently accessed and the majority of cold data is rarely accessed, while the dynamic property means that the data hotness is time-varying. To achieve both high access performance and high storage efficiency, practical storage systems can store hot data with high-redundancy erasure coding for high performance, while storing cold data with low-redundancy erasure coding for low-cost persistence. When data hotness varies, we perform redundancy transitioning to update coding parameters [41], [43].
- **Adaptation to varying reliability.** Data in different lifetime period requires different degrees of reliability [34], so storage systems need to dynamically tune coding parameters to meet different reliability demands. Also, disk reliability varies as the disks age [17]. To provide an efficient trade-off between storage overhead and fault tolerance, large-scale data centers should elastically adjust coding parameters [16], [17].
- **Generation of wide stripes.** Recent work explores wide stripes for erasure-coded storage to suppress the fraction of parity blocks in a stripe to achieve extreme storage savings [12]. To efficiently support wide stripes, storage systems can deploy narrow stripes with modestly low redundancy for newly-written data, and later transition the aged data into wide stripes with extremely low redundancy [44].

In this work, we explore a *stripe merging* approach [43], [44] for redundancy transitioning of LRC in *clustered* storage systems, which organize storage nodes hierarchically into multiple clusters, such that the cross-cluster network bandwidth is much more scarce than the inner-cluster bandwidth [5], [7], [35]. By stripe merging, we refer to the transitioning from multiple small-size LRC stripes to a larger-size LRC stripe. We show that stripe merging is critical for LRC as it can improve the storage efficiency or fault tolerance ability of the system without compromising other performance (Section II-B).

Efficient stripe merging for LRC is a challenging issue, as its performance heavily depends on how the stripes are

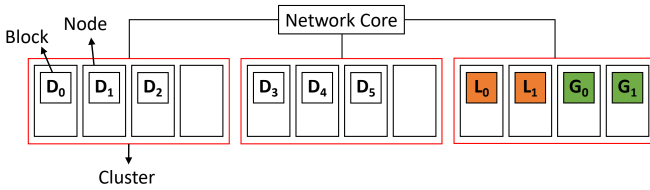


Figure 1. Clustered storage architecture under LRC$(6,2,2)$ with $c=3$ clusters; a red rectangle indicates a cluster.

distributed across the clusters. As we show, the uncoordinated *random* data placement of small-size LRC stripes can amplify the cross-cluster transitioning traffic (Section III). To this end, we design an *optimal* data placement scheme that coordinates the placement of multiple small-size LRC stripes in different clusters, so as to minimize the cross-cluster traffic for stripe merging in LRC. Our contributions include:

- We formally analyze the impact of the uncoordinated random data placement of multiple LRC stripes on the cross-cluster transitioning traffic (Section IV-A). We design an optimal data placement scheme that judiciously controls the number of blocks that need to be merged in the same cluster, while maintaining the repair efficiency of LRC. We formally prove that our optimal data placement scheme minimizes the cross-cluster transitioning traffic (Section IV-B).
- We implement two extreme points of our data placement scheme: the *dispersed* placement that completely distributes the blocks to be merged over distinct clusters and the *aggregated* placement that places the blocks to be merged in the same cluster. Experiments on a local cluster show that the dispersed placement reduces the transitioning time of the baseline by up to 43.2% for one setting, while the aggregated placement reduces the transitioning time of the baseline by up to 35.3% for another setting (Section V).

II. Background and Problem

A. Clustered Storage under LRC

Clustered storage architecture. Modern clustered storage systems typically have a two-layer hierarchical architecture [7], [12], under which a system organizes storage nodes (or *nodes* in short) into c *clusters* (where $c>1$) and connects the c clusters via the network core. Figure 1 shows a clustered storage system with $c=3$ clusters. In clustered storage, the inner-cluster network bandwidth is sufficient, while the cross-cluster bandwidth is scarce [5], [7], [35]. Thus, we assume that the cross-cluster transfer is the performance bottleneck.

Locally repairable codes (LRC). We denote an LRC construction by LRC(k,l,g), which encodes k *data blocks* (denoted by $D_0,D_1,\cdots,D_{k-1}$) into l *local parity blocks* (denoted by $L_0,L_1,\cdots,L_{l-1}$) and g *global parity blocks* (denoted by $G_0,G_1,\cdots,G_{g-1}$), such that the set of $k+l+g$ data/parity blocks (collectively called a *stripe*) are distributed over $k+l+g$ nodes. Practical storage systems typically comprise multiple stripes that are encoded independently.

An LRC can be constructed in a variety of ways [14], [18], [29]. In this work, we consider Azure's LRC construction called the Local Reconstruction Code [14]. Azure's LRC assumes that k is divisible by l. It divides k data blocks evenly into l equal-size groups, and computes a bitwise-XOR of each group of $\frac{k}{l}$ data blocks to form a local parity block. It further computes g global parity blocks based on all k data blocks as in Reed-Solomon codes [28]. Let $b=\frac{k}{l}$. We call the collection of b data blocks and their corresponding local parity block a *local group*. For example, Figure 1 shows an LRC stripe with $(k,l,g)=(6,2,2)$, where D_0-D_2 and L_0 form one local group, while D_3-D_5 and L_1 form another local group.

LRC is proposed to mitigate the bandwidth and I/O for a single-data-block repair by limiting the repair within a local group [14], [18], [41]. For example, in Figure 1, repairing D_0 only needs to access the remaining blocks in the same local group (i.e., D_1, D_2, and L_0). We define the *repair cost* of LRC as the average amount (in units of blocks) of cross-cluster network transfer to repair all data blocks. For example, in Figure 1, repairing any data block always incurs a cross-cluster transfer of one block, so the repair cost is one block.

Cluster-aware fault tolerance of LRC. By storing each stripe of $k+l+g$ data/parity blocks across $k+l+g$ nodes, the storage system provides multi-node fault tolerance. We further store multiple blocks of an LRC stripe in a single cluster, such that the system provides single-cluster fault tolerance [41]. Note that cluster failures are more rare events than node failures in real systems [24]. Under single-cluster fault tolerance, we can place no more than $g+i$ blocks (of an LRC stripe) that span i local groups into a single cluster, where $1 \le i \le l$, as shown in [41]. The main reason is that the number of parity blocks that can be used to decode these blocks is $g+i$. For example, in Figure 1, we can place D_0-D_2 in one cluster and D_3-D_5 in another cluster, subject to single-cluster fault tolerance.

B. Stripe Merging for Redundancy Transitioning of LRC

In this paper, we exploit a *stripe merging* approach for redundancy transitioning of LRC. Specifically, we merge x ($x \ge 2$) small-size LRC stripes into a larger-size stripe with all the data blocks in the small-size stripes being maintained. Prior studies [43], [44] have also studied stripe merging for Product Codes [11], [19] and Reed-Solomon Codes [28], but they address limited settings (see Section VI for details).

Problem definition. We consider merging x LRC(k,l,g) stripes to an LRC(xk,l',g') stripe for some special cases of l' and g'. Note that the parameters of LRC exhibit a certain trade-off among *repair locality* (denoted by r), *storage overhead* (denoted by s), and *minimum hamming distance* (denoted by d) [10], [18]. Specifically, we have: (i) $r=b=\frac{k}{l}$, which indicates the bandwidth cost for single-data block repair; (ii) $s=\frac{k+l+g}{k}$, which specifies the redundancy overhead; and (iii) $d=g+2$, which indicates the fault tolerance ability of LRC. Our idea is to fix two trade-off dimensions of LRC, while varying the third dimension. To this end, we consider two specific stripe merging problems:

- **Problem (1)**: fixing r and d, i.e., merging x LRC(k,l,g) into LRC(xk,xl,g). By doing so, we decrease the storage overhead s from $\frac{k+l+g}{k}$ to $\frac{k+l}{k}+\frac{g}{xk}$.

- **Problem (2)**: fixing r and s, i.e., merging x LRC(k,l,g) into LRC(xk,xl,xg). By doing so, we increase the minimum hamming distance d from $g+2$ to $xg+2$.

We do not consider stripe merging that fixes s and d, i.e., merging x LRC(k,l,g) into LRC$(xk,xl+(x-1)g,g)$. The reason is that xk may not be divisible by $xl+(x-1)g$ and there may not exist an LRC with parameters $(xk,xl+(x-1)g,g)$.

Significance. The redundancy transitioning in Problem (1) can improve the overall storage efficiency with the repair performance and fault tolerance ability being maintained, so it is useful for wide stripe generation [12], [44]. The redundancy transitioning in Problem (2) can improve the fault tolerance ability without changing the repair performance and storage efficiency, so it can efficiently adapt to the varying reliability demands [16], [17], [34].

Goals. Our primary goal is to minimize the cross-cluster network traffic for stripe merging, while preserving both the minimized repair cost for LRC(k,l,g) under single-cluster fault tolerance and the minimized repair cost for LRC(xk,l',g') under single-cluster fault tolerance. This maintains access efficiency before and after stripe merging.

III. Challenges and Motivations

A. Bandwidth-Intensive Operations in Stripe Merging

Stripe merging inevitably incurs data transfers when merging multiple small-size stripes into a large-size stripe. In the following, we identify two bandwidth-intensive operations in stripe merging.

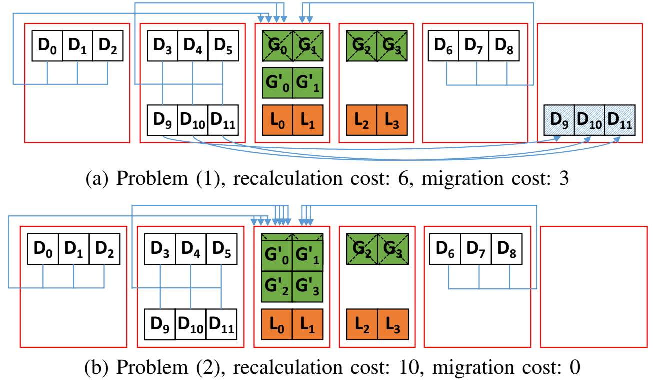
First, to merge x LRC(k,l,g) stripes into an LRC(xk,l',g') stripe, the number of data blocks of a stripe increases from k to xk. Thus, we need to recalculate the g' new global parity blocks from all xk data blocks; we call this operation *recalculation*. Note that we do not need to recalculate the local parity blocks, since each local parity block is still calculated from the same $b=\frac{k}{l}=\frac{xk}{l'}=\frac{xk}{xl}$ data blocks before and after stripe merging.

Second, while we guarantee single-cluster fault tolerance for each LRC(k,l,g) stripe, the data and local parity blocks of different LRC(k,l,g) stripes may aggregate in the same cluster, causing the violation of single-cluster fault tolerance for the resulting LRC(xk,l',g') stripe. Thus, we need to migrate some data and local parity blocks of some LRC(k,l,g) stripes to maintain fault tolerance for the resulting LRC(xk,l',g') stripe; we call this operation *migration*.

Both recalculation and migration can incur substantial cross-cluster traffic. Accordingly, we define the *recalculation cost* and *migration cost* as the amounts of cross-cluster traffic (in units of blocks) for recalculation and migration, respectively.

B. Challenges

We argue that simple data placement of LRC stripes can lead to substantial recalculation and migration costs. Here, we study *random data placement*, in which we place the blocks of each LRC(k,l,g) stripe into distinct nodes residing in a number of clusters (both the nodes and clusters are randomly selected), such that the single-cluster fault tolerance and the minimum repair cost are achieved. To minimize the repair cost, we determine the number of clusters by the prior work [41] (we will formally analyze the choice of the number of clusters in Section IV-B). Note that prior studies on stripe merging [43], [44] also consider random data placement to distribute the blocks to distinct nodes (while they are not cluster-aware).



(a) Problem (1), recalculation cost: 6, migration cost: 3

(b) Problem (2), recalculation cost: 10, migration cost: 0

Figure 2. Random data placement: (a) Problem (1): $x=2$ LRC$(6,2,2)$ to LRC$(12,4,2)$; and (b) Problem (2): $x=2$ LRC$(6,2,2)$ to LRC$(12,4,4)$.

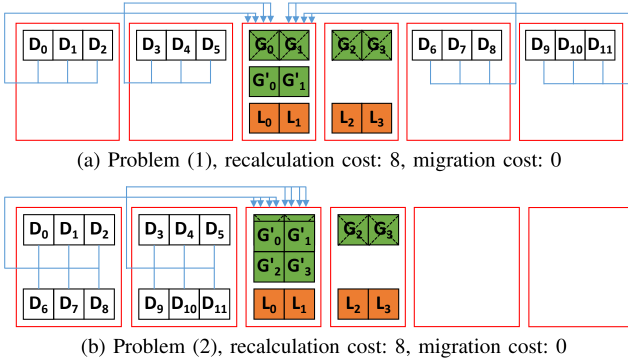
To illustrate, we use an example that applies stripe merging to $x=2$ LRC$(6,2,2)$ stripes over a system with $c=6$ clusters. For each LRC$(6,2,2)$ stripe, we place the blocks into $c=3$ randomly selected clusters to achieve both single-cluster fault tolerance and minimum repair cost. Specifically, the data blocks of two local groups are placed in two separate clusters, while all local and global parity blocks are placed in another cluster, as shown in Figure 2.

Challenge in Problem (1). Figure 2(a) shows the random data placement for Problem (1), in which we merge $x=2$ LRC$(6,2,2)$ stripes into an LRC$(12,4,2)$ stripe. For recalculation , we apply *encoding-and-transferring* [13], [21], [41] to the three clusters that store data blocks. Specifically, each cluster generates and transfers two partial blocks corresponding to the two new global parity blocks from its resident data blocks. Thus, the recalculation cost is six.

However, D_3-D_5 of the first LRC$(6,2,2)$ stripe and D_9-D_{11} of the second LRC$(6,2,2)$ stripe happen to be stored in the same cluster. After stripe merging into an LRC$(12,4,4)$ stripe, single-cluster fault tolerance is violated. The reason is that D_3-D_5 and D_9-D_{11} span two local groups, while the number of data blocks (i.e., 6) is larger than $g'+2=4$ (Section II-A). Thus, we need to migrate D_9-D_{11} to another cluster, and the migration cost is three.

Problem (1) shows that if the blocks to be merged are aggregated in a limited number of clusters, the migration traffic can be significant.

Challenge in Problem (2). Figure 2(b) shows the same random data placement for Problem (2), in which we merge $x=2$ LRC$(6,2,2)$ stripes into an LRC$(12,4,4)$ stripe. For recalculation, the second cluster encodes and transfers four partial blocks, while for each of the first and fifth clusters, we adopt encoding-and-transferring and transfer four partial blocks. To reduce the traffic, we directly transfer the three data



(a) Problem (1), recalculation cost: 8, migration cost: 0

(b) Problem (2), recalculation cost: 8, migration cost: 0

Figure 3. Optimal data placement: (a) Problem (1): 2 LRC$(6,2,2)$ to LRC$(12,4,2)$; and (b) Problem (2): 2 LRC$(6,2,2)$ to LRC$(12,4,4)$.

blocks in each of the first and fifth clusters to the cluster that recalculates the new global parity blocks. Thus, the dispersion of D_0-D_2 and D_6-D_8 needs to transfer across-cluster six blocks. In total, the recalculation cost is 10. For LRC$(12,4,4)$ after merging, we can guarantee single-cluster fault tolerance with no block migration, so the migration cost is zero.

Problem (2) shows that if the blocks to be merged are dispersed across different clusters, the recalculation traffic can be significant.

C. Motivations

Our idea is that by coordinating the data placement of x LRC(k,l,g) stripes, we can minimize the transitioning bandwidth incurred in stripe merging.

Our motivation for Problem (1). Figure 3(a) shows another data placement for Problem (1), in which the blocks of the two stripes are dispersed across different clusters. For recalculation, each of the four clusters that store data blocks encodes and transfers two partial blocks to the cluster that recalculates the new global parity blocks, so the recalculation cost is eight. After merging, the block distribution of LRC$(12,4,2)$ still satisfies single-cluster fault tolerance without any block migration, so the migration cost is zero. Note that the data placement of LRC$(12,4,2)$ still maintains the minimum repair cost. Figure 3(a) shows that we can disperse the blocks to be merged over distinct clusters to eliminate the migration cost and hence save the overall transitioning bandwidth.

Our motivation for Problem (2). Figure 3(b) shows another data placement for Problem (2), in which the data blocks of the two stripes are aggregated in two clusters. For recalculation, each of the two clusters that store data blocks encodes and transfers four partial blocks, so the recalculation cost is eight. The migration cost remains zero. Also, the block distribution of LRC$(12,4,4)$ guarantees the minimum repair cost under single-cluster fault tolerance. Figure 3(b) shows that by aggregating the blocks to be merged in the same cluster, we can mitigate the recalculation cost and hence the overall transitioning bandwidth.

IV. Analysis and Design

We first analyze how the random data placement of the x LRC(k,l,g) stripes affects the recalculation and migration costs. We then design a data placement scheme that is guaranteed to achieve the minimum bandwidth costs for the transitionings in both Problem (1), i.e., x LRC(k,l,g) to LRC(xk,xl,g), and Problem (2), i.e., x LRC(k,l,g) to LRC(xk,xl,xg).

A. Impact of Random Data Placement

Main findings. Propositions 1-3 show that if the blocks to be merged are aggregated into a limited number of clusters, then the migration cost will be significantly increased in Problem (1) (e.g., Figure 2(a)). Propositions 4-5 show that if the blocks to be merged are dispersed across a large number of clusters, then the recalculation cost will be enlarged in Problem (2) (e.g., Figure 2(b)).

Proposition 1. *If $g+i$ data and local parity blocks of one LRC(k,l,g) stripe that span i $(1 \le i \le l)$ local groups, and $g+j$ data and local parity blocks of a second stripe that span j $(1 \le j \le l)$ local groups, are aggregated in one cluster, then single-cluster fault tolerance is violated for LRC(xk,xl,g) without block migration.*

Proof. For LRC(xk,xl,g), the blocks span $i+j$ local groups, but the number of blocks (i.e., $2g+i+j$) is larger than $g'+i+j=g+i+j$. Thus, single-cluster fault tolerance is violated without any block migration [41]. □

Proposition 2. *To restore fault tolerance, as well as guarantee the minimum repair cost after merging, we need to migrate all $g+j$ data and local parity blocks of the second stripe.*

Proof. By Proposition 1, to restore fault tolerance, we need to perform block migration. We try to migrate blocks of the second stripe. From [41], to guarantee the minimum repair cost for LRC(xk,xl,g), we need to guarantee that the $g+j$ data and local parity blocks of the second stripe still reside in the same cluster. Hence, we need to migrate all $g+j$ data and local parity blocks of the second stripe. □

Proposition 3. *By pre-distributing the $g+j$ data and local parity blocks of the second stripe in a different cluster, we can save the total transitioning bandwidth.*

Proof. If we pre-distribute the $g+j$ data and local parity blocks of the second stripe in a different cluster, then this part of block migration is eliminated (i.e., the migration cost is reduced by $g+j$). However, for recalculation, we need to generate $g'=g$ partial blocks from the pre-distributed data blocks, and transfer these g partial blocks. Thus, the recalculation cost is increased by g. That is, we trade increased recalculation cost for decreased migration cost. Overall, the total bandwidth cost is reduced by j $(j \ge 1)$. □

Proposition 3 implies that if the blocks to be merged are aggregated into a small number of clusters, then the total bandwidth cost is amplified in Problem (1).

Proposition 4. *Even if the blocks from x small-size stripes (including $g+i_n$ data and local parity blocks from the n-th stripe that span i_n $(1 \le i_n \le l)$ local groups, where $0 \le n \le x-$*

1*) are aggregated in one cluster, single-cluster fault tolerance is guaranteed for LRC*(xk,xl,xg) *without block migration.*

Proof. For LRC(xk,xl,xg), the blocks span $\sum_{n=0}^{x-1} i_n$ local groups, and the number of blocks (i.e., $xg+\sum_{n=0}^{x-1} i_n$) equals $g'+\sum_{n=0}^{x-1} i_n$. Thus, single-cluster fault tolerance is guaranteed without any block migration [41]. □

Despite that the blocks to be merged can be aggregated into a single cluster, the random data placement inevitably causes dispersion of the blocks. For example, in Figure 2(b), D_0-D_2 of the first LRC$(6,2,2)$ stripe and D_6-D_8 of the second stripe are dispersed over two clusters.

Proposition 5. *By pre-aggregating the blocks from x small-size stripes (including* $g+i_n$ *data and local parity blocks from the n-th stripe, where* $0 \le n \le x-1$*) into a single cluster, we can save the total transitioning bandwidth.*

Proof. (i) As these blocks span $\sum_{n=0}^{x-1} i_n$ local groups, there are at most $\sum_{n=0}^{x-1} i_n$ local parity blocks (or equivalently, at least xg data blocks)

(ii) For a cluster that stores data blocks, if the number of data blocks is no less than $g'=xg$, then this cluster transfers xg partial blocks for recalculation; otherwise, this cluster directly transfers all its data blocks for recalculation.

(iii) Suppose that these blocks are dispersed across two or more clusters. If there exists one cluster that stores no less than xg data blocks, then the recalculation cost caused by this cluster alone is xg (from (ii)); otherwise, we directly transfer all original data blocks of all clusters for recalculation (from (ii)). From (i), the number of data blocks is at least xg, so the recalculation cost is at least xg.

(iv) Suppose that we pre-aggregate these blocks into a single cluster. From (i), the number of data blocks is no less than xg. Then from (ii), the recalculation cost is exactly xg. Compared to (iii), we save the overall transitioning bandwidth. □

Proposition 5 implies that if the blocks to be merged are dispersed over different clusters, the total bandwidth cost is enlarged in Problem (2).

B. Design

Overall idea. Section IV-A shows that we can reduce the transitioning bandwidth costs for Problem (1) and Problem (2), by controlling the number of blocks to be merged in the same cluster. Based on this insight, we design an optimal data placement scheme for the x LRC(k,l,g) small-size stripes. We derive two extreme points of our data placement scheme: the dispersed data placement (called *DIS*) that disperses the blocks to be merged over distinct clusters, and the aggregated data placement (called *AGG*) that aggregates the blocks to be merged in the same cluster. We prove that DIS achieves the minimum transitioning cost (i.e., the sum of the migration and recalculation costs) for Problem (1), while AGG achieves the minimum transitioning cost for Problem (2).

Preliminaries. We first derive the placement subject to both minimum repair cost and single-cluster fault tolerance. To achieve the minimum repair cost, we should place a local group (i.e., $b=\frac{k}{l}$ data blocks and one local parity block) into the minimum number of clusters. Under single-cluster fault tolerance, we can place at most $g+1$ data blocks of a local group into one cluster. Thus, we place every $g+1$ data blocks of each local group into one separate cluster. By doing so, we minimize the number of clusters each local group spans to $\lfloor \frac{b}{g+1} \rfloor + 1$, and the repair cost to $\lfloor \frac{b}{g+1} \rfloor$.

After this, there remain $(m = b \mod (g+1)) \le g$ data blocks in each local group. We call the remaining m data blocks and the corresponding local parity block a *remaining local group*. Let θ be the maximum number of remaining local groups that can be collocated together. By the fault tolerance constraint, the number of blocks (i.e., $\theta \times m + \theta$) that span θ local groups, cannot exceed $g+\theta$, meaning that $\theta = \lfloor \frac{g}{m} \rfloor$. Thus, we collocate every θ remaining local groups into one separate cluster to minimize the number of clusters a single LRC(k,l,g) stripe spans. We place all global parity blocks in a different cluster.

To facilitate our analysis, we assume that g is divisible by m, such that every θ remaining local groups contain g data blocks and θ local parity blocks.

If $(m = b \mod (g+1)) = 0$, then each remaining local group contains only one local parity block. We then simply collocate all local and global parity blocks into a separate cluster.

For example in Figure 2, for a single LRC$(6,2,2)$ stripe, we put every three data blocks into a separate cluster, and all local and global parity blocks into a third cluster.

Let α be the number of clusters a single LRC(k,l,g) stripe spans subject to both minimum repair cost and single-cluster fault tolerance. According to the above analysis, α can be readily calculated as follows (assuming that $l \mod \theta = 0$).

$$\alpha = \begin{cases} l \times \lfloor \frac{b}{g+1} \rfloor + \frac{l}{\theta} + 1, \text{ if } (m = b \mod (g+1)) \neq 0; \\ l \times \lfloor \frac{b}{g+1} \rfloor + 1, \text{ if } (m = b \mod (g+1)) = 0. \end{cases} \tag{1}$$

For example in Figure 2, we can easily compute that $\alpha = 3$.

We define *aggregation degree* of the x LRC(k,l,g) stripes (denoted by β) as the number of clusters that aggregate the data blocks from x LRC(k,l,g) stripes. Since we can aggregate the data blocks from x stripes in at most $\alpha-1$ clusters (recall that the last one cluster stores the global parity blocks), β ranges from 0 to $\alpha-1$.

Data placement design. Algorithm 1 takes several essential parameters as input (e.g., x, (k,l,g), β), and generates a data placement for the x LRC(k,l,g) stripes as output.

First, we initialize x block sets $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$ as $\emptyset$, and a cluster ID (denoted as d) as -1 (Lines 1-2).

For the i-th ($0 \le i \le l-1$) local groups of the x stripes, we try to aggregate or disperse every $x(g+1)$ data blocks from the x stripes. Thus, for $x(g+1)$ data blocks from the x stripes, we allocate a new cluster randomly from the c clusters (Lines 6-7). Then, according to the aggregation degree (i.e., β), we aggregate the $x(g+1)$ data blocks into the allocated cluster, or disperse these blocks across x separate clusters (Lines 8-9). We also derive the i-th remaining local groups for the x stripes (Line 11).

Algorithm 1 Multi-stripe data placement scheme

Input: x, (k,l,g), c (#clusters), β (aggregation degree)
Output: Data placement for multiple LRC(k,l,g) stripes
1: Initialize x block sets $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$ as $\emptyset$
2: Initialize cluster ID d as -1
3: **for** the i-th $(0 \le i \le l-1)$ local groups of x stripes **do**
4: // Place every $x(g+1)$ data blocks
5: **for** every $x(g+1)$ data blocks from x stripes **do**
6: Randomly select a new cluster
7: Cluster ID $d \leftarrow d+1$
8: $\mathcal{B}_n \leftarrow g+1$ data blocks from the n-th stripe, where $0 \le n \le x-1$
9: AGGORDIS(d, β, $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$, the cluster with ID d)
10: **end for**
11: Derive the i-th remaining local groups for the x stripes
12: **end for**
13: // Place every $x\theta$ remaining local groups
14: **for** every $x\theta$ remaining local groups from x stripes **do**
15: Randomly select a new cluster
16: Cluster ID $d \leftarrow d+1$
17: $\mathcal{B}_n \leftarrow \theta$ remaining local groups from the n-th stripe, where $0 \le n \le x-1$
18: AGGORDIS(d, β, $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$, the cluster with ID d)
19: **end for**
20: // Place global parity blocks of x stripes
21: Randomly select x new clusters
22: Distribute the global parity blocks of x stripes to the x clusters
23: **procedure** AGGORDIS(d, β, $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$, the cluster with ID d)
24: **if** $d < \beta$ **then**
25: Aggregate $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$ in the cluster with ID d
26: **else**
27: Disperse $\mathcal{B}_0, \mathcal{B}_1, \cdots, \mathcal{B}_{x-1}$ across x different clusters
28: **end if**
29: **end procedure**

We now try to aggregate or disperse every $x\theta$ remaining local groups from the x stripes. For $x\theta$ remaining local groups from the x stripes, we randomly choose a new cluster from the c clusters (Lines 15-16). Next, we disperse the $x\theta$ remaining local groups from the x stripes over x different clusters, or collocate these blocks into the chosen cluster, by comparing the current cluster ID with the aggregation degree (Lines 17-18).

The global parity blocks of the x stripes are dispersed over x dedicated clusters as they do not affect the recalculation and migration costs (Lines 21-22). Procedure aggOrDis is to completely aggregate or disperse the x block sets according to the aggregation degree (Lines 23-29).

If $(m = b \mod (g+1)) = 0$, then we simply place all local and global parity blocks of each stripe into a separate cluster.

Examples. By inputting $\beta = 0$, we get the DIS placement. For example, in Figure 3(a), we distribute the first LRC$(6,2,2)$ stripe and the second stripe to the first three clusters and the next three clusters, respectively.

Alternatively, by inputting $\beta = \alpha - 1$, we get the AGG placement. For example, in Figure 3(b), we collocate D_0-D_2 of the first stripe and D_6-D_8 of the second stripe in the first cluster, and D_3-D_5 of the first stripe and D_9-D_{11} of the second stripe in the second cluster. The parity blocks of the two stripes are placed onto the third and fourth clusters, respectively.

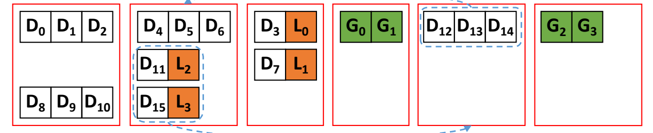


Figure 4. Converting random placement to our data placement for 2 LRC$(8,2,2)$ with less cost. The random placement has bandwidth 14 and 13, while our data placement has bandwidth 14 and 12 for Problems (1) and (2).

Cost analysis for Problem (1). For a data placement with aggregation degree β, there are β clusters that aggregate the data blocks from x LRC(k,l,g) stripes. According to Proposition 2, we need to migrate the data and local parity blocks of $x-1$ stripes for each of the β clusters. Recall that a single stripe spans α clusters, where each of the first $l \times \lfloor \frac{b}{g+1} \rfloor$ ones stores $g+1$ data blocks, and each of the next $\frac{l}{\theta}$ ones stores $g+\theta$ data and local parity blocks (Equation (1)). Thus, if $\beta <= l \times \lfloor \frac{b}{g+1} \rfloor$, then we migrate $(x-1) \times (g+1)$ data blocks for each of the β clusters; otherwise, we migrate $(x-1) \times (g+1)$ data blocks for each of the $l \times \lfloor \frac{b}{g+1} \rfloor$ clusters, and $(x-1) \times (g+\theta)$ data and local parity blocks for each of the $\beta - l \times \lfloor \frac{b}{g+1} \rfloor$ clusters.

$$\text{migration cost} = \begin{cases} \beta \times (x-1) \times (g+1), \text{ if } \beta <= l \times \lfloor \frac{b}{g+1} \rfloor; \\ (\beta - l \times \lfloor \frac{b}{g+1} \rfloor) \times (x-1) \times (g+\theta) + \\ l \times \lfloor \frac{b}{g+1} \rfloor \times (x-1) \times (g+1), \text{ otherwise.} \end{cases} \quad (2)$$

For recalculation, as the data blocks of the x stripes span $\beta + (\alpha - 1 - \beta) \times x$ clusters and the number of data blocks in each cluster is no less than $g' = g$ (each of the first β clusters has at least xg data blocks, while each of the next $(\alpha - 1 - \beta) \times x$ clusters has at least g data blocks), we encode and transfer g partial blocks from each cluster.

$$\text{recalculation cost} = (\beta + (\alpha - 1 - \beta) \times x) \times g. \quad (3)$$

Trade-off between recalculation cost and migration cost in Problem (1). From Equations (2) and (3), a large β (e.g., AGG) results in small recalculation cost but large migration cost, while a small β (e.g., DIS) brings small migration cost but large recalculation cost. There exists a trade-off between the recalculation cost and the migration cost.

For example, for $x = 2$ LRC$(6,2,2)$, the placement with $\beta = \alpha - 1 = 2$ (i.e., AGG) has recalculation cost (4) and migration cost (6); the placement with $\beta = 1$ has recalculation cost (6) and migration cost (3); and the placement with $\beta = 0$ (i.e., DIS) has recalculation cost (8) and migration cost (0). We gradually trade increased recalculation cost for decreased migration cost.

By Proposition 3, the data placement with a smaller β has smaller total bandwidth cost, so DIS has the smallest transitioning cost among all data placements with varied β.

Cost analysis for Problem (2). From Proposition 4, block migration is eliminated in Problem (2), so we solely focus on the recalculation process. Note that the data blocks of the x stripes reside in $\beta + (\alpha - 1 - \beta) \times x$ clusters. Since each of the first β clusters has at least xg data blocks, we transfer $g' = xg$

partial blocks for each such cluster for recalculation. Each of the next $(\alpha-1-\beta)\times x$ clusters stores at most $g+1$ data blocks. Therefore, for each such cluster, we directly transfer the original data blocks. If $\beta <= l\times\lfloor\frac{b}{g+1}\rfloor$, then we transfer $g+1$ data blocks for each of $(l\times\lfloor\frac{b}{g+1}\rfloor-\beta)\times x$ clusters and g data blocks for each of $(\alpha-1-l\times\lfloor\frac{b}{g+1}\rfloor)\times x$ clusters; otherwise, we transfer g data blocks for each of $(\alpha-1-\beta)\times x$ clusters.

$$\text{recalculation cost} = \begin{cases} (\alpha-1)\times x\times g-\beta\times x+ \\ l\times\lfloor\frac{b}{g+1}\rfloor\times x, \text{ if } \beta <= l\times\lfloor\frac{b}{g+1}\rfloor; \\ (\alpha-1)\times x\times g, \text{ otherwise.} \end{cases} \quad (4)$$

By Proposition 5, the data placement with a larger β ($\beta \leq l\times\lfloor\frac{b}{g+1}\rfloor$) has smaller total bandwidth cost, and AGG shows the smallest cost. For example, for 2 LRC$(6,2,2)$ stripes, the data placements with $\beta=0,1,2$ have costs 12, 10, 8, respectively.

Optimality guarantee. We first show that we can convert any random placement of the x LRC(k,l,g) stripes to our data placement derived in Algorithm 1 with saved bandwidth cost. If x sets of $g+1$ data blocks (or x sets of $g+\theta$ data and local parity blocks) from the x stripes are partially aggregated, then from Proposition 3 and Proposition 5, we can completely aggregate or disperse them to save the bandwidth cost. If one set of $g+1$ data blocks from one stripe are collocated with one set of $g+\theta$ data and local parity blocks from another stripe, we then switch the placements of these two block sets. By doing this, we save the bandwidth cost for Problem (2). Finally, if one set of $g+1$ data blocks (or one set of $g+\theta$ data and local parity blocks) are collocated with the global parity blocks, we then simply choose a different cluster to recalculate the new global parity blocks to avoid any block migration.

For example, in Figure 4, for 2 LRC$(8,2,2)$ stripes, we show a random placement, where D_4-D_6 of the first stripe are aggregated with D_{11}, L_2, D_{15}, L_3 of the second stripe. By switching the placements of D_{11}, L_2, D_{15}, L_3, and D_{12}-D_{14}, we get a designed data placement with $\beta=2$, which has smaller bandwidth cost than the random placement for Problem (2).

Theorem 1. *For any data placement of the x LRC(k,l,g) stripes that preserves both minimum repair cost and single-cluster fault tolerance, the minimum transitioning bandwidth cost is $x\times g\times(\alpha-1)$ for both Problem (1) and Problem (2).*

Proof. From the above analysis, we can transform any random placement into a designed data placement with reduced bandwidth cost. From the cost analysis, DIS has the smallest bandwidth cost in Problem (1), while AGG has the smallest cost in Problem (2), among all data placements with varied β. From Equations (2)-(4), the minimum transitioning bandwidth cost is $x\times g\times(\alpha-1)$ for Problem (1) and Problem (2). □

From Theorem 1, DIS achieves the minimum transitioning bandwidth cost (i.e., the sum of the migration and recalculation costs) for Problem (1), while AGG achieves the minimum transitioning cost for Problem (2).

Minimum repair cost for LRC(xk,xl,g'). After merging, the data placement of LRC(xk,xl,g') also achieves the minimum repair cost (i.e., $\lfloor\frac{b}{g+1}\rfloor$) in both Problem (1) and Problem (2).

V. Evaluation

We compare via numerical analysis and testbed experiments our optimal data placement (Section IV-B) with the random data placement (Section III-B). For the random data placement, we have two implementation approaches. The first approach (denoted by *Ran*) is to recalculate the new global parity blocks by directly accessing all original data blocks as in [43], [44], while the second approach (denoted by *Ran-P*) is to apply encoding-and-transferring to the recalculation process (e.g., the method in Figure 2) as in [41].

We summarize the overall results below. First, from numerical analysis, for Problem (1) (i.e., x LRC(k,l,g) to LRC(xk,xl,g)), DIS saves the cross-cluster traffic by up to 46.2%, while for Problem (2) (i.e., x LRC(k,l,g) to LRC(xk,xl,xg)), AGG saves the cross-cluster traffic by up to 34.1% (Section V-A). Second, from testbed experiments, for Problem (1), DIS reduces the transitioning time by up to 43.2%, while for Problem (2), AGG reduces the transitioning time by up to 35.3% (Section V-B).

A. Numerical Analysis

We analyze the total transitioning bandwidth costs during stripe merging for both Problem (1) and Problem (2).

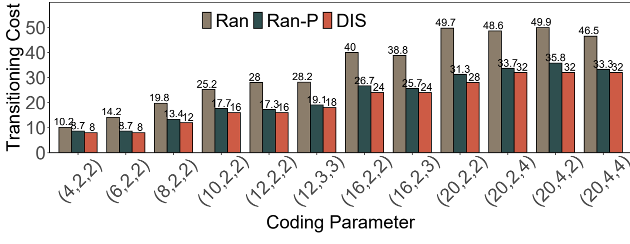
Ran and Ran-P. For Ran, we directly access all original data blocks for recalculation, so the recalculation cost is xk. We further apply encoding-and-transferring to obtain the recalculation cost of Ran-P. The migration cost is determined by the aggregation degree of the x stripes. We can refer to Proposition 2 to calculate the migration cost.

DIS for Problem (1) and AGG for Problem (2). From Theorem 1, the bandwidth costs of both DIS for Problem (1) and AGG for Problem (2) are $x\times g\times(\alpha-1)$.

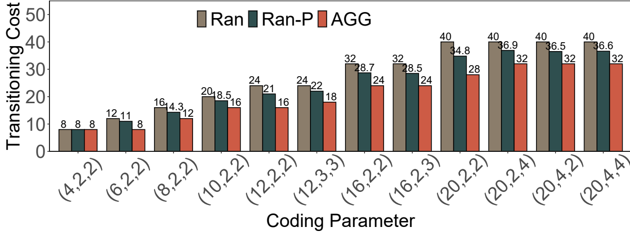
Setting. We consider different x and different parameters (k,l,g). We set the number of stripes as 100, and the number of clusters (i.e., c) as $x\alpha$, i.e., the number of clusters the DIS placement spans. We calculate the average transitioning cost for merging every x small-size stripes.

Results for different coding parameters. Figure 5 shows the transitioning cost results for $x=2$ under 12 sets of coding parameters. We summarize the following observations.

- From Figure 5(a), DIS has the minimum transitioning cost for Problem (1). DIS reduces the transitioning cost of Ran significantly, and also shows notable improvement over Ran-P. For example, for $(k,l,g)=(20,2,2)$, DIS saves the cross-cluster transitioning traffic of Ran and Ran-P by 43.7% and 10.5%, respectively.
- From Figure 5(b), AGG achieves the minimum transitioning cost for Problem (2). AGG can reduce the transitioning costs of Ran and Ran-P notably. For example, for $(k,l,g)=(6,2,2)$, AGG saves the cross-cluster traffic of Ran and Ran-P by 33.3% and 27.3%, respectively.
- In Figure 5(b), Ran, Ran-P, and AGG have the same costs for the case $(4,2,2)$. The reason is that the total number of

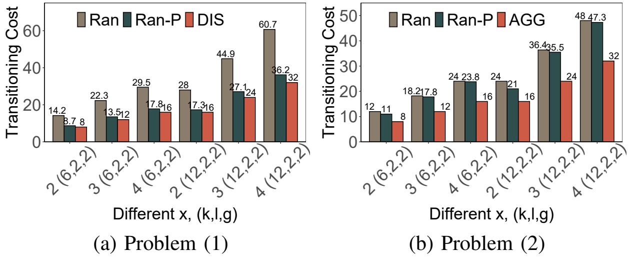


(a) Problem (1), x LRC(k,l,g) to LRC(xk,xl,g)



(b) Problem (2), x LRC(k,l,g) to LRC(xk,xl,xg)

Figure 5. Numerical results of the transitioning bandwidth costs.



(a) Problem (1) (b) Problem (2)

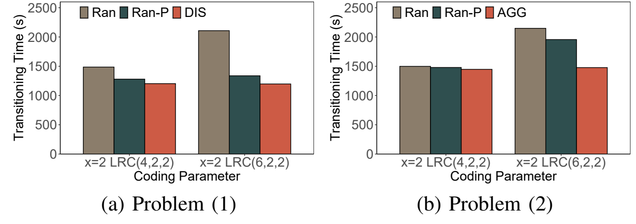
Figure 6. Numerical results under different values of x.

data blocks is $8 = lxg = lg'$, and the recalculation cost is always 8 regardless of the aggregation degree.

Results for different x. Figure 6 shows the transitioning cost results for different x. DIS constantly outperforms Ran and Ran-P under different x and different (k,l,g) in Problem (1), while AGG also shows stable improvements over Ran and Ran-P in Problem (2). For example, for LRC$(6,2,2)$, DIS reduces the cross-cluster transitioning traffic of Ran and Ran-P by 43.7%-46.2% and 8.0%-11.1% in Problem (1), and AGG reduces the cross-cluster traffic of Ran and Ran-P by 33.3%-34.1% and 27.3%-32.8%, across all values of x.

B. Testbed Experiments

We implement the placement schemes in a clustered storage system prototype, and conduct testbed experiments to understand their real transitioning performance. Our prototype is composed of a client, a coordinator (CN), and multiple datanodes (DNs) that are partitioned into clusters. The client interacts with the CN for metadata, and uploads data to the DNs. The CN sends commands to the DNs for redundancy transitioning. The DNs receive commands, and execute the actual data I/Os and transfer in parallel. Our prototype is implemented in C++ with about 4000 lines of code (LoC) and is open-sourced (Section I).



(a) Problem (1) (b) Problem (2)

Figure 7. Exp#1: Transitioning time under different coding parameters.

Setup. We deploy our prototype on a *Kubernetes* cluster with 9 physical nodes, each of which runs Ubuntu 16.04.7 LTS with a 40-core 2.40 GHz Intel(R) Xeon(R) Gold 5115, 128 GB RAM, a 1 TB SSD, and 1 Gbps bandwidth. Each node achieves 2000 MBps of disk read bandwidth and 350 MBps of write bandwidth. We deploy the client and CN on one node, and the DNs on 8 nodes. We use each node to emulate one cluster, so we can deploy multiple DNs on one node. We use the Wonder Shaper tool [4] to control the network bandwidth of each node, such that the ratio of the inner-node transfer speed to the cross-node transfer speed is around 8:1.

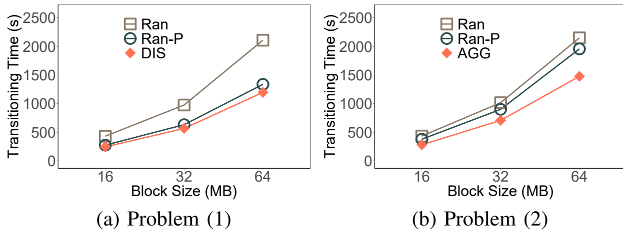
Methodology. We assume the following default configurations. We adopt the transitioning parameters $x = 2$ LRC$(6,2,2)$. We set the block size as 64 MB, and the packet size for network transfer as 1 MB. We upload 100 stripes of data and parity blocks, so the total data volume is around 60 GB. We set the number of clusters as 6. We vary different parameters in our experiments. We measure the time for transitioning all stripes. The results of each experiment are averaged over five runs.

Experiment 1 (Performance for different parameters). We first evaluate the performance under different coding parameters. We adopt two sets of parameters, $x = 2$ LRC$(4,2,2)$, and $x = 2$ LRC$(6,2,2)$. All other settings are the same as the defaults. Figure 7 shows the results.

From Figure 7(a), in Problem (1), DIS reduces the transitioning time of Ran and Ran-P by 43.2% and 10.5% under $x = 2$ LRC$(6,2,2)$, respectively, and by 19.0% and 6.0% under $x = 2$ LRC$(4,2,2)$, respectively. DIS greatly outperforms Ran as it exploits encoding-and-transferring to save the recalculation traffic. DIS also notably improves Ran-P since it disperses the two small-size stripes to eliminate the migration traffic.

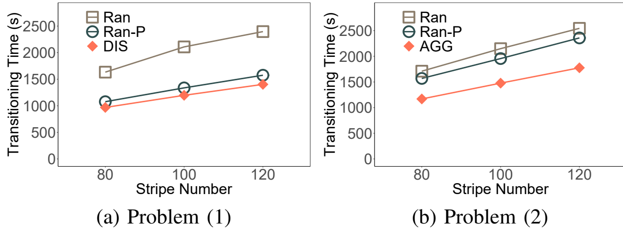
From Figure 7(b), in Problem (2), AGG reduces the transitioning time of Ran and Ran-P by 31.3% and 24.5% under $x = 2$ LRC$(6,2,2)$, respectively, since AGG efficiently saves the recalculation traffic via fully aggregating the data blocks. For 2 LRC$(4,2,2)$, Ran, Ran-P, and AGG have similar transitioning time, which confirms to our theoretical results (Section V-A).

Experiment 2 (Impact of block size). We now evaluate the impact of the block size, varied from 16 MB to 64 MB. We adopt $x = 2$ LRC$(6,2,2)$. Figure 8 shows the results. The transitioning time increases with a larger block size, while DIS constantly outperforms Ran and Ran-P in Problem (1), and AGG constantly outperforms Ran and Ran-P in Problem (2). Overall, DIS reduces the transitioning time of Ran and Ran-P by 42.0%-43.2% and 10.4%-10.8% in Problem (1), respectively,



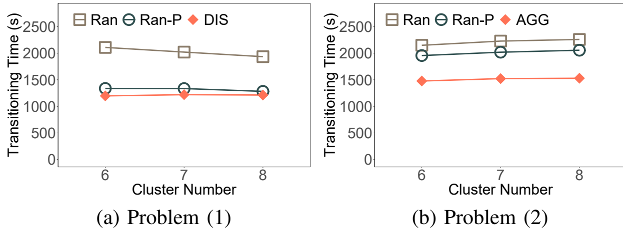
(a) Problem (1) (b) Problem (2)

Figure 8. Exp#2: Transitioning time under different block sizes.



(a) Problem (1) (b) Problem (2)

Figure 9. Exp#3: Transitioning time under different number of stripes.



(a) Problem (1) (b) Problem (2)

Figure 10. Exp#4: Transitioning time under different number of clusters.

while AGG reduces the transitioning time of Ran and Ran-P by 30.4%-35.3% and 21.5%-25.8% in Problem (2), respectively, across all block sizes.

Experiment 3 (Impact of number of stripes). We evaluate the impact of the number of stripes. We adopt $x = 2$ LRC$(6,2,2)$, fix the block size as 64 MB, and change the number of stripes from 80 to 120. Figure 9 shows the results. As the number of stripes increases, the transitioning time increases. In Problem (1), DIS outperforms Ran and Ran-P in the transitioning time by 40.7%-43.2% and 10.0%-11.0% under different number of stripes, respectively. In Problem (2), AGG outperforms Ran and Ran-P in the transitioning time by 30.2%-31.6% and 24.5%-25.5%, respectively.

Experiment 4 (Impact of number of clusters). We now study the impact of the number of clusters (i.e., c). We adopt $x = 2$ LRC$(6,2,2)$, fix the block size as 64 MB and the stripe number as 100, and vary c from 6 to 8. Figure 10 shows the results.

From Figure 10(a), in Problem (1), DIS reduces the transitioning time of Ran and Ran-P by 43.2% and 10.5%, 39.7% and 8.7%, and 37.3% and 5.4%, when the number of clusters is 6, 7, and 8, respectively. This indicates that DIS has larger performance gains over Ran and Ran-P with a smaller cluster number. From Figure 10(b), in Problem (2), AGG reduces the transitioning time of Ran and Ran-P by 31.3% and 24.5%, 31.7% and 24.6%, and 32.6% and 26.0%, when the number of clusters is 6, 7, and 8, respectively.

VI. Related Work

LRC designs. Several studies analyze the theoretical properties of LRC, such as minimum repair I/O [10], [27] and minimum hamming distances [31]–[33]. On the applied side, LRC is also implemented and evaluated in Azure [14], Facebook [29], and Ceph [18]. Recent study [41] analyzes the optimal trade-off between the repair performance and the redundancy transitioning between the parameters (k,l,g) and (k,l',g) in LRC, and further designs data placement that attains the optimal trade-off for a single stripe. Our work extends [41] and designs an optimal data placement scheme in the context of stripe merging, in which multiple stripes are merged with the minimum transitioning cost.

Redundancy transitioning. Several studies address redundancy transitioning in distributed storage, such as from replication to erasure coding [8], [20], [37], [38], efficient data redistribution for RAID arrays [39], [45], [47], erasure-coded storage [15]–[17], [42], [46], and in-memory erasure-coded KV stores [6], [34], [40]. Unlike the prior studies, we focus on applying stripe merging for bandwidth-efficient transitioning.

Prior studies [43], [44] also apply stripe merging in redundancy transitioning. HACFS [43] focuses on a limited setting that merges three small-size stripes into one large-size stripe for Product Codes [11], [19]. StripeMerge [44] also considers a specific setting that merges two stripes of Reed-Solomon Codes [28]. In contrast, our work considers the merge of an arbitrary number of LRC stripes. In addition, both studies [43], [44] consider flat storage systems without considering cross-cluster bandwidth, while we consider hierarchical clustered storage systems where the cross-cluster transfer is the bottleneck. The studies [22], [23] propose Convertible Codes to realize bandwidth-efficient stripe merging. Our work focuses on LRC and the optimal data placement for stripe merging.

Erasure coding in clustered storage. Recent studies focus on the deployment of erasure coding in clustered storage systems with hierarchical topologies. Some studies minimize the cross-cluster bandwidth for repair in erasure coding [12], [13], [21], [25], [30], while the study [41] minimizes the cross-cluster bandwidth for transitioning a single stripe. In contrast, our work addresses the stripe merging problem in clustered storage.

VII. Conclusion

We analyze the stripe merging problems for transitioning LRC in clustered storage systems. We prove that the placement of the multiple LRC stripes has a great impact on the cross-cluster transitioning traffic. We design a data placement scheme for the multiple stripes that is guaranteed to be optimal for two stripe merging problems. Both numerical studies and testbed experiments validate the effectiveness of our placement scheme. The authors have provided public access to their code at **https://zenodo.org/record/5797775**.

Acknowledgements: This work is supported by NSFC (61832011), Research Grants Council of HKSAR (AoE/P-404/18). The corresponding author is Patrick P. C. Lee.

REFERENCES

[1] Apache Hadoop 3.0.0. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html,2017.

[2] Erasure coding in Ceph. https://ceph.com/planet/erasure-coding-in-ceph/ ,2014.

[3] IDC. Data age 2025. https://www.seagate.com/our-story/data-age-2025/.

[4] The Wonder Shaper 1.4. https://github.com/magnific0/wondershaper.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant Mapreduce clusters. In *Proc. of USENIX ATC*, 2014.

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

[7] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. of ACM SIGCOMM*, 2013.

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

[9] 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.

[10] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Trans. on Information Theory*, 58(11):6925–6934, 2012.

[11] J. Hafner. HoVer erasure codes for disk arrays. In *Proc. of IEEE/IFIP DSN*, 2006.

[12] Y. Hu, L. Cheng, Q. Yao, P. P. Lee, W. Wang, and W. Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.

[13] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

[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] 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.

[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] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in Locally Repairable Codes. In *Proc. of USENIX ATC*, 2018.

[19] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Trans. on Storage*, 4(4):1–22, 2009.

[20] 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.

[21] X. Li, R. Li, P. P. Lee, and Y. Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, 2019.

[22] F. Maturana, C. Mukka, and K. Rashmi. Access-optimal linear MDS Convertible Codes for all parameters. In *Proc. of IEEE ISIT*, 2020.

[23] 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.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.

[25] N. Prakash, V. Abdrashitov, and M. Médard. The storage versus repair-bandwidth trade-off for clustered storage systems. *IEEE Trans.on Information Theory*, 64(8):5783–5805, Aug 2018.

[26] 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.

[27] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath. Locality and availability in distributed storage. *IEEE Trans. on Information Theory*, 62(8):4481–4493, 2016.

[28] 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.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel erasure codes for big data. In *Proc. of VLDB Endowment*, pages 325–336, 2013.

[30] Z. Shen, J. Shu, and P. P. Lee. Reconsidering single failure recovery in clustered file systems. In *Proc. of IEEE/IFIP DSN*, 2016.

[31] N. Silberstein, A. S. Rawat, O. O. Koyluoglu, and S. Vishwanath. Optimal Locally Repairable Codes via Rank-Metric Codes. In *Proc. of IEEE International Symposium on Information Theory*, 2013.

[32] I. Tamo and A. Barg. A family of optimal Locally Recoverable Codes. *IEEE Trans. on Information Theory*, 60(8):4661–4676, 2014.

[33] I. Tamo, D. S. Papailiopoulos, and A. G. Dimakis. Optimal Locally Repairable Codes and connections to Matroid theory. *IEEE Trans. on Information Theory*, 62(12):6661–6671, 2016.

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

[35] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *Proc. of USENIX NSDI*, 2015.

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

[37] S. Wei, Y. Li, Y. Xu, and S. Wu. DSC: Dynamic stripe construction for synchronous encoding in clustered file system. In *Proc. of IEEE INFOCOM*, 2017.

[38] 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.

[39] C. Wu and X. He. GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling. In *Proc. of IEEE ICPP*, 2012.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] G. Zhang, K. Li, J. Wang, and W. Zheng. Accelerate RDP RAID-6 scaling by reducing disk I/Os and XOR operations. *IEEE Trans. on Computers*, 64(1):32–44, 2015.

[46] 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.

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