Cauchy-Merge: An Efficient Cauchy Matrix based Stripe Merging Method for Reed-Solomon Codes

Huangzhen Xue[1], Chentao Wu[12*], Jie Li[123], Minyi Guo[1], Liyang Zhou[4], Shaoteng Liu[4], and Xiangyu Chen[4]
[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2] Yancheng Blockchain Research Institute, Hengyang, China
[3] MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China
[4] Huawei Technologies Co., Ltd., Shenzhen, China
[*] Corresponding Author: wuct@cs.sjtu.edu.cn

***Abstract*—Erasure codes are now widely deployed in cloud storage systems to ensure data reliability while maintaining low storage overhead. Wide-stripe erasure codes offer an opportunity for additional monetary cost reduction; however, they also introduce high recovery complexity, which can negatively impact the overall system performance. To improve the trade-off between storage overhead and system performance, it is crucial to transition cold data from narrow stripes to wide stripes efficiently. Nevertheless, generating wide stripes poses substantial challenges: existing methods involve significant computational and transmission overhead, high degraded read latency during maintenance tasks, as well as deployment limitations in practical systems.**

To address these issues, we propose Cauchy-Merge, a novel approach that effectively merges narrow stripes using merge-friendly Cauchy matrices. Cauchy-Merge reduces both the computational and network transmission overhead during the merging process and decreases degraded read latency under cluster maintenance through partial data migration. We also provide the definition and explicit constructions for merge-friendly Cauchy matrices along with efficient algorithms for the construction of encoding matrices and stripe merging. Numerical analyses, simulations, and testbed evaluations validate the effectiveness of Cauchy-Merge. Experimental results show that Cauchy-Merge significantly outperforms the state-of-the-art approaches such as DISMerge, reducing stripe merging time by up to 89.53% and degraded read latency by up to 69.17%.

I. Introduction

In modern cloud storage systems, where data volumes have reached exabytes (EB) [1], [2], the challenge of data failures is inevitable [3]. To ensure high data reliability, redundant mechanisms are essential. Erasure coding is preferred over replication due to its lower storage costs and equivalent fault tolerance [4]. Reed-Solomon (RS) codes [5], which encode k data blocks into r parity blocks to form a stripe that can tolerate any r block failures, are the most widely used erasure codes in production systems [1], [6]–[13].

To further reduce storage costs, wide-stripe erasure codes[1] have been proposed [14]–[18]. These codes increase the number of data blocks k within a stripe while maintaining the same fault tolerance capacity r, thus reducing the storage cost ratio $\frac{k+r}{k}$, which approaches 1 as k increases. However, this cost-saving comes at the expense of system performance due to higher recovery overhead. As k increases, so does the number of blocks needed to recover a single failed block, leading to greater network and decoding computation costs [19].

Data access frequency typically diminishes over time [6], and to balance system performance with storage overhead, it is common to encode new and frequently accessed ("hot") data using narrow stripes [20], [21]. As data becomes less accessed ("cold"), it is re-encoded into wide stripes to benefit from lower storage costs [22], [23]. Consequently, managing the redundancy transition from narrow to wide stripes is both crucial and essential.

The critical challenge of transitioning from narrow to wide stripes in erasure-coded storage systems significantly impacts network and computational resources. The conventional method, which involves re-encoding data blocks to create new parity blocks for wide stripes, is highly inefficient. It requires substantial network bandwidth to transfer data for encoding and considerable computational power for the encoding process itself, leading to reduced encoding throughput when moving from narrow to wide stripes [14]. To mitigate these issues, extensive research has been conducted to improve the redundancy transition process. Notable advancements include NCScale [24], [25], which uses network coding techniques to lessen network transmission overhead, thereby reducing bandwidth consumption during the transition. Methods like StripeMerge [17], Zebra [26], [27], and Convertible Codes [28], [29] leverage existing parity blocks to generate new ones, significantly reducing computational and network overhead by avoiding the re-encoding of data blocks from the ground up. Additionally, DISMerge [30] optimizes data placement for stripe merging, strategically positioning data to minimize costs associated with data movement and calculations during the merging process, leading to more efficient utilization of network and computational resources.

These innovations represent a concerted effort to make the redundancy transition from narrow to wide stripes more efficient. By reducing the need for extensive data movement and complex computations, these solutions aim to facilitate a more seamless and resource-efficient process, which is crucial for maintaining performance and reliability in large-scale storage systems. However, preceding approaches still exhibit limitations in certain aspects. The introduction of Cauchy-Merge presents a promising solution to these limitations:

[1]Wide stripes typically consist of a minimum of 20 data blocks [2], [14].

1) **High Computational and Transmission Overhead:** Existing methods like NCScale [24], [25] and DIS-Merge [30] still require significant merging overhead because they access all data blocks for merging and compute additional parity information (parity delta blocks). Cauchy-Merge reduces this overhead by effectively reusing existing parity blocks, which reduces the need for data block access and additional computations.
2) **Restrict Deployment Limitations:** StripeMerge [17] , Zebra [26], [27] and Convertible Codes [28], [29] have deployment limitations that are impractical in existing systems. StripeMerge and Convertible Code-I have to adopt a larger Galois filed than $GF(2^8)$, such as $GF(2^{16})$ to ensure the Maximum Distance Separable (MDS) property of wide stripes, which will bring out performance degradation and heightened implementation complexity [31]. Convertible Code-II reduces the number of parity blocks after merging, leading to decreased fault tolerance capability. Zebra requires that pre-merging stripes with the same parameters (k, r) use different encoding matrices, which is unrealistic in many storage systems. Most storage systems utilize a single erasure code, i.e., a single encoding matrix, especially for the same encoding parameters (k, r) [1], [2], [6]–[13]. Cauchy-Merge circumvents these issues through the use of carefully designed merge-friendly Cauchy matrices, making it more adaptable to existing systems.
3) **High Degraded Read Latency:** Cauchy-Merge also aims to improve the read performance of post-merging wide stripes during maintenance activities. By allowing partial data migration, it minimizes the impact on degraded read latency that typically occurs when maintenance zones are unavailable due to cluster maintenance.

In essence, Cauchy-Merge provides a more efficient and practical approach to stripe merging by addressing the computational and network inefficiencies of previous methods, offering a solution that is feasible in real-world systems without imposing restrictive prerequisites or data layout constraints. Moreover, it enhances system availability and read performance during maintenance periods, which is critical for large-scale storage systems where uptime and data access speed are paramount.

Our contribution can be summarized as:

- We define a merge-friendly Cauchy matrix and provide a method for constructing such matrices. An algorithm is developed allowing for the creation of pre-merging and post-merging encoding matrices for any given parameters (k, r) and merging stripe numbers β, which is a flexible solution adaptable to various storage scenarios without restrictive deployment requirements.
- We propose Cauchy-Merge, a method for merging narrow RS stripes into wider RS or Locally Repairable Code (LRC) stripes. It reduces the computational and network overhead that typically accompanies the merging process by leveraging the merge-friendly Cauchy matrix.
- We employ partial data migration, which is intended to maintain low degraded read latency even when certain parts of the storage system are undergoing maintenance. We demonstrate the effectiveness of Cauchy-Merge through numerical analyses, simulations, and practical experiments conducted on a testbed. The results show a significant reduction in both stripe merging time and degraded read latency compared to other methods.

TABLE I
NOTATIONS USED IN THIS PAPER.

Notation	Description
k	the number of data blocks in a pre-merging stripe
r	the number of parity blocks in a stripe
β	the number of stripes to be merged
$A_{i,j}$	the j-th element in the i-th row of the encoding matrix
$\mathbf{A}_{i,j}$	the j-th coefficient vector in the i-th row of the encoding matrix (i.e., $\mathbf{A}_{i,j} = [A_{i,jk}, \ldots, A_{i,jk+k-1}]$)
$D_{i,j}$	the j-th data block in the i-th pre-merging stripe
$\mathbf{D}_i$	the column vector of data blocks in the i-th pre-merging stripe (i.e., $\mathbf{D}_i = [D_{i,0}, \ldots, D_{i,k-1}]^T$)
$P_{i,j}$	the j-th parity block in the i-th pre-merging stripe
P_i^0	the i-th parity block in the post-merging stripe

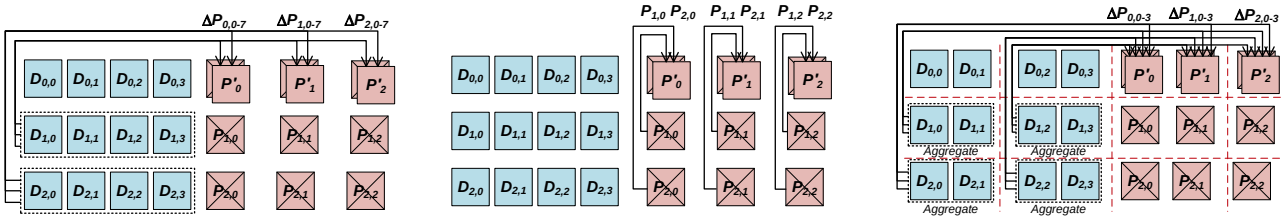
The rest of the paper is organized as follows: Section II introduces the related work and our motivations. Section III illustrates the Cauchy-Merge approach based on the merge-friendly Cauchy matrix in detail. Section IV presents the evaluation of Cauchy-Merge and other methods, and finally, we summarize the paper in Section V.

II. Background and Related Work

In this section, we provide a concise introduction to erasure codes. Then, we present existing approaches for redundancy transition along with their respective limitations. Finally, the motivation of this work is proposed. To facilitate comprehension, Table I lists the notations used in this paper.

A. Basics of Erasure Codes

Reed-Solomon (RS) Codes. RS codes [5], represented by $RS(k, r)$, are the most commonly used erasure codes [1], [6]–[9]. $RS(k, r)$ encodes a set of k data blocks to generate an additional r parity blocks. This assembly of $n = k + r$ blocks constitutes a stripe, distributed across n different nodes. Any k blocks within an $RS(k, r)$ stripe are sufficient to reconstruct the entire stripe, a property known as Maximum Distance Separable (MDS). Mathematically, the encoding process of $RS(k, r)$ involves a linear combination of k data blocks within the arithmetic framework of the Galois field $GF(2^w)$. For a systematic RS code $RS(k, r)$, a $(k + r) \quad k$ matrix, denoted as $\hat{G}$, functions as the generator matrix, and the initial k rows of $\hat{G}$ constitute an identity matrix, allowing us to represent $\hat{G}$ as $\hat{G} = \begin{bmatrix} I \\ G \end{bmatrix}$ where G is the encoding matrix. Two typical encoding matrices of RS codes are Cauchy matrix [32],



(a) NCScale: Each of $D_{1,i}$ and $D_{2,i}$ calculates and sends three parity delta blocks $\Delta P_{0,i}$, $\Delta P_{1,i}$, and $\Delta P_{2,i}$ to three new parity blocks, respectively. New $P_0^0 = P_{0,0} + \sum_i \Delta P_{0,i}$; new $P_1^0 = P_{0,1} + \sum_i \Delta P_{1,i}$; new $P_2^0 = P_{0,2} + \sum_i \Delta P_{2,i}$.

(b) StripeMerge/Zebra/Convertible Codes: The parity blocks with the same index are gathered together; for example, $P_{1,0}$ and $P_{2,0}$ are sent to $P_{0,0}$. For StripeMerge and Convertible Code-I ,new $P_0^0 = P_{0,0} + P_{1,0} + P_{2,0}$; new $P_1^0 = P_{0,1} + 2^4 P_{1,1} + 2^8 P_{2,1}$; new $P_2^0 = P_{0,2} + 2^8 P_{1,2} + 2^{16} P_{2,2}$. For Zebra, new $P_i^0 = P_{0,i} + P_{1,i} + P_{2,i}$.

(c) DISMerge: Data blocks in one cluster, such as $D_{1,0}$ and $D_{1,1}$, are aggregated, and three parity delta blocks $\Delta P_{0,i}$, $\Delta P_{1,i}$, and $\Delta P_{2,i}$ are computed and sent to three new parity blocks, respectively. New $P_0^0 = P_{0,0} + \sum_i \Delta P_{0,i}$; new $P_1^0 = P_{0,1} + \sum_i \Delta P_{1,i}$; new $P_2^0 = P_{0,2} + \sum_i \Delta P_{2,i}$.

Fig. 1. The merging processes of several state-of-the-art approaches.

denoted as G_C, and Vandermonde matrix [33], denoted as G_V , which are defined as follows

$$\mathbf{G}_C = \begin{bmatrix} \frac{1}{x_0 \quad y_0} & \frac{1}{x_0 \quad y_1} & \cdots & \frac{1}{x_0 \quad y_{k-1}} \\ \frac{1}{x_1 \quad y_0} & \frac{1}{x_1 \quad y_1} & \cdots & \frac{1}{x_1 \quad y_{k-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_{r-1} \quad y_0} & \frac{1}{x_{r-1} \quad y_1} & \cdots & \frac{1}{x_{r-1} \quad y_{k-1}} \end{bmatrix},$$

$$\mathbf{G}_V = \begin{bmatrix} 1^0 & 1^1 & \cdots & 1^{k-1} \\ 2^0 & 2^1 & \cdots & 2^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ r^0 & r^1 & \cdots & r^{k-1} \end{bmatrix},$$

where $X = \{x_0, \ldots, x_{r-1}\}, Y = \{y_0, \ldots, y_{k-1}\}$ and $X \cap Y = \emptyset$. While both Vandermonde and Cauchy matrices can serve as encoding matrices for RS codes, Cauchy matrices are more favored for wide-stripe RS codes. This preference is due to the reliability of Cauchy matrices in ensuring the MDS property for any systematic RS code $RS(k, r)$ within the commonly used Galois field $GF(2^8)$, as long as $k + r < 2^8$. Conversely, RS codes that employ Vandermonde matrices are subject to constraints on the stripe length. For instance, if $r = 4$, then k must not exceed 21 [34]. Consequently, we adopt Cauchy matrices as our encoding matrix, particularly because the stripes tend to be wide after they are merged.

Locally Repairable Codes (LRCs). LRCs are another class of popular erasure codes [12], [23], [35]. LRCs are RS-based codes that can be represented as $LRC(k, l, g)$, where k, l, and g denote the number of data blocks, local parity blocks, and global parity blocks, respectively. LRCs introduce local parity blocks to reduce the computational and network overhead during the recovery process. $LRC(k, l, g)$ divides k data blocks into l local groups, each accompanied by one local parity block. In addition, g global parity blocks are computed from all k data blocks, similar to RS codes. In the event of a single block failure, it can be reconstructed by accessing only the $\frac{k}{l}$ surviving blocks within the same local group.

B. Existing Redundancy Transition Methods

To achieve a cost-effective storage system with optimal performance, the transition from narrow to wide stripes is of paramount significance. Presently, two primary categories of methods are employed to execute this transition: scaling-based methods [24], [25], [36]–[41] and merging-based methods [17], [28]–[30]. Scaling-based methods convert a (k, r) stripe to a $(k + s, r)$ stripe, aiming to minimize conversion costs when adding s data blocks. Merging-based methods involve merging β (k, r) stripes into a single $(\beta k, r)$ stripe. We subsequently explore existing approaches used for redundancy transition, elucidating their respective strengths and limitations.

1) Scaling-based Methods:

NCScale. NCScale [24], [25] is the state-of-the-art scaling method that achieves the information-theoretically minimum scaling bandwidth and minimizes network transmission during the scaling process through network coding. An example of merging three $RS(4, 3)$ stripes via NCScale is illustrated in Figure 1(a). All data blocks from the second and third stripes, denoted as $D_{1,i}$ and $D_{2,i}$ respectively (where $0 \quad i \quad 3$), compute and transmit three parity delta blocks—$\Delta P_{0,i}$, $\Delta P_{1,i}$, and $\Delta P_{2,i}$—to $P_{0,0}$, $P_{0,1}$, and $P_{0,2}$, respectively, to update the post-merging parity blocks. However, NCScale has a limitation due to its reliance on storage scaling to accommodate an arbitrary number of scaling nodes. This constraint hinders the utilization of parity blocks, resulting in relatively high computational and network transmission overhead compared to other merging-based methods.

2) Merging-based Methods:

StripeMerge. StripeMerge [17] is a technique tailored for merging stripes of Vandermonde-based RS codes, as shown in Figure 1(b). For Vandermonde-based RS codes, the post-merging parity blocks can be expressed as linear combinations of the original stripes' parity blocks. For example, when merging three $RS(4, 3)$ stripes into one $RS(12, 3)$ stripe, the second parity blocks of the three pre-merging stripes are denoted as $P_{j,1} = \sum_{i=0}^{3} 2^i D_{j,i}$, for $j = 0, 1, 2$. Then, the

TABLE II
SUMMARY ON STRIPE MERGING METHODS

Merging Methods	Computation Pattern of Parity Blocks	Computation and Network Transmission Overhead	Deployment Limitations	Degraded Read Latency
NCScale [24], [25]	Data only	High	None	High
StripeMerge [17]	Parity only	Optimal	Expand field size	High
DISMerge [30]	Data only	Medium	None	High
Zebra [26], [27]	Parity only	Optimal	Prerequisite of different pre-merging encoding matrices	High
Convertible Codes-I [28], [29]	Parity only	Optimal	Expand field size	High
Convertible Codes-II [28], [29]	Parity only	Optimal	Decrease fault tolerance capability	High
Cauchy-Merge	Mostly parity + a small fraction of data	Low	None	Low

second parity block of the post-merging stripe, labeled as P_1^0, can be expressed as $P_1^0 = \sum_{i=0}^{3} 2^i D_{0,i} + \sum_{i=0}^{3} 2^{i+4} D_{1,i} + \sum_{i=0}^{3} 2^{i+8} D_{2,i} = P_{0,1} + 2^4 P_{1,1} + 2^8 P_{2,1}$. This computing pattern extends to the first and third parity blocks as well. Notably, StripeMerge significantly reduces both network transmission overhead and computational complexity by exclusively relying on parity blocks during the stripe merging process. However, it faces deployment challenges in real-world systems because systematic RS codes based on Vandermonde matrices do not guarantee the MDS property within $GF(2^8)$ for wide stripes [34], [42], unless the Galois field's size is expanded, which leads to performance degradation and increased implementation complexity [31].

DISMerge. DISMerge [30] has identified an optimal data placement strategy that minimizes inter-cluster network transmission during the merging of LRC stripes. Further reductions in network transmission overhead are achieved through encode-by-transfer mechanisms. An illustrative example of DISMerge is depicted in Figure 1(c). It aggregates data blocks within a cluster to compute and transmit parity delta blocks to update the post-merging parity blocks. For example, $D_{1,0}$ and $D_{1,1}$ are aggregated to calculate and transmit three parity delta blocks for P_0^0, P_1^0, and P_2^0 respectively, and this aggregation process is applied to other data blocks in the same cluster to efficiently perform the merging operation. However, DISMerge utilizes conventional Cauchy matrices for encoding, which does not take advantage of pre-merging parity blocks, resulting in relatively high computational and network transmission overhead compared to methods that could reuse existing parity blocks more effectively. Moreover, DISMerge's data placement strategy requires data blocks to be centrally located within local groups, which can lead to increased degraded read latency during system maintenance. When a block goes offline, global recovery is needed for decoding, which can slow down the read operations and affect the system's overall performance.

Zebra. Zebra [26], [27] leverages the property that every submatrix of a Cauchy matrix is itself a Cauchy matrix. It dissects a Cauchy matrix into multiple sub-Cauchy matrices, with each serving as an encoding matrix for pre-merging stripes. The concatenation of these sub-Cauchy matrices then forms the encoding matrix for post-merging stripes. Consequently, it yields a computational pattern similar to that of StripeMerge, as depicted in Figure 1(b). The parity blocks of the post-merging stripes can be efficiently computed through XOR operations with their corresponding counterparts from the pre-merging stripes. This approach significantly reduces the computational and network overhead during stripe merging. However, Zebra introduces an impractical requirement: it necessitates that the pre-merging stripes with identical (k, r) parameters employ distinct encoding matrices, which is unfeasible in real-world storage systems. In practice, most storage systems utilize a single encoding matrix for the same encoding parameters (k, r) [1], [2], [6]–[13].

Convertible Codes. Convertible Codes [28], [29] formally establish the theoretical lower bound of access costs during the stripe merging of linear MDS codes. They provide two explicit constructions of encoding matrices that achieve the theoretical lower bound, substantially reducing I/O and network overhead in the stripe merging process. However, both constructions face practical deployment limitations. Convertible Code-I relies on Vandermonde matrices, similar to StripeMerge, necessitating the Galois field size to be $O(2^{n^3})$[2] [29], which implies an exponential field size requirement relative to the number of blocks in the stripe. Convertible Code-II, based on superregular Hankel arrays, requires a significantly lower (polynomial) field size. It selects submatrices from the Hankel array to serve as both pre-merging and post-merging encoding matrices. Each row of the post-merging encoding matrix is a concatenation of specific rows from the pre-merging encoding matrix to facilitate the efficient reuse of existing parity blocks. This construction has a tradeoff between the field size and the maximum value of the number of parity blocks after merging, denoted as r^0. However, even at the extreme end, where the maximum r^0 is supported and the field size requirement is $O(kr)$, it still has to reduce the number of parity blocks, since $r^0 \quad r \quad \beta + 1$ and $\beta \quad 2$ [29], leading to a decrease in fault tolerance capability. Consequently, both constructions

[2]This bound is not a tight one. In fact, $GF(2^{16})$ can cover a wide range of (k, r) to ensure the MDS property of the Vandermonde-based RS codes, such as $RS(200, 4)$. However, compared to $GF(2^8)$, $GF(2^{16})$ will result in performance degradation and increased implementation complexity [31].

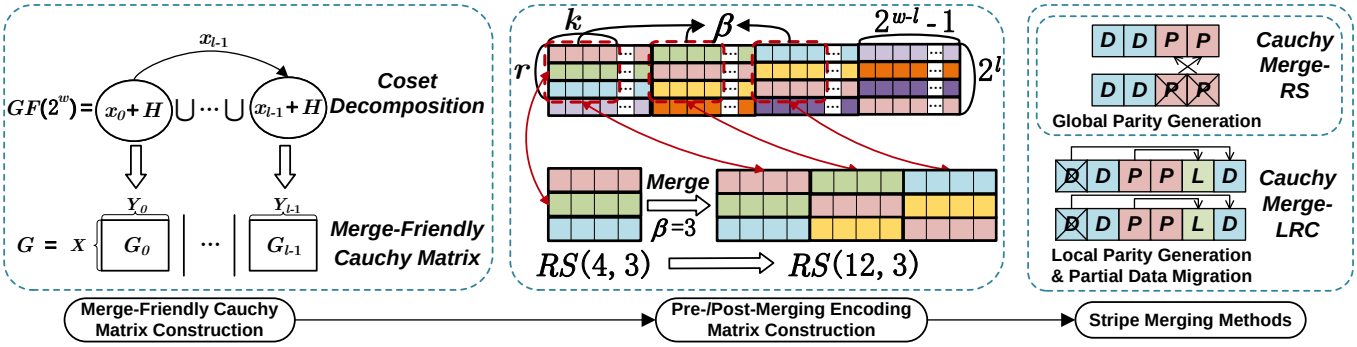


Fig. 2. The overview of Cauchy-Merge Approach

of Convertible Codes encounter deployment challenges within real-world systems.

C. Motivation

The properties of various redundancy transition approaches for merging narrow stripes into wide stripes are summarized in Table II. As analyzed earlier, NCScale [24], [25] and DISMerge [30] do not optimize the computation pattern for the post-merging parity blocks; thereby, they require reading all data blocks to compute the post-merging parity blocks. As a result, they do not fundamentally reduce computational complexity and network transmission overhead. StripeMerge, Zebra, and Convertible Codes implement the optimal computation pattern, in which only parity blocks are used to calculate the post-merging parity blocks, thereby effectively reducing computational complexity and network transmission overhead. However, they all face deployment limitations in real-world systems. StripeMerge and Convertible Code-I impose an increase in the field size to satisfy the MDS property, leading to performance degradation and heightened implementation complexity [31]. Convertible Code-II reduces the number of parity blocks after merging, leading to a reduction in fault tolerance capability. Zebra requires the existence of different encoding matrices for the same parameters (k, r) within the system, which is impractical [1], [2], [6]–[13].

To address these issues, we aim to design a stripe merging method, Cauchy-Merge, that leverages Cauchy matrices, which ensure the MDS property, to merge narrow stripes into wide stripes. By adopting the stripe merging method, we can alter the computation pattern by reusing existing parity blocks as much as possible, leading to reduced computational complexity and network overhead. Furthermore, we will ensure that the pre-merging stripes with the same parameters (k, r) are encoded by the same encoding matrices, which is the most practical scenario in real-world systems.

III. Cauchy Merge

A. Overview of Cauchy-Merge

RS codes and LRCs are the two most commonly used RS-based erasure codes in contemporary cloud storage systems, such as HDFS [13] and Windows Azure Storage [12]. Therefore, our focus is on stripe merging from RS codes to RS codes and LRCs (note that merging from LRCs to LRCs can be regarded as a special case of merging from RS codes to LRCs). We propose Cauchy-Merge, a methodology for merging narrow RS stripes into wide RS or LRC stripes. Cauchy-Merge reduces the computational and network transmission overhead of the merging process by using merge-friendly Cauchy matrices, which allow for the efficient utilization of parity blocks from the original stripes to generate both global and local parity blocks within the wide stripe. Furthermore, we employ partial data migration to distribute each local group across different maintenance zones, thereby reducing degraded read latency during cluster maintenance. Figure 2 presents an overview of Cauchy-Merge, which consists of three main components: merge-friendly Cauchy matrix construction, pre-merging and post-merging encoding matrix construction, and stripe merging methods.

Merge-Friendly Cauchy Matrix Construction. Cauchy-Merge defines a merge-friendly Cauchy matrix and constructs it through coset decomposition, as detailed in Section III-B.

Pre-Merging and Post-Merging Encoding Matrix Construction. Cauchy-Merge constructs a pair of pre-merging and post-merging encoding matrices for RS codes to facilitate efficient stripe merging. This construction is based on the merge-friendly Cauchy matrix, as described in Section III-C.

Stripe Merging Methods. Cauchy-Merge introduces two efficient methods for merging RS stripes into RS stripes or LRC stripes. These methods utilize existing parity blocks to reduce network transmission and computational overhead. The methods are based on the merge-friendly Cauchy matrix and are elaborated in Section III-D.

B. Merge-Friendly Cauchy Matrix

First, we consider the specific problem of merging r $RS(k, r)$ stripes into a single $RS(kr, r)$ stripe. We will discuss merging β $RS(k, r)$ stripes into a single $RS(\beta k, r)$ stripe in Section III-C.

The size of the encoding matrix changes from $r \quad k$ to $r \quad kr$ when merging r $RS(k, r)$ stripes. To reuse parity blocks from the original stripes during the merging process, one approach is

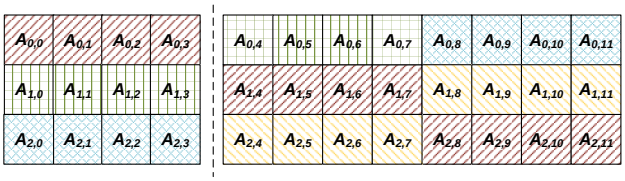


Fig. 3. The merge-friendly encoding matrix for $RS(12,3)$ is depicted, with its leftmost four columns forming the encoding matrix corresponding to the pre-merging $RS(4,3)$ matrix. Coefficient vectors are highlighted using identical colors and filling patterns to indicate identity.

to construct the post-merging encoding matrix such that some of its coefficients align with those of the pre-merging encoding matrices. This allows original parity blocks to contribute to the partial sums in the encoding computation of the post-merging stripe. Specifically, if we view the k elements in each row of the pre-merging encoding matrices as coefficient vectors, the post-merging encoding matrix should have a block symmetric form, with these coefficient vectors serving as its partitioned blocks. For instance, Figure 3 illustrates an encoding matrix for $RS(12,3)$, where the first 4 columns form the encoding matrix for $RS(4,3)$ and the full 12 columns represent the post-merging encoding matrix. The coefficient vectors are highlighted with different colors and filling patterns to emphasize their identity. This matrix structure facilitates the efficient reuse of the original parity blocks during the stripe merging process, which is further discussed in Section III-D and Section III-E. Consequently, we define a Cauchy matrix with a block symmetric structure as a merge-friendly Cauchy matrix and provide an explicit construction for it.

Definition 1 (Merge-Friendly Cauchy Matrix)**.** Let G be an $r \quad kr$ matrix over a field F. We define G as a merge-friendly Cauchy matrix if G is both a Cauchy matrix and a block symmetric matrix with r row partitions and r column partitions. Each block of G is an $1 \quad k$ row vector, referred to as a coefficient vector.

Following the definition of a merge-friendly Cauchy matrix, we proceed to present a general and explicit construction of this matrix through coset decomposition. To begin, we introduce some preliminary concepts in group theory and coset decomposition.

Definition 2 (Finite Groups and Subgroups)**.** A **finite group** $(G,+)$ is a set G that is finite and closed under a binary operation $+$ satisfying the properties of closure, associativity, identity, and invertibility. A subset $H \quad G$ is a **subgroup** of G if H is also closed under the binary operation $+$. The **order** of a group is defined as the number of its elements.

Definition 3 (Generating Set and Basis)**.** Let G be a group under a binary operation $+$, and let S be a non-empty subset of G. The set S is termed a **generating set** of G if every element in G can be expressed as a finite sum of elements in S and their inverses under the binary operation $+$. The **basis** of G is the generating set that has the smallest cardinality.

Definition 4 (Cosets and Representatives)**.** Let H be a subgroup of a group G and let g be a **representative** in G. Then the set $fg + h \; j \; h \; 2 \; Hg$ is called the **coset** generated by g with respect to H, denoted by $g + H$.

Theorem 1 (Lemma 2.40 in [43])**.** *Let H be a subgroup of a group G and let $g_1, g_2 \; 2 \; G$, then $g_1 + H = g_2 + H$ if and only if $g_1 + g_2 \; 2 \; H$,.*

Theorem 2 (Coset Decomposition [43])**.** *Let H be a subgroup of a group G. Then the cosets of H in G partition G. In other words, G is the union of the disjoint cosets of H in G.*

$$G = H \; [\; (g_1 + H) \; [\; (g_2 + H) \; [\quad [\; (g_{jGj/jHj \;\; 1} + H),$$

where $g_1, g_2, \ldots$ are elements of G chosen such that all cosets are distinct.

The encoding matrices of RS codes are defined over the Galois field, which is a specific type of finite group. Consequently, we can leverage the concept of coset decomposition in group theory to construct merge-friendly Cauchy matrices.

Construction 1. Let F be a Galois field with order 2^w and let H be a subgroup of F with order 2^l, where $l < w$.

1) Let $H = fy_0, y_1, \ldots, y_{2^{w \;\; l} \;\; 1}g$, with $y_{2^{w \;\; l} \;\; 1} = 0$ and let $\alpha_0, \alpha_1, \ldots, \alpha_{w \;\; l \;\; 1}$ form a basis of H. By Definition 2 and Definition 3, there exist $\alpha_{w \;\; l}, \alpha_{w \;\; l+1}, \ldots, \alpha_{w \;\; 1}$ such that $\alpha_0, \alpha_1, \ldots, \alpha_{w \;\; 1}$ form a basis of F.
2) For $i \; 2 \; [0, 2^l \quad 1]$, i has a binary representation $(i_0, i_1, \ldots, i_{l \;\; 1})$, i.e., $i = \sum_{j=0}^{l \;\; 1} i_j \;\; 2^j$. Define
$$x_i = \sum_{j=0}^{l \;\; 1} i_j \alpha_{w \;\; l+j}. \tag{1}$$
3) For $i \; 2 \; [0, 2^l \quad 1], j \; 2 \; [0, 2^{w \;\; l} \quad 1]$, denote $x_i + y_j$ as $y_{i,j}$. Let
$$\begin{aligned} x_i + H &= fx_i + y_0, x_i + y_1, \ldots, x_i + y_{2^{w \;\; l} \;\; 1}g \\ &= fy_{i,0}, y_{i,1}, \ldots, y_{i,2^{w \;\; l} \;\; 1}g. \end{aligned} \tag{2}$$
4) For $i \; 2 \; [0, 2^l \quad 1]$, denote
$$Y_i = (x_i + H) nf x_i g = fy_{i,0}, y_{i,1}, \ldots, y_{i,2^{w \;\; l} \;\; 2}g. \tag{3}$$
The second equality can be derived from the fact that $y_{2^{w \;\; l} \;\; 1} = 0$.
5) Define the matrix $A \; 2 \; F^{[0,2^l \;\; 1] \;\; [0,2^{w \;\; 2^l \;\; 1}]}$ with the (m,n)-th entry being
$$A_{m,n} = A_{m,a \; 2^l + b} = \frac{1}{x_m + y_{a,b}}. \tag{4}$$

Construction 1 provides a class of encoding matrix for $RS(2^w \quad 2^l, 2^l)$. For example, when $w = 4$, $l = 2$, we have $F = f0, 1, \ldots, 15g$ and let's define $H = f4, 8, 12, 0g$. Recall that the operation of addition in a Galois field is equivalent to XOR; therefore, the basis of H is $\alpha_0 = 4$ and $\alpha_1 = 8$, while the basis of F is $\alpha_0 = 4$, $\alpha_1 = 8$, $\alpha_2 = 1$, and $\alpha_3 = 2$ (step

1). Next, we can deduce that $x_0 = 0$, $x_1 = \alpha_2 = 1$, $x_2 = \alpha_3 = 2$, and $x_3 = \alpha_2 + \alpha_3 = 3$ (step 2). Consequently, we have $Y_0 = \{4, 8, 12\}$, $Y_1 = \{5, 9, 13\}$, $Y_2 = \{6, 10, 14\}$, and $Y_3 = \{7, 11, 15\}$ (steps 3-4). Finally, the matrix is defined by (4) as the encoding matrix for $RS(12, 4)$(step 5). Here we present the example where H is selected such that $x_i = i$. In fact, we can choose any H to complete the construction steps, as long as H is a subgroup of F with order 2^l.

Next, we shall prove that the matrices constructed by Construction 1 are merge-friendly Cauchy matrices.

Lemma 1. *The matrix A constructed by Construction 1 is a Cauchy matrix.*

Proof. For any $i, j \in [0, 2^l - 1]$ with $i \neq j$, using (1), it is straightforward to derive that $x_i \neq x_j$ and $x_i + x_j \notin H$ because the basis of $\{x_i\}$ is disjoint from the basis of H. Applying Theorem 1, we conclude that $x_i + H \neq x_j + H$, implying the disjointness of $x_i + H$ and $x_j + H$. Consequently, $\{x_i + H\}$ forms a coset decomposition of F, according to Theorem 2. Then all elements in $\{x_i + H\}$ are distinct, and we have $x_i \notin x_j + H$ implying that $x_i \notin Y_j$. Meanwhile, the definition of Y_i in (3) ensures that $x_i \notin Y_i$. Hence, $\{x_i\}$ and $\{Y_i\}$ satisfy the conditions needed to construct a Cauchy matrix A. □

Lemma 2. *The matrix A constructed by Construction 1 is a block symmetric matrix.*

Proof. To prove that A is a block symmetric matrix, it suffices to demonstrate that $A_{m,a\,2^l+b} = A_{a,m\,2^l+b}$. According to (4), this is equivalent to proving that $x_m + y_{a,b} = x_a + y_{m,b}$. Recall the definition $y_{i,j} = x_i + y_j$ as given in (2); it holds that $x_m + y_{a,b} = x_m + (x_a + y_b) = x_a + (x_m + y_b) = x_a + y_{m,b}$. □

Theorem 3. *The matrix G constructed by Construction 1 is a merge-friendly Cauchy matrix.*

Proof. This follows directly from Lemma 1 and Lemma 2 □

C. Pre-merging and Post-merging Encoding Matrix

From Construction 1, we can construct a merge-friendly Cauchy matrix of size $2^l \times (2^w - 2^l)$. This matrix directly yields the post-merging encoding matrix for the stripes of $RS(2^w - 2^l, 2^l)$ and the corresponding pre-merging encoding matrix for the stripes of $RS(2^{w-l} - 1, 2^l)$. However, the coding parameters (k, r) specified by Construction 1 are stringent. It mandates that the pre-merging stripe has $k = 2^{w-l} - 1$, $r = 2^l$, and each merging operation involves exactly $\beta = 2^l$ stripes. Adhering strictly to these parameter requirements from Construction 1 is unrealistic and unnecessary. Another construction method, based on Construction 1, allows us to generate pre-merging and post-merging encoding matrices while relaxing the constraints on the coding parameters. Specifically, for the pre-merging stripes, $k \leq 2^{w-l} - 1$, $r \geq 2^l$, and the number of stripes merged each time, $\beta \geq 2^l$. This relaxation of parameters is achieved by carefully selecting submatrices from the merge-friendly Cauchy matrix. These submatrices, once selected, act as the pre-merging and post-merging encoding matrices while still maintaining the properties of a merge-friendly Cauchy matrix. This allows for the efficient reuse of existing parity blocks during the computation of new parity blocks in the process of stripe merging.

Algorithm 1 Construct A Pair of Encoding Matrices

1: **procedure** CONSTRUCT(k, r, β)
2: $l \leftarrow \lceil \log_2^r \rceil$
3: $H \leftarrow \{i \cdot 2^l\}$: subgroup of $GF(2^w)$ with $|H| = 2^{w-l}$
4: $x_0 \leftarrow 0$
5: **for** $i = 1$ to $2^l - 1$ **do**
6: $x_i \leftarrow$ smallest element of $G \setminus \bigcup_{j=0}^{i-1}(x_j + H)$
7: **end for**
8: $X \leftarrow \{x_0, x_1, \ldots, x_{2^l-1}\}$
9: $Y_i \leftarrow (x_i + H) \setminus x_i$
10: $X \leftarrow$ first r elements of X
11: $Y_i \leftarrow$ first k elements of Y_i
12: $Y \leftarrow \{Y_0, Y_1, \ldots, Y_{\beta-1}\}$
13: **return** (X, Y_0) and (X, Y)
14: **end procedure**

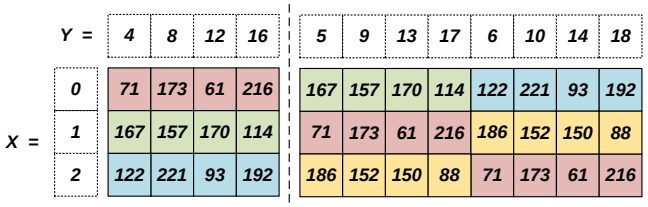


Fig. 4. An example of the merge-friendly Cauchy matrix. The leftmost column and the uppermost row represent X and Y, respectively, generated by Algorithm 1, determining the Cauchy encoding matrix. The 3×4 submatrix to the left of the dashed line is the encoding matrix of the pre-merging $RS(4, 3)$ code and the entire 3×12 matrix is the encoding matrix of the post-merging $RS(12, 3)$ code. The same coefficient vector is highlighted in the same color.

Algorithm 1 outlines the process of constructing this pair of encoding matrices, relaxing the constraints on k, r, and β compared to Construction 1. Algorithm 1 initially follows the steps of Construction 1 (lines 2-9). Then, it selects the first r representatives to comprise X instead of the entire 2^l representatives (line 10). Furthermore, it takes the first k elements from each coset to form Y_i instead of all $2^{w-l} - 1$ elements (line 11). Lastly, it choose β cosets to comprise Y, which determines the post-merging encoding matrix with the size $r \times \beta k$ (line 12).

Figure 4 presents a specific example of the construction of the encoding matrices for pre-merging $RS(4, 3)$ and post-merging $RS(12, 3)$ codes via Algorithm 1 over $GF(2^8)$. Initially, we determine $l = \lceil \log_2 3 \rceil = 2$ (line 2), and define H as a subgroup of order 2^6 in $GF(2^8)$. Let $H = \{i \cdot 2^l\} = \{4i\}$ (line 3). We then compute all cosets and their representatives for H, with each new representative being the minimal element in $GF(2^8)$ that does not belong to any pre-existing coset (lines 5-7). The cosets generated by these representatives form a coset decomposition of $GF(2^8)$ (lines 8-9). From the 2^l representatives, we select the first r representatives to comprise X (line 10), and from the cosets induced by these r representatives, we take the first k elements of each coset to

constitute Y (lines 11-12). The first coset (H itself) contributes its first k elements to constitute Y_0 of the pre-merging RS code. Thus, we obtain (X, Y_0) and (X, Y) for the encoding matrices of the pre-merging and post-merging RS codes, respectively.

D. Stripe Merging Methods

Based on the merge-friendly Cauchy matrix, Cauchy-Merge can efficiently merge β stripes of $RS(k, r)$ into a single stripe of either $RS(\beta k, r)$ or $LRC(\beta k, \beta, r)$. We propose two merging algorithms for Cauchy-Merge: Cauchy-Merge-RS for merging into an RS stripe and Cauchy-Merge-LRC for merging into an LRC stripe.

1) Cauchy-Merge-RS: By leveraging the properties of the merge-friendly Cauchy matrix, the parity blocks in the post-merging stripe can be computed using existing parity blocks from the original stripes and some additional data blocks. For $i \in [0, r-1], j \in [0, k-1], i_1 \in [0, \beta-1], i_2 \in [0, r-1]$, let P_i^0 represent the i-th parity block of the post-merging stripe, P_{i_1,i_2} denote the i_2-th parity block of the i_1-th original stripe, $D_{i,j}$ be the j-th data block of the i-th original stripe, $\boldsymbol{D}_i$ be the column vector of the data blocks of the i-th stripe, i.e., $\boldsymbol{D}_i = [D_{i,0}, \ldots, D_{i,k-1}]^T$, $\boldsymbol{A}_{i_1,i_2}$ be the i_2-th coefficient vector on the i_1-th row of the merge-friendly Cauchy matrix, i.e., $\boldsymbol{A}_{i_1,i_2} = [A_{i_1,i_2 k}, \ldots, A_{i_1,i_2 k+k-1}]$. Notice that $\boldsymbol{A}_{i,i} = \boldsymbol{A}_{0,0}$ and $\boldsymbol{A}_{i_1,i_2} = \boldsymbol{A}_{i_2,i_1}$ according to the block symmetric property of the merge-friendly Cauchy matrix. Combined with the fact that $P_{i_1,i_2} = \boldsymbol{A}_{i_2,0}\boldsymbol{D}_{i_1}$ (if P_{i_1,i_2} exists), we can derive that when $i = 0$:

$$
\begin{aligned}
P_0^0 &= \sum_{i=0}^{\beta-1} \boldsymbol{A}_{0,i}\boldsymbol{D}_i = \sum_{i=0}^{\beta-1} \boldsymbol{A}_{i,0}\boldsymbol{D}_i \\
&= \begin{cases} \sum_{i=0}^{\beta-1} P_{i,i}, & \text{if } \beta \le r \\ \sum_{i=0}^{r-1} P_{i,i} + \sum_{i=r}^{\beta-1} \boldsymbol{A}_{i,0}\boldsymbol{D}_i, & \text{if } \beta > r \end{cases}
\end{aligned} \tag{5}
$$

when $1 \le i \le r-1$:

$$
\begin{aligned}
P_i^0 &= \boldsymbol{A}_{i,0}\boldsymbol{D}_0 + \sum_{j_1=1}^{i-1} \boldsymbol{A}_{i,j_1}\boldsymbol{D}_{j_1} + \boldsymbol{A}_{i,i}\boldsymbol{D}_i + \sum_{j_2=i+1}^{\beta-1} \boldsymbol{A}_{i,j_2}\boldsymbol{D}_{j_2} \\
&= \boldsymbol{A}_{i,0}\boldsymbol{D}_0 + \sum_{j_1=1}^{i-1} \boldsymbol{A}_{i,j_1}\boldsymbol{D}_{j_1} + \boldsymbol{A}_{0,0}\boldsymbol{D}_i + \sum_{j_2=i+1}^{\beta-1} \boldsymbol{A}_{i,j_2}\boldsymbol{D}_{j_2} \\
&= \begin{cases} P_{0,i} + P_{i,0} + \sum_{\substack{m=1 \\ m \ne i}}^{\beta-1} \sum_{n=0}^{k-1} A_{i,km+n} D_{m,n}, & \text{if } i < \beta \\ P_{0,i} + \sum_{m=1}^{\beta-1} \sum_{n=0}^{k-1} A_{i,km+n} D_{m,n}, & \text{if } i \ge \beta. \end{cases}
\end{aligned} \tag{6}
$$

Algorithm 2 summarizes the process of Cauchy-Merge-RS, which merges β narrow $RS(k, r)$ stripes into one wide $RS(\beta k, r)$ stripe. It first addresses the calculation of post-merging parity blocks from the sender's perspective, determining which blocks to transmit to which nodes to complete the computation of parity blocks (lines 2-11), as opposed to the receiver's perspective described in equations (5) and (6). For the original 0-th stripe, it only needs to transmit its own j-th parity block (if it exists) to the node corresponding to the j-th post-merging parity block (lines 2-4). For other i-th original stripes ($i > 0$), there are three parts to transmit (lines 5-11): the first part involves sending the 0-th parity block to the node corresponding to the i-th post-merging parity block (line 6); the second part involves sending the i-th parity block to the node corresponding to the 0-th post-merging parity block (line 7); the third part entails transmitting its own data block to the nodes corresponding to all post-merging parity blocks except the 0-th and i-th (lines 8-10). After transmitting these blocks, the post-merging stripe can compute the new parity blocks based on equations (5) and (6) (lines 12-14).

Algorithm 2 Cauchy-Merge-RS

```
1:  procedure MERGE_TO_RS(D_{i,j}, P_{i,j}, k, r, β)
        // D_{i,j}: the j-th data block of the i-th original stripe
        // P_{i,j}: the j-th parity block of the i-th original stripe
        // P_i^0: the i-th parity block of the post-merging stripe
2:      for j = 0 to r − 1 do    // i = 0
3:          Send P_{0,j} (if exists) to P_j^0
4:      end for
5:      for i = 1 to β − 1 do    // i > 0
6:          Send P_{i,0} to P_i^0
7:          Send P_{i,i} (if exists) to P_0^0
8:          for j = 1 to r − 1, j ≠ i do
9:              Send D_{j,0} ... D_{j,k−1} to P_j^0
10:         end for
11:     end for
12:     for i = 0 to r − 1 do
13:         Calculate P_i^0 by (5) and (6)
14:     end for
15:     return {P_i^0}
16: end procedure
```

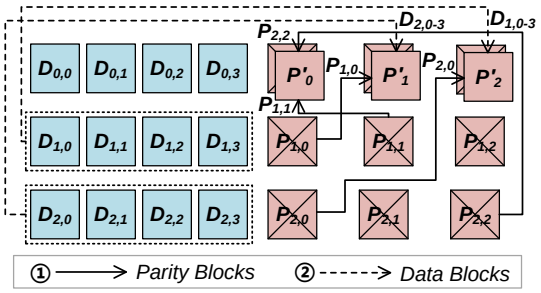


Fig. 5. Merging three $RS(4, 3)$ stripes into an $RS(12, 3)$ stripe by Cauchy-Merge-RS. $P_{1,1}$ and $P_{2,2}$ are sent to $P_{0,0}$. New $P_0^0 = P_{0,0} + P_{1,1} + P_{2,2}$. $P_{1,0}$ and $D_{2,0} \sim D_{2,3}$ are sent to $P_{0,1}$. New $P_1^0 = P_{0,1} + P_{1,0} + \sum_{i=0}^{3} A_{1,i+8} D_{2,i}$. $P_{2,0}$ and $D_{1,0} \sim D_{1,3}$ are sent to $P_{0,2}$. New $P_2^0 = P_{0,2} + P_{2,0} + \sum_{i=0}^{3} A_{2,i+4} D_{1,i}$.

Figure 5 illustrates the process of merging three $RS(4, 3)$ stripes into an $RS(12, 3)$ stripe. According to equations (5) and (6), $P_0^0 = P_{0,0} + P_{1,1} + P_{2,2}$, $P_1^0 = P_{0,1} + P_{1,0} + \sum_{i=0}^{3} A_{1,i+8} D_{2,i}$, and $P_2^0 = P_{0,2} + P_{2,0} + \sum_{i=0}^{3} A_{2,i+4} D_{1,i}$. The block $P_{0,0}$ receives $P_{1,1}$ and $P_{2,2}$. The block $P_{0,1}$ receives $P_{1,0}$ and $D_{2,0} \sim D_{2,3}$. The block $P_{0,2}$ receives $P_{2,0}$ and $D_{1,0} \sim D_{1,3}$. Then, all post-merging parity blocks P_0^0, P_1^0 and

P_2^0 can be calculated.

2) Cauchy-Merge-LRC: The recovery overhead of RS codes is notably substantial, especially in scenarios involving wide stripes. RS decoding requires the retrieval of k surviving blocks, which can lead to a decline in system performance. To alleviate the recovery overhead associated with such wide-stripe erasure codes, the introduction of local parity blocks is a common practice [2], [14], [15].

However, the data placement strategy is crucial for LRC, as it is essential to fully leverage the recovery performance advantages offered by local parity blocks. It is imperative to ensure that blocks from a local group are distributed across different maintenance zones to mitigate performance degradation due to cluster maintenance, which is both common and foreseeable [2]. By doing so, when a degraded read request is triggered during cluster maintenance, it only requires reading other blocks within the local group to decode the data block. Otherwise, this degraded read might trigger global RS decoding, leading to significant performance degradation and a decline in service quality. Hence, careful attention must be paid to the placement of data and parity blocks during the stripe merging process of LRC.

We propose Cauchy-Merge-LRC, which merges narrow RS stripes into wide LRC stripes while considering the block placement strategy through the merging process. It consists of three main steps: Local Parity Block Generation, Global Parity Block Generation, and Partial Data Migration.

Local Parity Block Generation. Cauchy-Merge-LRC initially adopts the first parity blocks of the existing stripes as the local parity blocks for the post-merging LRC stripe during the Local Parity Block Generation step. These local parity blocks are then replicated on a dedicated node where none of the data blocks from the local group are stored, ensuring they are not within the same maintenance zone.

Global Parity Block Generation. Subsequently, Cauchy-Merge-LRC executes the Global Parity Block Generation step, which is the same as the procedure of Cauchy-Merge-RS, introduced in Section III-D1.

Partial Data Migration. Finally, we employ Partial Data Migration to ensure that each maintenance zone contains at most one block from a local group. This ensures that all degraded read requests caused by foreseeable cluster maintenance can be decoded within their respective local groups. Partial Data Migration examines all local groups to check whether the blocks are distributed across different maintenance zones and migrate blocks of a local group that are in the same maintenance zone to other unused maintenance zones.

Algorithm 3 summarizes the process of Cauchy-Merge-LRC for merging β narrow $RS(k,r)$ stripes into one wide $LRC(\beta k, \beta, r)$ stripe. Initially, it replicates existing parity blocks to unused maintenance zones to serve as local parity blocks for the post-merging stripe. Subsequently, it utilizes Cauchy-Merge-RS to compute the global parity blocks for the post-merging stripe. Finally, it examines the placement of data blocks and ensures, through partial data migration, that each maintenance zone stores at most one data or local parity block from a local group.

Algorithm 3 Cauchy-Merge-LRC

procedure MERGE_TO_LRC($D_{i,j}, P_{i,j}, k, r, \beta$)
  // $D_{i,j}$: the j-th data block of the i-th original stripe
  // $P_{i,j}$: the j-th parity block of the i-th original stripe
  // L_i^0: the i-th local parity block of the post-merging stripe
  // P_i^0: the i-th global parity block of the post-merging stripe
  // Generate local parity blocks
  for i = 0 to β 1 **do**
    Send $P_{i,0}$ to an unused maintenance zone as L_i^0
  end for
  // Generate global parity blocks
  fP_i^0g $Merge\_to\_RS(D_{i,j}, P_{i,j}, k, r, \beta)$
  // Partial data migration
  for i = 0 to β 1 **do**
    while Two blocks are stored in one maintenance zone **do**
      Send one block to an unused maintenance zone
    end while
  end for
  return fL_i^0g, fP_i^0g
end procedure

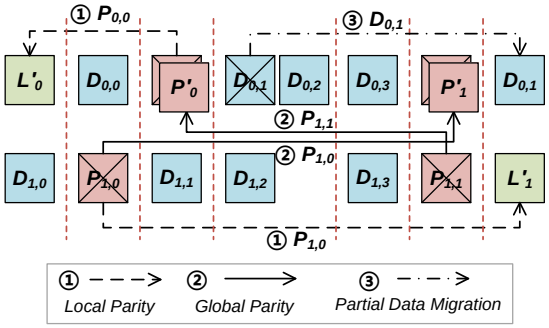


Fig. 6. Merging two $RS(4,2)$ stripes into an $LRC(8,2,2)$ stripe by Cauchy-Merge-LRC. Different maintenance zones are marked by red dotted lines.

An example of merging two $RS(4,2)$ stripes into an $LRC(8,2,2)$ stripe is illustrated in Figure 6. First, $P_{0,0}$ and $P_{1,0}$ are duplicated to serve as the new local parity blocks. Then, $P_{1,0}$ and $P_{1,1}$ are transmitted to $P_{0,1}$ and $P_{0,0}$, respectively, to complete the generation of global parity blocks, as in Cauchy-Merge-RS. Finally, partial data migration identifies that $D_{0,1}$ and $D_{0,2}$ are located within the same maintenance zone, and consequently, $D_{0,1}$is migrated to another zone.

E. Properties of Cauchy-Merge

In this subsection, we analyze the properties of Cauchy-Merge to demonstrate its effectiveness. First, we examine the proportion of reused parity blocks during the stripe merging process. We focus on the generation of RS parity blocks, since this process is required in both Cauchy-Merge-RS and Cauchy-Merge-LRC, and constitutes all the computational overhead during the stripe merging process. We can prove that when merging β $RS(k,r)$ stripes and computing RS parity blocks,

TABLE III
THE REUSE RATE OF EXISTING PARITY BLOCKS (CALCULATION SAVINGS RATE) DURING THE COMPUTATION OF RS PARITY BLOCKS IN THE STRIPE MERGING PROCESS FOR DIFFERENT r AND β.

r \ β	2	3	4
2	100%	–	–
3	83.33%	77.78%	58.33%
4	75%	66.67%	62.5%

out of the original βr parity blocks, $r \quad 2 + 2\min\{\beta, r\}$ of them are reused.

Theorem 4. *Cauchy-Merge reuses a total of* $r \quad 2 + 2\min\{\beta, r\}$ *out of* βr *existing parity blocks when merging* β $RS(k, r)$ *stripes into a single* $RS(\beta k, r)$ *stripe.*

Proof. **Case 1:** $\beta \quad r$**.** In the computation of P_0^0, β parity blocks associated with the first β coefficient vectors constituting the first row of the post-merging encoding matrix are reused, as directly derived from (5). In the computation of P_i^0 $(1 \quad i \quad \beta \quad 1)$, as formulated in (6), two parity blocks associated with the coefficient vectors comprising the first column and main diagonal of the post-merging encoding matrix are reused. Meanwhile, in the computation of P_i^0 $(\beta \quad i \quad r \quad 1)$, one parity block associated with the coefficient vectors comprising the first column of the post-merging encoding matrix is reused. Hence, a total of $\beta + 2(\beta \quad 1) + (r \quad \beta) = 2\beta + r \quad 2$ original parity blocks are reused when computing the parity blocks of the post-merging stripe.

Case 2: $\beta > r$**.** In the computation of P_0^0, r parity blocks associated with all coefficient vectors constituting the first row of the post-merging encoding matrix are reused, which directly derived from (5). In the computation of P_i^0 $(1 \quad i \quad r \quad 1)$, as formulated in (6), two parity blocks associated with the coefficient vectors comprising the first column and main diagonal of the post-merging encoding matrix are reused. Hence, a total of $r + 2(r \quad 1) = 3r \quad 2$ original parity blocks are reused when computing the parity blocks of the post-merging stripe.

By considering both case 1 and case 2, a total of $r \quad 2 + 2\min\{\beta, r\}$ original parity blocks are reused when computing the parity blocks of the post-merging stripe. □

Theorem 4 demonstrates the number of parity blocks that can be reused during the merging process. Typically, a stripe contains 2, 3, or 4 parity blocks. The reuse rate (or calculation savings rate) during the computation of RS parity blocks in the stripe merging process for different values of r and β is presented in Table III.

Additionally, Cauchy-Merge imposes no restrictions on the selection of stripes for merging, as this choice does not affect the reutilization of parity blocks. The method does not rely on specific stripe combinations and can merge any set of r stripes with equal efficiency in terms of reusing original parity blocks.

IV. PERFORMANCE EVALUATION

In this section, we conduct a series of mathematical analyses, simulations, and testbed experiments to demonstrate the effectiveness of Cauchy-Merge under various parameters.

A. Evaluation Methodology

To evaluate the effectiveness of Cauchy-Merge, we compare it with two advanced redundancy transition methods: NCScale [24], [25] and DISMerge [30]. However, StripeMerge [17], Zebra [26], [27] and Convertible Codes [28], [29] are excluded from our evaluation in Table II because they have specific deployment requirements and can not adapt to existing storage systems, as analyzed in Section II. For Cauchy-Merge, we have implemented two approaches: Cauchy-Merge-RS, merging into RS stripes, and Cauchy-Merge-LRC, merging into LRC stripes (denoted as CM-RS and CM-LRC, respectively, for simplicity in figures). We evaluate the performance of these methods when merging β $RS(k, r)$ stripes into one $RS(\beta k, r)$ stripe for NCScale and Cauchy-Merge-RS, or into one $LRC(\beta k, \beta, r)$ stripe for DISMerge and Cauchy-Merge-LRC.

1) Metrics for Simulations and Numerical Analyses: We use the **Merging Transmission Cost**, **Algorithm Running Time** as the metrics for simulations and **Merging Computation Cost** as the metrics for numerical analyses.

Merging Transmission Cost is defined as the total number of blocks transmitted during the merging process.
Merging Computation Cost is defined as the total number of XOR operations and multiplications over Galois field during the merging process.
Algorithm Running Time is defined as the total time taken to generate the stripe merging schemes.

2) Metrics for Testbed Evaluation: We use **Merging Time** and **Degraded Read Latency** as the metrics for testbed evaluation.

Merging Time is measured by the average time taken to merge a fixed number of stripes, including transmission time and computing time.
Degraded Read Latency is measured by the average latency experienced during degraded reads when the nodes of one maintenance zone are offline.

3) Simulator Environment: We implement a simulator for Cauchy-Merge to evaluate the merging transmission cost and algorithm running time. NCScale and DISMerge are also implemented in the simulator for comparison of the merging transmission cost. The simulator runs on a physical server equipped with an Intel Xeon Processor E5-2620, 192GB of memory, and 8TB of disks. The default setup involves merging 6000 stripes, with the blocks distributed randomly across 40 nodes.

4) Testbed Environment: We conduct our experiments on Amazon EC2 in the US-East-2 (Ohio) region, utilizing 40 m5.large instances as storage nodes and an additional instance as the scheduling node. These instances are interconnected with a 10 Gbps network. The scheduling node manages the

entire stripe merging task, while storage nodes handle the assigned transmitting and computing tasks. We implement the parity block computation of Cauchy-Merge using Intel ISA-L [44] and develop a test program to measure the merging time, including transmission and computation time, along with degraded read latency. All experiment results are averaged over 5 rounds.

We randomly distribute all blocks of 6000 stripes across 40 storage nodes. To simulate maintenance tasks in the cluster, we evenly distribute nodes into different maintenance zones and remove the nodes from one maintenance zone when evaluating degraded read latency. By default, the block size is fixed at 64MB, and the number of maintenance zones is 20.

B. Simulation and Numerical Results

In this subsection, we evaluate the merging transmission cost, merging computation cost, and algorithm running time of various methods during the stripe merging process under different parameters. The merging transmission cost and algorithm time are evaluated in simulations, while the merging computation cost is analyzed mathematically.

1) Merging transmission cost versus (k, r): Figure 7 illustrates the merging transmission cost of different methods under various (k, r) parameters. The results demonstrate that Cauchy-Merge-RS/LRC effectively reduces the number of transmitted blocks during the merging process. Specifically, when $\beta = 2$, Cauchy-Merge-RS/LRC achieves a reduction of 47.18% to 93.12% compared to DISMerge. When $\beta = 3$, the reduction ranges from 48.48% to 66.08%. For $\beta = 4$, the reduction varies between 40.51% and 48.07%.

As β increases, Cauchy-Merge-RS/LRC shows a decreasing trend in the reduction ratio of merging transmission cost. This behavior is attributed to the fact that the parity block reuse ratio in Cauchy-Merge is given by $\frac{r \quad 2+2\min\{\beta, r\}}{\beta r}$. With increasing β, the reuse ratio diminishes, as shown in Table III.

2) Merging computation cost versus (k, r): We analyze the merging computation cost mathematically and present the numerical results of different methods under various (k, r) parameters in Figures 8 and 9.

XOR Operations. When merging β $RS(k, r)$ stripes, the number of XOR operations ($\#_{\oplus}$) for Cauchy-Merge is

$$\#_{\oplus} = \begin{cases} k(\beta \quad 1)(\beta \quad 2) + k(\beta \quad 1)(r \quad \beta) + 2(\beta \quad 1), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \beta \quad r, \\ k(r \quad 1)(\beta \quad 2) + k(\beta \quad r) + 2(r \quad 1), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \beta > r. \end{cases}$$

The number of XOR operations for DISMerge and NCScale is $\#_{\oplus} = k(\beta \quad 1)r$. Figure 8 demonstrates that Cauchy-Merge effectively reduces the XOR computations involved in the merging process compared to DISMerge and NCScale. For example, when $\beta = 2$, Cauchy-Merge shows a reduction of 45.83% to 83.33%.

TABLE IV
TIME BREAKDOWN OF CAUCHY-MERGE FOR MERGING 6000 STRIPES

(k, r, β)	RS Transfer (s)	RS Compute (s)	LRC Transfer (s)
$(8, 2, 2)$	110.18	2.18	23.94
$(10, 3, 3)$	533.90	9.43	33.21
$(12, 4, 4)$	1009.94	41.08	81.65

GF Multiplications[3]**.** When merging β $RS(k, r)$ stripes, the number of GF Multiplications ($\#_{\otimes}$) for Cauchy-Merge is

$$\#_{\otimes} = \begin{cases} k(\beta \quad 1)(\beta \quad 2) + k(\beta \quad 1)(r \quad \beta), & \text{if } \beta \quad r, \\ k(r \quad 1)(\beta \quad 2) + k(\beta \quad r), & \text{if } \beta > r. \end{cases}$$

The number of GF Multiplications for DISMerge and NCScale is $\#_{\otimes} = k(\beta \quad 1)r$. Figure 9 demonstrates that Cauchy-Merge also reduces GF Multiplications in the merging process. For example, when $\beta = 2$, Cauchy-Merge reduces GF Multiplications by 50% to 100%.

The results show that Cauchy-Merge effectively reduces both XOR computations and GF Multiplications compared to DISMerge and NCScale. However, the reduction ratio decreases as β increases, similar to the merging transmission cost, due to the declining parity block reuse ratio in Table III.

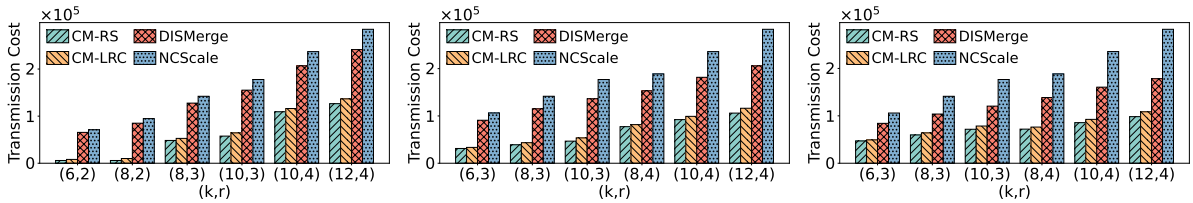
3) Running time versus the number of stripes: Figure 10 illustrates the algorithm's running time for Cauchy-Merge-RS/LRC under a varying number of stripes. The results indicate that Cauchy-Merge-LRC is consistently slower than Cauchy-Merge-RS. This is because Cauchy-Merge-LRC incurs additional running time for partial data migration compared to Cauchy-Merge-RS. The overhead caused by running Cauchy-Merge-RS/LRC to generate the merging schemes is negligible when compared to the merging time, as evaluated in the following subsection.

C. Testbed Evaluation

In this subsection, we measure the breakdown of stripe merging time for Cauchy-Merge. Next, we examine the impact of different parameters, including the coding parameters (k, r), the merging number β, block size, and the number of maintenance zones, on the stripe merging time of different methods. Lastly, we measure the degraded read latency, thereby demonstrating the effectiveness of Partial Data Migration.

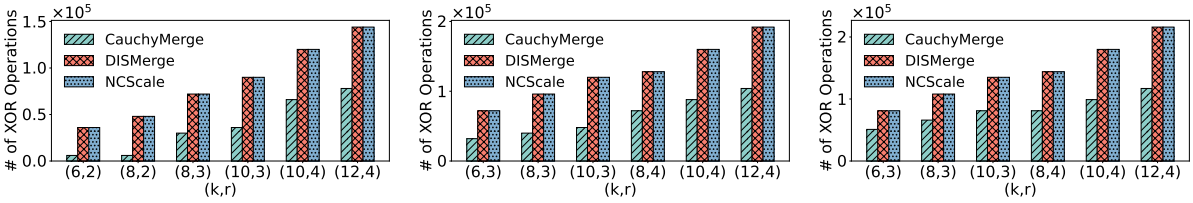
1) Time breakdown: We measured the breakdown of Cauchy-Merge stripe merging time, demonstrating the additional overhead introduced by Cauchy-Merge-LRC compared to Cauchy-Merge-RS. We categorize the Cauchy-Merge time into three parts: RS Transfer, RS Compute, and LRC Transfer. RS Transfer refers to the time taken to transmit data blocks and parity blocks for computing post-merging RS parity blocks; RS Compute is the time taken to compute post-merging RS parity blocks; and LRC Transfer is the time taken to generate local parity blocks and perform partial data migration. The total time for Cauchy-Merge-RS is the sum of the first two parts, while the total time for Cauchy-Merge-LRC is the sum

[3]GF Multiplications is an abbreviation for multiplications over the Galois field.



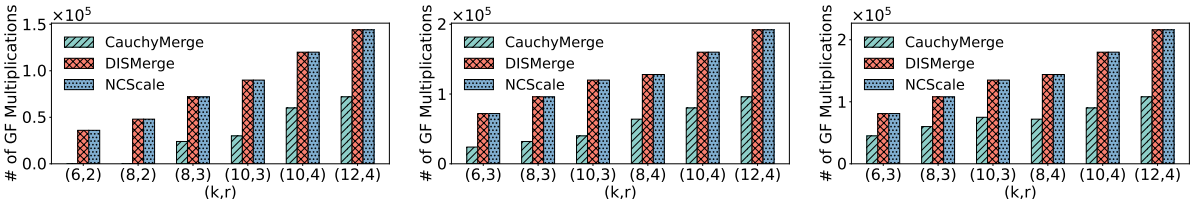
(a) $\beta = 2$. (b) $\beta = 3$. (c) $\beta = 4$.

Fig. 7. Comparisons among different redundancy transition approaches in terms of merging transmission cost under various (k, r) parameters.



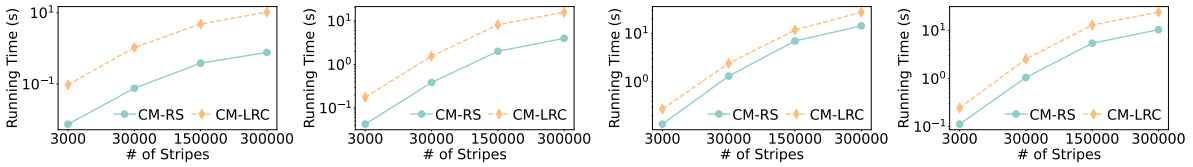
(a) $\beta = 2$. (b) $\beta = 3$. (c) $\beta = 4$.

Fig. 8. Comparisons among different redundancy transition approaches in terms of the number of XOR operations under various (k, r) parameters.



(a) $\beta = 2$. (b) $\beta = 3$. (c) $\beta = 4$.

Fig. 9. Comparisons among different redundancy transition approaches in terms of the number of GF Multiplications under various (k, r) parameters.



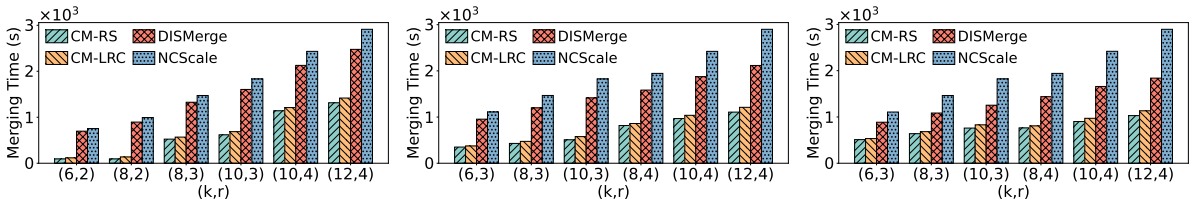
(a) $k = 6, r = 2, \beta = 2$. (b) $k = 8, r = 3, \beta = 3$. (c) $k = 10, r = 4, \beta = 2$. (d) $k = 10, r = 4, \beta = 3$.

Fig. 10. Comparisons between Cauchy-Merge-RS and Cauchy-Merge-LRC in terms of algorithm running time under various numbers of stripes.

of all three parts. We tested the breakdown of merging time for different (k, r), and β, and Table IV presents the breakdown of time. Compared to Cauchy-Merge-RS, the overhead of Cauchy-Merge-LRC is 21.74%, 6.22%, and 8.08%, respectively, for different (k, r), and β.

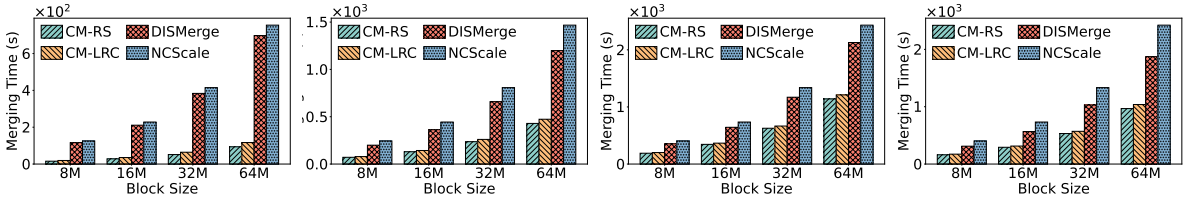
2) Merging time versus (k, r): We conducted experiments to measure the merging time of different (k, r), and β. The experimental results, depicted in Figure 11, reveal that, compared to DISMerge, Cauchy-Merge-RS/LRC reduces the total merging time by 41.18% to 89.53% and 38.41% to 84.77%, respectively. The reduction is attributed to the efficient reuse of existing parity blocks. As β increases, the reduction in merging time decreases. This trend is due to the diminishing parity block reuse ratio as β increases. Additionally, the extra overhead of Cauchy-Merge-LRC increases with the growth of k. For example, with $\beta = r = 3$, the overhead rises from 3.2% to 11.52% as k increases from 6 to 10. This is due to the higher likelihood of storing multiple blocks from the same local group within a maintenance zone as k increases, leading to more blocks being transferred during partial data migration.

3) Merging time versus block size: We measured the total merging time for different block sizes ranging from 8MB to 64MB, and the results are illustrated in Figure 12. It is demonstrated that Cauchy-Merge consistently provides similar improvements across varying block sizes. For example, Cauchy-Merge achieves reductions in merging time ranging from 56.70% to 61.69% compared to DISMerge when $k = 8, r = 3, \beta = 3$ across different block sizes. This suggests the



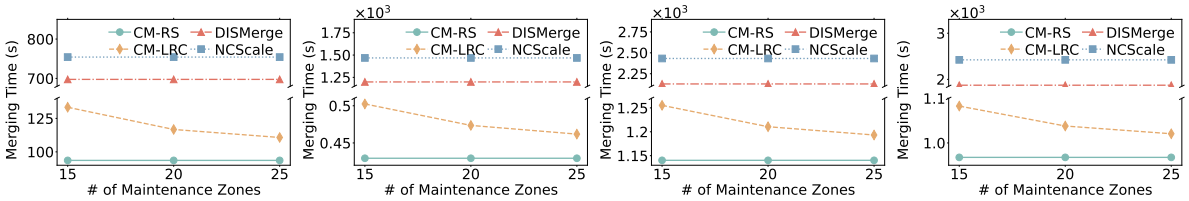
(a) $\beta = 2$. (b) $\beta = 3$. (c) $\beta = 4$.

Fig. 11. Comparisons among different redundancy transition approaches in terms of merging time under various (k, r) parameters.



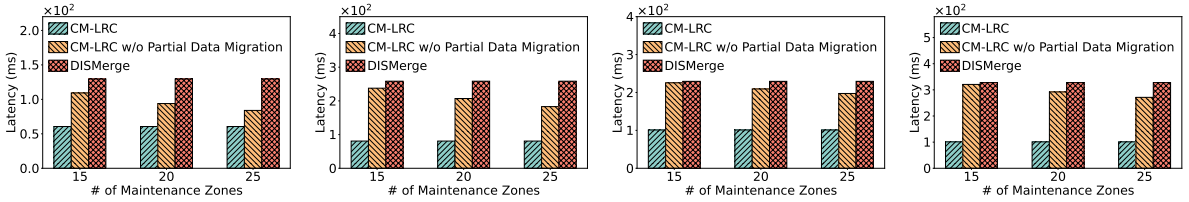
(a) $k = 6, r = 2, \beta = 2$. (b) $k = 8, r = 3, \beta = 3$. (c) $k = 10, r = 4, \beta = 2$. (d) $k = 10, r = 4, \beta = 3$.

Fig. 12. Comparisons among different redundancy transition approaches in terms of merging time under various block sizes.



(a) $k = 6, r = 2, \beta = 2$. (b) $k = 8, r = 3, \beta = 3$. (c) $k = 10, r = 4, \beta = 2$. (d) $k = 10, r = 4, \beta = 3$.

Fig. 13. Comparisons among different redundancy transition approaches in terms of merging time under various numbers of maintenance zones.



(a) $k = 6, r = 2, \beta = 2$. (b) $k = 8, r = 3, \beta = 3$. (c) $k = 10, r = 4, \beta = 2$. (d) $k = 10, r = 4, \beta = 3$.

Fig. 14. Comparisons among different redundancy transition approaches in terms of degraded read latency under various numbers of maintenance zones.

effectiveness of Cauchy-Merge regardless of block sizes.

4) Merging time versus the number of maintenance zones: We evaluated the merging time with different numbers of maintenance zones. Figure 13 presents the experimental results. The merging time for Cauchy-Merge-RS, DISMerge, and NCScale remains constant regardless of the number of maintenance zones. Cauchy-Merge-LRC exhibits a reduction in merging time as the number of maintenance zones increases. For instance, when $k = 6$, $r = 2$, and $\beta = 2$, Cauchy-Merge-LRC shows reductions of 10.23% and 12.60% in merging time with 20 and 25 maintenance zones, respectively, compared to 15 maintenance zones. This is because of the decreased probability of a maintenance zone storing multiple blocks from the same local group, resulting in fewer data blocks being transferred in partial data migration.

5) Degraded read latency versus the number of maintenance zones: We conducted experiments to measure the degraded read latency of merged LRC stripes with an unavailable maintenance zone under different numbers of maintenance zones. The results, as shown in Figure 14, reveal that Cauchy-Merge-LRC exhibits the lowest degraded read latency, and the number of maintenance zones does not significantly affect it. Cauchy-Merge-LRC without Partial Data Migration's degraded read latency falls between the other two methods and decreases with an increasing number of maintenance zones. DISMerge exhibits the highest degraded read latency due to its restrictions on the data layout, centralizing the placement of data blocks within a local group. Consequently, it consistently

requires global recovery to decode the offline block during maintenance tasks. Compared to Cauchy-Merge-LRC without Partial Data Migration, Cauchy-Merge-LRC significantly reduces degraded read latency, especially when k and β are large, since Cauchy-Merge-LRC only needs to retrieve k blocks while other methods need to retrieve all βk blocks.

V. Conclusion

In this paper, we propose Cauchy-Merge, a redundancy transition method that merges narrow RS stripes into wide RS or LRC stripes, based on the merge-friendly Cauchy matrix. It reuses original parity blocks in the merging process, thereby reducing computation and network transmission overhead. Additionally, Cauchy-Merge employs partial data migration to lower degraded read latency during cluster maintenance. We evaluate the performance of Cauchy-Merge through numerical analyses and testbed experiments. The results demonstrate that Cauchy-Merge can reduce stripe merging time by up to 89.53% and degraded read latency by up to 69.17%.

VI. Acknowledgements

We thank anonymous reviewers for their insightful comments and suggestions. This work is partially sponsored by the Natural Science Foundation of China (NSFC) (No.61972246, No.61932014, and No.62232011).

References

[1] P. Narayanan, S. Samal, and S. Nanniyur, "Yahoo cloud object store-object storage at exabyte scale," 2017.

[2] S. Kadekodi, S. Silas, D. Clausen, and A. Merchant, "Practical design considerations for wide locally recoverable codes (LRCs)," in *Proc. of FAST*, 2023.

[3] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. of OSDI*, 2010.

[4] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *International Workshop on Peer-to-Peer Systems*, pp. 328–337, Springer, 2002.

[5] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[6] 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 OSDI*, 2014.

[7] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. of OSDI*, 2006.

[8] 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 MSST*, 2015.

[9] A. Fikes, "Storage architecture and challenges," *Talk at the Google Faculty Summit*, vol. 535, pp. 1–25, 2010.

[10] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. of the VLDB Endowment*, 2013.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proc. of SOSP*, 2003.

[12] 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 ATC*, 2012.

[13] "Hadoop distributed file system (hdfs)." http://hadoop.apache.org.

[14] 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 FAST*, 2021.

[15] G. Yang, H. Xue, Y. Gu, C. Wu, J. Li, M. Guo, S. Li, X. Xie, Y. Dong, and Y. Zhao, "XHR-code: An efficient wide stripe erasure code to reduce cross-rack overhead in cloud storage systems," in *Proc. of SRDS*, 2022.

[16] "VastData." https://vastdata.com/blog/providing-resilience-efficiently-part-ii.

[17] 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 ICDCS*, 2021.

[18] Q. Yu, L. Wang, Y. Hu, Y. Xu, D. Feng, J. Fu, X. Zhu, Z. Yao, and W. Wei, "Boosting multi-block repair in cloud storage systems with wide-stripe erasure coding," in *Proc. of IPDPS*, 2023.

[19] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[20] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "Ec-cache:load-balanced,low-latency cluster caching with online erasure coding," in *Proc. of OSDI*, 2016.

[21] 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 Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–30, 2017.

[22] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron, "Pelican: A building block for exascale cold data storage," in *Proc. of OSDI*, 2014.

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

[24] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou, "Toward optimal storage scaling via network coding: From theory to practice," in *Proc. of INFOCOM*, 2018.

[25] Y. Hu, X. Zhang, P. P. Lee, and P. Zhou, "NCScale: toward optimal storage scaling via network coding," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 271–284, 2021.

[26] J. Li and B. Li, "Zebra: Demand-aware erasure coding for distributed storage systems," in *Proc. of IWQoS*, 2016.

[27] J. Li and B. Li, "Demand-aware erasure coding for distributed storage systems," *IEEE Transactions on Cloud Computing*, vol. 9, no. 2, pp. 532–545, 2018.

[28] F. Maturana, V. C. Mukka, and K. Rashmi, "Access-optimal linear mds convertible codes for all parameters," in *Proc. of ISIT*, 2020.

[29] F. Maturana and K. Rashmi, "Convertible codes: Enabling efficient conversion of coded data in distributed storage," *IEEE Transactions on Information Theory*, vol. 68, no. 7, pp. 4392–4407, 2022.

[30] S. Wu, Q. Du, P. P. Lee, Y. Li, and Y. Xu, "Optimal data placement for stripe merging in locally repairable codes," in *Proc. of INFOCOM*, 2022.

[31] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel simd instructions.," in *Proc. of FAST*, 2013.

[32] J. Blomer, "An XOR-based erasure-resilient coding scheme," *Technical report at ICSI*, 1995.

[33] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software: Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[34] "corrupted fragment on decode." https://github.com/intel/isa-l/issues/10.

[35] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Transactions on Information theory*, vol. 58, no. 11, pp. 6925–6934, 2012.

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

[37] S. Wu, Y. Xu, Y. Li, and Z. Yang, "I/O-efficient scaling schemes for distributed storage systems with CRS codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2639–2652, 2015.

[38] S. Wu, Z. Shen, and P. P. Lee, "On the optimal repair-scaling trade-off in locally repairable codes," in *Proc. of INFOCOM*, 2020.

[39] Z. Lin, H. Guo, C. Wu, J. Li, G. Xue, and M. Guo, "Rack-Scaling: An efficient rack-based redistribution method to accelerate the scaling of cloud disk arrays," in *Proc. of IPDPS*, 2021.

[40] W. Zheng and G. Zhang, "FastScale: Accelerate RAID scaling by minimizing data migration," in *Proc. of FAST*, 2011.

[41] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie, "Scale-RS: An efficient scaling scheme for RS-coded storage clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1704–1717, 2014.

[42] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," *Software-Practice and Experience*, vol. 35, no. 2, p. 189, 2005.

[43] J. J. Rotman, *Advanced modern algebra*, vol. 114. American Mathematical Soc., 2010.

[44] "Intel ISA-L." https://github.com/intel/isa-l.