Locally Repairable Convertible Codes: Erasure Codes for Efficient Repair and Conversion

Francisco Maturana and K. V. Rashmi
Carnegie Mellon University
Computer Science Department
{fmaturan,rvinayak}@cs.cmu.edu

2023 IEEE International Symposium on Information Theory (ISIT) | 978-1-6654-7554-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISIT54713.2023.10206604

***Abstract*—Erasure codes are typically used in distributed storage systems in order to protect against failures and unavailabilities with low storage overhead. An important disadvantage of classic erasure codes (such as Reed-Solomon codes) is the high cost of repairing failures. Locally repairable codes (LRCs) reduce the repair cost at the cost of higher storage overhead. In practice, the parameters of LRCs are chosen based on several factors, such as failure rates, workloads, and budget constraints. However, encoded data is stored for long periods of time, and during that time these factors can vary, and thus the ideal parameters can change. The process of changing the code parameters on encoded data is called *code conversion*. The default approach to code conversion is to read all data, re-encode it, and write it back, which can be prohibitively expensive. To address this problem, we propose a new construction technique for designing LRCs that can perform code conversion at a lower cost than the default approach. We apply this technique to design codes for several code conversion scenarios which are of practical interest.**

I. Introduction

Nowadays, large-scale distributed storage systems store petabytes of data across thousands of disks. In such systems, failures and unavailabilities are common, and thus erasure coding has become essential to protect data. Several factors affect the choice of code, such as failure rates [2], workloads [3], and budget constraints. However, stored data lives for a long time, over which these factors vary, prompting one to change the code. The process of converting already-encoded data into its encoding under a different code is known as *code conversion* [4]. The default approach to code conversion is to read all data, encode it under the new code, and write it back. Yet, in most cases it is possible to convert data more efficiently. The key idea behind efficient code conversion is to design the code in such a way that existing code symbols can be used in creating the new code symbols without needing access to all the message symbols. Existing works [4]–[7] on code conversion focus on conversion of *maximum-distance-separable* (MDS) codes from length n^I and dimension k^I to n^F and k^F, respectively.

Recently, there has been increased interest in *wide codes*, i.e. codes with large k, as they can achieve lower storage overhead given a target level of failure tolerance. One important drawback of wide codes is that even if a single node becomes unavailable, one must incur high resource-costs to repair it. For example, in the case of an MDS code, one must read k different nodes and reconstruct the original data to repair a node. In practice, repair operations are common enough that those costs negatively affect the performance of the cluster [8]–[10]. *Locally repairable codes* (LRCs) [11], [12] mitigate this problem by encoding data in a way that allows nodes to be repaired by accessing $r \ll k$ nodes only.

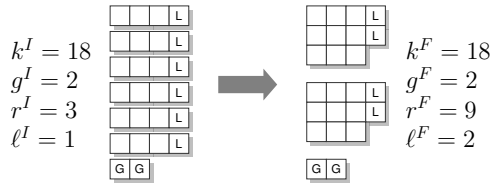

Fig. 1: Example of LRC conversion. Empty boxes are message symbols. L and G are local and global parities respectively.

To change the optimal repair properties over time, we study the code conversion problem for LRCs (see Fig. 1). This work focuses on codes with (r, ℓ) *data locality*, where k data nodes are divided into groups of size r, each with ℓ *local parities* that are a function of those r data nodes only. In addition, the code has g *global parities* which are a function of all k data nodes. We focus on LRCs with *optimal distance* [13]. As the cost of conversions, we consider *conversion bandwidth*, defined as the total amount of data communicated between nodes during conversion. Our contribution is a new construction technique for LRCs with efficient conversion. This technique can be applied to different types of conversions: in this paper we focus on *global conversions*, which only change k and g. Even though it is possible to do this type of conversion with existing constructions for MDS codes [6], [7], the constructions presented in this paper are able to further reduce conversion bandwidth by using both local and global parities. E.g., our construction achieves the conversion of (k, g, r, ℓ) from $(40, 2, 10, 2)$ to $(20, 3, 10, 2)$ with 17.89% less conversion bandwidth than existing constructions [7]. Proofs of this paper can be found in the extended version [1].

II. Background and related work

Let $[i] \coloneqq \{1, \ldots, i\}$. Let $\mathbf{v}_{[i,j]}$ denote entries i through j of a vector $\mathbf{v}$. A linear $[n, k, d, \alpha]$ *vector code* $\mathcal{C}$ over finite field $\mathbb{F}$ is a linear subspace of $\mathbb{F}^{\alpha n}$ of dimension αk. We refer to each coordinate (an element of $\mathbb{F}$) as a *symbol*. A codeword $\mathbf{c} \in \mathcal{C}$ is divided into n *nodes* $\mathbf{c}_i \coloneqq (c_{i,j})_{j=1}^{\alpha}$ $(i \in [n])$. The minimum distance of $\mathcal{C}$ is d, and it is defined as the minimum Hamming distance over $\mathbb{F}^{\alpha}$ between distinct codewords in $\mathcal{C}$. The code $\mathcal{C}$ is said to be MDS if $d = n - k + 1$ (in which case d is omitted). Data $\mathbf{m} \in \mathbb{F}^{\alpha k}$ is encoded via a $\alpha k \times \alpha n$ generator matrix $\mathbf{G}$ as $\mathbf{c} = \mathbf{mG}$. As an abuse of notation, we

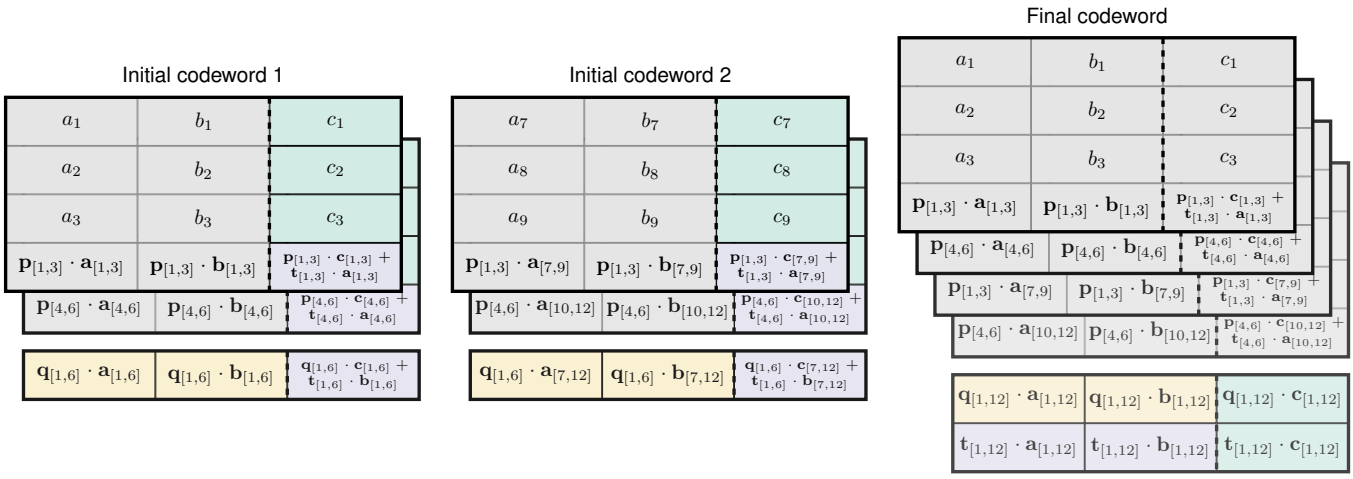

Fig. 2: (Ex. 1) Example of global merge conversion with parameters $k^I = 6$, $k^F = 12$, $g^I = 1$, $g^F = 2$, $r = 3$, $\ell = 1$.

denote the encoding of $\mathbf{m}$ under $\mathcal{C}$ as $\mathcal{C}(\mathbf{m})$. Code $\mathcal{C}$ is said to be *systematic* if $\mathbf{c}_i = \mathbf{m}_i := (m_{i,j})_{j=1}^{\alpha}$ for $i \in [k]$. The *support* of a code symbol is the set of data symbols corresponding to the non-zero indices in its generator matrix column; the support of a node is the union of the supports of its symbols.

A systematic code $\mathcal{C}$ is said to have (r, ℓ) *data locality* if for each data node $\mathbf{c}_i$ there exists a set of indices $\Gamma(i)$ containing i such that $|\Gamma(i)| \leq r + \ell$ and the restriction of $\mathcal{C}$ to $\Gamma(i)$ has minimum distance at least $\ell + 1$. Prior work [14] has shown that a code with (r, ℓ) data locality satisfies:

$$d \leq n - k + 1 - \ell\left(\left\lceil \frac{k}{r} \right\rceil - 1\right). \tag{1}$$

In this paper, we consider codes defined by parameters (k, g, r, ℓ), denoting a $[n, k, d, \alpha]$ vector code with $n := k + \lceil \frac{k\ell}{r} \rceil + g$, having (r, ℓ) data locality, and minimum distance d satisfying (1) with equality (i.e. optimal distance); we assume $r \mid k$ and treat α as a free variable. The constructions that we present are systematic codes with the following structure: the code has $m := \frac{k}{r}$ disjoint *local groups* each with r data nodes and ℓ local parity nodes, and g additional global parity nodes.

A. Systematic Vandermonde code

A systematic Vandermonde code is an $[n, k, d, \alpha{=}1]$ code defined by a generator matrix that is the concatenation of a $k \times k$ identity matrix and a $k \times (n - k)$ Vandermonde matrix with evaluation points $(\xi_i)_{i=1}^{n-k}$. If the field is large enough, choosing $\xi_i := \theta^{i-1}$, where θ is a primitive element, guarantees the MDS property (construction in [5, §V]). Column i of a Vandermonde matrix has the following property: consider a subvector and scale it by a power of ξ_i; this is equivalent to shifting the subvector by i entries. In particular, let $k := \lambda t$, and $\mathbf{h}^{(i)} := (\xi_i^{j-1})_{j=1}^{k}$ be the i-th encoding vector; then $\mathbf{h}^{(i)}_{[(m-1)t+1,mt]} = \xi_i^{(m-1)t}\mathbf{h}^{(i)}_{[1,t]}$ for all $i \in [n-k]$ and $m \in [\lambda]$.

B. Basic pyramid code [11]

One method for constructing a code with (r, ℓ) data locality and optimal distance is to start with a $[k + \ell + g, k, \alpha]$ MDS systematic linear code $\mathcal{C}$. Then, the generator matrix column of local parity $j \in [\ell]$ in local group $i \in [\frac{k}{r}]$ is constructed by taking the column of parity j in $\mathcal{C}$, and setting all the entries outside of rows $\{(j-1)r + 1, \ldots, jr\}$ to 0.

C. Piggybacking framework [15]

The *piggybacking framework* constructs an $[n, k, d, \alpha]$ vector code, by using α instances of an $[n, k, d]$ *base code* and adding special functions (called *piggybacks*) to certain symbols. I.e. symbol $c_{i,j}$ is the encoding of $(m_{i,j})_{i=1}^{k}$ under the base code, plus an specially designed piggyback. We refer to the non-piggyback part of a symbol as the *base*. A piggybacked code must have a *decoding order* for the instances of the base code given by a permutation $\sigma : [\alpha] \to [\alpha]$. To satisfy σ, the piggybacking functions used in instance i can only use data from instance j if $\sigma(i) > \sigma(j)$. Thus, when decoding by the order σ, the already-decoded instances are used to remove piggybacks, and the bodies are decoded with the base code. The utility of piggybacks is that they can store useful information which can be retrieved by subtracting the base.

D. Other related work

Codes designed to have small localities were first proposed in [11], [16], and a bound on the minimum distance of LRCs was proved in [13]. LRCs have been the subject of a wide range of works [9], [11], [12], [14], [17]–[30], which has proposed constructions, bounds on field size, and stronger recoverability properties than optimal distance (such as *maximum recoverability*).

The general problem of code conversion was introduced in [5]. Several works [4]–[7], [31], [32] have proposed constructions for code conversion. The results in these works consider two types of cost (access cost and conversion bandwidth) and focus on constructions and lower bounds for code conversions in which both the initial and final codes are MDS.

To the best of our knowledge, the idea of converting between different LRCs was first considered in [3] (called *up/downcoding*). Xia et al. [3] propose a conversion procedure for converting between two specific LRCs with different r parameter (and constant $\ell = 1, k, g$). This conversion procedure

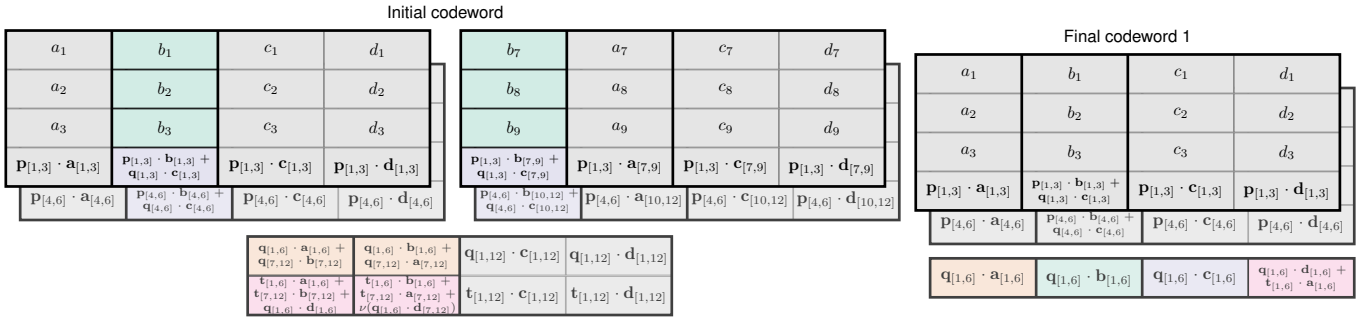

Fig. 3: (Ex. 2) Example of global split conversion with parameters $k^I = 12$, $k^F = 6$, $g^I = 2$, $g^F = 1$, $r = 3$, $\ell = 1$. In the code, $\nu := \xi_3^6$. For compactness, only one final codeword is shown; the other final codeword has the same encoding.

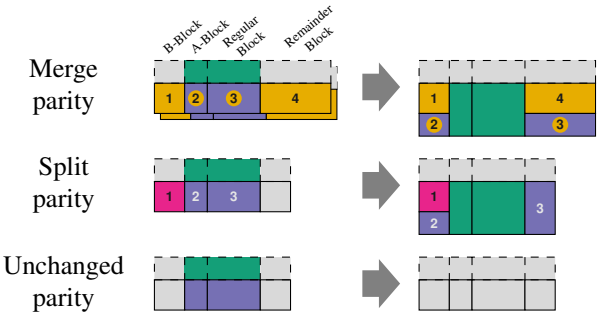

Fig. 4: Parity designs. Data is shown with a dashed box; parities with a solid box. Parities have one special block (B-block then A-block), a regular block, and a remainder block. Numbers indicate how initial parity symbols are used in the final parities.

can be viewed as reducing the number of nodes read during conversion (i.e. *access cost* [5]). In this paper, we focus on reducing conversion bandwidth instead. Minimization of conversion bandwidth for MDS codes was studied in [6], [7].

Recently, [33]–[36] studied LRC conversion (also called *scaling*) in a *clustered* setting, where code symbols are placed in clusters with the goal of reducing inter-cluster communication and satisfying some fault-tolerance constraints. The present paper is, to the best of our knowledge, the first one to focus on LRC conversion bandwidth (i.e. inter-node communication).

III. Conversion of LRCs

We study the LRC conversion from initial parameters (k^I, g^I, r^I, ℓ^I) to final parameters (k^F, g^F, r^F, ℓ^F). Conversion is carried by a *converter* which reads data from nodes, computes new symbols, and writes them. Cost is measured as *conversion bandwidth* [6]: the total amount of data communicated to and from the converter. We focus on reducing *read* conversion bandwidth (i.e. number of symbols read), since the number of symbols written is fixed. We denote read conversion bandwidth as γ, and normalize it as $\tilde{\gamma} := \gamma/\alpha$. Codes must satisfy:

- P1) the initial $[n^I, k^I, d^I, \alpha]$ code has (r^I, ℓ^I) data locality and optimal distance d^I,
- P2) the final $[n^F, k^F, d^F, \alpha]$ code has (r^F, ℓ^F) data locality and optimal distance d^F,
- P3) there is a conversion procedure from initial code to final code that is efficient in conversion bandwidth.

As in the code conversion literature [5], k is changed by considering $M := \text{lcm}(k^I, k^F)$ data nodes evenly divided among $\lambda^I := \frac{M}{k^I}$ codewords in the initial code and $\lambda^F := \frac{M}{k^F}$ codewords in the final code. Our approach is to construct a code with the piggybacking framework and using the piggybacks to reduce conversion bandwidth. As the base code, we use a basic pyramid code derived from a systematic Vandermonde code, which guarantees optimal distance. Our constructions combine a small number of techniques, which simplifies their description and analysis. We start by presenting two running examples used throughout the paper to illustrate our techniques. Details will be made clear as we explain our construction approach.

Example 1: Figure 2 shows an example of conversion from $(k^I{=}6, g^I{=}1, r^I{=}3, \ell^I{=}1)$ to $(k^F{=}12, g^F{=}2, r^F{=}3, \ell^F{=}1)$. We refer to this type of conversion as a global merge conversion. In the example, data corresponds to $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, and the encoding vectors of the base code are $\mathbf{p}$ (local parity) and $(\mathbf{q}, \mathbf{t})$ (global parities). The non-gray symbols in the initial codewords are read and used in generating the colored symbols in the final codeword (where colors denote techniques that will be described later). Conversion uses the property of Vandermonde codes that, e.g, $\mathbf{p}_{[4,6]} = \xi_1^3 \mathbf{p}_{[1,3]}$. The decoding order in the initial and final codes is $(1, 2, 3)$. By using this construction, conversion requires $\tilde{\gamma} = 7\frac{1}{3}$, compared to 12 (default approach) or 8 (MDS code in [6]). ▸

Example 2: Figure 3 shows conversion from $(k^I{=}12, g^I{=}2, r^I{=}3, \ell^I{=}1)$ to $(k^F{=}6, g^F{=}1, r^F{=}3, \ell^F{=}1)$. We refer to this type of conversion as a global split conversion. As in the previous example, data corresponds to $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$, encoding vectors are $\mathbf{p}$ (local parity) and $(\mathbf{q}, \mathbf{t})$ (global parities), and non-gray symbols in the initial codeword are read and used in generating the non-gray symbols in the final codewords. The decoding order is $(3, 4, 1, 2)$ in the initial code, and $(3, 1, 4, 2)$ in the final code. Conversion requires $\tilde{\gamma} = 5$, compared to 12 (default approach) or $5\frac{1}{3}$ (MDS code in [7]). ▸

These examples show that it is possible to reduce conversion bandwidth compared to other approaches. Now, we describe our general approach in detail.

A. Base code

Let $\tilde{k} := \max\{k^I, k^F\}$ and $\tilde{g} := \max\{\ell^I + g^I, \ell^F + g^F\}$. First, we construct a systematic Vandermonde MDS code $\mathcal{C}$ (§II-A) of length $\tilde{k} + \tilde{g}$ and dimension $\tilde{k}$. Then, we shorten and

puncture $\mathcal{C}$ by removing the last $\tilde{k} - k^I$ rows, the last $\tilde{k} - k^I$ data columns, and the last $\tilde{g} - \ell^I - g^I$ parity columns from the generator matrix to obtain $\mathcal{C}^{\tilde{I}}$. Finally, we derive the initial base code as a basic pyramid code (§II-B) of $\mathcal{C}^{\tilde{I}}$ (and likewise for the final base code).

B. Conversion techniques

For ease of exposition, we first present the techniques that will be used in designing conversion-bandwidth efficient codes:

Direct computation (DC). A final parity symbol is computed from the data symbols in its support. E.g., this is used in Ex. 1 to compute $\mathbf{q}_{[1,12]} \cdot \mathbf{c}_{[1,12]}$ from $\mathbf{c}$.

Projection (Pr). A final parity symbol with support S' is computed from an initial parity symbol with support $S \supsetneq S'$ and data symbols in $S \setminus S'$. E.g., used in Ex. 2 to compute $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]}$ from $(\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]} + \mathbf{q}_{[7,12]} \cdot \mathbf{b}_{[7,12]})$ and $\mathbf{b}$.

Piggybacks (Pb). A final parity symbol for instance $j \in [\alpha]$ is stored as a piggyback on an initial parity symbol of instance $i \in [\alpha]$ such that $\sigma(i) > \sigma(j)$. The piggyback is recovered by computing and subtracting the base of the initial parity using the data in instance i. E.g., this is used in Ex. 1 to compute $\mathbf{t}_{[1,12]} \cdot \mathbf{b}_{[1,12]}$ from $(\mathbf{q}_{[1,6]} \cdot \mathbf{c}_{[1,6]} + \mathbf{t}_{[1,6]} \cdot \mathbf{b}_{[1,6]})$, $(\mathbf{q}_{[1,6]} \cdot \mathbf{c}_{[7,12]} + \mathbf{t}_{[1,6]} \cdot \mathbf{b}_{[7,12]})$, and $\mathbf{c}$.

Projected piggybacks (PP). A final parity symbol for instance $j \in [\alpha]$ is stored as a piggyback on an initial parity symbol of instance $i \in [\alpha]$ with $\sigma(i) > \sigma(j)$. The base of the initial parity symbol (with support S) is projected using the data in a subset $S' \subsetneq S$; the remaining part (with support $S \setminus S'$) becomes a piggyback in the final parity symbol. In the final code, i and j are swapped in the decoding order. E.g., this is used in Ex. 2 to compute $(\mathbf{q}_{[1,6]} \cdot \mathbf{d}_{[1,6]} + \mathbf{t}_{[1,6]} \cdot \mathbf{a}_{[1,6]})$ from $(\mathbf{t}_{[1,6]} \cdot \mathbf{a}_{[1,6]} + \mathbf{t}_{[7,12]} \cdot \mathbf{b}_{[7,12]} + \mathbf{q}_{[1,6]} \cdot \mathbf{d}_{[1,6]})$ and $\mathbf{b}$.

Linear combination (LC). A final parity symbol with support T is computed as a linear combination of symbols with support S_i such that $T = \bigcup_i S_i$. The linear combination is determined by the base code. E.g., this is used in Ex. 1 to compute $\mathbf{q}_{[1,12]} \cdot \mathbf{a}_{[1,12]}$ from $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]}$ and $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[7,12]}$.

Instance reassignment (IR). During conversion, the data symbols associated to data node $i \in [k]$ are reassigned to instances via some permutation $\pi_i : [\alpha] \to [\alpha]$. That is, data in the final code is interpreted as $\mathbf{m}'_i = (m_{i,\pi_i(j)})_{j=1}^{\alpha}$ for $i \in [k]$. This reassignment affects the supports of parities, but it does not modify data nodes. E.g., this is used in Ex. 2 to exchange $\mathbf{a}$ and $\mathbf{b}$ during conversion in some nodes.

We denote linear combination of multiple piggybacks as Pb+LC, e.g., as used in the piggybacks of local parities in both examples. In diagrams, we denote the use of IR with letters, and use the following colors to distinguish the other techniques:

DC Pr Pb PP LC Pb+LC

C. General strategy

As the output of conversion, the converter constructs new parity nodes, called *target parities*. Target parities are grouped into s sets, such that parity nodes that have the same support are in the same set. Data nodes are divided into s disjoint batches of equal size, corresponding to the supports of the s sets of target parities. In other words, target parities in set i are in the span of batch i ($i \in [s]$). E.g., in Ex. 1, there is single target parity and s=1 set., while in Ex. 2 there are two final parities (one in each final codeword) and thus s=2 sets.

The α instances are divided into s blocks of size B, plus a remainder block of size R (i.e. $\alpha := sB + R$), where s, B, and R are positive integers set depending on the type of conversion. E.g., in Ex. 1, B=3 and R=0, while in Ex. 2 B=1 and R=2.

We refer to block $i \in [s]$ of nodes in batch i as a *special block*, and to blocks $j \neq i \in [s]$ as *regular blocks*. Special blocks are divided into two sub-blocks: an *accessed sub-block* (A-block) of size E and an *unaccessed sub-block* (B-block) of size $B - E$. In initial parity nodes, block $i \in [s]$ is special if its support and the data in batch i (i.e. data in a special block i) have a non-empty intersection; otherwise, the block is regular. Notice that for each $i \in [s]$, there is a single batch whose nodes have block i as special. In particular, when $s = 1$, all nodes have a single special block, and no regular blocks. E.g., in Ex. 1, each node has a single block (special) and E=1. In Ex. 2, each node has one regular, special, and remainder block; the special block corresponds to $\mathbf{b}$ and E=0.

D. Design of parities and conversion

We describe three types of parity design: *merge parities*, *split parities*, and *unchanged parities* (see Fig. 4). In each design, we describe the techniques associated which each symbol.

During conversion, for each batch, the converter downloads all symbols in regular blocks and A-blocks of data nodes (i.e. B-blocks and remainder blocks are not read). In addition, the converter downloads symbols from initial parities and uses them as specified by the parity type. To ensure the final code has optimal distance, each initial parity symbol is used in constructing at most one final parity symbol (which avoids linear dependencies that reduce distance). Thus, we assign at most one technique to each initial parity symbol. In addition, piggybacks in local parities must be a function of data in their local group, and piggybacks in global parities must be a functions of the data in their codeword.

In all parity types, A-blocks and regular blocks are designed the same way: these blocks use Pb or Pb+LC. For symbols in these blocks, all data in their supports is read during conversion, and so piggybacks in them can be recovered. Piggybacks in A-blocks are chosen as parity symbols of instances in the corresponding B-blocks; piggybacks from regular blocks are chosen as parity symbols of instances in the remainder block. Target parity symbols that are a function of data in A-blocks or regular blocks use DC.

Merge parities: This design is used for parities whose support is a strict subset of the support of a target parity. E.g., in Ex. 1 the initial global parities are merge parities. When $d^I \geq d^F$, the B-block and remainder block of target parities can be fully constructed via LC of initial parity symbols in the respective blocks. Otherwise, we use LC to construct the B-block and remainder block of some target parities, and use Pb or Pb+LC from A-blocks and regular blocks for other target parities.

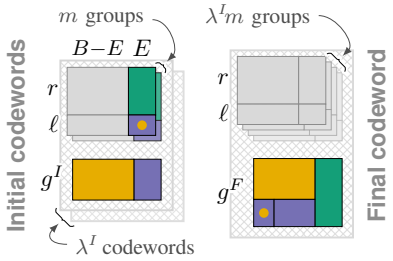

Fig. 5: Global merge conversion ($r = \frac{k^I}{2}$, $\lambda^I = 2$, and $g^I < g^F$).

Split parities: This design is used for parities whose support is a strict superset of the support of a target parity. E.g., in Ex. 2 the initial global parities are split parities. The remainder block of split parities is unused. When $d^I \geq d^F$, then the B-block of target parities can be fully constructed via Pr of split parities in B-blocks. If $d^I > d^F$, the rest of the initial parity symbols in B-blocks use PP to construct final parity symbols in a remainder block. When $d^I < d^F$, the whole B-block of split parities uses Pr. The rest of the final parity symbols in the B-block use Pb from A-blocks.

Unchanged parities: Both B-blocks and remainder blocks are unused. E.g., in both examples local parities are unchanged parities.This type of parity can be kept in the final code.

E. Instance reassignment

In conversions where the number of codewords increases, we have to ensure that final codewords use the same code. Otherwise, systems would need to keep extra metadata for each codeword, which induces extra complexity and overhead. The template described so far does not meet this requirement:we use IR to correct this. Let $\text{batch}(i) := \left\lfloor \frac{(i-1)s}{k^I} \right\rfloor$. For data node i, we use permutation:

$$\pi_i(j) := \begin{cases} ((j - \text{batch}(i)B - 1) \bmod sB) + 1, & \text{if } j \leq sB, \\ j, & \text{otherwise.} \end{cases}$$

Theorem 1: The construction template presented in this section yields codes satisfying properties P1–3. ■

IV. Conversion of Global Parameters

In this section, we describe constructions where both k and g vary, with r and ℓ constant. These conversions are useful to alter the *durability* of the code. In particular, we explore two types: *global merge conversions*, which combine multiple codewords into one; and *global split conversions*, which divide one codeword into multiple. In both types, g changes arbitrarily.

One way to achieve these conversions is to ignore local parities, and use existing constructions for MDS codes [6], [7]. The new constructions also use local parities in conversion, and thus can reduce the conversion bandwidth compared to previous constructions.

A. Global merge conversion

In global merge conversions, $\lambda^I \geq 2$ codewords are merged into one, i.e. $k^F = \lambda^I k^I$. Local parities are designed as unchanged parities, and global parities as merge parities.

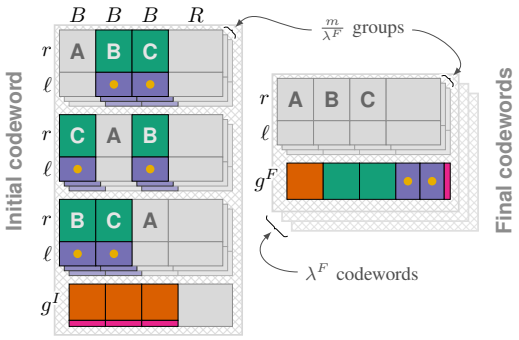

Fig. 6: Global split conversion ($r = \frac{k^I}{9}$, $\lambda^F = 3$, and $g^I > g^F$).

Theorem 2: The construction presented in this section achieves the following conversion bandwidth:

$$\tilde{\gamma} = \begin{cases} \lambda^I g^F, & \text{if } g^F \leq g^I, \\ \lambda^I \left(\frac{(k^I + m^I \ell)(g^F - g^I)}{g^F + \ell} + g^I \right), & \text{otherwise.} \end{cases}$$ ■

This construction generalizes the MDS construction ($\ell = 0$).

Case $g^F \leq g^I$: Conversion is carried out using only global parities, as in the MDS case [6].

Case $g^F > g^I$: In this construction (see Fig. 5), we set:

$$s = 1, \qquad B = g^F + \ell, \qquad R = 0, \qquad E = g^F - g^I.$$

During conversion, LC is used in the global parities to construct symbols in the first g^I final global parities. The rest of the final symbols are constructed via Pb, Pb+LC, or DC.

B. Global split conversion

In global split conversions, a single initial codeword is split into $\lambda^F \geq 2$, i.e. $k^I = \lambda^F k^F$. Local parities are designed as unchanged parities, and global parities as split parities.

Theorem 3: The construction presented in this section achieves the following conversion bandwidth:

$$\tilde{\gamma} = \begin{cases} \lambda^F g^F \frac{(\lambda^F - 1)(k^F + m^F \ell) + g^I}{(\lambda^F - 1) g^F + g^I + \ell(\lambda^F - 1)}, & \text{if } g^F \leq g^I, \\ \frac{\lambda^F g^F \left((k^F + m^F \ell)(\lambda^F g^F - g^I) + g^I g^F \right)}{\lambda^F g^F (g^F + \ell) - g^I \ell}, & \text{otherwise.} \end{cases}$$ ■

This construction generalizes the MDS construction ($\ell = 0$).

Case $g^I \geq g^F$: Variables are set as follows (see Fig. 6):

$$s = \lambda^F, \quad B = g^F, \quad R = \ell(\lambda^F - 1) + g^I - g^F, \quad E = 0.$$

The first g^F global parities use Pr to construct symbols in the final global parities; the remaining initial global parities use PP to construct symbols in remainder blocks. Local parities use Pb+LC to construct symbols from remainder blocks.

Case $g^I < g^F$: We set the construction variables as follows:

$$s = \lambda^F, \quad B = (g^F)^2, \quad R = \lambda^F \ell g^F - \ell g^I, \quad E = g^F(g^F - g^I).$$

During conversion, initial global parities use Pr to construct symbols for the B-blocks of the first g^I global parities in each final codeword. The rest of the symbols are constructed via Pb and Pb+LC from the A-blocks. The remainder block of final global parities is constructed via Pb+LC on local parities.

REFERENCES

[1] "Locally repairable convertible codes: erasure codes for efficient repair and conversion." http://www.cs.cmu.edu/~rvinayak/papers/LRC_conversion_ISIT2023_extension.pdf.

[2] S. Kadekodi, K. V. Rashmi, and G. R. Ganger, "Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity," in *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019* (A. Merchant and H. Weatherspoon, eds.), pp. 345–358, USENIX Association, 2019.

[3] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in HDFS," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (J. Schindler and E. Zadok, eds.), pp. 213–226, USENIX Association, 2015.

[4] F. Maturana, V. S. C. Mukka, and K. V. Rashmi, "Access-optimal linear MDS convertible codes for all parameters," in *IEEE International Symposium on Information Theory, ISIT 2020, Los Angeles, California, USA, June 21-26, 2020*, 2020.

[5] F. Maturana and K. V. Rashmi, "Convertible codes: enabling efficient conversion of coded data in distributed storage," *IEEE Transactions on Information Theory*, vol. 68, pp. 4392–4407, 2022.

[6] F. Maturana and K. V. Rashmi, "Bandwidth cost of code conversions in distributed storage: fundamental limits and optimal constructions," in *IEEE International Symposium on Information Theory, ISIT 2021, Melbourne, Australia, July 12-20, 2021*, pp. 2334–2339, IEEE, 2021.

[7] F. Maturana and K. V. Rashmi, "Bandwidth cost of code conversions in the split regime," (Espoo, Finland), pp. 3262–3267, IEEE, 2022.

[8] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *5th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'13, San Jose, CA, USA, June 27-28, 2013* (A. Gulati, ed.), USENIX Association, 2013.

[9] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: novel erasure codes for big data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[10] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A" hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 331–342, 2014.

[11] C. Huang, M. Chen, and J. Li, "Pyramid codes: flexible schemes to trade space for access efficiency in reliable data storage systems," *ACM Transactions on Storage*, vol. 9, pp. 3:1–3:28, Mar. 2013.

[12] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Transactions on Information Theory*, vol. 60, no. 10, pp. 5843–5855, 2014.

[13] 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.

[14] N. Prakash, G. M. Kamath, V. Lalitha, and P. V. Kumar, "Optimal linear codes with a local-error-correction property," in *Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*, pp. 2776–2780, IEEE, 2012.

[15] K. V. Rashmi, N. B. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," *IEEE Transactions on Information Theory*, vol. 63, no. 9, pp. 5802–5820, 2017.

[16] J. Han and L. A. Lastras-Montaño, "Reliable memories with subline accesses," in *IEEE International Symposium on Information Theory, ISIT 2007, Nice, France, June 24-29, 2007*, pp. 2531–2535, IEEE, 2007.

[17] A. S. Rawat, O. O. Koyluoglu, N. Silberstein, and S. Vishwanath, "Optimal locally repairable and secure codes for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 60, no. 1, pp. 212–236, 2013.

[18] M. Blaum, J. L. Hafner, and S. Hetzler, "Partial-MDS codes and their application to RAID type of architectures," *IEEE Transactions on Information Theory*, vol. 59, no. 7, pp. 4510–4519, 2013.

[19] N. Silberstein, A. S. Rawat, O. O. Koyluoglu, and S. Vishwanath, "Optimal locally repairable codes via rank-metric codes," (Istanbul, Turkey), pp. 1819–1823, IEEE, 2013.

[20] I. Tamo and A. Barg, "A family of optimal locally recoverable codes," *IEEE Transactions on Information Theory*, vol. 60, no. 8, pp. 4661–4676, 2014.

[21] P. Gopalan, C. Huang, B. Jenkins, and S. Yekhanin, "Explicit maximally recoverable codes with locality," *IEEE Transactions on Information Theory*, vol. 60, no. 9, pp. 5245–5256, 2014.

[22] V. R. Cadambe and A. Mazumdar, "Bounds on the size of locally recoverable codes," *IEEE Transactions on Information Theory*, vol. 61, no. 11, pp. 5787–5794, 2015.

[23] I. Tamo, A. Barg, and A. A. Frolov, "Bounds on the parameters of locally recoverable codes," *IEEE Transactions on Information Theory*, vol. 62, no. 6, pp. 3070–3083, 2016.

[24] I. Tamo, D. S. Papailiopoulos, and A. G. Dimakis, "Optimal locally repairable codes and connections to matroid theory," *IEEE Transactions on Information Theory*, vol. 62, no. 12, pp. 6661–6671, 2016.

[25] S. L. Frank-Fischer, V. Guruswami, and M. Wootters, "Locality via partially lifted codes," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA* (K. Jansen, J. D. P. Rolim, D. Williamson, and S. S. Vempala, eds.), vol. 81(43) of *LIPIcs*, pp. 1–17, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[26] A. Barg, K. Haymaker, E. W. Howe, G. L. Matthews, and A. Várilly-Alvarado, "Locally recoverable codes from algebraic curves and surfaces," in *Algebraic Geometry for Coding Theory and Cryptography* (E. W. Howe, K. E. Lauter, and J. L. Walker, eds.), (Cham), pp. 95–127, Springer International Publishing, 2017.

[27] A. Mazumdar, "Capacity of locally recoverable codes," in *IEEE Information Theory Workshop, ITW 2018, Guangzhou, China, November 25-29, 2018*, pp. 1–5, IEEE, 2018.

[28] A. Agarwal, A. Barg, S. Hu, A. Mazumdar, and I. Tamo, "Combinatorial alphabet-dependent bounds for locally recoverable codes," *IEEE Transactions on Information Theory*, vol. 64, no. 5, pp. 3481–3492, 2018.

[29] S. Gopi, V. Guruswami, and S. Yekhanin, "Maximally recoverable LRCs: A field size lower bound and constructions for few heavy parities," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019* (T. M. Chan, ed.), pp. 2154–2170, SIAM, 2019.

[30] V. Guruswami, C. Xing, and C. Yuan, "How long can optimal locally repairable codes be?," *IEEE Transactions on Information Theory*, vol. 65, no. 6, pp. 3662–3670, 2019.

[31] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Enabling node repair in any erasure code for distributed storage," in *2011 IEEE International Symposium on Information Theory Proceedings, ISIT 2011, St. Petersburg, Russia, July 31 - August 5, 2011* (A. Kuleshov, V. M. Blinovsky, and A. Ephremides, eds.), pp. 1235–1239, IEEE, 2011.

[32] S. Mousavi, T. Zhou, and C. Tian, "Delayed parity generation in MDS storage codes," in *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pp. 1889–1893, IEEE, 2018.

[33] S. Wu, Z. Shen, and P. P. C. Lee, "On the optimal repair-scaling trade-off in locally repairable codes," in *2020 IEEE Conference on Computer Communications, INFOCOM 2020, Virtual Conference, July 6-9, 2020*, IEEE, 2020.

[34] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen, "Exploiting combined locality for wide-stripe erasure coding in distributed storage," in *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021* (M. K. Aguilera and G. Yadgar, eds.), pp. 233–248, USENIX Association, 2021.

[35] S. Wu, Z. Shen, P. P. C. Lee, and Y. Xu, "Optimal repair-scaling trade-off in locally repairable codes: analysis and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 56–69, 2022.

[36] S. Wu, Q. Du, P. P. C. Lee, Y. Li, and Y. Xu, "Optimal data placement for stripe merging in locally repairable codes," (London, United Kingdom), pp. 1669–1678, IEEE, 2022.