Balancing Repair Bandwidth and Sub-Packetization in Erasure-Coded Storage via Elastic Transformation

Kaicheng Tang[†], Keyun Cheng[†], Helen H. W. Chan[†], Xiaolu Li[‡]
Patrick P. C. Lee[†], Yuchong Hu[‡], Jie Li[*], and Ting-Yi Wu[*]
[†]*The Chinese University of Hong Kong* [‡]*Huazhong University of Science and Technology*
[*]*Huawei Technologies Co., Ltd., Hong Kong*

***Abstract*—Erasure coding provides high fault-tolerant storage with significantly low redundancy overhead, at the expense of high repair bandwidth. While there exist access-optimal codes that theoretically minimize both the repair bandwidth and the amount of disk reads, they also incur a high sub-packetization level, thereby leading to non-sequential I/Os and degrading repair performance. We propose *elastic transformation*, a framework that transforms any base code into a new code with smaller repair bandwidth for all or a subset of nodes, such that it can be configured with a wide range of sub-packetization levels to limit the non-sequential I/O overhead. We prove the fault tolerance of elastic transformation and model numerically the repair performance with respect to a sub-packetization level. We further prototype and evaluate elastic transformation atop HDFS, and show how it reduces the single-block repair time of the base codes and access-optimal codes in a real network setting.**

I. Introduction

Modern distributed storage systems adopt *erasure coding* as a low-cost storage redundancy technique to protect data against failures; in particular, Reed-Solomon (RS) codes [32] are the most popular erasure codes adopted in production (e.g., at Google [10], Facebook [25], Backblaze [7], CERN [27], etc.). At a high level, we can construct an RS code, denoted by $\mathrm{RS}(n,k)$, with two input parameters n and k (where $k<n$). $\mathrm{RS}(n,k)$ encodes data in fixed-size units called *packets*. It encodes every k data packets into $n-k$ *parity* packets of the same size, such that any k out of n packets suffice to reconstruct the k data packets. Compared with traditional replication, erasure coding achieves multiple orders of magnitude higher reliability (in mean-time-to-failure) with the same redundancy [37].

Erasure coding achieves high storage efficiency at the expense of high *repair bandwidth* (i.e., the amount of data transferred from the non-failed nodes) when repairing the lost data in node failures. For example, to repair any lost packet, $\mathrm{RS}(n,k)$ reads and transfers k packets from k non-failed nodes, leading to $k\times$ bandwidth of the packet size. New erasure codes have been proposed in the literature to reduce the repair bandwidth, among which minimum-storage regenerating (MSR) codes [9] are the first proven erasure codes that minimize the repair bandwidth (with significantly less than k packets in repair), with the same redundancy as RS codes. However, the early constructions of MSR codes [9] incur high I/O access, as they require non-failed nodes to read and encode all their locally stored data and send the encoded data for repair.

Some studies extend MSR codes with *access-optimal MSR codes* (e.g., [8], [26], [28], [36]), which minimize both repair bandwidth and I/O access by eliminating the need of encoding in non-failed nodes during repair, such that each non-failed node directly transfers the data being read from its local storage. The core idea of access-optimal MSR codes is *sub-packetization*, which partitions each packet into smaller sub-packets and performs encoding and repair at the sub-packet granularity. During repair, a non-failed node only reads and transfers a subset of sub-packets.

Despite the theoretical guarantees of access-optimal MSR codes, the practical repair performance gain is often limited. The major drawback is that under sub-packetization, access-optimal MSR codes need to read a subset of sub-packets that are not sequentially placed. Thus, accessing sub-packets incurs *non-sequential I/Os*, which incur non-negligible disk seek overhead (albeit less amount of data being read) compared with sequential I/Os [5, Ch. 37]. For example, in SATA hard disks, a non-sequential I/O to a 4 KB disk block takes milliseconds (or hundreds of KB/s in throughput), while sequential I/Os can reach over 100 MB/s. Such performance asymmetry is also found in solid-state drives [5, Ch. 44]. To amortize non-sequential I/O overhead, some studies exploit a larger packet size in system implementation [15], [22], [31], [34]. Unfortunately, the *sub-packetization level* (i.e., the number of sub-packets per packet) is provably exponential for access-optimal MSR codes [6] and inevitably aggravates non-sequential I/O overhead.

Our insight is that there exists a trade-off between repair bandwidth (i.e., network transmissions) and sub-packetization (i.e., non-sequential I/Os) in erasure coding deployment. To this end, we propose an *elastic transformation* framework that transforms any base erasure code into another erasure code with smaller repair bandwidth, subject to a configurable sub-packetization level. By elastic, we mean that the framework can be (i) configured with a wide range of sub-packetization levels (starting from two) to limit non-sequential I/O overhead; (ii) applied to all or a subset of nodes; and (iii) applied to a variety of erasure codes. Our framework is more flexible than the prior transformation framework [20] that always increases the sub-packetization level by $n-k$ times in each transformation. To summarize, we make the following contributions.

- We design a building block, called the *transformation array*, for the elastic transformation framework to transform an array of sub-packets to reduce their repair bandwidth, while limiting the non-sequential I/O overhead. The transformation array can be configured in any size (based on the sub-packetization level and the number of packets), and its main idea is to overlap two square transformation arrays into a non-square transformation array.

- We show how our elastic transformation reduces the repair bandwidth of various erasure codes, including RS codes [32], Azure's LRC [13], Hitchhiker [31], and HashTag [18].
- On the theoretical side, we prove the fault tolerance of elastic transformation, model the lower bound of repair bandwidth for a given sub-packetization level, and model the repair time subject to the bandwidth and I/O conditions.
- Existing code transformation solutions [11], [12], [19], [20] only focus on theoretical analysis, but do not consider empirical evaluation. To fill this void, we prototype elastic transformation based on OpenEC [23] and evaluate it atop Hadoop 3.0.0 HDFS [1]. Experiments on our prototype, called OpenEC-ET, show that it reduces the repair time of the base RS codes (without transformation) by up to 56.3% in low-bandwidth settings and reduces the repair time of the access-optimal MSR codes by up to 51.4% in high-bandwidth settings. The source code of OpenEC-ET is at: **http://adslab.cse.cuhk.edu.hk/software/openec-et**.

II. Background and Related Work

A. Basics of Erasure Coding

We consider RS(n,k), which performs encoding on fixed-size *packets*. It encodes a set of k original (uncoded) data packets into $n-k$ (coded) parity packets, and the n data/parity packets collectively form a *stripe*. RS encoding ensures that any k out of the n packets in a stripe suffice to decode the original k data packets. To mitigate the I/O overhead in read/write operations, practical distributed storage systems (e.g., HDFS [35]) store data in fixed-size units called *blocks* that have much larger sizes than packets; for example, the default packet and block sizes are 1 MiB and 128 MiB in Hadoop 3.0.0 HDFS [1], respectively. Each set of n blocks contains multiple stripes of packets (where the packets of each stripe reside at the same block offset) that are encoded/decoded independently and identically, and the n blocks are distributed across n nodes of a distributed storage system to provide fault tolerance against $n-k$ node failures.

RS codes are popularly deployed (§I) by addressing three key properties: (i) *generality*, meaning that RS(n,k) supports general parameters n and $k<n$; (ii) *maximum distance separable (MDS)*, meaning that the redundancy (i.e., n/k times the original data size) is the minimum among all erasure codes for tolerating any $n-k$ node failures; and (iii) *systematic*, meaning that the original k data packets are preserved in a stripe after encoding. Our work preserves the three properties when transforming RS codes into repair-friendly codes.

We consider sub-packetization to mitigate the repair bandwidth. Sub-packetization divides a packet into α sub-packets, where α is called the *sub-packetization level*. Each set of n sub-packets at the same offset of a packet is called a *sub-stripe* (i.e., there are α sub-stripes in a stripe). Repairing a packet can be done by retrieving a subset of sub-packets from each available packet in a stripe.

We define the following notations. Consider a stripe of n packets that are stored in n nodes, denoted by $N_1, N_2, \cdots, N_n$. Let $\boldsymbol{d}_i = (d_{i,1}, d_{i,2}, \cdots, d_{i,k})$ be the vector of k data sub-packets in the i-th sub-stripe, where $1 \le i \le \alpha$; let f_j be the parity function for encoding d_i into the j-th parity sub-packet in the i-th sub-stripe, where $1 \le j \le n-k$. Figure 1 shows RS$(9,6)$ with $\alpha = 2$ sub-stripes. Note that in RS(n,k), each sub-stripe remains independently and identically encoded. Since each node stores one packet, in this paper, the terms "nodes" and "packets" are interchangeably used.

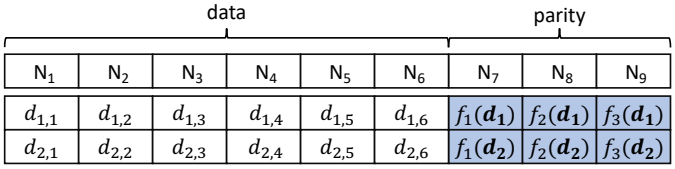


Figure 1: RS$(9,6)$ with $\alpha = 2$.

B. Related Work on Erasure-Coded Repair

Repair-friendly codes. *Access-optimal MSR codes* extend the classical MSR codes [9] to minimize both repair bandwidth and I/O access (i.e., the minimum repair bandwidth is the same as the amount of I/Os accessed from storage), while maintaining storage optimality (i.e., the MDS property). FMSR codes [8] are non-systematic codes for $n-k=2$. PM-RBT [28], Butterfly codes [26], and Clay codes [36] are systematic codes for $n \ge 2k-1$, $n-k=2$, and general (n,k), respectively. Access-optimal MSR codes theoretically incur an exponential sub-packetization level $\alpha \ge (n-k)^{\lceil n/(n-k) \rceil}$ [6], leading to significant non-sequential I/Os. Our work can transform a base code into an access-optimal MSR code for general (n,k).

Locally repairable codes [13], [16], [33] reduce the number of packets to retrieve in repair and hence the repair bandwidth. In this work, we focus on Azure's Local Reconstruction Codes (or LRC in short) [13]. LRC is configured by three parameters, n, $k<n$, and ℓ, and is denoted by LRC(n,k,ℓ). It divides k data packets into ℓ groups with $\frac{k}{\ell}$ data packets each (assuming k is divisible by ℓ). It adds a local parity packet to each group, and encodes all k data packets to form $n-k-\ell$ global parity packets. Repairing a data packet or a local parity packet can locally retrieve the $\frac{k}{\ell}$ available packets within the same group, so as to reduce the repair bandwidth. Note that LRC still retrieves the k available data packets to repair a global parity packet. LRC keeps $\alpha = 1$, but is non-MDS and incurs higher storage overhead than RS and MSR codes.

Piggybacking codes [30], [31] construct a stripe from multiple sub-stripes of an RS code, and add the data of a sub-stripe into the parities of another sub-stripe through some piggybacking functions. Repairing a data packet of a stripe can retrieve the sub-packets across sub-stripes. In this work, we focus on Hitchhiker [31], which has 25-50% savings of repair bandwidth compared with RS codes. Hitchhiker is MDS, and supports $\alpha \ge 2$ [30], albeit higher repair bandwidth than access-optimal MSR codes. However, it improves the repair for data packets only, while parity packets are still repaired by retrieving k available packets as in RS codes.

In this work, we also consider HashTag [18], a repair-friendly code that supports general (n,k) and any $\alpha \ge 2$ with the same motivation as ours to mitigate non-sequential I/O overhead. However, like Hitchhiker, HashTag supports efficient repair for

data packets only. HashTag+ [17] extends HashTag to support efficient repair for parity packets as well by applying code transformation [20]. However, the sub-packetization level α, after code transformation, needs to be a multiple of $n-k$ (see details below). In contrast, elastic transformation maintains a small $\alpha \geq 2$ for both data and parity packets. Also, we show how elastic transformation can be applied to RS codes, LRC, Hitchhiker, and HashTag, while keeping a small α.

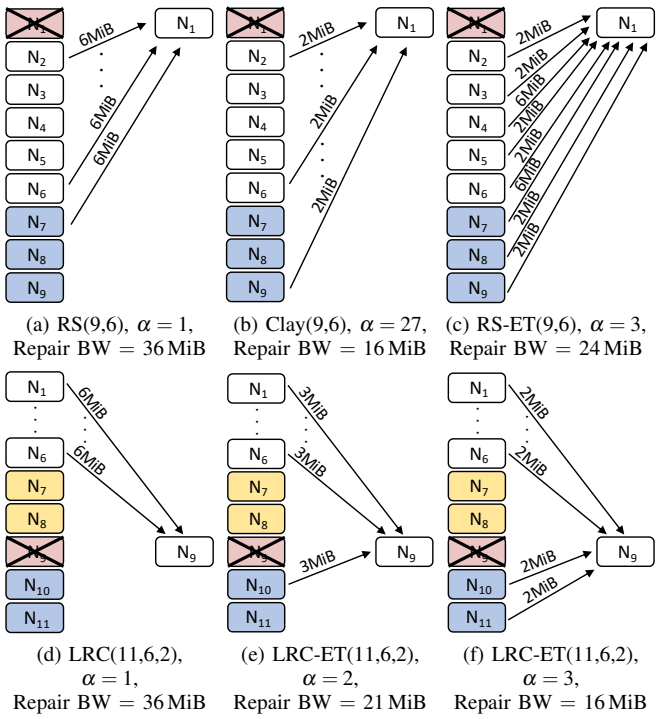
Code transformation. Li et al. [20] propose a generic transformation framework that converts a non-binary MDS code into a new MDS code that minimizes the repair bandwidth for $n-k$ nodes, while the sub-packetization level α of the new code increases by $n-k$ times. By repeatedly applying the transformation to $\lceil n/(n-k) \rceil$ groups of $n-k$ nodes, the final code becomes an access-optimal MSR code, while α meets the lower bound $(n-k)^{\lceil n/(n-k) \rceil}$. Follow-up studies extend generic transformation for binary MDS codes [11], [12], [19]. However, generic transformation [20] poses a stringent requirement on the sub-packetization level α, which always increases by $n-k$ times in each transformation (e.g., when RS$(14,10)$ is transformed into an access-optimal MSR code, α becomes 4, 16, 64, and 256 only). In contrast, we consider a much wider range of α (e.g., for any small α where $2 \leq \alpha \leq n-k$, and for various values of large α where $n-k < \alpha \leq (n-k)^{\lceil n/(n-k) \rceil}$).

Repair-efficient algorithms. Recent studies propose repair-efficient algorithms that apply to general erasure-coded storage (e.g., via parallelization [21], [24]). To mitigate non-sequential I/O overhead, Hitchhiker [31] proposes a hop-and-couple method to keep a large sub-packet size, while Geometric partitioning [34] parallelizes the repair for access-optimal MSR codes (e.g., Clay codes [36]) with geometric sub-packet sizes. Repair-efficient algorithms focus on system-level repair optimization, while our work focuses on code transformation.

C. Motivating Examples

We show via examples how elastic transformation balances the trade-off between repair bandwidth and sub-packetization.

Examples for MDS codes. We first consider MDS codes. Suppose that we store a file of size 36 MiB with RS$(9,6)$, where $k=6$ data blocks (of size 6 MiB each) are stored in nodes N_1 to N_6, while $n-k=3$ parity blocks (of size 6 MiB each) are stored in nodes N_7 to N_9. Suppose that N_1 fails. Figure 2(a) shows the repair of N_1 in RS(9,6) (where $\alpha=1$), which transfers 6×6 MiB $= 36$ MiB of data from N_2 to N_7. Figure 2(b) shows the repair of N_1 in an access-optimal MSR code Clay(9,6) [36], which transfers 8×2 MiB $= 16$ MiB of data from N_2 to N_9. Clay(9,6) has the same storage redundancy as RS(9,6), while its repair bandwidth is minimized [36] and is 55.6% less than that of RS$(9,6)$. However, Clay(9,6) also has a high sub-packetization level $\alpha=27$. Figure 2(c) shows the repair of N_1 in the elastic transformation for RS$(9,6)$ (called RS-ET$(9,6)$) configured with $\alpha=3$. RS-ET$(9,6)$ transfers 2-6 MiB of data from each of N_2 to N_9 (note that it retrieves a different amount of data from each node), with a total of repair bandwidth of 24 MiB. It incurs 33.3% less repair bandwidth than RS$(9,6)$, while limiting non-sequential I/Os with $\alpha=3$.



(a) RS(9,6), $\alpha=1$, Repair BW $=$ 36 MiB
(b) Clay(9,6), $\alpha=27$, Repair BW $=$ 16 MiB
(c) RS-ET(9,6), $\alpha=3$, Repair BW $=$ 24 MiB
(d) LRC(11,6,2), $\alpha=1$, Repair BW $=$ 36 MiB
(e) LRC-ET(11,6,2), $\alpha=2$, Repair BW $=$ 21 MiB
(f) LRC-ET(11,6,2), $\alpha=3$, Repair BW $=$ 16 MiB

Figure 2: Motivating examples of elastic transformation.

Examples for non-MDS codes. Elastic transformation can also be applied to non-MDS codes to reduce the repair bandwidth for a subset of nodes. Consider LRC [13], which reduces the repair bandwidth for data blocks and local parity blocks only, but still retrieves k blocks for repairing any global parity block (§II-B). We show how elastic transformation can reduce the repair bandwidth for global parity blocks, while the data blocks and local parity blocks are still locally repairable. Suppose that we store a file of size 36 MiB with LRC$(11,6,2)$, in which $k=6$ data blocks (of size 6 MiB each) are stored in N_1 to N_6, $\ell=2$ local parity blocks are stored in N_7 and N_8, and the remaining $n-k-\ell=3$ global parity blocks are stored in N_9 to N_{11}. Suppose that N_9 fails. Figure 2(d) shows LRC (with $\alpha=1$), in which the repair of N_9 (where a global parity block is stored) retrieves 6×6 MiB $= 36$ MiB of data as in RS codes.

Figures 2(e) and 2(f) show the elastic transformation on LRC$(11,6,2)$ (called LRC-ET$(11,6,2)$) configured with $\alpha=2$ and $\alpha=3$, respectively, in which the repair of N_9 reduces the repair bandwidth to 21 MiB and 16 MiB (i.e., 41.7% and 55.6% less than LRC), respectively. Note that the repair bandwidth of LRC-ET$(11,6,2)$ with $\alpha=3$ (Figure 2(f)) is less than RS-ET$(9,6)$ with $\alpha=3$ (Figure 2(c)), since elastic transformation is applied to reduce the repair bandwidth for a subset of nodes in LRC-ET (i.e., N_9 to N_{11}) instead of for all nodes in RS-ET.

III. Elastic Transformation

We propose an elastic transformation framework that transforms a base code into a repair-friendly code with a configurable sub-packetization level α. Our goal is to reduce the repair bandwidth for repairing a single lost packet as in existing studies (§II-B), assuming that having a single lost packet in a

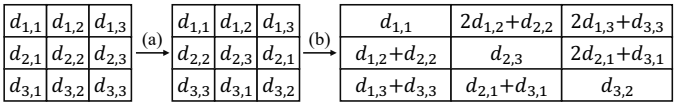


Figure 3: Example of transformation on a square array for $\alpha = 3$: (a) rotation and (b) pairwise coupling.

stripe is much more common than having multiple lost packets concurrently in practice [13], [29].

A. Construction of Transformation Arrays

A key building block of our framework is a *transformation array*, which specifies an array of data/parity sub-packets to which our transformation is applied. Each row of a transformation array corresponds to a sub-stripe, while each column represents the sub-packets stored in a node. There are two types of transformation arrays, as explained below.

Square transformation arrays. We define a *square transformation array* (or *square array* in short) as an $\alpha \times \alpha$ array of sub-packets in a code, where α is the sub-packetization level of the transformed code. Without loss of generality, we consider how the transformation operates on a square array of data sub-packets $[d_{i,j}]$ for $1 \le i,j \le \alpha$; note that the array may also contain both data and parity sub-packets. The transformation on a square array is based on *pairwise coupling* [20], [36], which linearly combines the sub-packets within the array. It contains the following steps:

- *Step 1 (Rotation):* It cyclically rotates the i-th row to the left by $i-1$ sub-packets, where $1 \le i \le \alpha$. Each new sub-packet $d'_{i,j}$ after rotation is given by:

$$d'_{i,j} = d_{i,((j+i-2) \bmod \alpha)+1}. \tag{1}$$

- *Step 2 (Pairwise coupling):* Given the rotated array, it linearly combines the sub-packets at the axially symmetric positions (i.e., $d'_{i,j}$ and $d'_{j,i}$ for $i \neq j$ and are said to be *coupled*) and retains the sub-packets in the diagonal positions (i.e., $d'_{i,i}$). Each new sub-packet $d''_{i,j}$ after pairwise coupling is:

$$d''_{i,j} = \begin{cases} d'_{i,j} + d'_{j,i} & \text{for } i > j, \\ d'_{i,j} & \text{for } i = j, \\ e_{i,j} \times d'_{i,j} + d'_{j,i} & \text{for } i < j, \end{cases} \tag{2}$$

for some field element $e_{i,j} \in \mathbb{F}_q \backslash \{0,1\}$. In this paper, we set $e_{i,j} = 2$, which sufficiently preserves fault tolerance in our implementation (§III-C). Figure 3 shows an example of the transformation on a square array for $\alpha = 3$.

After transformation, the resulting sub-packets $\{d''_{i,j}\}$ are in non-systematic form. Thus, we apply *systematic transformation* [20], [28] to revert each $d''_{i,j}$ to the original data sub-packet $d_{i,j}$ and recompute all parity sub-packets accordingly.

- *Step 3 (Systematic transformation):* From Equations (1) and (2), we can express $d_{i,j}$ as a function of $d''_{i,j}$:

$$d_{i,j} = \begin{cases} \frac{e_{p(i,j),i} \times d''_{i,p(i,j)} - d''_{p(i,j),i}}{e_{p(i,j),i} - 1} & \text{for } i > p(i,j), \\ d''_{i,p(i,j)} & \text{for } i = p(i,j), \\ \frac{d''_{i,p(i,j)} - d''_{p(i,j),i}}{e_{i,p(i,j)} - 1} & \text{for } i < p(i,j), \end{cases} \tag{3}$$

where $p(i,j) = ((j-i) \bmod \alpha) + 1$. To map each data sub-packet $d''_{i,j}$ with the original data sub-packet $d_{i,j}$, we can view that each $d''_{i,j}$ is transformed into $d_{i,j}$ via a sequence of arithmetic operations. Each parity sub-packet is also recomputed by applying the same sequence of arithmetic operations to its input data sub-packets. For example, in Figure 4(a), we have $d''_{1,2} = 2d_{1,2} + d_{2,2}$ and $d''_{2,1} = d_{1,2} + d_{2,2}$ before systematic transformation. We can show that $d_{1,2} = d''_{1,2} - d''_{2,1}$ and $d_{2,2} = 2d''_{2,1} - d''_{1,2}$. Thus, we replace $d''_{1,2}$ and $d''_{2,2}$ by $d_{1,2}$ and $d_{2,2}$, respectively. Also, we replace the second inputs to $\boldsymbol{d}_1$ and $\boldsymbol{d}_2$ by $d_{1,2} - d_{2,1}$ and $2d_{2,1} - d_{1,2}$, respectively. We similarly update the inputs to $\boldsymbol{d}_1$, $\boldsymbol{d}_2$, and $\boldsymbol{d}_3$, and finally obtain the systematic form (Figure 4(b)).

N_1	N_2	N_3	N_4	N_5	N_6
$d_{1,1}$	$2d_{1,2}+d_{2,2}$	$2d_{1,3}+d_{3,3}$	$f_1(\boldsymbol{d_1})$	$f_2(\boldsymbol{d_1})$	$f_3(\boldsymbol{d_1})$
$d_{1,2}+d_{2,2}$	$d_{2,3}$	$2d_{2,1}+d_{3,1}$	$f_1(\boldsymbol{d_2})$	$f_2(\boldsymbol{d_2})$	$f_3(\boldsymbol{d_2})$
$d_{1,3}+d_{3,3}$	$d_{2,1}+d_{3,1}$	$d_{3,2}$	$f_1(\boldsymbol{d_3})$	$f_2(\boldsymbol{d_3})$	$f_3(\boldsymbol{d_3})$

(a) Before systematic transformation, where $\boldsymbol{d}_1 = (d_{1,1}, d_{1,2}, d_{1,3})$, $\boldsymbol{d}_2 = (d_{2,1}, d_{2,2}, d_{2,3})$, and $\boldsymbol{d}_3 = (d_{3,1}, d_{3,2}, d_{3,3})$

N_1	N_2	N_3	N_4	N_5	N_6
$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$f_1(\boldsymbol{d_1})$	$f_2(\boldsymbol{d_1})$	$f_3(\boldsymbol{d_1})$
$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$f_1(\boldsymbol{d_2})$	$f_2(\boldsymbol{d_2})$	$f_3(\boldsymbol{d_2})$
$d_{3,1}$	$d_{3,2}$	$d_{3,3}$	$f_1(\boldsymbol{d_3})$	$f_2(\boldsymbol{d_3})$	$f_3(\boldsymbol{d_3})$

(b) After systematic transformation, where $\boldsymbol{d}_1 = (d_{1,1},\ d_{1,2} - d_{2,1},\ d_{1,3} - d_{3,1})$, $\boldsymbol{d}_2 = (d_{2,3} - d_{3,2},\ 2d_{2,1} - d_{1,2},\ d_{2,2})$, and $\boldsymbol{d}_3 = (2d_{3,2} - d_{2,3},\ d_{3,3},\ 2d_{3,1} - d_{1,3})$

Figure 4: A 3×3 square array applied to N_1 to N_3 in the (6,3) code.

Example. We show how the repair bandwidth is reduced by applying a square array. Consider Figure 4(b), which shows the systematic code after we apply a 3×3 square array to the packets in N_1 to N_3 in RS(6,3). Suppose that N_1 fails. The repair of N_1 only retrieves five sub-packets, i.e., $d_{1,2}, d_{1,3}, f_1(\boldsymbol{d}_1), f_2(\boldsymbol{d}_1)$, and $f_3(\boldsymbol{d}_1)$. It first uses $f_1(\boldsymbol{d}_1)$, $f_2(\boldsymbol{d}_1)$, and $f_3(\boldsymbol{d}_1)$ to solve for $d_{1,1}$, $d_{1,2} - d_{2,1}$, and $d_{1,3} - d_{3,1}$. It uses the results, plus $d_{1,2}$ and $d_{1,3}$, to solve for all sub-packets $d_{1,1}$, $d_{2,1}$, and $d_{3,1}$ in N_1. In contrast, the conventional repair of RS(6,3) reads nine sub-packets (from any $k = 3$ available packets), so the repair bandwidth reduces by 44.4%.

Non-square transformation arrays. An (n,k) code in general cannot be evenly divided into square arrays. Thus, we also define a *non-square transformation array* (or *non-square array* in short) as an $\alpha \times \gamma$ array of sub-packets in a code, where $\alpha < \gamma < 2\alpha$. The non-square array is operated by the overlapping of two square arrays. Without loss of generality, consider the transformation on a non-square array of data sub-packets $[d_{i,j}]$ for $1 \le i \le \alpha$ and $1 \le j \le \gamma$. It contains the following steps:

- *Step 1 (Rotation):* As in a square array, it cyclically rotates the i-th row to the left by $i-1$ sub-packets.
- *Step 2 (Pairwise coupling):* Given the rotated non-square array, it performs the first pairwise coupling on the (square) array in N_1 to N_α, such that each new sub-packet in these nodes is given by Equation (2). It then performs the second pairwise coupling on the (square) array in $N_{\gamma-\alpha+1}$ to N_γ, where the coupled results in $N_{\gamma-\alpha+1}$ to N_α from the first

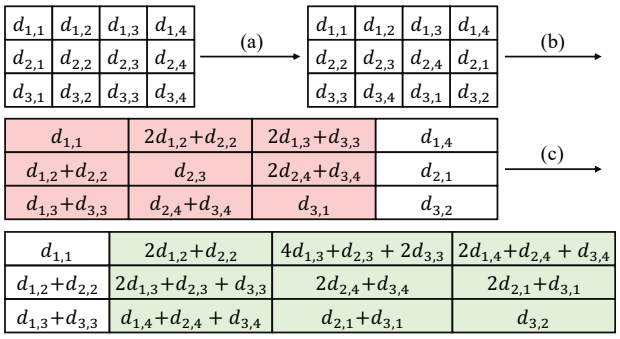


Figure 5: Example of transformation on a non-square array for $\alpha = 3$ and $\gamma = 4$: (a) rotation; (b) pairwise coupling on the first overlapped square array; and (c) pairwise coupling on the second overlapped square array.

N_1	N_2	N_3	N_4	N_5	N_6	N_7
$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$d_{1,4}$	$f_1(\boldsymbol{d_1})$	$f_2(\boldsymbol{d_1})$	$f_3(\boldsymbol{d_1})$
$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$d_{2,4}$	$f_1(\boldsymbol{d_2})$	$f_2(\boldsymbol{d_2})$	$f_3(\boldsymbol{d_2})$
$d_{3,1}$	$d_{3,2}$	$d_{3,3}$	$d_{3,4}$	$f_1(\boldsymbol{d_3})$	$f_2(\boldsymbol{d_3})$	$f_3(\boldsymbol{d_3})$

$\boldsymbol{d}_1 = (d_{1,1},\ d_{1,2}-d_{2,1},\ d_{1,3}-d_{2,2}-d_{3,1},\ d_{1,4}-d_{3,2})$,
$\boldsymbol{d}_2 = (d_{2,4}-d_{3,3},\ 2d_{2,1}-d_{1,2},\ 2d_{2,2}-d_{1,3},\ d_{1,4}+d_{2,3}-2d_{3,2})$,
$\boldsymbol{d}_3 = (2d_{3,3}-d_{2,4},\ d_{3,4},\ 2d_{3,1}-d_{2,2}-d_{1,3},\ 4d_{3,2}-d_{2,3}-d_{1,4})$

Figure 6: Example of transformation on a N_1 to N_4 through a 3×4 non-square transformation in RS(7,4).

pairwise coupling are used as the inputs to the second pairwise coupling.

Figure 5 shows an example of the transformation on a non-square array for $\alpha = 3$ and $\gamma = 4$.

- *Step 3 (Systematic transformation):* Similar to the square arrays, it expresses each original data sub-packet $d_{i,j}$ as a function of the non-systematic data sub-packets obtained after Steps 1 and 2. It then maps each non-systematic data sub-packet to the original data sub-packet, and replaces the inputs in the parity sub-packets accordingly.

Example. We show how the repair bandwidth is reduced by applying a non-square array. Figure 6 shows the systematic code after we apply a 3×4 non-square array to the packets in N_1 to N_4 in RS(7,4). Suppose that N_1 fails. The repair of N_1 only retrieves $d_{1,2}, d_{1,3}, d_{1,4}, d_{2,2}, d_{3,2}, f_1(\boldsymbol{d}_1), f_2(\boldsymbol{d}_1)$, and $f_3(\boldsymbol{d}_1)$ to repair all sub-packets in N_1. It first uses $d_{1,4}$ and $d_{3,2}$ to compute $d_{1,4}-d_{3,2}$. It uses $d_{1,4}-d_{3,2}$ as an input to $f_1(\boldsymbol{d}_1)$, $f_2(\boldsymbol{d}_1)$, and $f_3(\boldsymbol{d}_1)$, and solves them for $d_{1,1}$, $d_{1,2}-d_{2,1}$, and $d_{1,3}-d_{2,2}-d_{3,1}$. Finally, it uses $d_{1,2}$, $d_{1,3}$, and $d_{2,2}$ to solve for all sub-packets $d_{1,1}$, $d_{2,1}$, and $d_{3,1}$ in N_1. The repair bandwidth reduces from 12 (in RS(7,4)) to eight (i.e., 33.3% less).

Discussion. We point out that after we apply a transformation array to a subset of packets, the repair bandwidth for other non-transformed packets remains unchanged. We provide the intuitive reasons as follows. With the rotation step (Equation (1)), we ensure that the coupled sub-packets in fact correspond to the original sub-packets in the *same* node (e.g., the coupled sub-packets $2d_{1,2}+d_{2,2}$ and $d_{1,2}+d_{2,2}$ correspond to $d_{1,2}$ and $d_{2,2}$ in N_2). When we retrieve sub-packets from some node (say N) in a transformation array to repair a non-transformed node, we either (i) retrieve both coupled sub-packets whose original sub-packets are in N or (ii) retrieve the sub-packet directly if it has no other coupled sub-packet. Thus, *we do not retrieve extra sub-packets in the repair after transformation*. For example, in Figure 3, we retrieve $2d_{1,2}+d_{2,2}$, $d_{1,2}+d_{2,2}$, and $d_{3,2}$, which correspond to the sub-packets $d_{1,2}$, $d_{2,2}$, and $d_{3,2}$ in N_2. The formal proof can be based on [20, Theorem 3], and we omit the details here in the interest of space.

B. Applications of Elastic Transformation

We show how elastic transformation is applied to general erasure codes. We start with RS(n,k), and show how it is applied to other codes.

Application to RS. Recall that RS(n,k) always retrieves k packets to repair a single lost packet. We apply elastic transformation to *all* n packets in RS(n,k) into RS-ET(n,k) with less repair bandwidth. We first consider a small α, where $2 \le \alpha \le n-k$, and later extend our analysis with $\alpha > n-k$.

Given an α (where $2 \le \alpha \le n-k$), we first generate α sub-stripes of RS(n,k), where each sub-stripe is independently encoded. We divide the k data packets into $\lfloor \frac{k}{\alpha} \rfloor$ groups, such that the first $\lfloor \frac{k}{\alpha} \rfloor - 1$ groups are $\alpha \times \alpha$ square arrays, while the last group is an $\alpha \times \alpha$ square array if k is divisible by α, or an $\alpha \times \gamma$ non-square array otherwise (where $\gamma = k \bmod \alpha + \alpha$). Similarly, we divide the $n-k$ parity packets into groups. Then we perform rotation and pairwise coupling for each transformation array. We also apply systematic transformation to the data sub-packets and update the parity sub-packets accordingly.

To repair a lost packet in RS-ET(n,k), we perform the following steps:

- *Step 1 (Selecting the major sub-stripe):* We first select the s-th sub-stripe (where $1 \le s \le \alpha$) as the *major sub-stripe*, which determines the sub-packets to be retrieved for the repair. Suppose that the lost packet resides in N_f (where $1 \le f \le n$). We derive s as:

$$s = \begin{cases} ((f-1) \bmod \alpha)+1 & \text{for } 1 \le f \le k-\alpha, \\ ((f-k+\alpha-1) \bmod \alpha)+1 & \text{for } k-\alpha+1 \le f \le k, \\ ((f-k-1) \bmod \alpha)+1 & \text{for } k+1 \le f \le n-\alpha, \\ ((f-n+\alpha-1) \bmod \alpha)+1 & \text{for } n-\alpha+1 \le f \le n. \end{cases} \quad (4)$$

  The first and second cases choose the sub-stripe corresponding to the modulo index of f in a square array of the data packets; if k is not divisible by α, the second case chooses the sub-stripe based on the last overlapping square array. The third and fourth cases are for the parity packets.
- *Step 2 (Retrieving sub-packets):* We retrieve three sets of sub-packets: (i) $\mathcal{S}_1$, which includes any k out of $n-\alpha$ sub-packets in the major sub-stripe, except the lost sub-packet and $\alpha-1$ sub-packets coupled with the lost sub-packets (i.e., α sub-packets in total); (ii) $\mathcal{S}_2$, which includes the coupled sub-packets for the k sub-packets in $\mathcal{S}_1$; and (iii) $\mathcal{S}_3$, which includes the sub-packets that are coupled with the lost sub-packets. Note that the repair bandwidth can slightly vary for different subsets of k sub-packets being retrieved (§IV-A).

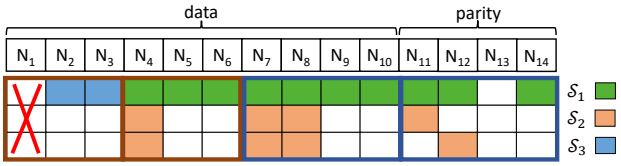


Figure 7: Elastic transformation to RS$(14,10)$ for $\alpha = 3$.

- *Step 3 (Repairing the inputs to the data vector):* We use the retrieved sub-packets in $\mathcal{S}_1$ and $\mathcal{S}_2$ to repair the k inputs to the data vector $\boldsymbol{d}_s$.
- *Step 4 (Repairing the lost sub-packets):* We use the sub-packets in $\mathcal{S}_3$, together with the k inputs resolved from Step 3, to repair all lost sub-packets.

Figure 7 shows an example of applying elastic transformation to all packets of RS$(14,10)$ for $\alpha = 3$ and how the sub-packets are retrieved for repairing N_1.

We can extend elastic transformation for RS codes for $\alpha > n-k$, where α is some composite number. Suppose that $\alpha = \alpha_1 \cdot \alpha_2$, for some $2 \le \alpha_1, \alpha_2 \le n-k$. We first apply an $\alpha_1 \times \alpha_1$ square array to the packets in N_1 to N_{α_1}. We treat the transformed code as a base code by aggregating its α_1 sub-stripes as one big sub-stripe, and apply elastic transformation to the packets in the remaining nodes N_{α_1+1} to N_n for α_2. If the number of remaining nodes is less than α_2 (i.e., $n-\alpha_1 < \alpha_2$), we apply an $\alpha_2 \times \alpha_2$ square array to nodes $N_{n-\alpha_2+1}$ to N_n, in which nodes $N_{n-\alpha_2+1}$ to N_{α_1} have been transformed (similar to applying an overlapped square array to a non-square array). In general, we can construct a transformed code for any $\alpha = \prod_i \alpha_i$, where $2 \le \alpha_i \le n-k$.

Elastic transformation can convert a base code into an access-optimal MSR code as in [20] when $\alpha = (n-k)^{\lceil n/(n-k) \rceil}$. Specifically, we apply elastic transformation $\lceil \frac{n}{n-k} \rceil$ times. In the i-th transformation (where $1 \le i \le \lceil \frac{n}{n-k} \rceil - 1$), we apply an $(n-k) \times (n-k)$ square array to nodes $N_{(i-1)(n-k)+1}$ to $N_{i(n-k)}$, while in the last transformation, we apply an $(n-k) \times (n-k)$ square array to nodes N_{k+1} to N_n. The transformed code is the same as the access-optimal MSR code in [20].

Application to other codes. We can apply elastic transformation to other codes to reduce the repair bandwidth for a subset of packets. For example, LRC [13] reduces the repair bandwidth (via local repair) for data and local parity packets, but still retrieves k packets to repair global parity packets; Hitchhiker [31] and HashTag [18] only reduce the repair bandwidth for the data packets, but not the parity packets. We can apply transformation arrays to cover the global parity packets for LRC and the parity packets for Hitchhiker and HashTag. Note that we do not need to apply systematic transformation to parity packets, which are originally coded.

C. Proof of Fault Tolerance

We prove that elastic transformation preserves fault tolerance. Specifically, we prove that when elastic transformation is applied to an MDS base code, there exists a lower bound of the field size that the transformed code is still MDS. Our proof is similar to those in prior work [3], [12], [18], yet the lower bound is different as our problem setting is different.

Theorem 1. *If the size q of a finite field $\mathbb{F}_q$ is larger than*

$$2\left(\binom{n-1}{k-1} - \binom{\lceil n/\alpha \rceil - 1}{\lceil k/\alpha \rceil - 1}\right), \tag{5}$$

then there exist coefficients $e_{i,j}$'s (see Equation (2)) over $\mathbb{F}_q$ to ensure that the transformed code is MDS.

Proof. Suppose that the base code is an (n,k) MDS code and the transformed code has a sub-packetization level α. We can view the transformed code as encoding $k\alpha$ data sub-packets over $\mathbb{F}_q$ into $n\alpha$ encoded sub-packets. Each encoded sub-packet can be expressed as a linear equation with $k\alpha$ variables, and there are in total $n\alpha$ equations for all encoded sub-packets (each of the n nodes is associated with α equations). We require that the original $k\alpha$ data sub-packets can be reconstructed from a system of $k\alpha$ equations in any k out of n nodes to solve for the $k\alpha$ variables. To do so, the system of $k\alpha$ equations needs to be linearly independent; in other words, the determinant of the coefficient matrix of the system of equations is non-zero.

There are $\binom{n}{k}$ cases for selecting any k out of n nodes. For each case, the determinant of the coefficient matrix is a polynomial over $e_{i,j}$ (Equation (2)). For a square array, there is one choice for each $e_{i,j}$ as shown in Equation (2); for a non-square array, if we treat it as two overlapped square arrays, there are up to two choices for each $e_{i,j}$. Thus, the degree of each $e_{i,j}$ in the polynomial is at most two. For all $\binom{n}{k}$ cases, we can form a polynomial (as the product of polynomials for each case). Each $e_{i,j}$ will appear up to $\binom{n-1}{k-1}$ times of all cases. Thus, the degree of each $e_{i,j}$ in the polynomial (denoted by $\theta_{i,j}$) will satisfy $\theta_{i,j} \le 2\binom{n-1}{k-1}$. By [4, Theorem 1.2], if the field size q is larger than each $\theta_{i,j}$, there exists a non-zero solution for the polynomial. Thus, the original data sub-packets can be decoded (i.e., the MDS property is achieved).

Note that there exist cases where the encoded sub-packets for a specific set of k nodes can directly reconstruct all data sub-packets without using $e_{i,j}$. From [20, Theorem 1], if we read the two coupled sub-packets together, we can directly solve for the two original sub-packets. Thus, when the k nodes include square or non-square arrays in entirety, the data sub-packets can be reconstructed without $e_{i,j}$. For example, for RS-ET$(9,6)$ with $\alpha = 3$, if we choose the k nodes as N_4 to N_9, we have two complete 3×3 square arrays. Then we can directly decode all sub-packets in N_4 to N_9 from their coupled sub-packets without $e_{i,j}$, and all data sub-packets in N_1 to N_6 can be reconstructed. There are $\binom{\lceil n/\alpha \rceil - 1}{\lceil k/\alpha \rceil - 1}$ such special cases. By subtracting them from the $\binom{n-1}{k-1}$ cases, the theorem follows. □

Theorem 1 provides a smaller lower bound than reported in prior studies, such as $\binom{n}{k}(n-k)^{\alpha+1}$ [3], $\binom{n}{k}\frac{\alpha(\alpha-1)}{2}\lfloor \frac{k}{\alpha} \rfloor$ [12], and $\binom{n}{k}(n-k)\alpha$ [18]. For example, for $(n,k) = (16,12)$ and $\alpha = 4$, their lower bounds are 1,863,680, 32,760, and 29,120, respectively, while ours is 2,724.

While Theorem 1 states that elastic transformation maintains the MDS property for a sufficiently large field size, we find (by enumerating all $\binom{n}{k}$ combinations) that the MDS property is achievable in GF(2^8) with $e_{i,j} = 2$ (for all i and j) for medium

ranges of (n,k) (e.g., up to $(n,k)=(16,12)$). Thus, elastic transformation can be feasibly implemented in practice.

IV. Numerical Analysis

We conduct numerical analysis on elastic transformation. We prove the lower bound of repair bandwidth for a given α under elastic transformation, for RS(n,k) and LRC(n,k,ℓ). We show that our elastic transformation implementation, OpenEC-ET, can match closely to the lower bound. We further show that when the I/O overhead is also considered, the actual repair performance can be negated as α increases.

A. Modeling of Single-Packet Repair Bandwidth

RS. Theorem 2 models the lower bound of repair bandwidth for any single lost packet in RS(n,k). Here, we focus on $\alpha \le n-k$. We can extend the theorem for $\alpha > n-k$ by expressing α as $\prod_i \alpha_i$, where $2 \le \alpha_i \le n-k$ (§III-B). We obtain the bound for each α_i from Theorem 2 and sum all the bounds.

Theorem 2. *Suppose that we apply elastic transformation to RS(n,k) into RS-ET(n,k) for all n packets, with $\alpha \le n-k$. The repair bandwidth for a single lost packet (denoted by β) is lower bounded by (in number of sub-packets):*

$$\beta \ge \begin{cases} k+\alpha-1 & \text{for } k < \lfloor \frac{n}{\alpha} \rfloor - 1, \\ 2k - \lfloor \frac{n}{\alpha} \rfloor + \alpha & \text{for } k \ge \lfloor \frac{n}{\alpha} \rfloor - 1. \end{cases} \quad (6)$$

Proof. We prove the theorem by counting the numbers of sub-packets in $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$.

For $\mathcal{S}_1$, $|\mathcal{S}_1| = k$, as we always retrieve k sub-packets.

For $\mathcal{S}_2$, we first identify the property that in an $\alpha \times \alpha$ square array, there exist one non-coupled sub-packet and $\alpha-1$ coupled sub-packets in each sub-stripe (Equation (2)). Note that $|\mathcal{S}_2|$ depends on the sub-packets in $\mathcal{S}_1$ and decreases when there are more non-coupled sub-packets in $\mathcal{S}_1$. If $k < \lfloor \frac{n}{\alpha} \rfloor - 1$, we can select k non-coupled sub-packets from any k out of $\lfloor \frac{n}{\alpha} \rfloor - 1$ square arrays, so $|\mathcal{S}_2| = 0$. Otherwise, if $k \ge \lfloor \frac{n}{\alpha} \rfloor - 1$, we can select $\lfloor \frac{n}{\alpha} \rfloor - 1$ non-coupled sub-packets from the $\lfloor \frac{n}{\alpha} \rfloor - 1$ square arrays (as shown in the property) and $k - (\lfloor \frac{n}{\alpha} \rfloor - 1)$ coupled sub-packets in $\mathcal{S}_1$ from $\lfloor \frac{n}{\alpha} \rfloor - 1$ square arrays without the lost packet. We retrieve the corresponding coupled sub-packet from each of the $k - (\lfloor \frac{n}{\alpha} \rfloor - 1)$ coupled sub-packets into $\mathcal{S}_2$, so $|\mathcal{S}_2| \ge k - (\lfloor \frac{n}{\alpha} \rfloor - 1)$.

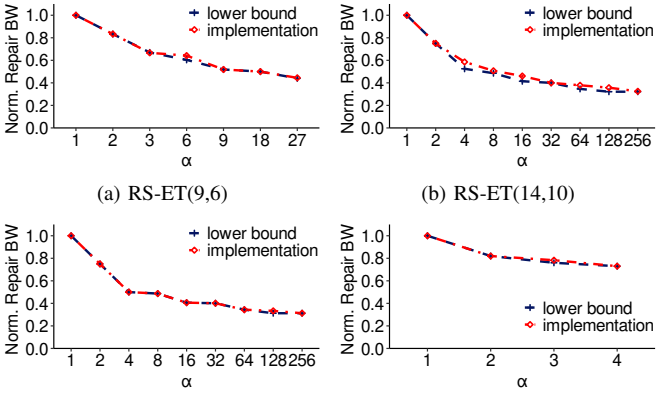
For $\mathcal{S}_3$, it contains at least $\alpha-1$ sub-packets coupled with the α lost sub-packets. Thus, we obtain $|\mathcal{S}_3| \ge \alpha - 1$.

By summing up the bounds for $|\mathcal{S}_1|$, $|\mathcal{S}_2|$, and $|\mathcal{S}_3|$, the theorem follows. □

LRC. We next consider LRC, which does not optimize the repair for global parity packets (§II-B). We apply elastic transformation to only the global parity packets.

Theorem 3. *Suppose that we apply elastic transformation to the $n-k-\ell$ global parity packets in LRC(n,k,ℓ) into LRC-ET(n,k,ℓ), with $\alpha \le n-k-\ell$. The repair bandwidth for a single lost global parity packet (denoted by β') is lower bounded by (in number of sub-packets):*

$$\beta' \ge k + \alpha - 1. \quad (7)$$



(a) RS-ET(9,6) (b) RS-ET(14,10)

(c) RS-ET(16,12) (d) LRC-ET(16,10,2)

Figure 8: Modeled single-packet repair bandwidth, normalized with respect to $\alpha = 1$.

Proof. We count the numbers of sub-packets in $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$. The bounds of $|\mathcal{S}_1|$ and $|\mathcal{S}_3|$ are the same as in Theorem 2. For $\mathcal{S}_2$, as we only apply transformation to global parity packets, we can select k data sub-packets in the major sub-stripe to form $\mathcal{S}_1$. The k data sub-packets are not transformed, so there is no coupled sub-packet and $|\mathcal{S}_2| = 0$. By summing up the bounds for $|\mathcal{S}_1|$, $|\mathcal{S}_2|$, and $|\mathcal{S}_3|$, the theorem follows. □

The average repair bandwidth of LRC-ET(n,k,ℓ) (in number of sub-packets) is hence given by $((k+\ell)\frac{k\alpha}{\ell} + (n-k-\ell)\beta')/n$.

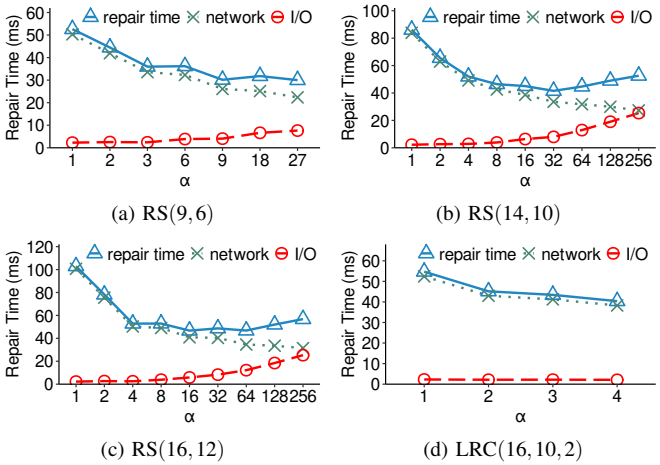
Analysis. We plot the lower bounds of repair bandwidth for RS-ET and LRC-ET versus α. We also show the repair bandwidth achieved by our implementation, OpenEC-ET (§V), in which we select the k sub-packets with the least coupled sub-packets in the major sub-stripe into $\mathcal{S}_1$, so as to reduce the number of coupled sub-packets to be retrieved into $\mathcal{S}_2$.

Figures 8(a)-8(c) show the lower bound and the repair bandwidth of OpenEC-ET for RS-ET(n,k), normalized with respect to RS(n,k) (i.e., $\alpha = 1$). The repair bandwidth decreases with α, and reaches the access-optimal MSR point when $\alpha = (n-k)^{\lceil \frac{n}{n-k} \rceil}$. OpenEC-ET matches closely the lower bound. For example, RS-ET$(14,10)$ reduces the repair bandwidth of RS$(14,10)$ by 25.0-67.5% for $2 \le \alpha \le 256$.

Figure 8(d) shows the repair bandwidth results for LRC-ET$(16,10,2)$, normalized with respect to LRC$(16,10,2)$ (the same parameters are also considered in [16]). By reducing the repair bandwidth for global parity packets, LRC-ET$(16,10,2)$ reduces the average repair bandwidth for all packets of LRC$(16,10,2)$ by up to 27% (for $\alpha = 4$).

B. Modeling of Single-Packet Repair Time

We now model the single-packet repair time by taking into account both network transmissions and I/Os, and show how it can be adversely affected by non-sequential I/Os. Let b be the available network bandwidth, $R(\alpha)$ be the average single-packet repair bandwidth (in number of sub-packets) (§IV-A), p be the packet size, $\phi(\alpha)$ be the number of alive nodes from which the repair operation retrieves available packets, and $\tau(\alpha)$ be the time to read a sub-packet of size $\frac{p}{\alpha}$. Our model assumes



(a) RS(9,6) (b) RS(14,10)

(c) RS(16,12) (d) LRC(16,10,2)

Figure 9: Modeled single-packet repair time versus α.

that coding computations have negligible overhead compared with network transmissions and I/Os.

RS. We first consider RS(n,k). We derive $R(\alpha)$ based on OpenEC-ET (which is close to the lower-bound repair bandwidth) from §IV-A, and set $\phi(\alpha) = \min(n-1, k+\alpha-1)$ based on the repair steps in §III-B. For $\tau(\alpha)$, since it depends on the network bandwidth and the sub-packet size $\frac{p}{\alpha}$, we measure the read time for a sub-packet for different α's on HDFS in our testbed (see §V for testbed details). For example, when the network bandwidth is 1 Gb/s and the packet size is 1 MiB, we have $\tau(\alpha) = 2.2$, 1.5, 1.1, 0.7, and 0.4 (in ms) for $\alpha = 1$, 4, 16, 64, and 256, respectively. Thus, the single-packet repair time T is modeled as:

$$T = \frac{R(\alpha)p}{b\alpha} + \frac{R(\alpha)}{\phi(\alpha)}\tau(\alpha), \tag{8}$$

where the first term represents the network transmission time, and the second term represents the I/O time. For the second term, we assume that we issue reads to $\frac{R(\alpha)}{\phi(\alpha)}$ sub-packets evenly from each of the $\phi(\alpha)$ alive nodes to simplify our analysis, although the numbers of sub-packets read from alive nodes are generally different (e.g., see Figure 7).

OpenEC-ET enhances read performance by issuing a read to multiple sub-packets that are sequentially placed instead of reading sub-packets individually, so as to issue fewer reads. Also, it issues reads to multiple packets within a block in parallel via multi-threading. Thus, our model overestimates the I/O time in OpenEC-ET, yet it still effectively captures the trend of repair time versus α, as shown in our evaluation (§V).

LRC. We also model the average single-packet repair time of LRC(n,k,ℓ) based on Equation (8). To repair a data packet or a local parity packet, the repair time (denoted by T_1) is $T_1 = \frac{kp}{\ell b} + \tau(1)$ (as we read a single packet from each of the $\frac{k}{\ell}$ alive nodes). To repair a global parity packet, we derive its repair time (denoted by T_2) by substituting $R(\alpha)$ in Equation (8) with the repair bandwidth for a global parity packet in OpenEC-ET (§IV-A). Thus, the average single-packet repair time of LRC is $((k+\ell)T_1 + (n-k-\ell)T_2)/n$.

Analysis. Figure 9 shows the single-packet repair time versus α, where $b = 1$ Gb/s and $p = 1$ MiB; it also shows a breakdown for the network and I/O times. Elastic transformation initially reduces the repair time as α increases from $\alpha = 1$ by reducing the repair bandwidth, but the repair time starts to increase for large α due to I/O overhead, especially for RS-ET(14,10) and RS-ET(16,12) at the access-optimal MSR point $\alpha = 256$. For example, RS-ET$(14,10)$ reduces the repair time by up to 51.8% for $\alpha = 32$ compared with RS$(14,10)$, but the reduction drops to 39.0% for $\alpha = 256$. LRC-ET(16,10,2) reduces the repair time by up to 26.2% for $\alpha = 4$, as α remains small.

V. Testbed Evaluation

We conduct experiments for elastic transformation in a real local cluster. We aim to address two questions: (i) Does the practical repair performance match the numerical analysis results? (ii) How do sub-packetization, network bandwidth, and packet size affect the practical repair performance?

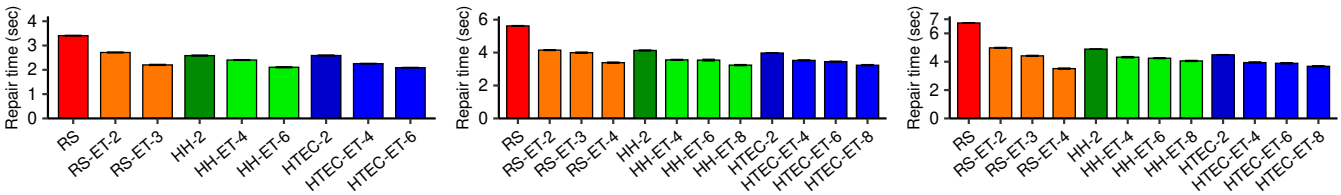
Implementation. We implemented elastic transformation with OpenEC [23], a middleware system realizing erasure coding in the form of direct acyclic graphs atop HDFS (on Hadoop 3.0.0 [1]). Our implementation, OpenEC-ET, supports RS codes [32], LRC [13], Hitchhiker [31], and HashTag [18]. OpenEC-ET is written in C++ and uses ISA-L [2] to implement erasure coding operations. It adds 4.8 K LoC to OpenEC.

HDFS comprises a single NameNode for storage management and multiple DataNodes for data storage. Recall that HDFS organizes data in *blocks*, each of which contains multiple packets (§II-A). Suppose that we repair a single lost block and store the repaired block in a DataNode. The DataNode first queries the NameNode for the DataNodes that contain the available blocks. It then retrieves the sub-packets from the DataNodes and decodes the lost block.

Methodology. We conduct experiments in a local cluster with 17 machines, each of which has a quad-core 3.4 GHz Intel Core i5 CPU, 32 GiB RAM, and a 7200 RPM 1 TB SATA hard disk. All machines are connected via a 10 Gb/s Ethernet switch. We assign one machine as the NameNode and the remaining machines as DataNodes. We use Wondershaper [14] to configure the network bandwidth in each machine. By default, we set the HDFS block size and packet sizes as 64 MiB and 1 MiB, respectively, and the network bandwidth as 1 Gb/s as in prior work [23], [33]. We also evaluate the impact of network bandwidth (Experiment 2) and packet size (Experiment 3).

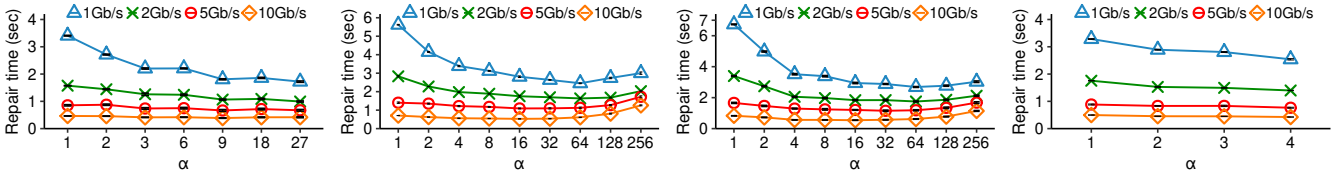
We evaluate the repair performance based on the *single-block repair time*, defined as the time from issuing a repair request until a failed block is repaired. Specifically, we first write n stripes to HDFS. We erase one block in each stripe and ensure that each of the n nodes has exactly one erased block, so that the repair operation includes both data and parity blocks. We measure the single-block repair time by averaging the time to repair each of the n blocks. We obtain the average results over 10 runs, with an error bar showing the 95% confidence interval based on the student's t-distribution; most error bars are invisible due to negligible deviations.

Experiment 1 (Overall repair performance). We study how elastic transformation affects the single-block repair time of different codes. We focus on MDS codes, including RS codes,



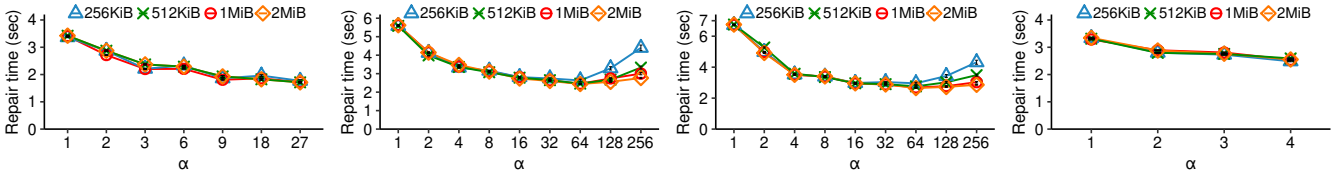
(a) $(n,k)=(9,6)$ (b) $(n,k)=(14,10)$ (c) $(n,k)=(16,12)$

Figure 10: Experiment 1: Overall repair performance. The number next to each code represents the sub-packetization level ('ET' means with elastic transformation; 'HH' means Hitchhiker; 'HTEC' means HashTag).



(a) RS; $(n,k)=(9,6)$ (b) RS; $(n,k)=(14,10)$ (c) RS; $(n,k)=(16,12)$ (d) LRC; $(n,k,\ell)=(16,10,2)$

Figure 11: Experiment 2: Impact of sub-packetization and network bandwidth.



(a) RS; $(n,k)=(9,6)$ (b) RS; $(n,k)=(14,10)$ (c) RS; $(n,k)=(16,12)$ (d) LRC; $(n,k,\ell)=(16,10,2)$

Figure 12: Experiment 3: Impact of packet size.

Hitchhiker, and HashTag, so that they are compared with the same storage overhead for a given (n,k). We configure the base codes of RS codes, Hitchhiker, and HashTag with sub-packetization levels as one, two, and two, respectively, and apply elastic transformation that increases their sub-packetization levels by at most $n-k$ times.

Figure 10 shows the single-block repair time for different (n,k)'s. Elastic transformation reduces the repair time of the base codes by reducing the repair bandwidth while limiting the sub-packetization overhead. For example, for $(14,10)$, elastic transformation reduces the repair times of RS codes, Hitchhiker, and HashTag by 26.2-39.8%, 13.9-21.6%, and 11.3-18.6%, respectively.

Experiment 2 (Impact of sub-packetization and network bandwidth). We apply elastic transformation to all packets in RS codes and global parity packets in LRC. We study the impact of sub-packetization and network bandwidth.

Figure 11 shows the single-block repair time results of RS$(9,6)$, RS$(14,10)$, RS$(16,12)$, and LRC$(16,10,2)$. We first examine the impact of α. The repair time trend in the empirical results under 1 Gb/s matches our numerical analysis (§IV-B); that is, elastic transformation reduces the repair time of RS codes and LRC for a small α, but the repair time increases for a large α. For example, when the network bandwidth is 1 Gb/s, RS-ET(14,10) with $\alpha=64$ reduces the repair times of RS(14,10) (i.e., $\alpha=1$) and the access-optimal MSR point (i.e., $\alpha=256$) by 56.3% and 18.3%, respectively.

We also examine the impact of network bandwidth. As the network bandwidth increases, the access-optimal MSR point suffers from non-sequential I/O overhead. For example, when the network bandwidth is 10 Gb/s, RS-ET$(14,10)$ with $\alpha=64$ reduces the repair times of RS$(14,10)$ and the access-optimal MSR point by 13.4% and 51.4%, respectively.

Experiment 3 (Impact of packet size). We study the impact of packet size on the single-block repair time, where the packet size varies from 256 KiB to 2 MiB. Figure 12 shows the repair time results. A small packet size (e.g., 256 KiB) aggravates the non-sequential I/O overhead. For example, RS-ET$(14,10)$ with $\alpha=64$ reduces the repair time of the access-optimal MSR point by 40.1% when the packet size is 256 KiB.

VI. Conclusions

We propose an elastic transformation framework to transform a base code into a code with less repair bandwidth subject to a configurable sub-packetization level, so as to limit non-sequential I/Os. We study the problem from both theoretical and applied perspectives. We present a fault tolerance proof and the modeling of repair performance. We also prototype elastic transformation on HDFS. Both numerical analysis and testbed experiments demonstrate the benefits of elastic transformation.

Acknowledgements: This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No. 2021YFB0301400), Key Laboratory of Information Storage System Ministry of Education of China, Research Grants Council of HKSAR (AoE/P-404/18), and Research Matching Grant Scheme. The corresponding author is Patrick P. C. Lee.

REFERENCES

[1] Apache Hadoop 3.0.0. https://hadoop.apache.org/docs/r3.0.0/.

[2] ISA-L. https://github.com/intel/isa-l.

[3] G. K. Agarwal, B. Sasidharan, and P. V. Kumar. An alternate construction of an access-optimal regenerating code with optimal sub-packetization level. In *Proc. of the 21st NCC*, 2015.

[4] N. Alon. Combinatorial nullstellensatz. *Combinatorics, Probability and Computing*, 8(1-2):7–29, 1999.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, Aug 2018.

[6] S. B. Balaji and P. V. Kumar. A tight lower bound on the sub-packetization level of optimal-access MSR and MDS codes. In *Proc. of IEEE ISIT*, 2018.

[7] B. Beach. Backblaze Vaults: Zettabyte-scale cloud storage architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2019.

[8] H. C. H. Chen, Y. Hu, P. P. C. Lee, and Y. Tang. NCCloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Trans. on Computers*, 63(1):31–44, 2014.

[9] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.

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

[11] H. Hou and P. P. Lee. Binary MDS array codes with optimal repair. *IEEE Trans. on Information Theory*, 66(3):1405–1422, 2019.

[12] H. Hou, P. P. Lee, and Y. S. Han. Multi-layer transformed MDS codes with optimal repair access and low sub-packetization. *arXiv preprint arXiv:1907.08938*, 2019.

[13] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.

[14] B. Hubert, J. Geul, and S. Séhier. Wonder Shaper. https://github.com/magnific0/wondershaper, 2012.

[15] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

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

[17] K. Kralevska and D. Gligoroski. An explicit construction of systematic MDS codes with small sub-packetization for all-node repair. In *Proc. of IEEE DASC/PiCom/DataCom/CyberSciTech*, 2018.

[18] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby. Hashtag erasure codes: From theory to practice. *IEEE Trans. on Big Data*, 4(4):516–529, 2018.

[19] J. Li, X. Tang, and C. Hollanti. A generic transformation for optimal node repair in mds array codes over f2. *IEEE Trans. on Communications*, 2021.

[20] J. Li, X. Tang, and C. Tian. A generic transformation to enable optimal repair in MDS codes for distributed storage systems. *IEEE Trans. on Information Theory*, 64(9):6257–6267, 2018.

[21] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[22] R. Li, J. Lin, and P. P. C. Lee. Enabling concurrent failure recovery for regenerating-coding-based storage systems: From theory to practice. *IEEE Trans. on Computers*, 64(7):1898–1911, Jul 2015.

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

[24] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR) a distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[25] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, pages 383–398, 2014.

[26] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[27] A.-J. Peters, M. K. Simon, and E. A. Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.

[28] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[29] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[30] K. Rashmi, N. B. Shah, and K. Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Trans. on Information Theory*, 63(9):5802–5820, 2017.

[31] 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 *Proc. of ACM SIGCOMM*, 2014.

[32] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[33] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. *Proc. of the VLDB Endowment*, 6(5):325–336, 2013.

[34] Y. Shan, K. Chen, T. Gong, L. Zhou, T. Zhou, and Y. Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proc. of ACM SOSP*, 2021.

[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[36] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[37] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, 2002.