# LESS is More for I/O-Efficient Repairs in Erasure-Coded Storage

Keyun Cheng[1*], Guodong Li[2*], Xiaolu Li[3], Sihuang Hu[2], and Patrick P. C. Lee[1]

[1]*The Chinese University of Hong Kong* [2]*Shandong University*
[3]*Huazhong University of Science and Technology*

## Abstract

I/O efficiency is critical for erasure-coded repair performance in modern distributed storage. We propose LESS, a family of repair-friendly erasure code constructions that reduces both the amount of data accessed and the number of I/O seeks in single-block repairs, while ensuring balanced reductions across blocks. LESS layers multiple extended sub-stripes formed by widely deployed Reed-Solomon coding, and is configurable to balance the trade-off between the amount of data accessed and I/O seeks. Evaluation shows that LESS on HDFS reduces both single-block repair and full-node recovery times compared to state-of-the-art I/O-optimal erasure codes.

## 1 Introduction

Erasure coding is widely adopted in modern distributed storage systems (e.g., [3, 5, 10, 21]) to ensure fault tolerance with significantly lower storage overhead than traditional replication [37]. However, erasure coding incurs high repair costs. Reconstructing a single failed block needs to access and transfer much more data than the block size, leading to amplified bandwidth and I/O costs. Recent surveys [1, 33] highlight extensive studies, from both coding theory and systems communities, that aim to improve repair efficiency in erasure-coded storage. These studies focus on improving access performance (e.g., optimizing degraded reads [6, 13]) and storage reliability (e.g., reducing mean-time-to-data-loss [9, 10]). In particular, extensive research has proposed repair-friendly erasure code constructions to mitigate bandwidth and I/O costs (see §2.2 for related work).

In modern distributed storage systems, I/O efficiency has become an increasingly critical design factor compared to bandwidth efficiency due to rapid advancements in network technologies (e.g., InfiniBand, RDMA, and CXL) that support high-speed interconnects. However, I/O performance cannot match such network improvements, especially under random-access workloads. This motivates us to construct repair-friendly erasure codes that make I/O efficiency a "first-class citizen" for high repair performance.

Existing repair-friendly erasure codes often have sub-optimal I/O performance in repairs (§2.2). For example, Clay codes [36] are state-of-the-art minimum-storage regenerating (MSR) codes that provably minimize the amount of data accessed from local storage in single-block repairs, while minimizing storage redundancy for fault tolerance. Their core idea is *sub-packetization*, which divides an erasure-coded *stripe* (i.e., a set of blocks encoded together) into smaller *sub-stripes* composed of sub-blocks at the same block offset. During a single-block repair, Clay codes retrieve a minimum number of sub-blocks across sub-stripes for reconstruction. Despite minimizing data access, Clay codes require exponential sub-packetization for optimality, and issue numerous non-contiguous I/O seeks to retrieve sub-blocks in a repair. This significantly degrades repair performance in I/O-constrained environments due to amplified I/O requests. Locally repairable codes (e.g., Azure's LRC [10]) achieve I/O-efficient repairs, but incur higher storage redundancy. Other erasure codes support small sub-packetization with minimum storage redundancy [15, 16, 29, 35], yet they have various limitations in code constructions that lead to repair inefficiency.

We present LESS, a family of erasure code constructions designed for I/O-efficient repairs, aiming to (i) reduce both the amount of data accessed and the number of I/O seeks with small sub-packetization (as low as *two* sub-stripes per stripe) and (ii) maintain balanced reductions across all blocks within a stripe (i.e., all blocks have similar data access and I/O seek costs). LESS builds on the notion of layering extended sub-stripes, by strategically stacking multiple extended sub-stripes (each with a longer stripe length) atop a regular stripe, such that a single-block repair (and some cases of multi-block repairs) can be done within an extended sub-stripe. LESS is designed with practicality in mind. It builds on the widely used Reed-Solomon (RS) codes [30] (§2.1) for a readily understandable design, and preserves the practical properties of RS codes. It further allows configurable sub-packetization to balance the trade-off between data access and I/O seeks.

We implement and evaluate LESS on HDFS [34] in a local cluster. LESS reduces the single-block repair and full-node recovery times of Clay codes by up to 83.3% and 36.6%, respectively. We release the source code of LESS at: https://github.com/adslabcuhk/less.

## 2 Background

### 2.1 Basics of Erasure Coding

We consider distributed storage systems that organize data in large fixed-size *blocks* (e.g., 128 MiB in HDFS [34] and 256 MiB in Facebook's f4 [21]) to mitigate I/O overhead. Such systems are typically I/O-bound, where network band-

[*] The first two authors contributed equally to this work.

width and disk I/Os (on the order of MiB/s) are primary performance bottlenecks, rather than the computation overhead of encoding and decoding, as observed in prior work [6,9,18,19]. Erasure coding ensures fault tolerance against block failures, which is essential for maintaining high data persistence in applications such as archival storage. We focus on RS codes [30], which are widely deployed in production [3,5,21,23]. RS codes are configured by two parameters $n$ and $k$ (where $n > k$). An $(n,k)$ RS code encodes $k$ uncoded data blocks into $n-k$ coded parity blocks, forming a *stripe* of $n$ blocks, and ensures that any $k$ out of $n$ blocks can reconstruct all $k$ data blocks (i.e., any $n-k$ failed blocks are tolerable). In distributed storage systems, multiple stripes are independently encoded, with each stripe distributed across $n$ nodes for tolerating any $n-k$ node failures.

RS codes satisfy three practical properties: (i) *maximum distance separable (MDS)*, i.e., the redundancy ($\frac{n}{k}$ times the data size) is minimized for tolerating any $n-k$ failed blocks; (ii) *general*, i.e., any $n$ and $k$ (where $n > k$) can construct RS codes for a sufficiently large field size; and (iii) *systematic*, i.e., each stripe keeps $k$ data blocks for direct access.

RS encoding and decoding use linear combinations over a Galois Field GF($2^w$) with $w$-bit words, where $n \leq 2^w + 1$ [25]. Each block can be expressed as a linear combination of any $k$ blocks under Galois Field arithmetic (see §3.1 for details).

RS codes incur high *repair bandwidth*, defined as the amount of data transferred across nodes during a repair. Repairing a single failed block needs to retrieve $k$ blocks of the same stripe from other nodes; we call this process *conventional repair*, which applies to all $(n,k)$ MDS codes. It also incurs high *repair I/O*, defined as the amount of data accessed from local storage during a repair. In high-speed networks, repair I/O becomes the dominant bottleneck (§1).

## 2.2 Related Work on Repair Optimization

Prior surveys [1,33] review extensive research on optimizing single-block repair performance in erasure-coded storage. We review several representative repair-friendly erasure codes and their limitations.

**Minimum-storage regenerating (MSR) codes.** MSR codes [4] minimize repair bandwidth for single-block repairs, while preserving the MDS property (i.e., minimum redundancy). They build on *sub-packetization*, which divides each block into $\alpha > 1$ sub-blocks and forms $\alpha$ *sub-stripes* of $n$ sub-blocks at the same offset across blocks. Each sub-block is a linear combination of other $n\alpha - 1$ sub-blocks in the stripe over GF($2^w$). A single-block repair transfers a fraction of sub-blocks, with the minimum repair bandwidth.

Traditional MSR codes [4] require nodes to read and encode all local sub-blocks, leading to high repair I/O. *I/O-optimal MSR codes* minimize repair I/O, by allowing accessed data to be sent directly without encoding. For example, Functional MSR (F-MSR) codes [6, 8] require $n-k = 2$ [8] or $n-k = 3$ [6], with a linear $\alpha = n-k$, but are non-systematic. PM-RBT codes [27] are systematic but require $n \geq 2k-1$, while Butterfly codes [22] are also systematic but require $n-k = 2$ and incur an exponential $\alpha = 2^{k-1}$.

Clay codes [36] are state-of-the-art I/O-optimal MSR codes that are systematic, support general $(n,k)$, and are deployed in Ceph [38]. However, Clay codes impose an exponential $\alpha = (n-k)^{\lceil n/(n-k) \rceil}$ (note that exponential sub-packetization is necessary for I/O-optimal MSR codes [2]). Even though Clay codes minimize repair I/O, they access numerous non-contiguous sub-blocks and incur substantial *I/O seeks*, defined as the number of non-contiguous reads to local storage in a repair. This leads to sub-optimal repair performance due to the overhead of handling extensive I/O requests [35].

**Locally repairable codes (LRCs).** LRCs [10,12,14,31] reduce repair I/O for single-block repairs. For example, Azure-LRC [10] partitions data blocks into local groups and adds a local parity block per local group. Repairing a failed data or local parity block accesses only the blocks of its local group. However, Azure-LRC encodes all data blocks of a stripe into global parity blocks, whose repairs still rely on conventional repair. Some LRC variants support local repairs for global parity blocks [14], but LRCs are non-MDS and have higher redundancy than RS codes.

**MDS codes with small sub-packetization.** To limit I/O seeks, some MDS codes allow a small $\alpha$ with reduced repair I/O. Hitchhiker codes [29] use $\alpha = 2$ sub-stripes of RS codes by combining sub-blocks across sub-stripes via piggybacking functions, and reduce repair I/O for data blocks by 25-45% compared to RS codes; however, parity-block repairs still follow conventional repair. HashTag codes [16] reduce repair I/O for data blocks with general $(n,k)$ and $\alpha \geq 2$; their extended codes [15] reduce repair I/O for both data and parity blocks, but require $\alpha \geq 4$ and $\alpha$ as a multiple of $n-k$.

Elastic Transformation (ET) [35] converts an RS code into a repair-friendly code with a configurable $\alpha \geq 2$. However, the transformation restricts flexibility in code construction and limits further repair improvements.

**Repair-efficient algorithms.** Our study focuses on erasure code construction, while some studies design repair-efficient algorithms, such as search of minimum-I/O recovery for XOR-based erasure codes [13,39] or repair parallelization [17,18, 20,32]. They do not minimize repair I/O due to constraints from underlying erasure codes.

## 2.3 Goals

Table 1 summarizes the repair-friendly codes and their limitations, including: non-MDS (e.g., Azure-LRC [10]), restrictive parameters (e.g., F-MSR [6,8], PM-RBT [27], Butterfly [22]), non-systematic forms (e.g., F-MSR), exponential sub-packetization (e.g., Butterfly, Clay [36]), and data-block-only improvements (e.g., Hitchhiker [29], HashTag [16]).

The limitations inspire LESS, a family of repair-friendly erasure codes with I/O efficiency in mind. We observe that

| Codes | MDS | Parameters | Systematic | Sub-packetization | Reduced repair I/O for data/parity |
|---|---|---|---|---|---|
| F-MSR [6, 8] | Yes | $n-k \leq 3$ | No | $\alpha = n-k$ | Yes |
| PM-RBT [27] | Yes | $n \geq 2k-1$ | Yes | $k-1 \leq \alpha \leq n-k$ | Yes |
| Butterfly [22] | Yes | $n-k=2$ | Yes | $\alpha = 2^{k-1}$ | Yes |
| Clay [36] | Yes | general $(n,k)$ | Yes | $\alpha = (n-k)^{\lceil \frac{n}{n-k} \rceil}$ | Yes |
| Azure-LRCs [10] | No | general $(n,k)$ | Yes | $\alpha = 1$ | No |
| Hitchhiker [29] | Yes | general $(n,k)$ | Yes | $\alpha = 2$ | No |
| HashTag [16] | Yes | general $(n,k)$ | Yes | $2 \leq \alpha \leq (n-k)^{\lceil \frac{k}{n-k} \rceil}$ | No |
| HashTag+ [15] | Yes | general $(n,k)$ | Yes | $4 \leq \alpha \leq (n-k)^{\lceil \frac{n}{n-k} \rceil}$ | Yes |
| ET [35] | Yes | general $(n,k)$ | Yes | $2 \leq \alpha \leq (n-k)^{\lceil \frac{n}{n-k} \rceil}$ | Yes |
| LESS | Yes | general $(n,k)$ | Yes | $2 \leq \alpha \leq n-k$ | Yes |

**Table 1:** Comparisons of repair-efficient erasure codes.

repair performance depends on both repair I/O and I/O seeks. Minimizing repair I/O introduces an exponential $\alpha$ [2, 36] (and hence high I/O seeks) and can negate repair performance gains. LESS targets near-minimum repair I/O with a small and configurable $\alpha$ to limit I/O seeks. Its design goals include:

- **Preserving RS code properties:** LESS is MDS, supports general $(n,k)$, and remains systematic, as in RS codes.
- **I/O-efficient repair:** LESS achieves I/O-efficient repairs in three aspects: (i) *reduced repair I/O*, (ii) *reduced I/O seeks* (e.g., $\alpha = 2$, 3, or 4); and (iii) *balanced reductions* (i.e., similar reductions in repair I/O and I/O seeks across data and parity blocks). LESS also allows $\alpha$ to be configurable.
- **Single- and multi-block repair efficiency:** Earlier studies focus on optimizing single-block repairs, which dominate in practice (e.g., 98% of failures in $(14,10)$-coded stripes [28]). However, wide-stripe codes (i.e., $n$ and $k$ are large) emerge and make multi-block repairs more common [12]. LESS addresses both scenarios.

# 3 LESS Design

LESS is constructed with configurable parameters $(n,k,\alpha)$, where $k < n$ and $2 \leq \alpha \leq n-k$. Its key idea is to layer multiple *extended sub-stripes*, each constructed from sub-blocks and encoded using Vandermonde-based RS codes, such that a single-block repair always retrieves the sub-blocks from a single extended sub-stripe. LESS also enables I/O-efficient multi-block repairs if the failed blocks share the same extended sub-stripe.

## 3.1 Vandermonde-Based RS Codes

LESS builds on Vandermonde-based RS codes. An $(n,k)$ Vandermonde-based RS code uses a *parity-check matrix* [24] to define the relationships between $k$ data blocks $B_1, B_2, \cdots, B_k$ and $n-k$ parity blocks $B_{k+1}, \cdots, B_n$ in GF($2^w$) based on the *parity-check equation*:

$$[\mathbf{v}_1,\ \mathbf{v}_2,\ \cdots,\ \mathbf{v}_n] \cdot [B_1,\ B_2,\ \cdots,\ B_n]^T = \textstyle\sum_{i=1}^{n} B_i \mathbf{v}_i = \mathbf{0}, \quad (1)$$

where $\mathbf{v}_i = [1, v_i, v_i^2, \cdots, v_i^{n-k-1}]^T$ is a column vector of length $n-k$, and $v_i$ is a distinct coding coefficient in GF($2^w$) associated with block $B_i$, for $1 \leq i \leq n$. In GF($2^w$), an addition is equivalent to a bitwise-XOR operation. The $(n-k) \times n$ parity-check matrix $[\mathbf{v}_1,\ \mathbf{v}_2,\ \cdots,\ \mathbf{v}_n]$ ensures that any $n-k$ out of $n$ columns are linearly independent, so as to allow reconstruction of any $n-k$ blocks from the remaining $k$ blocks.

The encoding computes the $n-k$ parity blocks from the $k$ data blocks based on Equation (1):

$$[B_{k+1}, \cdots, B_n]^T = [\mathbf{v}_{k+1}, \cdots, \mathbf{v}_n]^{-1} [\mathbf{v}_1, \cdots, \mathbf{v}_k][B_1, \cdots, B_k]^T. \quad (2)$$

The decoding of any $n-k$ blocks multiplies: (i) the inverted sub-matrix of their associated column vectors, (ii) the sub-matrix of the column vectors associated with the $k$ remaining blocks, and (iii) the $k$ remaining blocks, from Equation (1).

## 3.2 Motivating Example

We consider LESS via an example. Figure 1 depicts an $(n,k,\alpha) = (6,4,2)$ LESS stripe, where each block $B_i$ ($1 \leq i \leq 6$) has two sub-blocks $b_{i,1}$ and $b_{i,2}$. LESS organizes the 12 sub-blocks into three $(8,6)$ RS-coded extended sub-stripes, $X_1$, $X_2$, and $X_3$. Each extended sub-stripe can tolerate any two sub-block failures, and the entire stripe can tolerate any two block failures due to the MDS property.

**Construction.** First, we partition the 12 sub-blocks, such that each sub-block belongs to exactly two extended sub-stripes. This overlapping structure ensures that each extended sub-stripe can be derived from the other two extended sub-stripes. For example, $X_3$ comprises four data sub-blocks and four parity sub-blocks, each of which also belongs to either $X_1$ or $X_2$, but not both, so $X_3$ can be derived from $X_1$ and $X_2$.

Next, we choose coding coefficients for the extended sub-stripes based on Vandermonde-based RS codes. We select 12 distinct coding coefficients $v_{i,j}$'s for $b_{i,j}$'s from GF($2^8$) ($1 \leq i \leq 6$ and $1 \leq j \leq 2$).

Finally, we compute the parity blocks $B_5$ and $B_6$ by encoding the first two extended sub-stripes $X_1$ and $X_2$. We compute $b_{5,1}$ and $b_{6,1}$ with the six data blocks from $X_1$, and

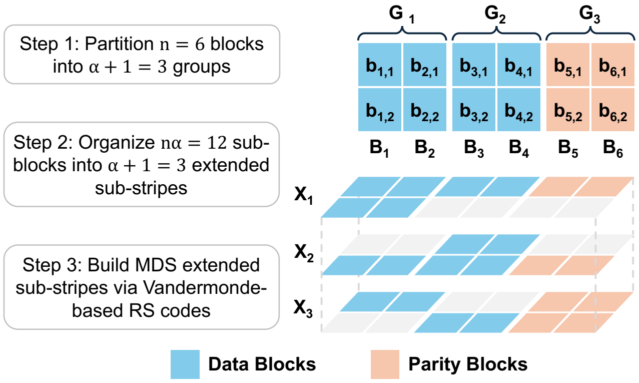

Step 1: Partition n = 6 blocks into α + 1 = 3 groups
Step 2: Organize nα = 12 sub-blocks into α + 1 = 3 extended sub-stripes
Step 3: Build MDS extended sub-stripes via Vandermonde-based RS codes
G1
G2
G3
b1,1
b2,1
b3,1
b4,1
b5,1
b6,1
b1,2
b2,2
b3,2
b4,2
b5,2
b6,2
B1
B2
B3
B4
B5
B6
X1
X2
X3
Data Blocks
Parity Blocks

**Figure 1:** Code design of an $(n,k,\alpha) = (6,4,2)$ LESS stripe.

$b_{5,2}$ and $b_{6,2}$ with the other six data blocks from $X_2$ via (8,6) Vandermonde-based RS encoding (based on Equation (1)) as:

$$(\sum_{i=1}^{6} b_{i,1}\mathbf{v}_{i,1}) + b_{1,2}\mathbf{v}_{1,2} + b_{2,2}\mathbf{v}_{2,2} = \mathbf{0},$$

$$(\sum_{i=1}^{6} b_{i,2}\mathbf{v}_{i,2}) + b_{3,1}\mathbf{v}_{3,1} + b_{4,1}\mathbf{v}_{4,1} = \mathbf{0},$$

where $\mathbf{v}_{i,j} = [1,\ v_{i,j}]^T$. This completes the encoding process. Our coding coefficients are carefully chosen to ensure that $X_3$ inherently forms an (8,6) Vandermonde-based RS stripe without being explicitly encoded. The reason is that $X_3$ also satisfies the parity-check equation (Equation (1)), which is exactly the sum of the parity-check equations of $X_1$ and $X_2$:

$$\begin{aligned} & b_{1,1}\mathbf{v}_{1,1} + b_{2,1}\mathbf{v}_{2,1} + b_{3,2}\mathbf{v}_{3,2} + b_{4,2}\mathbf{v}_{4,2} \\ + & b_{5,1}\mathbf{v}_{5,1} + b_{5,2}\mathbf{v}_{5,2} + b_{6,1}\mathbf{v}_{6,1} + b_{6,2}\mathbf{v}_{6,2} = \mathbf{0}. \end{aligned}$$

**Single-block repair.** Suppose $B_1$ fails. LESS repairs $b_{1,1}$ and $b_{1,2}$ from $X_1$ by retrieving six sub-blocks $b_{2,1}$, $b_{2,2}$, $b_{3,1}$, $b_{4,1}$, $b_{5,1}$, and $b_{6,1}$, with 25% less I/O than the conventional repair for the $(6,4)$ RS code (four full blocks). Note that LESS's repair I/O reductions apply to all data and parity blocks. For any block, its sub-blocks must reside in one of the extended sub-stripes. Thus, a single-block repair always retrieves six sub-blocks.

### 3.3 Construction

We now elaborate on how to construct a LESS stripe for the configurable parameters $(n,k,\alpha)$, where $k < n$ and $2 \le \alpha \le n$. An $(n,k,\alpha)$ LESS stripe comprises $n$ blocks $B_i$'s $(1 \le i \le n)$, each with $\alpha$ sub-blocks $b_{i,j}$'s $(1 \le j \le \alpha)$ (i.e., $\alpha$ sub-stripes). LESS organizes the sub-blocks into $\alpha+1$ extended sub-stripes, each tolerating any $n-k$ sub-block failures. Its construction process proceeds as follows.

*Step 1 (Grouping blocks).* We partition $n$ blocks into $\alpha+1$ *block groups* $G_z$'s $(1 \le z \le \alpha+1)$, whose numbers of blocks differ by at most one. The number of blocks in $G_z$ $(|G_z|)$ is:

$$|G_z| = \begin{cases} \lceil \frac{n}{\alpha+1} \rceil & \text{if } z \le n \bmod (\alpha+1) \\ \lfloor \frac{n}{\alpha+1} \rfloor & \text{otherwise.} \end{cases} \tag{3}$$

In Figure 1, we divide the six blocks into three block groups: $G_1 = \{B_1, B_2\}$, $G_2 = \{B_3, B_4\}$, and $G_3 = \{B_5, B_6\}$.

*Step 2 (Layering extended sub-stripes).* We organize $n\alpha$ sub-blocks into $\alpha+1$ extended sub-stripes $X_z$'s $(1 \le z \le \alpha+1)$. Let $g_i$ be the index of the block group containing block $B_i$ (i.e., if $B_i \in G_z$, then $g_i = z$). Each of the first $\alpha$ $X_z$'s (for $1 \le z \le \alpha$) includes all sub-blocks in $G_z$ plus all sub-blocks in the $z$-th sub-stripe, while $X_{\alpha+1}$ includes all sub-blocks in $G_{\alpha+1}$ and all sub-blocks $b_{i,j}$'s where $g_i = j$. The number of sub-blocks in $X_z$ $(|X_z|)$ is:

$$|X_z| = n + (\alpha-1)|G_z|, \tag{4}$$

where $n$ accounts for the sub-blocks in a sub-stripe, and $(\alpha-1)|G_z|$ accounts for the remaining sub-blocks in $G_z$ that do not belong to the same sub-stripe. Each sub-block belongs to exactly two extended sub-stripes. In particular, $X_{\alpha+1}$ includes the sub-blocks from the first $\alpha$ extended sub-stripes, where each sub-block appears in exactly one of the first $\alpha$ extended sub-stripes. In Figure 1, we organize the sub-blocks into three extended sub-stripes ($X_1$, $X_2$ and $X_3$), each of which contains eight sub-blocks.

*Step 3 (Building MDS extended sub-stripes).* Our goal is to make each extended sub-stripe $X_z$ $(1 \le z \le \alpha+1)$ form an $(|X_z|, |X_z|-n+k)$ Vandermonde-based RS stripe that can tolerate any $n-k$ sub-block failures. The encoding process proceeds as follows. First, we select $n\alpha$ distinct coding coefficients $v_{i,j}$'s $(1 \le i \le n$ and $1 \le j \le \alpha)$ from GF($2^w$), such that $X_z$ must satisfy the parity-check equation:

$$\sum\nolimits_{b_{i,j} \in X_z} b_{i,j}\mathbf{v}_{i,j} = \mathbf{0}, \tag{5}$$

where $\mathbf{v}_{i,j} = [1 \quad v_{i,j} \quad v_{i,j}^2 \quad \cdots \quad v_{i,j}^{n-k-1}]^T$ is a Vandermonde column vector of length $n-k$. The linear independence of any $n-k$ Vandermonde column vectors ensures that $X_z$ is an RS-coded stripe. Next, we compute the parity blocks $B_i$'s $(k+1 \le i \le n)$ by encoding the first $\alpha$ extended sub-stripes $X_z$'s in sequence (i.e., from $X_1$ to $X_\alpha$). Specifically, for each $X_z$ $(1 \le z \le \alpha)$, we compute the $n-k$ parity sub-blocks $b_{i,z}$ $(k+1 \le i \le n)$ using RS encoding (based on Equations (2) and (5)). $X_{\alpha+1}$ does not require explicit encoding, as it automatically forms an Vandermonde-based RS stripe after the first $\alpha$ extended sub-stripes are encoded. In Figure 1, we compute $B_5$ and $B_6$ with $X_1$ and $X_2$ based on Vandermonde-based RS encoding, after which $X_3$ also forms an $(8,6)$ Vandermonde-based RS stripe.

**MDS property.** For a LESS stripe to tolerate any $n-k$ block failures, we must carefully select $n\alpha$ distinct coding coefficients $v_{i,j}$'s from a Galois Field. Theorem 1 shows that such coefficients can always be found in a sufficiently large field size. The detailed proof is in the Appendix.

**Theorem 1.** *For $(n,k,\alpha)$ LESS, we can always find $n\alpha$ distinct coding coefficients $v_{i,j}$'s $(1 \le i \le n$ and $1 \le j \le \alpha)$ in GF($2^w$) for MDS when $2^w \ge n\alpha + (n-k-1)\binom{n-1}{k}$.*

Theorem 1 provides a sufficient condition for the Galois Field size. In practice, for typical coding parameters, the required coefficients can be found in smaller fields (GF($2^8$) or GF($2^{16}$)). In our implementation, we can generate the $n\alpha$ coding coefficients using a primitive element $p$ with a multiplicative approach:

$$v_{i,j} = p^{(h_i(\alpha+1)+g_i)\alpha+j},\ 1 \le i \le n \text{ and } 1 \le j \le \alpha, \qquad (6)$$

where $h_i$ ($1 \le i \le |G_z|$) is the position of $B_i$ in its block group $G_z$ (i.e., $B_i$ is the $h_i$-th block in $G_z$). For example, in Figure 1, $B_2$ is the second block in $G_1$, so $g_2 = 1$ and $h_2 = 2$. The primitive element is $p = 2$ for $(6,4,2)$ LESS, while $v_{2,1}$ and $v_{2,2}$ are $2^{15}$ and $2^{16}$ in GF($2^8$), respectively. We can find the feasible primitive elements based on brute-force search (i.e., checking linear independence by enumerating any possible sub-matrices of the parity-check matrix). Note that the search of feasible primitive elements is done only once for construction. In the Appendix, we show how the parity-check matrix is formed and provide the feasible primitive elements for common $(n,k,\alpha)$ ($n-k \le 4$ and $2 \le \alpha \le 4$).

## 3.4 Repair

LESS supports efficient repair of single-block failures within an extended sub-stripe and certain multi-block failures when the failed blocks reside in the same extended sub-stripe.

**Single-block repair.** For a failed block $B_i$ ($1 \le i \le n$) in $G_z$ ($1 \le z \le \alpha$), we repair $B_i$ using sub-blocks within $X_z$, which contains all sub-blocks of $G_z$. Since $X_z$ is an $(|X_z|, |X_z|-n+k)$ RS-coded stripe, it tolerates any $n-k \ge \alpha$ sub-block failures. The $\alpha$ failed sub-blocks can be reconstructed using the parity-check equation (Equation (5)) for $X_z$. To reduce accessed blocks, we prioritize transferring $|G_z|(\alpha-1)$ available sub-blocks from $G_z$, followed by $|X_z|-n+k-|G_z|(\alpha-1)$ available sub-blocks from other block groups.

**Multi-block repair.** LESS is also beneficial for a multi-block repair, where we simultaneously repair multiple failed blocks on a single stripe. When $\lfloor \frac{n-k}{\alpha} \rfloor \ge 2$, LESS can repair any $\lfloor \frac{n-k}{\alpha} \rfloor$ block failures within one extended sub-stripe if all failed blocks reside in the same block group, say $G_z$ ($1 \le z \le \alpha+1$). With all failed sub-blocks in $X_z$, since $\alpha \lfloor \frac{n-k}{\alpha} \rfloor \le n-k$, the failed sub-blocks can be repaired within $X_z$. For other multi-block failure cases, LESS resorts to conventional repair (which retrieves $k$ blocks). For example, in a $(14,10,2)$ LESS stripe (Figure 2), repairing two failed blocks $B_1$ and $B_2$ in $G_1$ (i.e., the sub-blocks $b_{1,1}$, $b_{1,2}$, $b_{2,1}$, and $b_{2,2}$ are in $X_1$) can leverage $X_1$'s $(19,15)$ RS coding. This requires 15 sub-blocks, with 25% less repair I/O than the conventional repair of the $(14,10)$ RS code.

## 3.5 I/O-Efficient Repairs in LESS

LESS achieves I/O-efficient repairs in three aspects: (i) repair I/O, (ii) I/O seeks, and (iii) balanced I/O reductions.

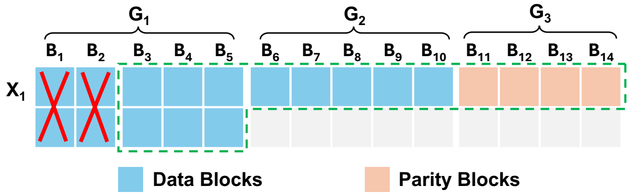


**Figure 2:** Two-block repair in $(n,k,\alpha) = (14,10,2)$ LESS.

**Repair I/O.** For block $B_i$ ($1 \le i \le n$) in $G_z$ (recall $z = g_i$), We retrieve $|X_z|-n+k$ sub-blocks, as we repair the block with the $(|X_z|, |X_z|-n+k)$ RS code. From Equations (3) and (4), the repair I/O (in sub-blocks) of $B_i$ is:

$$\text{IO}_i = \begin{cases} k+(\alpha-1)\left\lceil \frac{n}{\alpha+1} \right\rceil & \text{if } g_i \le n \bmod (\alpha+1) \\ k+(\alpha-1)\left\lfloor \frac{n}{\alpha+1} \right\rfloor & \text{otherwise.} \end{cases} \qquad (7)$$

This is strictly less than the I/O for the conventional repair of an $(n,k)$ RS code when $k > \lceil \frac{n}{3} \rceil$ (common in practice), as $\text{IO}_i \le k+(\alpha-1)\lceil \frac{n}{\alpha+1} \rceil \le k+(\alpha-1)\lceil \frac{n}{3} \rceil < k\alpha$.

**I/O seeks.** For a single-block failure, LESS limits I/O seeks to $k+\alpha-1$. Repairing $B_i$ in $G_z$ requires $|X_z|-n+k$ sub-blocks from $X_z$, including: (i) the $\alpha(|G_z|-1)$ contiguous sub-blocks from $G_z$, which can be retrieved with $|G_z|-1$ seeks, and (ii) $|X_z|-n+k-\alpha(|G_z|-1)$ sub-blocks from other blocks, each requiring one seek. The total number of I/O seeks is $(|G_z|-1)+|X_z|-n+k-\alpha(|G_z|-1) = k+\alpha-1$, from Equations (3) and (4).

**Balanced I/O reductions.** The repair I/O is similar for the repair of any data or parity block in a stripe, and differs by at most $\alpha-1$ sub-blocks from Equation (7). Also, the repair incurs *exactly one* I/O seek from each of the $k+\alpha-1$ available blocks. Thus, LESS balances I/O reductions.

# 4 Evaluation

We evaluate LESS via numerical analysis and testbed experiments, and address two questions: (i) Does LESS's empirical repair performance conform to its theoretical improvements? (ii) How do system configurations affect repair performance?

## 4.1 Numerical Analysis

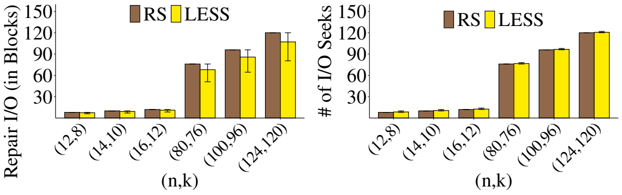
**Exp#A1 (Single-block repair).** We compare LESS against systematic MDS codes (RS, Clay, Hitchhiker, HashTag, and ET) in a single-block repair. We vary $\alpha$ from 2 to $n-k$ for HashTag, ET, and LESS. We measure the average, minimum, and maximum repair I/O (in blocks) and total number of I/O seeks when repairing each of the $n$ blocks. We focus on $(n,k) = (14,10)$, a default configuration in Facebook's f4 [21]. Table 2 shows that for $\alpha \ge 3$, LESS has less repair I/O than other codes (except Clay, which minimizes repair I/O) with limited I/O seeks. For $\alpha = 4$, LESS reduces the average repair I/O of RS, Hitchhiker, HashTag ($\alpha = 4$), and ET ($\alpha = 4$) by 53.6%, 38.1%, 23.1%, and 20.7%, respectively, and reduces the average number of I/O seeks of Clay by 95.5%.

| Codes | $\alpha$ | Repair I/O | | # I/O seeks | |
|---|---|---|---|---|---|
| | | Avg | Min/Max | Avg | Min/Max |
| RS | 1 | 10.00 | 10.00 / 10.00 | 10.000 | 10 / 10 |
| Clay | 256 | 3.25 | 3.25 / 3.25 | 286.00 | 13 / 832 |
| Hitchhiker | 2 | 7.50 | 6.50 / 10.00 | 10.86 | 10 / 13 |
| HashTag | 2 | 7.07 | 5.50 / 10.00 | 10.71 | 10 / 11 |
| | 3 | 6.67 | 4.34 / 10.00 | 12.64 | 10 / 20 |
| | 4 | 6.04 | 4.00 / 10.00 | 12.14 | 10 / 13 |
| ET | 2 | 7.50 | 7.50 / 7.50 | 11.00 | 11 / 11 |
| | 3 | 7.14 | 6.67 / 7.67 | 13.43 | 12 / 14 |
| | 4 | 5.86 | 5.50 / 6.25 | 14.29 | 13 / 15 |
| LESS | 2 | 7.36 | 7.00 / 7.50 | 11.00 | 11 / 11 |
| | 3 | 5.71 | 5.33 / 6.00 | 12.00 | 12 / 12 |
| | 4 | 4.64 | 4.00 / 4.75 | 13.00 | 13 / 13 |

**Table 2:** Analysis for a single-block repair for $(14,10)$.

| Codes / $(n,k)$ | Avg. Repair I/O / Avg. # of I/O seeks | | |
|---|---|---|---|
| | $(80,76)$ | $(100,96)$ | $(124,120)$ |
| RS | 76 / 76 | 96 / 96 | 120 / 120 |
| LESS ($\alpha = 4$) | 31 / 79 | 39 / 99 | 48.6 / 123 |

**Table 3:** Analysis for a single-block repair for wide-stripe codes.



(a) Repair I/O

(b) Number of I/O seeks

**Figure 3:** Analysis for two-block repairs for different $(n,k)$.

LESS also improves wide-stripe repairs. Table 3 shows the results (some parameters are studied in [9, 12]). For example, for $(124,120)$, LESS ($\alpha = 4$) reduces the repair I/O of RS by 59.5% with limited extra I/O seeks.

**Exp#A2 (Multi-block repair).** We study two-block repair I/O and I/O seeks for LESS and RS codes (conventional repair) for different $(n,k)$'s. We average the results over all $\binom{n}{2}$ block failure combinations, with minimum/maximum error bars. Figure 3 shows the results. LESS reduces repair I/O for 27.3-32.8% of two-block failure cases, with limited I/O seek overhead. For example, for $(14,10)$, LESS ($\alpha = 2$) reduces the average repair I/O of RS by 7.4% and improves repairs for 28.6% of cases. For wide stripes, say $(124,120)$, LESS ($\alpha = 2$) reduces the average repair I/O of RS by 10.8% and improves repairs for 32.8% of cases.

## 4.2 Testbed Experiments

**Implementation.** We implement LESS on OpenEC (an erasure coding middleware) [19] atop Hadoop 3.3.4 HDFS [7] and use Jerasure [26] for coding operations. We also include RS, Clay, Hitchhiker, HashTag, and ET in our comparisons. Our prototype adds 8.7 K LoC to OpenEC in C++.

HDFS uses a NameNode for storage management and multiple DataNodes for storage. HDFS organizes data in fixed-size blocks, each further partitioned into multiple *packets*. The packets at the same block offset are encoded together, allowing pipelined coding operations across packets. Our prototype follows this packet-level pipelined implementation.

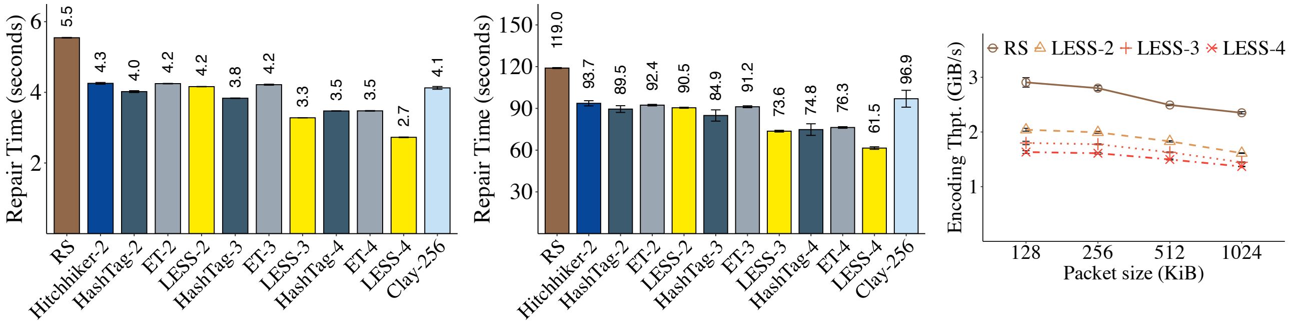
**Methodology.** We conduct testbed experiments in a 15-machine local cluster connected via a 10 Gbps Ethernet switch. Each machine is equipped with a quad-core 3.4 GHz Intel i5-7500 CPU, 16 GiB RAM, a 7200 RPM 1 TB SATA HDD, and Ubuntu 22.04. We use Wondershaper [11] to configure the network bandwidth of each machine. We use one machine for the NameNode and 14 machines for DataNodes. By default, we set $(n,k) = (14,10)$, 64 MiB blocks, 256 KiB packets, and 1 Gbps network bandwidth as in prior work [19, 31, 35].

We evaluate the (i) *single-block repair time* (i.e., the average time from issuing a repair request to a failed block until the failed block is repaired, averaged over $n$ blocks) and (ii) *full-node recovery time* (i.e., the total time to repair all blocks in a single failed DataNode). In full-node recovery, we repair 20 blocks from different stripes (one block per stripe) [18], where node IDs are randomly assigned in the cluster across stripes, so that the lost blocks of a failed node correspond to different block positions across stripes. We average results over 10 runs, with 95% confidence intervals based on the Student's t-distribution.

**Exp#B1 (Single-block repair).** Figure 4(a) shows that LESS effectively reduces the single-block repair time due to the reductions in repair I/O and I/O seeks. For example, LESS ($\alpha = 4$) reduces the single-block repair times of RS, Hitchhiker, HashTag ($\alpha = 4$), ET ($\alpha = 4$), and Clay by 50.8%, 35.9%, 21.5%, 21.5%, and 33.9%, respectively.

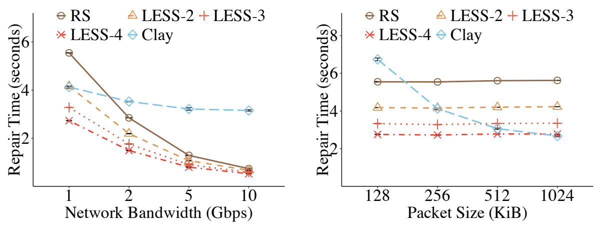
**Exp#B2 (Full-node recovery).** Figure 4(b) shows that LESS also reduces the full-node recovery time. For example, the reductions of LESS ($\alpha = 4$) compared with RS, Hitchhiker, HashTag ($\alpha = 4$), ET ($\alpha = 4$), and Clay are 48.3%, 34.3%, 17.8%, 19.4%, and 36.6%, respectively.

**Exp#B3 (Encoding throughput).** We measure the encoding throughput (i.e., the amount of data encoded per second) on a single machine. We construct a $(14,10)$ stripe in memory and put random bytes in the stripe for encoding. Figure 4(c) shows the results for packet sizes from 128 KiB to 1024 KiB. For 256 KiB packets, RS achieves 2.8 GiB/s, while LESS ($\alpha = 4$) achieves 1.6 GiB/s. For wide stripes, say $(124,120)$ and 256 KiB packets, RS and LESS ($\alpha = 4$) achieve 2.6 GiB/s and 1.1 GiB/s, respectively (not shown in the figure). The encoding throughput for both codes decreases with larger packet sizes, as more data needs to be fit into the CPU cache for encoding. LESS has lower encoding throughput than RS due to sub-packetization, yet its encoding throughput exceeds 1 GiB/s and its computational overhead remains limited compared to bandwidth and I/O bottlenecks (§2). Our evaluation measures single-threaded performance; the encoding throughput can be further improved via multi-threading across stripes.



(a) Single-block repair time (b) Full-node recovery time (c) Encoding throughput

**Figure 4:** Repair performance for $(14,10)$. In figures (a) and (b), we put $\alpha$ next to each code (e.g., LESS-4 has $\alpha = 4$).



(a) Impact of network bandwidth (b) Impact of packet size

**Figure 5:** Impact of configurations on (14,10) single-block repair.

**Exp#B4 (Impact of network bandwidth).** We study the impact of network bandwidth, varied from 1 Gbps to 10 Gbps via Wondershaper, on the single-block repair time. Figure 5(a) shows that as the network bandwidth increases, Clay suffers from significant I/O seek overhead, while LESS effectively reduces the single-block repair time due to small sub-packetization. For example, for 10 Gbps bandwidth, LESS ($\alpha = 4$) reduces the single-block repair times of RS and Clay by 28.6% and 83.3%, respectively. Overall, LESS maintains a low single-block repair time for varying network bandwidth.

**Exp#B5 (Impact of packet size).** We study the impact of packet size, varied from 128 KiB to 1024 KiB. Figure 5(b) shows that for small packet sizes, Clay incurs significant I/O overhead for processing a large number of sub-blocks, while LESS keeps stable repair performance. For example, for 128 KiB packets, LESS ($\alpha = 4$) reduces the single-block repair times of RS and Clay by 59.1% and 50.4%, respectively.

## 5 Conclusion

LESS is a family of erasure codes designed for I/O-efficient repairs by layering extended sub-stripes, so as to reduce both repair I/O and I/O seeks and ensure balanced reductions across blocks. It has several practical properties: MDS, general parameters, systematic, and configurable sub-packetization. Numerical analysis and testbed experiments show LESS's repair benefits over state-of-the-art repair-friendly codes.

## Acknowledgments

We thank our shepherd, Ramnatthan Alagappan, and the anonymous reviewers for their comments. This work was supported in part by the National Key R&D Program of China (2021YFA1001000), the National Natural Science Foundation of China (12231014 and 62302175), the Postdoctoral Fellowship Program and China Postdoctoral Science Foundation (BX20250065), Research Grants Council of Hong Kong (AoE/P-404/18), and Research Matching Grant Scheme. The corresponding author is Xiaolu Li.

## References

[1] S. B. Balaji, M. Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P. Vijay Kumar. Erasure coding for distributed storage: an overview. *Science China Information Sciences*, 61(100301):100301:1–100301:45, 2018.

[2] S. B. Balaji, Myna Vajha, and P. Vijay Kumar. Lower bounds on the sub-packetization level of MSR codes and characterizing optimal-access MSR codes achieving the bound. *IEEE Trans. on Information Theory*, 68(10):6452–6471, 2022.

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

[4] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.

[5] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[6] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, and Dan Feng. Revisiting network coding for warm blob storage. In *Proc. of USENIX FAST*, 2025.

[7] Hadoop 3.3.4. https://hadoop.apache.org/docs/r3.3.4/, 2022.

[8] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.

[9] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for Wide-Stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.

[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.

[11] Bert Hubert, Jacco Geul, and Simon Séhier. Wonder Shaper. https://github.com/magnific0/wondershaper, 2012.

[12] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical design considerations for wide locally recoverable codes (LRCs). In *Proc. of USENIX FAST*, 2023.

[13] Osama Khan, Randal Burns, James Plank, and William Pierce. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[14] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[15] Katina Kralevska and Danilo Gligoroski. An explicit construction of systematic MDS codes with small subpacketization for all-node repair. In *Proc. of IEEE DASC/PiCom/DataCom/CyberSciTech*, 2018.

[16] Katina Kralevska, Danilo Gligoroski, Rune E. Jensen, and Harald Øverby. HashTag erasure codes: From theory to practice. *IEEE Trans. on Big Data*, 4(4):516–529, 2018.

[17] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[18] Xiaolu Li, Keyun Cheng, Kaicheng Tang, Patrick P. C. Lee, Yuchong Hu, Dan Feng, Jie Li, and Ting-Yi Wu. ParaRC: Embracing sub-packetization for repair parallelization in MSR-coded storage. In *Proc. of USENIX FAST*, 2023.

[19] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, 2019.

[20] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[21] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.

[22] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[23] Andreas-Joachim Peters, Michal Kamil Simon, and Elvin Alin Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.

[24] James S. Plank and Cheng Huang. Tutorial: Erasure coding for storage applications. http://web.eecs.utk.edu/~jplank/plank/papers/FAST-2013-Tutorial.html, 2013.

[25] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.

[26] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - version 2.0. Technical Report CS-08-627, 2014.

[27] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan 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.

[28] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan 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.

[29] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[30] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[31] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: novel erasure codes for big data. In *Proc. of VLDB Endowment*, 2013.

[32] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning:

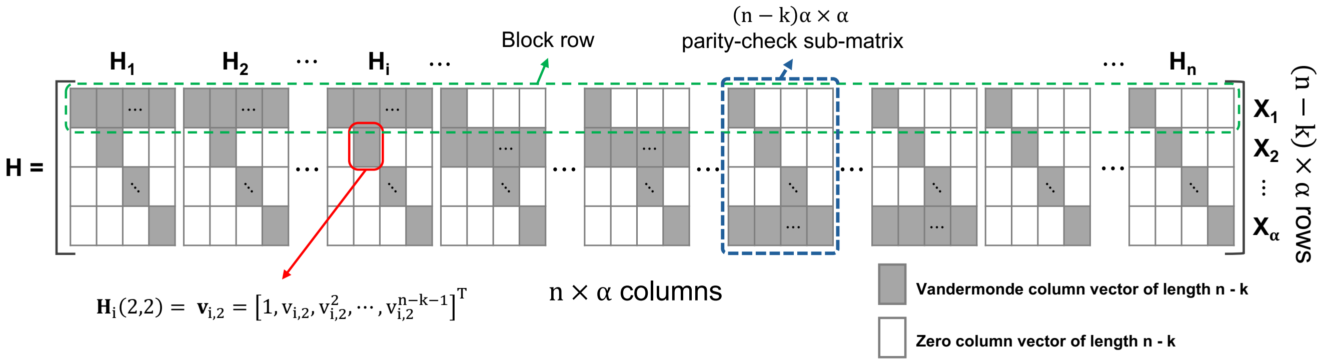

H =
H_1
H_2
···
H_i
···
Block row
(n − k)α × α
parity-check sub-matrix
···
H_n
X_1
X_2
⋮
X_α
(n − k) × α rows
H_i(2,2) = v_{i,2} = [1, v_{i,2}, v_{i,2}^2, ···, v_{i,2}^{n−k−1}]^T
n × α columns
Vandermonde column vector of length n - k
Zero column vector of length n - k

**Figure 6:** Parity-check matrix of an $(n,k,\alpha)$ LESS stripe.

Explore the boundary of optimal erasure code repair. In *Proc. of ACM SOSP*, 2021.

[33] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick P. C. Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. *ACM Trans. on Storage*, 21(1):1–39, 2025.

[34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. of IEEE MSST*, 2010.

[35] Kaicheng Tang, Keyun Cheng, Helen H. W. Chan, Xiaolu Li, Patrick P. C. Lee, Yuchong Hu, Jie Li, and Ting-Yi Wu. Balancing repair bandwidth and sub-packetization in erasure-coded storage via elastic transformation. In *Proc. of IEEE INFOCOM*, 2023.

[36] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[37] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[38] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, 2006.

[39] Yunfeng Zhu, Patrick P. C. Lee, Yinlong Xu, Yuchong Hu, and Liping Xiang. On the speedup of recovery in large-scale erasure-coded storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 25(7):1830–1840, 2014.

## A Appendix: Proof of MDS Property

We prove Theorem 1 that LESS is MDS for a sufficiently large field size. Our proof is based on the parity-check matrix of LESS.

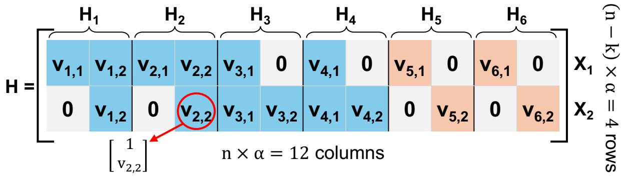


**Figure 7:** Parity-check matrix of a $(6,4,2)$ LESS stripe.

**Parity-check matrix of LESS.** The parity-check matrix ($\mathbf{H}$) of an $(n,k,\alpha)$ LESS stripe is an $(n-k)\alpha \times n\alpha$ matrix. We view $\mathbf{H}$ as an $\alpha \times n\alpha$ *block matrix*, where each entry is a column vector of length $n-k$. We partition $\mathbf{H}$ into $n$ *parity-check sub-matrices* as $\mathbf{H} = [\mathbf{H}_1, \mathbf{H}_2, \cdots, \mathbf{H}_n]$, where each $\mathbf{H}_i$ (where $1 \le i \le n$) is an $\alpha \times \alpha$ block matrix. We define a *block row* as $n\alpha$ entries in a row of $\mathbf{H}$. The entry of $\mathbf{H}_i$ on the $z$-th block row and the $j$-th column (where $1 \le z \le \alpha$ and $1 \le j \le \alpha$) is given by

$$\mathbf{H}_i(z,j) = \begin{cases} \mathbf{v}_{i,j} & \text{if } b_{i,j} \in X_z \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{8}$$

Figure 6 shows the parity-check matrix of LESS. We write block $B_i = [b_{i,1}, b_{i,2}, \cdots, b_{i,\alpha}]^T$ as a column vector of the sub-blocks. From the parity-check matrix, the $\alpha$ parity-check equations of the first $\alpha$ extended sub-stripes can be written as

$$[\mathbf{H_1}, \mathbf{H_2}, \cdots, \mathbf{H_n}][B_1^T, B_2^T, \cdots, B_n^T]^T = \sum_{i=1}^{n} \mathbf{H_i} B_i = \mathbf{0}, \tag{9}$$

Based on Equations (5) and (9), the $z$-th block row can be written as the parity-check equation of the extended sub-stripe $X_z$ as follows:

$$\begin{aligned} & \sum_{i=1}^{n} [\mathbf{H}_i(z,1), \mathbf{H}_i(z,2), \cdots, \mathbf{H}_i(z,\alpha)] B_i \\ = & \sum_{i=1}^{n} \left[ \sum_{j=1}^{\alpha} (\mathbf{H}_i(z,j) b_{i,j}) \right] = \sum_{b_{i,j} \in X_z} b_{i,j} \mathbf{v}_{i,j} = \mathbf{0}. \end{aligned}$$

Figure 7 shows the parity-check matrix of $(6,4,2)$ LESS. The $4 \times 12$ parity-check matrix $\mathbf{H}$ comprises two block rows

| $n-k$ | $\alpha$ | $n$ | Galois Fields | $p$ |
|---|---|---|---|---|
| 2 | 2 | $n \leq 127$ | GF$(2^8)$ | 2 |
| 3 | 2 | $n \leq 44$ | GF$(2^8)$ | 50 |
|  |  | $n \leq 127$ | GF$(2^{16})$ | 2 |
|  | 3 | $n \leq 40$ | GF$(2^8)$ | 14 |
|  |  | $n \leq 127$ | GF$(2^{16})$ | 2 |
| 4 | 2 | $n \leq 23$ | GF$(2^8)$ | 6 |
|  |  | $n \leq 127$ | GF$(2^{16})$ | 46 |
|  | 3 | $n \leq 17$ | GF$(2^8)$ | 2 |
|  |  | $n \leq 127$ | GF$(2^{16})$ | 1362 |
|  | 4 | $n \leq 16$ | GF$(2^8)$ | 14 |
|  |  | $n \leq 127$ | GF$(2^{16})$ | 635 |

**Table 4:** Feasible primitive elements for commonly used coding parameters.

and six parity-check sub-matrices. The first and second block rows can be derived from the parity-check equations of $X_1$ and $X_2$, respectively.

**Proof of Theorem 1.** To ensure LESS is MDS, we can verify the parity-check matrix of LESS, such that any $n-k$ out of $n$ parity sub-matrices can form an $(n-k)\alpha \times (n-k)\alpha$ square invertible matrix. Recall that any invertible square matrix has a non-zero determinant. To ensure that all possible square matrices from the parity-check matrix are invertible, we can multiply the determinants of the square matrices and ensure that the product is non-zero.

There are $\binom{n}{n-k}$ cases for selecting $n-k$ matrices out of the $n$ parity-check sub-matrices to form a square matrix. The determinant of each square matrix can be viewed as a polynomial over the coding coefficients $v_{i,j}$'s (where $1 \leq i \leq n$ and $1 \leq j \leq \alpha$). Each $v_{i,j}$ appears in only one parity-check submatrix (i.e., $\mathbf{H}_i$), and there are $\binom{n-1}{k}$ possible square matrices that contain $\mathbf{H}_i$. Thus, the degree of $v_{i,j}$ in the determinant of such a square matrix is $n-k-1$. By multiplying the determinants of all square matrices, the degree of each $v_{i,j}$ is $(n-k-1)\binom{n-1}{k}$. We can choose $n\alpha$ distinct coding coefficients in GF($2^w$) to ensure the product of determinants is non-zero when $2^w \geq n\alpha + (n-k-1)\binom{n-1}{k}$, which can be proven from Noga Alon's Combinatorial Nullstellensatz. □

Table 4 shows the feasible primitive elements for commonly used coding parameters $n \leq 127$ and $n-k \leq 4$ in GF($2^8$) and GF($2^{16}$) based on brute-force search.

## B Artifact Appendix

### Abstract

LESS is a family of erasure codes designed to improve repair I/O efficiency by layering extended sub-stripes, aiming to reduce both repair I/O and I/O seeks and maintain balanced reductions across all blocks. We implement LESS on OpenEC (an erasure coding middleware) atop Hadoop 3.3.4 HDFS.

### Scope

The artifact is a research-driven prototype that can be used to validate the concepts, designs, and evaluation results of LESS presented in the paper.

### Contents

The artifact has the following contents:

- `src/`, which includes the implementation of LESS as a patch to the OpenEC codebase.
- `scripts/`, which includes the evaluation scripts and configuration files to reproduce our evaluation results.
- `README.md`, which overviews the implementation and provides essential information to run the prototype.
- `AE_INSTRUCTION.md`, which includes detailed instructions for artifact evaluation.

### Hosting

The artifact is hosted in GitHub at `https://github.com/adslabcuhk/less`. We use version `v1.0.0` for artifact evaluation.

### Requirements

#### Hardware Dependencies

We recommend 15 machines to run the experiments with our prototype and evaluation scripts. These machines need to be connected via a 10 Gbps network, such that they are reachable from each other. For each machine, we recommend a quad-core CPU, 16 GiB of memory, a 7200 RPM SATA HDD and above. We need a minimum of 15 machines to form a storage cluster for the default erasure coding parameters $(n,k) = (14,10)$. In particular, we use one machine to run the HDFS NameNode and the other machines to run the HDFS DataNodes.

#### Software Dependencies

The artifact is developed and tested on Ubuntu 22.04 LTS with the following software dependencies:

- **OpenEC**: `g++, cmake, redis, hiredis, ISA-L, gf-complete`
- **Hadoop HDFS**: `openjdk-8-jdk, maven`
- **scripts**: `expect, python3, wondershaper`

### Testbed Setup

Please set up the testbed with the following steps:

- Prepare the hardware dependencies as described.
- Download the artifact (version `v1.0.0`) from URL: `https://github.com/adslabcuhk/less/releases`.
- Extract the files with `tar -zxvf less.tar.gz`; run `cd less` to navigate to the project directory.
- Follow `AE_INSTRUCTION.md` to set up the testbed and run experiments with our provided evaluation scripts. The testbed setup takes around 5 hours, depending on the hardware specifications.

## Evaluation

### Artifact Claims

Our goal is to demonstrate LESS's effectiveness in improving repair performance. For numerical analysis (i.e., Exp#A1 and Exp#A2), we expect the results to match those in our paper. For testbed experiments (i.e., Exp#B1 to Exp#B5), we expect that LESS reduces the single-block repair and full-node recovery times compared to the baseline erasure codes. However, the testbed experiment results may vary from those in our paper due to different factors, such as cluster sizes, machine specifications, operating systems, and software packages.

### Experiments

**Exp#A1 (Single-block repair).** *Expected outcome:* Exp#A1 produces the results as shown in Tables 2 and 3, which demonstrate that LESS reduces the average repair I/Os of RS, Hitchhiker, HashTag, and ET, and reduces the average number of I/O seeks of Clay. *Approximate runtime:* 2 compute minutes.

**Exp#A2 (Multi-block repair).** *Expected outcome:* Exp#A2 produces the results as shown in Figure 3, which demonstrate that LESS reduces the average repair I/O of RS, and achieves similar average numbers of I/O seeks as RS. *Approximate runtime:* 1 compute minute.

**Exp#B1 (Single-block repair).** *Expected outcome:* Exp#B1 produces the results as shown in Figure 4(a), which demonstrate that LESS reduces the single-block repair times of RS, Hitchhiker, HashTag, ET, and Clay. *Approximate runtime:* 2 compute hours.

**Exp#B2 (Full-node recovery).** *Expected outcome:* Exp#B2 produces the results as shown in Figure 4(b), which demonstrate that LESS reduces the full-node recovery times of RS, Hitchhiker, HashTag, ET, and Clay. *Approximate runtime:* 2 compute hours.

**Exp#B3 (Encoding throughput).** *Expected outcome:* Exp#B3 produces the results as shown in Figure 4(c), which demonstrate that LESS has lower single-stripe encoding throughput than RS, yet the coding computation overhead is limited compared to bandwidth and I/O operations. *Approximate runtime:* 5 compute minutes.

**Exp#B4 (Impact of network bandwidth).** *Expected outcome:* Exp#B4 produces the results as shown in Figure 5(a). It shows that when the network bandwidth increases, while Clay suffers from significant I/O seek overhead, LESS still effectively reduces the single-block repair time due to small sub-packetization. *Approximate runtime:* 2 compute hours.

**Exp#B5 (Impact of packet size).** *Expected outcome:* It shows that for small packet sizes, while Clay incurs significant I/O overhead for processing a large number of sub-blocks, LESS maintains stable repair performance due to small sub-packetization. *Approximate runtime:* 2 compute hours.