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

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

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

USENIX FAST 2026

Erasure Coding

- Erasure coding provides storage-efficient fault tolerance
  - Widely adopted in production: Google, Facebook, Azure, CERN …
- ***(n, k)*** Reed-Solomon (RS) codes (where $k < n$)
  - Encode ***k*** uncoded blocks to ***n*** coded blocks (a **stripe**)
  - Maximum distance separable (**MDS**)
    → any k out of n blocks can recover all data with minimum redundancy n/k
    - (14,10) RS codes → redundancy 1.4x
    - 5-way replication → redundancy 5x

    Tolerate any four block failures
- Drawback: High repair penalty
  - Single-block repair accesses and transfers multiple blocks

Repair-friendly Erasure Codes

- Improve repair performance, yet with different design trade-offs
- Clay codes [FAST'18 Vajha]
  - ✓ Minimum single-block **repair I/O** (amount of data read), satisfying MDS
  - ✗ Exponential sub-packetization → substantial non-contiguous **I/O seeks**
- Locally repairable codes (LRCs) [ATC'12 Huang]
  - ✓ Local-group repair (< *k* blocks)
  - ✗ Non-MDS → higher redundancy
- MDS codes with small sub-packetization [SIGCOMM'14, Rashmi INFOCOM'23, Tang IEEE Trans. Big Data'18, Kralevska]
  - ✓ Reduced repair I/O and I/O seeks
  - ✗ Constraints in code constructions

I/O-efficient Repair

- I/O efficiency is more critical than bandwidth efficiency
  - E.g., emerging high-speed network technologies
- Goals
  - **Preserving RS code properties:** MDS, systematic, general in *(n,k)*
  - **I/O-efficient repair**
    - Reduced repair I/O (amount of data read)
    - Reduced I/O seeks (number of non-contiguous accesses)
    - Balanced reductions across all data and parity blocks
  - **Single- and multi-block repair efficiency**
    - Single-block failures dominate in practice
    - Multi-block failures are more common for wide stripes [FAST'23, Kadekodi]

Can we achieve all these properties?

Our Contributions

LESS: a family of erasure codes for I/O-efficient repair

- **Layering extended sub-stripes**
  - Readily builds on RS codes
  - Single-block repair (and some cases of multi-block repair) retrieves sub-blocks within a single extended sub-stripe
  - Allows small and configurable sub-packetization
- Implementation on Hadoop HDFS 3.3.4
- Up to **83.3%** and **36.6%** less single-block repair and full-node recovery times than Clay codes

LESS's Idea

- Input: (n, k, α)
  - α = number of sub-blocks per block (i.e., sub-packetization level)
- Construct extended stripes (n', k')
  - $n' = n + n(\alpha-1)/(\alpha+1)$; $k' = k + n(\alpha-1)/(\alpha+1)$
  - E.g., (n, k, α) = (6, 4, 2) → (n', k') = (8, 6)
- Idea:
  - Layering multiple (n', k') extended sub-stripes on a regular (n, k) stripe
  - Repairing a block in (n, k) stripe → repairing α sub-blocks in (n', k') stripe
  - E.g., (n, k, α) = (6, 4, 2)
    - Traditional erasure codes: Repairing a block retrieves k = 4 blocks
    - LESS: Repairing a block retrieves k' = 6 sub-blocks → 3 blocks (25% less)

** See paper on how we handle the case where n is not divisible by α+1.*

Vandermonde-based RS codes

- **Parity-check equation**

Parity-check matrix

$$\underbrace{\begin{bmatrix} 1^0 & 2^0 & 3^0 & 4^0 & 5^0 & 6^0 & 7^0 & 8^0 \\ 1^1 & 2^1 & 3^1 & 4^1 & 5^1 & 6^1 & 7^1 & 8^1 \end{bmatrix}}_{\mathbf{P}} \times \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \\ B_6 \\ B_7 \\ B_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- **Encoding**
- **Decoding** (similar to encoding)

$$\begin{bmatrix} B_7 \\ B_8 \end{bmatrix} = \overbrace{\underbrace{\begin{bmatrix} 7^0 & 8^0 \\ 7^1 & 8^1 \end{bmatrix}^{-1}}_{\mathbf{P}^{(2)}} \times \underbrace{\begin{bmatrix} 1^0 & 2^0 & 3^0 & 4^0 & 5^0 & 6^0 \\ 1^1 & 2^1 & 3^1 & 4^1 & 5^1 & 6^1 \end{bmatrix}}_{\mathbf{P}^{(1)}}}^{\text{Generator (encoding) matrix}} \times \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \\ B_6 \end{bmatrix}$$

Data Blocks

Parity Blocks

(8,6) Vandermonde-based RS code

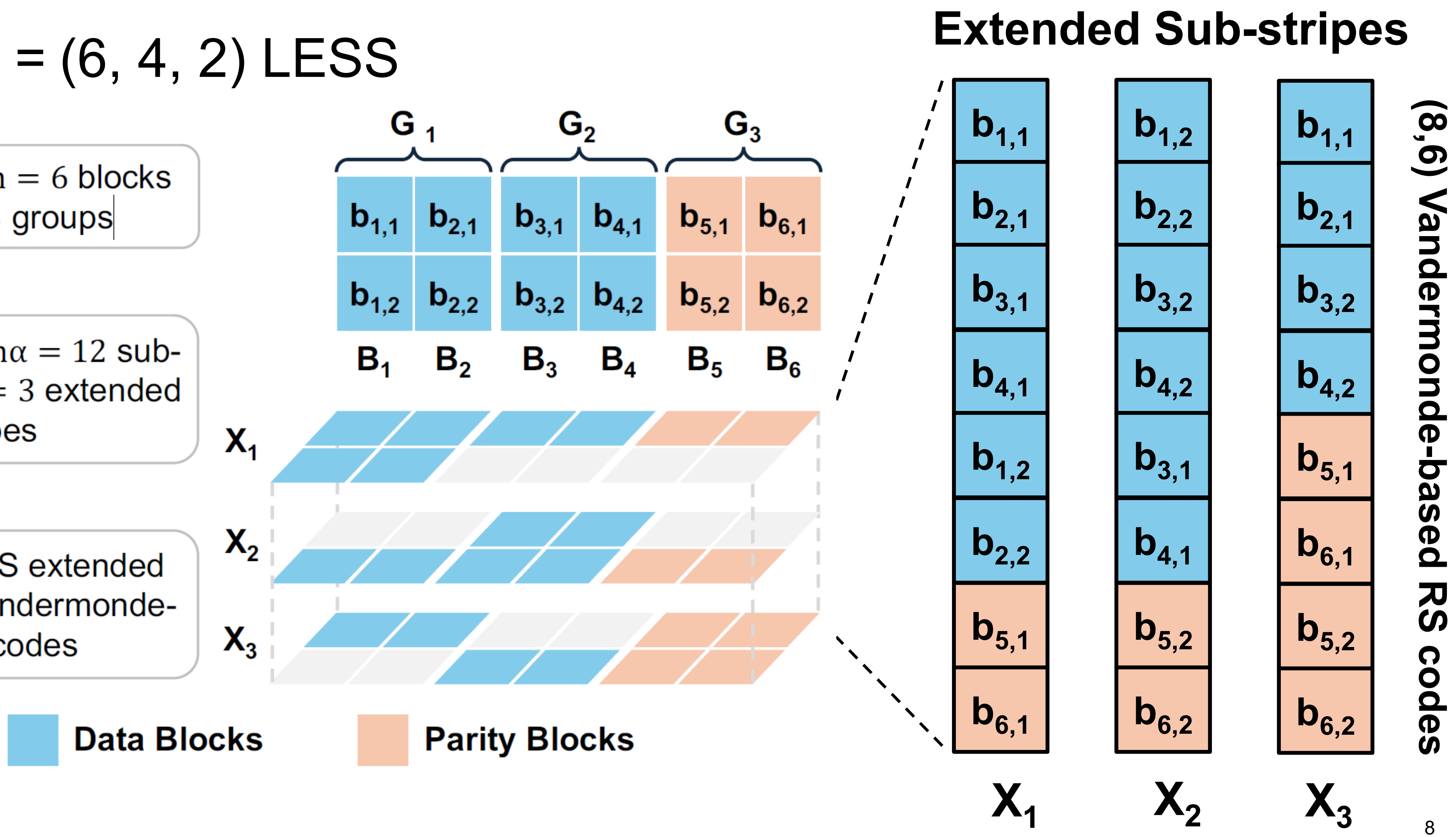
LESS Construction

- ➢ $(n, k, \alpha) = (6, 4, 2)$ LESS

Step 1: Partition $n = 6$ blocks into $\alpha + 1 = 3$ groups

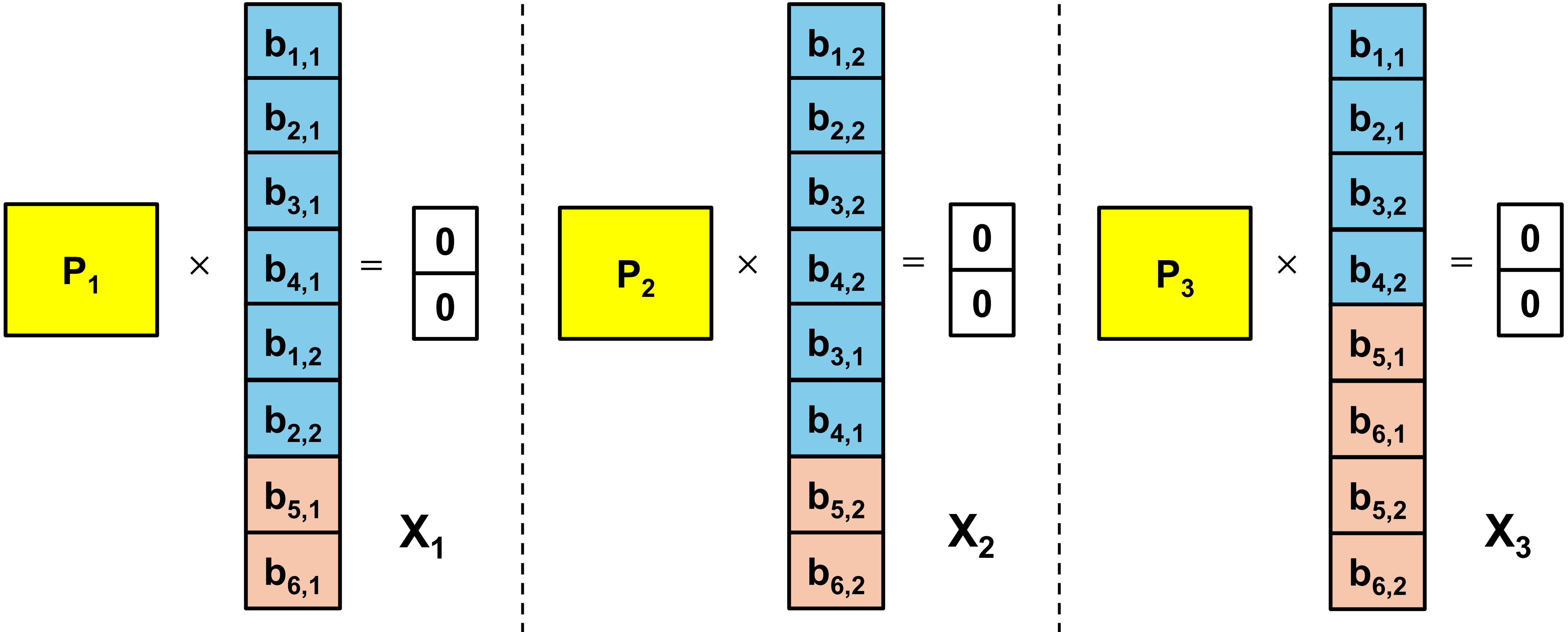
Step 2: Organize $n\alpha = 12$ sub-blocks into $\alpha + 1 = 3$ extended sub-stripes

Step 3: Build MDS extended sub-stripes via Vandermonde-based RS codes



Extended Sub-stripes

- Each has a longer stripe length than a regular stripe
- Satisfies parity-check equation: any *n* – *k sub*-block failures tolerable



LESS Encoding

- Encodes the first α extended sub-stripes in sequence
- The (α+1)-th extended sub-stripe inherently forms an RS stripe

$$\begin{bmatrix} b_{5,1} \\ b_{6,1} \end{bmatrix} = \left[P_1^{(2)}\right]^{-1} \times \left[P_1^{(1)}\right] \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \\ b_{4,1} \\ b_{1,2} \\ b_{2,2} \end{bmatrix}$$

$\mathbf{X_1}$

$$\begin{bmatrix} b_{5,2} \\ b_{6,2} \end{bmatrix} = \left[P_2^{(2)}\right]^{-1} \times \left[P_2^{(1)}\right] \times \begin{bmatrix} b_{1,2} \\ b_{2,2} \\ b_{3,2} \\ b_{4,2} \\ b_{3,1} \\ b_{4,1} \end{bmatrix}$$

$\mathbf{X_2}$

With carefully chosen coding parameters, $\mathbf{X_3}$ inherently forms an (8,6) stripe, **without explicit encoding**

LESS Encoding

- Encoding of $\mathbf{X}_1$ and $\mathbf{X}_2$:

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

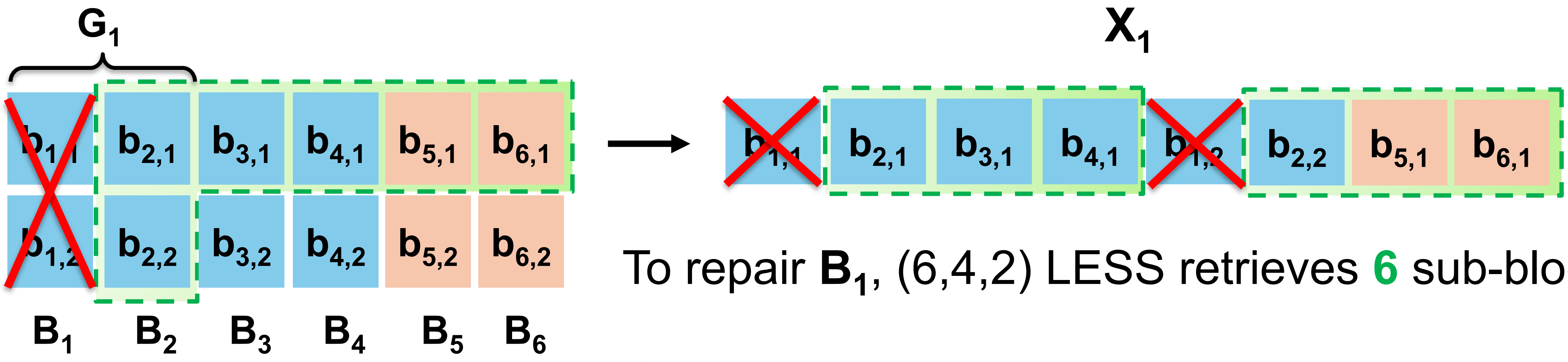
$$\left(\sum_{i=1}^{6} b_{i,2}\mathbf{v}_{i,2}\right) + b_{3,1}\mathbf{v}_{3,1} + b_{4,1}\mathbf{v}_{4,1} = \mathbf{0}$$

- $\mathbf{X}_3$ is the sum of the parity-check equations of $\mathbf{X}_1$ and $\mathbf{X}_2$, and also satisfies the parity-check equation in Vandermonde-based form

$$b_{1,1}\mathbf{v}_{1,1} + b_{2,1}\mathbf{v}_{2,1} + b_{3,2}\mathbf{v}_{3,2} + b_{4,2}\mathbf{v}_{4,2}$$
$$+ b_{5,1}\mathbf{v}_{5,1} + b_{5,2}\mathbf{v}_{5,2} + b_{6,1}\mathbf{v}_{6,1} + b_{6,2}\mathbf{v}_{6,2} = \mathbf{0}$$

Single-block Repair

- Retrieves sub-blocks within a single extended sub-stripe
  - Reduced repair I/O
- Keeps small α and contiguous sub-block access
  - ***One seek per block***

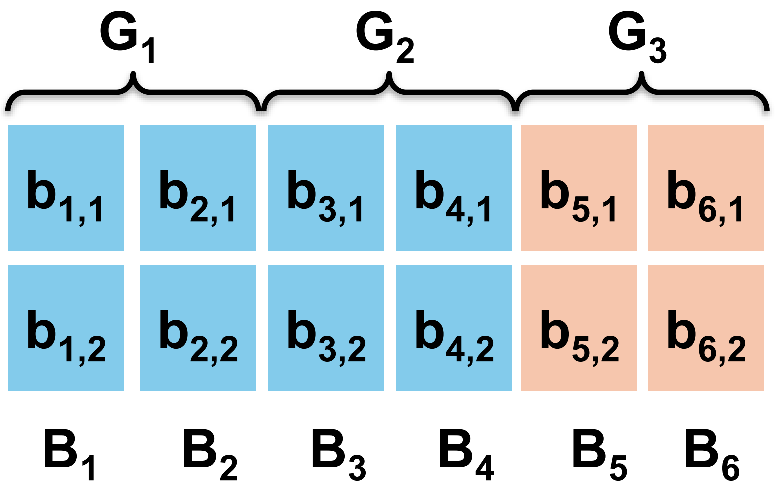


To repair B_1, (6,4,2) LESS retrieves **6** sub-blocks

→ **25%** less than (6,4) RS codes

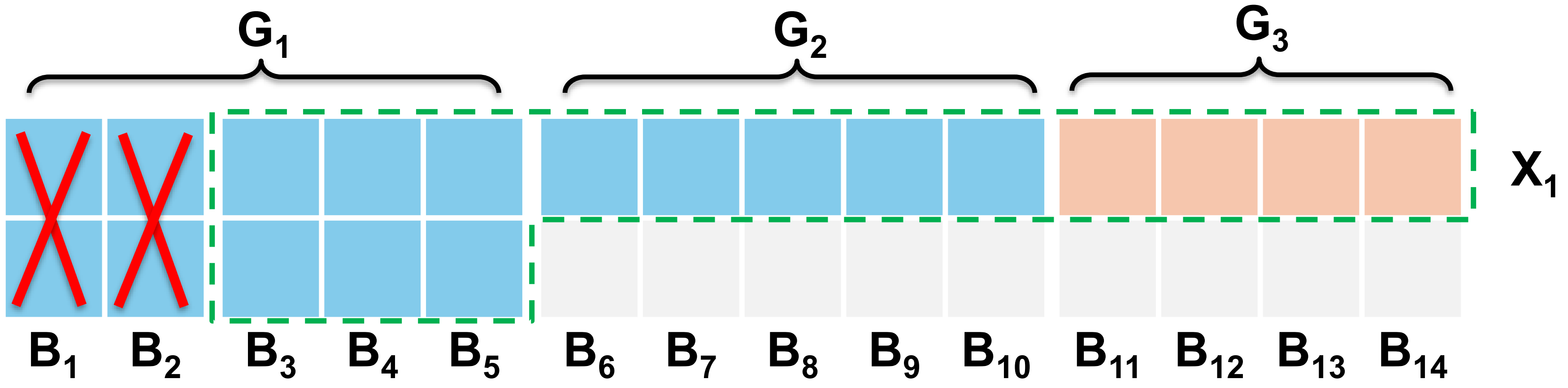
Balanced I/O Reductions

- Each block belongs to one extended sub-stripe
  - **B_1** , **B_2** (in **G_1**): use **X_1** to repair
  - **B_3** , **B_4** (in **G_2**): use **X_2** to repair
  - **B_5** , **B_6** (in **G_3**): use **X_3** to repair
- Similar repair I/O and I/O seeks across all blocks
  - All extended sub-stripes have similar stripe lengths



Multi-block Repair

- LESS benefits multi-block repairs when
  - $\left\lfloor \frac{n-k}{\alpha} \right\rfloor \geq 2$, and at most $\left\lfloor \frac{n-k}{\alpha} \right\rfloor$ failed blocks
  - All failed blocks belong to a single extended sub-stripe



To repair $\mathbf{B_1}$ and $\mathbf{B_2}$, (14,10, 2) LESS retrieves **15** sub-blocks

→ **25%** less than (14,10) RS codes

Preserving RS Code Properties

- MDS
  - LESS's coding coefficients are carefully chosen to ensure MDS property
  - Check our paper for detailed proofs
- Systematic
  - All data blocks are kept for direct access
- General in (n, k)
  - Supports commonly used coding parameters with $GF(2^8)$ and $GF(2^{16})$
  - $2 \leq \alpha \leq n - k$

Evaluation

- Implementation
  - We implement LESS on OpenEC [Li, FAST'19] running on Hadoop 3.3.4

- Numerical analysis
  - LESS reduces repair I/O and I/O seeks

- Testbed experiment
  - LESS reduces single-block repair and full-node recovery times

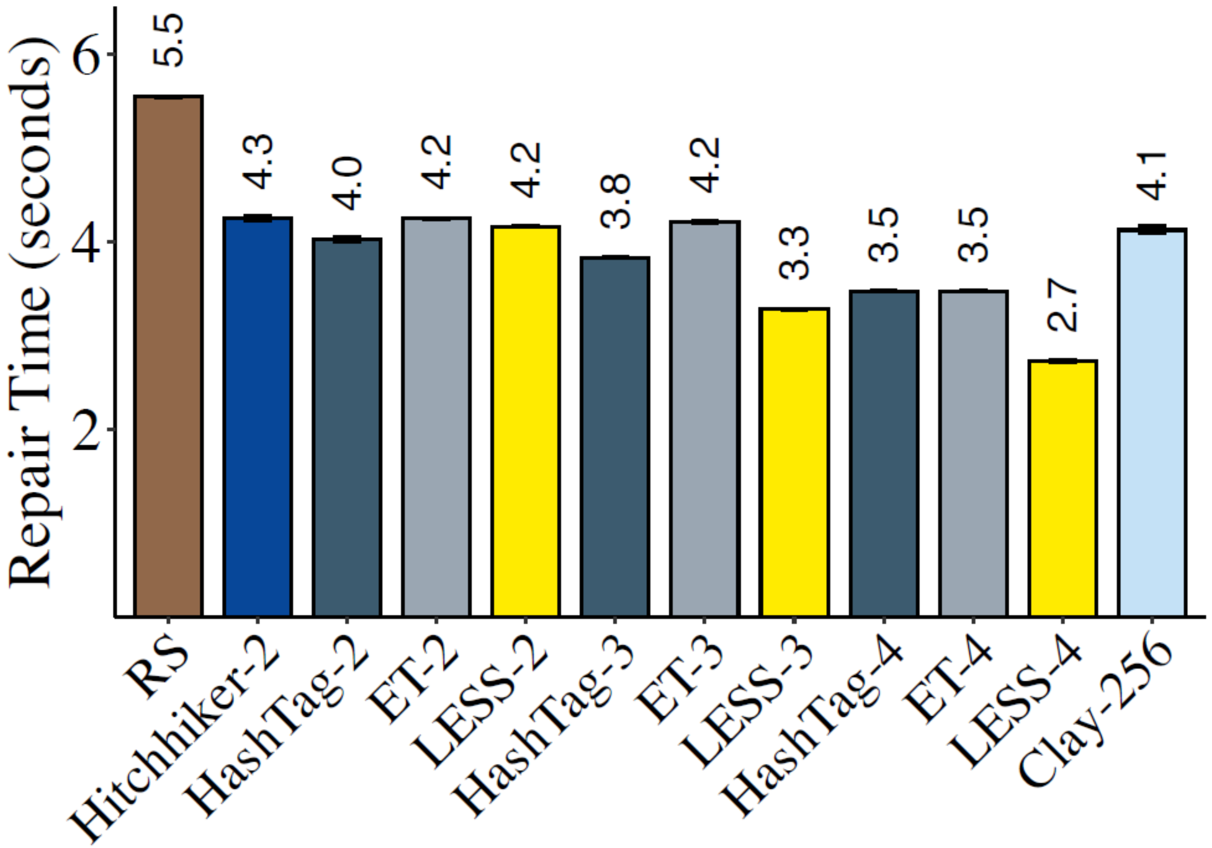
Numerical Analysis: Single-Block Repair

Codes	α	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

$(n, k) = (14,10)$

For (14,10), LESS ($\alpha = 4$) reduces 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 average number of I/O seeks of Clay by **95.5%**

Testbed Experiments: Single-Block Repair



(n, k) = (14,10)

95% confidence intervals based on Student's t-distribution over 10 runs

LESS (α = 4) reduces **single-block repair times** of RS, Hitchhiker, HashTag (α = 4), ET (α = 4), and Clay by **50.8%, 35.9%, 21.5%, 21.5%,** and **33.9%,** respectively.

Impact of Hardware Specifications

- We reproduced our results in Chameleon cloud in FAST'26 AE
  - LESS achieves consistent repair performance improvement

	FAST'26 AE	Our paper
Machine	Chameleon cloud cluster, bare-metal	Local cluster, bare-metal
CPU	Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz 24-core 48-threads	Intel i5-7500 CPU @ 3.80 GHz 4-core 4-threads
Memory	192 GiB	16 GiB
Disk	MTFDDAK480TDS 400 GiB SATA SSD	TOSHIBA 7200 RPM 1 TB SATA HDD
Network	10 Gbps	10 Gbps

Summary of results

- LESS shows similar repair times and consistently outperforms the baselines in different network bandwidths and packet sizes
- LESS and RS achieve 4.5 GiB/s and 7.9 GiB/s encoding throughput in Chameleon with improved CPUs, respectively (1.6 GiB/s and 2.8 GiB/s in our paper)

Conclusions

- **LESS**: a family of repair I/O efficient erasure codes
  - Builds on RS codes by layering extended sub-stripes
  - Reduced repair I/O and I/O seeks for single- and multi-block repairs
  - Implementation atop Hadoop HDFS
- Source code: **https://github.com/adslabcuhk/less**

ARTIFACT EVALUATED USENIX AVAILABLE

ARTIFACT EVALUATED USENIX FUNCTIONAL

ARTIFACT EVALUATED USENIX REPRODUCED