LightMSR: A Lightweight Wide-Stripe MSR Code with Matrix-Based Design for Efficient Repair

Qinhong Ma*, Jingwen Guo*, Yifei Chen*, Suzhen Wu*, Wei Wang†,
Zhanhong Lu†, Weichun Wang†, and Bo Mao*
*School of Informatics, Xiamen University, Xiamen, China
†Hikvision, Hangzhou, China
Corresponding Authors: Wei Wang (wangweiwhxt1@hikvision.com) and Bo Mao (maobo@xmu.edu.cn)

***Abstract*—Erasure codes are widely used in in-memory systems, and wide-stripe designs have emerged to further reduce redundancy. However, wide-stripe erasure codes incur high repair bandwidth. Existing solutions reduce repair bandwidth by increasing redundancy, which contradicts the wide-stripe goal of minimal redundancy. Minimum Storage Regenerating (MSR) codes can reduce repair bandwidth without additional redundancy, but existing MSR designs are too complex for wide-stripe settings and incur high computational overhead, limiting repair efficiency. To address these challenges, we propose LightMSR, a lightweight MSR code based on matrix operations. LightMSR is tailored for wide-stripe environments, supports multi-block recovery, and maintains low computational latency. We implement LightMSR using the ISA-L and Jerasure libraries, and evaluate it against wide-stripe RS and other MSR codes in a Memcached-based in-memory system. Experimental results show that LightMSR significantly reduces design complexity while improving repair performance.**

***Index Terms*—MSR, wide-stripe, repair algorithm, in-memory system**

I. Introduction

Compared with replication, erasure codes provide lower storage overhead for comparable reliability [39]. They have been widely adopted in distributed storage systems for data repair in the event of storage node failures [2], [5], [7], [18], [26], [32], [34], [42], [44], [47], [50]. In recent years, an increasing number of in-memory key–value (KV) stores have adopted erasure codes in place of replication-based redundancy [4], [22]. It indicates that erasure coding is no longer confined to disk-based storage but is being applied in in-memory systems, where I/O is substantially faster.

Among erasure codes, Reed-Solomon (RS) codes [41] are the most popular due to their maturity and effectiveness. RS codes divide the original data into k data blocks and encode them into m parity blocks of equal size. These $k+m$ blocks form a stripe from which any k blocks are sufficient to reconstruct the original data. The redundancy ratio is only $(k+m)/k$ [30].

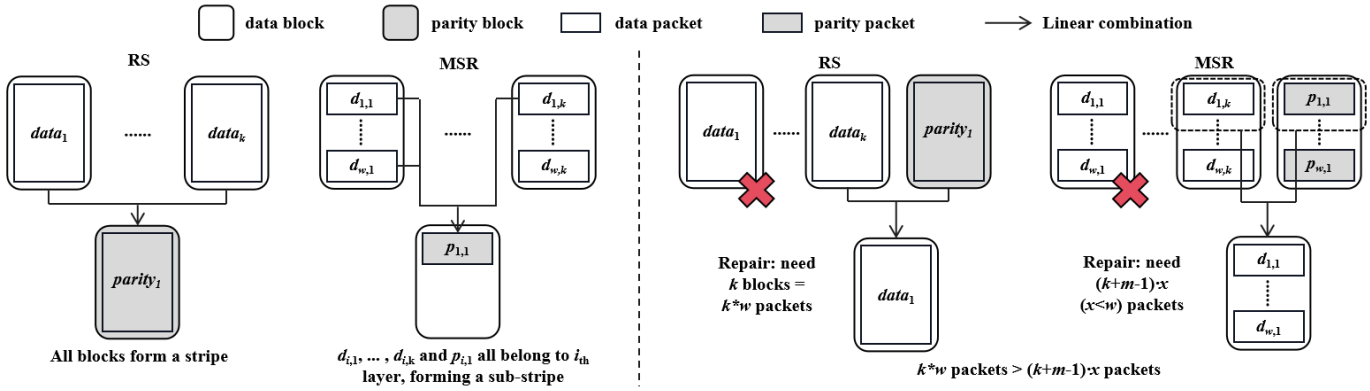
Meanwhile, the concept of wide-stripe erasure coding has been adopted in in-memory systems in recent years to further reduce redundancy overhead [4], [22], [48]. In wide-stripe RS, the number of data blocks k is made very large while the number of parity blocks m remains fixed (e.g., $m = 4$), thereby reducing overall redundancy while maintaining the same fault tolerance [10], [14], [21], [28], [45], [46].

Nevertheless, the wide-stripe RS codes exacerbate the repair bandwidth demand. In RS codes, reconstructing a missing block typically requires reading k surviving blocks; when k is large, even a single-block loss triggers substantial repair bandwidth [14], [33]. This issue is especially acute in in-memory systems: because I/O is extremely fast, network latency has a proportionally greater impact on repair efficiency.

To alleviate this issue for wide-stripe scenarios, prior works have proposed using Locally Repairable Codes (LRC) [14] and Minimum Bandwidth Regenerating (MBR) codes [22]. However, both approaches increase redundancy to achieve repair bandwidth reduction. LRCs introduce extra local parity blocks to reduce repair bandwidth for local failures [12], [15], [35], [40], while MBR codes combine replication and coding that replicate the same data multiple times [6], [31]. While effective in reducing bandwidth, these methods deviate from the fundamental design goal of wide-stripe erasure codes—minimizing redundancy.

Fortunately, Minimum Storage Regenerating (MSR) codes can achieve the same redundancy as RS codes while significantly reducing repair bandwidth [29]. MSR codes can be categorized into systematic codes [16], [24], [25], [27], [36], [38], which contain both original data and parity blocks, and non-systematic codes [1], [3], [8], [11], which store only encoded parity blocks. Non-systematic MSR codes require extra data access and computation even during normal read operations. In contrast, systematic MSR codes preserve original data blocks, ensuring low redundancy, low repair bandwidth, and no negative impact on read performance. These advantages have made systematic MSR codes the focus of increasing research efforts in recent years, and some have been deployed in real systems [19], [20], [37]. This work also focuses on systematic MSR codes.

In MSR, each data block is further divided into w data packets. These data packets form multiple sub-stripes, shifting computation from a traditional block-based to a packet-based way. Each parity packet is generated through a linear combination of data packets drawn from multiple sub-stripes. However, existing systematic MSR codes are mostly studied for small k, and most designs focus only on single-block failures [19], [20], [37]. There are two primary challenges: (1) As k increases or multiple block losses occur, existing MSR designs become significantly more complex [16], [25], [27], [36], [38], making



(a) The RS and MSR encoding processes. **(b) The RS and MSR decoding processes.**

Fig. 1: The comparison of the encoding and decoding processes between RS and MSR.

them infeasible for wide-stripe configurations; (2) Compared to RS codes, the MSR codes operate on packets rather than blocks, which increases encoding and decoding latency [16], [25], [38]. For in-memory wide-stripe settings [4], [22], where I/O speed is very high, the additional computation latency of MSR severely impacts data repair efficiency.

To address these challenges, this paper draws on the code structure of the HashTag [16] and proposes a new MSR algorithm called LightMSR. Under HashTag design, when k is fixed, the value w can be flexibly configured within a certain range, making it possible for wide-stripe configurations. In contrast, many other MSR codes—due to their algebraic constructions—require a much larger w in wide-stripe settings, which limits their practicality. However, the original HashTag algorithm suffers from complex encoding steps, incomplete repair algorithms, and high computational latency, making it impractical for large k. LightMSR effectively addresses the aforementioned challenges and targets wide-stripe settings with $m = 4$, which is a common practical configuration. This novel MSR algorithm makes the following contributions:

- Based on wide-stripe characteristics, it designs a lightweight MSR encoding process and derives a feasible range of w instead of relying on a large fixed w.
- It enables efficient packet localization during repair in wide-stripe scenarios and effectively handles multiple block failures, unlike existing MSR schemes, which primarily focus on single-block repair.
- It expresses all computations as matrix operations, enabling efficient acceleration and low computing latency in erasure coding implementations.

In summary, LightMSR adopts the HashTag code structure but fundamentally differs at the algorithmic level: It introduces a new encoding process tailored to wide-stripe characteristics with a refined formulation of w; It provides a complete and efficient repair algorithm; It reformulates the core procedures into matrix-based operations. These changes affect repair bandwidth, recoverability, and practical efficiency. To the best of our knowledge, existing practical MSR implementations face substantial challenges in wide-stripe settings. The core code is available at https://github.com/astl-xmu/LightMSR-paper.

Section II introduces the background and motivations. Section III describes the core algorithm of LightMSR. Section IV presents the implementation and provides a performance evaluation for LightMSR. Section V discusses the limitations of LightMSR. Section VI discusses related work. Section VII concludes this paper.

II. Background and Motivations

A. MSR Codes

RS codes perform encoding and decoding at the block level. In contrast, MSR codes divide each block into packets, and all computations are performed at the packet level. As illustrated in Figure 1 (a), each data block is split into w data packets. Since each data block is divided into the same number of data packets, the packets at the same layer can be regarded as a sub-stripe. Each sub-stripe contains k data packets and m parity packets, where the m parity packets are generated as linear combinations of the data packets from the same sub-stripe and other sub-stripes.

Each parity packet contains data packet information drawn from multiple sub-stripes. As a result, MSR codes enable data repair by accessing only a subset of packets from each remaining storage node, thereby reducing the repair bandwidth required to recover the lost block as illustrated in Figure 1 (b). However, different MSR codes employ distinct data-packet selection strategies and computation orders when linearly generating each parity packet. Consequently, the level of repair bandwidth optimization achieved over RS codes can also vary depending on the MSR design [16], [38].

Currently, three major systematic MSR constructions have been practically implemented and studied: Butterfly code [25], Clay code [38], and HashTag code [16]. Butterfly code supports only configurations where $m = 2$. Clay code is more general and can support arbitrary k and m values. However, like Butterfly code, Clay code enforces a fixed relationship between w and k, where w grows exponentially or polynomially as k increases. Although Clay code cannot be applied to

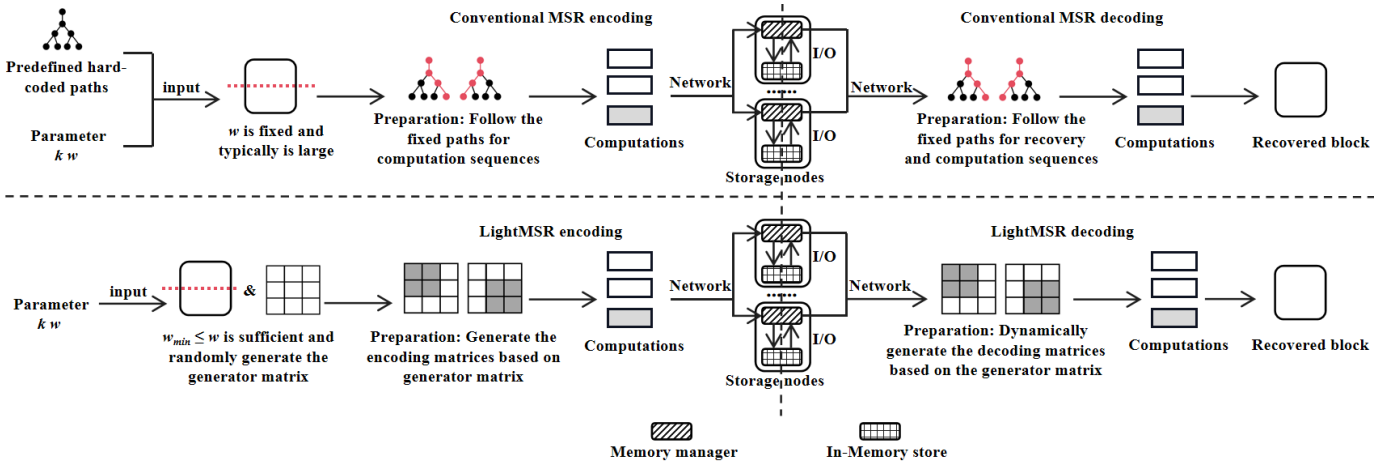


Fig. 2: The differences between conventional MSR and LightMSR in their encoding and decoding processes. When k, m, and w remain unchanged, the encoding and decoding processes in a distributed system primarily consist of four parts: I/O, Network, Preparation, and Computation.

wide-stripe settings, it achieves the theoretical minimum repair bandwidth for single-block repair and represents the current state-of-the-art in systematic MSR code design.

The HashTag code differs in that the w value is not tightly coupled with the k value due to its structural characteristics ($2 \leq w \leq m^{\frac{k}{m}}$). However, the HashTag algorithm suggests setting $w = m^{\frac{k}{m}}$ to facilitate related computations. Although LightMSR draws from the structure of HashTag codes, it incorporates wide-stripe characteristics and adopts a different encoding strategy, resulting in a redefined w range.

In the structure of HashTag codes, as adopted by LightMSR, each parity packet is generated either by linearly combining all k data packets within the same sub-stripe, or by linearly combining those k data packets together with an additional k/m data packets drawn from different sub-stripes. However, the HashTag code, like other traditional MSR codes, lacks lightweight encoding processes and efficient packet localization algorithms, which in turn necessitates hard-coded fixed paths to ensure encoding and decoding; consequently, it is unsuitable for wide-stripe settings and multiple block failures and suffers from high computational latency.

As illustrated in Figure 2, conventional MSR codes, including HashTag, rely on predefined hard-coded paths to perform both encoding and decoding. In practical implementations, they require a fixed and typically large w value, which is tightly coupled with the parameters k and m. During the Preparation stage of encoding and decoding, the system must search these predefined paths to determine packet selection and establish a valid computation order. In the Computation stage, packet combinations are then carried out according to the results of this path search.

Because these schemes fundamentally depend on hard-coded paths, existing MSR code constructions lack an efficient packet localization mechanism. As a result, most implementations primarily focus on single-block repair. In wide-stripe settings, as k increases, these predefined hard-coded paths become increasingly complex, substantially complicating implementation and resulting in high computational latency. This tight coupling between the encoding and decoding logic and fixed paths ultimately limits scalability.

In contrast, as shown in Figure 2, LightMSR adopts a fundamentally different algorithmic design. Both the encoding and decoding procedures in LightMSR are formulated purely as matrix operations, eliminating the dependence on predefined hard-coded paths. In the Preparation stage, LightMSR constructs the required matrices based on the generator matrix, without performing path search. The Computation stage then performs matrix multiplications without a predefined sequence. This design simplifies implementation and improves extensibility, especially in wide-stripe settings.

To better accommodate wide-stripe configurations, LightMSR introduces a lightweight and randomized encoding process and provides a more flexible range for selecting w values. Furthermore, by formulating both encoding and decoding purely as matrix operations, LightMSR enables a complete packet localization algorithm. This design supports multi-block repair, rather than being restricted to single-block recovery, making it significantly more scalable and practical for large-scale deployments.

B. Motivation

Compared to replication strategies, erasure coding not only requires additional computation during data repair but also incurs higher repair bandwidth [43]. In recent years, the concept of wide-stripe has emerged to reduce redundancy overhead [21]. In wide-stripe RS, k is set to a large value while m is typically fixed at 4, which further exacerbates the repair bandwidth issue. This problem is significant in in-memory erasure coding systems [4], [17], [22], [51] (I/O speed is fast) that network bandwidth and computation efficiency become the system bottlenecks for repair.

TABLE I: Encoding and decoding latencies of different codes under varying configurations.

Schemes	(k, m)	w	Encode (μs)	Decode (μs)
RS	(8, 2)	1	9	9
Butterfly [25]	(8, 2)	128	38341	1042
RS	(8, 4)	1	17	9
HashTag [16]	(8, 4)	16	722	294
Clay [38]	(8, 4)	64	17587	4793
RS	(12, 4)	1	25	13
HashTag	(12, 4)	64	5801	1534
Clay	(12, 4)	256	22430	6147

In the past years, works such as LRC [14] and MBR [22] have attempted to reduce repair bandwidth in wide-stripe settings by increasing redundancy. However, this contradicts the fundamental goal of wide-stripe design, which is to minimize redundancy. MSR codes promise both low redundancy and low repair bandwidth. However, they are currently not applicable to wide-stripe scenarios for two main reasons:

(1) The complexity of MSR design limits its applicability in wide-stripe environments: Existing MSR schemes face two main challenges: (i) Some impose specific constraints on k, m and w due to their mathematical construction, making them unsuitable for wide-stripe configurations [25], [27], [38]; (ii) The repair algorithms are overly complex and lack efficient packet localization, typically supporting only single-block recovery with small k [16], [23], [25], [27], [38]. To the best of our knowledge, prior practical designs have not demonstrated efficient deployment in wide-stripe settings.

(2) MSR codes suffer from extremely high encoding/decoding latency: We evaluated several implementations [19], [20], [37], measuring encoding latency and decoding latency for single-block repair with w set according to each code's mathematical construction requirements, and with a block size of 4KB [4] (In this test, encoding/decoding latency refers to the sum of Preparation latency and Computation latency). As shown in Table I, even for small k, MSR exhibits encoding/decoding latency that is orders of magnitude higher than that of RS, often by factors of hundreds or even thousands. The gap further widens as k increases (Under the wide-stripe setting, where $m = 4$, existing MSR implementations fail to support k values larger than those listed because these implementations have hard-coded the data repair paths in advance. When k exceeds the predefined range, data repair becomes impossible. In particular, Butterfly codes only support $m = 2$. Extending previous MSR implementations to support larger k values would encounter the issues discussed in (1), where the MSR algorithms become excessively complex and difficult to implement in practice).

Motivated by the limitations of current MSR schemes, this paper proposes LightMSR, a lightweight MSR algorithm designed for wide-stripe scenarios with $m = 4$, the most common configuration in practice.

III. The Design of LightMSR Code

This section introduces the LightMSR algorithm and its matrix-based encoding and decoding operations. It presents the conditions for parameter selection and the lightweight encoding process, followed by the decoding procedure including packet localization and computation. In addition, this section provides reliability analysis and examines the bounds and structural analysis of repair bandwidth.

As illustrated in Figure 2, both encoding and decoding in the in-memory system consist of four stages: I/O, Network, Preparation, and Computation. LightMSR primarily redesigns the algorithmic logic in the Preparation and Computation stages. However, the algorithmic logic will affect the I/O and Network stages.

A. Encoding

LightMSR is designed for practical wide-stripe deployments with large k and $m = 4$, which is the most common configuration. To simplify the coding process, we specialize the design to this setting and require both k and w to be multiples of m. Under this setting, the code contains four parity blocks, $parity_1$ through $parity_4$.

1) Definitions: Let the original data packets form a matrix (Assume each packet is 8 bits in size, which corresponds to one element in the $GF(2^8)$)

$$D = [d_{i,j}] \in GF(2^8)^{w \times k}.$$

To guide cross-layer packet selection, LightMSR pre-generates a matrix

$$\Lambda \in \{1, \ldots, w\}^{(k/m) \times w}.$$

Each row of Λ is a permutation of $\{1, 2, \ldots, w\}$ and is divided into $m = 4$ groups:

$$G_1^r, G_2^r, G_3^r, G_4^r,$$

where $r \in \{1, \ldots, k/4\}$. Each group has size

$$g_s = \frac{w}{m}.$$

Among all k groups generated across Λ, we require global uniqueness:

$$\forall\, i \neq j \in \{1, \ldots, k\}, \quad \{G_i\} \neq \{G_j\}.$$

That is, no two groups contain exactly the same set of elements, even if the ordering differs. The total number of candidate groups is

$$N(w, m) = \binom{w}{g_s} = \binom{mg_s}{g_s}.$$

Thus, the minimum valid w is

$$w_{\min}(m, k) = m \cdot \min_{g_s \in \mathbb{Z}_{\geq 1}} \left\{ g_s : \binom{mg_s}{g_s} \geq k \right\}.$$

This condition guarantees that the number of candidate groups is sufficient to assign k globally distinct groups, thereby ensuring that the uniqueness constraint on Λ is feasible.

Each group G_s^r is mapped to a column of D by

$$col_s^r = (r-1)\cdot 4 + s.$$

The Cauchy matrix

$$\Psi \in GF(2^8)^{4w\times(k+k/4)}$$

provides the encoding coefficients.

2) Encoding Rules: Let $p_{i,t}$ denote the ith packet in $parity_t$.

a) Same-layer encoding: For all $t \in \{1,2,3,4\}$,

$$p_{i,t}^{(same)} = \sum_{j=1}^{k} \Psi_{i+(t-1)w,\ j}\cdot d_{i,j}.$$

b) Cross-layer encoding: Only $parity_2$–$parity_4$ include cross-layer packets. For group G_s^r, the selected cross-layer source for $parity_t$ is

$$G_{\text{sel}}^{(t)} = G_{(s+t-1) \bmod 4+1}^r, \qquad t \in \{2,3,4\}.$$

Let

$$\gamma = \left\lfloor \frac{col_s^r - 1}{4} \right\rfloor + 1.$$

Then the cross-layer contribution is

$$p_{i,t}^{(cross)} = \sum_{z\in G_{\text{sel}}^{(t)}} \Psi_{i+(t-1)w,\ k+\gamma}\cdot d_{z,col_s^r}.$$

c) Final parity:

$$p_{i,t} = p_{i,t}^{(same)} + p_{i,t}^{(cross)}, \qquad t \in \{2,3,4\},$$

$$p_{i,1} = p_{i,1}^{(same)}.$$

Algorithm 1 presents the construction procedure of matrix Λ, which only needs to be executed once during initialization when k remains fixed. Under the above condition on w, Λ can be constructed to satisfy the required uniqueness property, which in turn guarantees deterministic recoverability.

A candidate row of Λ is accepted only when all its groups pass the global uniqueness check. Therefore, once Λ is generated and validated, the random seed does not affect the recoverability guarantee. The seed only affects the overlap pattern among valid groups, which is analyzed in Section III-D. In failure correlated environments, an unfavorable overlap pattern may affect repair bandwidth variation across different loss patterns, but it does not violate recoverability as long as the validated uniqueness constraint holds. A system can therefore use a fixed and validated Λ during deployment. The complete LightMSR encoding procedure is summarized in Algorithm 2.

Algorithm 1 Generation of Λ

Require: k, w, $m = 4$
Ensure: Λ

```
g_s ← w/m, U ← ∅
for r = 1 to k/4 do
    valid ← false
    while not valid do
        Generate a random permutation of {1, ..., w}
        Partition it into G_1^r, G_2^r, G_3^r, G_4^r
        valid ← true
        for s = 1 to 4 do
            if Sort(G_s^r) ∈ U then
                valid ← false
                break
            end if
        end for
    end while
    for s = 1 to 4 do
        Add Sort(G_s^r) to U
    end for
    Write the row into Λ
end for
```

3) Illustrative Example: Consider $k = 8$ and $w = 8$. The data matrix D has size 8×8. Each packet in $parity_1$ uses $k = 8$ same-layer packets. Each packet in $parity_2$-$parity_4$ uses the same 8 same-layer packets plus $k/4 = 2$ cross-layer packets. Assume

$$\Lambda = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 3 & 2 & 4 & 5 & 7 & 6 & 8 \end{bmatrix}.$$

The first group in the second row is $\{1,3\}$, which maps to column 5 in D. For $parity_2$, the associated group is $\{2,4\}$ (the first group among the remaining three groups in the same row), so the data packets $d_{2,5}$ and $d_{4,5}$ are included in the linear combination for the 1st and 3rd parity packets of $parity_2$. Similarly:

- $parity_3$ uses $\{5,7\}$, i.e., $d_{5,5}$ and $d_{7,5}$.
- $parity_4$ uses $\{6,8\}$, i.e., $d_{6,5}$ and $d_{8,5}$.

This logic is repeated for all groups of the Λ matrix.

4) Encoding Implementation: According to the above mathematical model, after the matrices Λ and Ψ are prepared according to the input parameters k and w, the encoding process is divided into two stages in our implementation: the Preparation stage and the Computation stage.

In the Preparation stage, based on matrix Λ and the parity packet generation procedure, the corresponding computation matrices are then derived (used to perform the linear computation of parity packets).

In the Computation stage, the parity packets are generated according to these computation matrices. Since all operations in this stage involve matrix computations, libraries such as

Algorithm 2 LightMSR encoding

Require: D, Λ, Ψ, k, w, $m = 4$
Ensure: $parity_1, parity_2, parity_3, parity_4$

for $i = 1$ to w **do**
  for $t = 1$ to 4 **do**
    Compute same-layer part:

$$p_{i,t}^{(same)} \leftarrow \sum_{j=1}^{k} \Psi_{i+(t-1)w,\, j} \cdot d_{i,j}$$

  end for
end for
for $r = 1$ to $k/4$ **do**
  for $s = 1$ to 4 **do**
    $col_s^r \leftarrow (r-1) \cdot 4 + s$
    $\gamma \leftarrow \left\lfloor \frac{col_s^r - 1}{4} \right\rfloor + 1$
    for $t = 2$ to 4 **do**
      Select cross-layer group:

$$G_{\text{sel}}^{(t)} \leftarrow G_{(s+t-1) \bmod 4+1}^{r}$$

      for $i = 1$ to w **do**
        Accumulate cross-layer part:

$$p_{i,t}^{(cross)} \leftarrow p_{i,t}^{(cross)} + \sum_{z \in G_{\text{sel}}^{(t)}} \Psi_{i+(t-1)w,\, k+\gamma} \cdot d_{z,col_s^r}$$

      end for
    end for
  end for
end for
for $i = 1$ to w **do**
  $p_{i,1} \leftarrow p_{i,1}^{(same)}$
  for $t = 2$ to 4 **do**
    $p_{i,t} \leftarrow p_{i,t}^{(same)} + p_{i,t}^{(cross)}$
  end for
end for

ISA-L or Jerasure can be utilized to implement and accelerate the computation process.

B. Decoding

Suppose q data columns are lost, where

$$lost\_cols = \{j_1, j_2, \ldots, j_q\} \subseteq [1, k], \qquad 1 \le q \le 4.$$

The goal is to recover all lost packets in these columns.

1) Definitions: Let

$$x \in GF(2^8)^{qw}$$

denote the unknown vector formed by the lost packets, ordered as

$$x = \{d_{1,j_1}, \ldots, d_{w,j_1},\ d_{1,j_2}, \ldots, d_{w,j_2},\ \ldots,\ d_{1,j_q}, \ldots, d_{w,j_q}\}.$$

For each lost column $j \in lost\_cols$, its corresponding row index and group index in Λ are

$$row\_idx = \left\lfloor \frac{j-1}{m} \right\rfloor + 1, \qquad group\_idx = (j-1) \bmod m + 1.$$

Algorithm 3 LightMSR decoding

Require: $lost\_cols$, Λ, Ψ, parity packets
Ensure: All lost packets in $lost\_cols$

Construct unknown vector x from all lost packets in $lost\_cols$
$B \leftarrow \emptyset$
for each $j \in lost\_cols$ **do**
  $row\_idx \leftarrow \left\lfloor \frac{j-1}{m} \right\rfloor + 1$
  $group\_idx \leftarrow (j-1) \bmod m + 1$
  $B \leftarrow B \cup G_{group\_idx}^{row\_idx}$
end for
Step 1: Construct the Base Equation Set
Use all $p_{i,t}$ with $i \in B$ and $t \in \{1, 2, 3, 4\}$ to form base equations
Move unknown terms to the left-hand side and known terms to the right-hand side
Construct

$$R \cdot x = p_b$$

Step 2: Construct the Supplementary Equation Set
Reorder columns of R so that variables with $i \in B$ appear first
Apply Gaussian elimination to R
Restore the original column order
Identify free variables $D_{extra} = \{d_{i,j} \in x \mid i \notin B\}$
if $D_{extra} \neq \emptyset$ **then**
  Add parity equations $p_{i,t}$ for each $d_{i,j} \in D_{extra}$ and $t \in \{1, \ldots, q\}$
  Move unknown terms to the left-hand side and known terms to the right-hand side
  Construct

$$Q \cdot x = p_n$$

else
  $Q \leftarrow R$, $p_n \leftarrow p_b$
end if
Step 3: Solve the Final Linear System
Compute the row reduction of Q
Recover

$$x = T \cdot p_n$$

Return all packets in x

Let $G_{group\_idx}^{row\_idx}$ be the corresponding group in Λ. We define

$$B = \bigcup_{j \in lost\_cols} G_{group\_idx}^{row\_idx}, \qquad B_{len} = |B|.$$

The packets indexed by B are used to construct the base equation set. If supplementary equations are needed, they are added on top of the base set to form the final linear system.

2) Decoding Rules:

a) Step 1: Construct the Base Equation Set: For each $i \in B$ and each parity block $t \in \{1, 2, 3, 4\}$, we use parity packet $p_{i,t}$ to form one linear equation. This gives $4B_{len}$ base equations in total.

After moving all terms involving lost packets to the left-hand side and all known terms to the right-hand side, the base system is written as

$$R \cdot x = p_b,$$

where

$$R \in GF(2^8)^{(4B_{len})\times(qw)}, \qquad p_b \in GF(2^8)^{4B_{len}}.$$

b) Step 2: Construct the Supplementary Equation Set: To preserve the variables indexed by $i \in B$ during elimination, we first reorder the columns of R so that these variables appear first, then apply Gaussian elimination, and finally restore the original column order.

If free variables still remain after elimination, define

$$D_{extra} = \{d_{i,j} \in x \mid i \notin B\}.$$

For each variable in D_{extra}, we add parity equations from rows i and from parity blocks $parity_1$ to $parity_q$. Let $\mathcal{E}$ denote the supplementary equation set, and let

$$E = q \cdot |D_{extra}|$$

be the number of added equations. Combining the base and supplementary equations yields

$$Q \cdot x = p_n,$$

where

$$Q \in GF(2^8)^{(4B_{len}+E)\times(qw)}, \qquad p_n \in GF(2^8)^{4B_{len}+E}.$$

c) Step 3: Solve the Final Linear System: We apply row reduction to Q. Let T be the corresponding transformation matrix. Then

$$T \cdot Q = \mathrm{rref}(Q), \qquad x = T \cdot p_n.$$

The recovered vector x gives all lost packets in $lost\_cols$. The complete LightMSR decoding procedure is summarized in Algorithm 3. Once Λ satisfies the global uniqueness constraint and Ψ is constructed from a Cauchy matrix, the recoverability problem becomes equivalent to solving the induced linear system for the considered $q \le m$ failures.

3) Illustrative Example: We illustrate the decoding procedure with an example from III-A3 when column 1 and column 5 of the original matrix D are lost ($lost\_cols = \{1,\ 5\}$).

According to the Λ matrix, these two lost columns correspond to the first and fifth groups of the matrix. The row indices associated with these groups are $\{1,\ 2\}$ and $\{1,\ 3\}$.

Taking the union and removing duplicates, we obtain:

$$B = \{1,\ 2,\ 3\}, \quad B_{len} = 3$$

From this, we select all parity packets from $parity_1$ to $parity_4$ whose row indices belong to B, i.e., rows 1, 2, and 3. These yield $4 \times 3 = 12$ base equations:

$$\{p_{1,1}, p_{2,1}, p_{3,1}, p_{1,2}, p_{2,2}, p_{3,2}, p_{1,3}, p_{2,3}, p_{3,3}, p_{1,4}, p_{2,4}, p_{3,4}\}.$$

As an example, one such equation is $p_{1,1}$, which can be expressed as:

$$\sum_{j=1}^{8} \Psi_{1,\ j} \cdot d_{1,j} = p_{1,1}$$

After moving all terms involving $d_{1,j}$ for $j \in lost\_cols = \{1,\ 5\}$ to the left-hand side, and the rest to the right-hand side, we obtain:

$$\Psi_{1,1} \cdot d_{1,1} + \Psi_{1,5} \cdot d_{1,5} = p_{1,1} + \sum_{\substack{j=1 \\ j\notin\{1,5\}}}^{8} \Psi_{1,j} \cdot d_{1,j}$$

Repeating this transformation for all 12 base equations yields a coefficient matrix $R \in GF(2^8)^{12\times16}$. Since $q = 2$ and $w = 8$, there are $q \cdot w = 16$ unknowns to recover. We then reorder the columns of R such that the variables $d_{i,j}$ with $i \in B = \{1, 2, 3\}$ appear at the front. Row reduction is performed, and the columns are returned to their original positions. Suppose after Gaussian elimination, two additional unknowns $d_{7,1}$ and $d_{8,1}$ ($i \notin B$) remain as free variables.

To resolve these variables, we add the corresponding parity equations from rows 7 and 8 in $parity_1$ and $parity_2$ (since $q = 2$). That is:

$$\mathcal{E} = \{p_{7,1},\ p_{8,1},\ p_{7,2},\ p_{8,2}\}, \quad E = 4$$

This extends the system from 12 to 16 equations. Each new equation is processed in the same way as before: unknown terms are moved to the left-hand side, and known terms to the right-hand side. We then construct a new matrix $Q \in GF(2^8)^{16\times16}$ and a right-hand side vector $p_n \in GF(2^8)^{16}$.

Finally, we compute the row-reduced form of Q using a transformation matrix T. The recovered vector x contains all 16 lost data packets in columns 1 and 5 of D.

4) Decoding Implementation: According to the above mathematical model, the decoding process is divided into two stages in our implementation: the Preparation stage and the Computation stage.

In the Preparation stage, the row and group indices of matrix Λ are determined according to the locations of the lost data blocks, and B is subsequently generated. Then, the Gaussian elimination operations on sub-matrices from matrix Ψ according to matrix Λ and B are performed to locate the packets required and to construct the corresponding computation matrices (used to perform the linear computation involved in restoring the data packets).

In the Computation stage, the lost data packets are regenerated using the corresponding computation matrices. Both the Gaussian elimination operations in Preparation stage and matrix multiplications in Computation stage are matrix operations. These operations can be efficiently implemented and accelerated using software libraries such as ISA-L or Jerasure.

C. Recoverability Analysis

a) Theorem: Let m and w be fixed. For any set $lost\_cols \subseteq \{1, \ldots, k\}$ with $|lost\_cols| = q \le m$, the decoding procedure of LightMSR recovers all lost packets; equivalently, the constructed linear system has full rank wq.

b) Assumptions: (1) Each row of Λ partitions $\{1, \ldots, w\}$ into m disjoint groups of size w/m, and all groups are globally distinct as sets. (2) For a lost column j, its group is denoted by G_j, and the remaining groups in the same Λ row are mapped bijectively to $parity_2, \ldots, parity_m$; moreover, cross-layer packets used by $parity_2, \ldots, parity_m$ are always selected from offsets different from the current offset. (3) The encoding coefficients are drawn from a Cauchy matrix Ψ.

c) Proof: Let $lost\_cols = \{j_1, \ldots, j_q\}$. The unknown packets are

$$x = \{d_{i,j} \mid 1 \le i \le w,\ j \in lost\_cols\},$$

with total number wq. For each $j \in lost\_cols$, let G_j be its associated group and let

$$B = \bigcup_{j \in lost\_cols} G_j.$$

For any fixed offset i and the q parity families $t = 1, \ldots, q$, consider the q equations $\{p_{i,t}\}$. After moving all known terms ($j \notin lost\_cols$) to the right-hand side, the coefficients of $\{d_{i,j_1}, \ldots, d_{i,j_q}\}$ in these q equations form the $q \times q$ submatrix

$$M_i = \Psi\big[\{i + (t-1)w\}_{t=1}^{q},\ \{j_1, \ldots, j_q\}\big].$$

Since Ψ is Cauchy and the selected rows/columns are distinct, M_i is invertible.

Step 1: Base equations. For each $i \in B$, LightMSR includes $\{p_{i,t} \mid 1 \le t \le m\}$, hence it includes in particular $\{p_{i,t} \mid 1 \le t \le q\}$. Therefore, for every $i \in B$, the system contains the q equations whose same-layer coefficient matrix on $\{d_{i,j_1}, \ldots, d_{i,j_q}\}$ is M_i, and thus provides q independent pivots. Hence Step 1 yields $q|B|$ pivots.

Step 2: Supplementary equations. For each offset $i \notin B$ (if any), LightMSR adds $\{p_{i,t} \mid 1 \le t \le q\}$. By Assumptions (1)–(2), any cross-layer term in these equations involves only unknowns $d_{z,j}$ with $z \neq i$; moreover, the disjoint partition within each Λ row and the global group uniqueness ensure that these cross-layer selections do not collapse into a degenerate self-dependence on the same offset. Hence, treating all unknowns with $z \neq i$ as parameters, the q supplementary equations at offset i reduce to a linear system in the q unknowns $\{d_{i,j_1}, \ldots, d_{i,j_q}\}$ with coefficient matrix M_i. Therefore, these q equations uniquely determine $\{d_{i,j_1}, \ldots, d_{i,j_q}\}$ and contribute q independent pivots for this offset. Repeating for all $i \notin B$ yields $q(w - |B|)$ additional pivots.

Step 3: Full rank. The total number of pivots is

$$q|B| + q(w - |B|) = wq.$$

Therefore the coefficient matrix has full rank wq, and all lost packets are uniquely recoverable. □

D. Bounds and Structural Analysis of Repair Bandwidth

a) Setup: Reconstructing q data blocks requires extracting wq equations from $parity$ (reading wq parity packets). Let F be the number of data packets read; total packets read: $wq + F$.

b) Lower bound: When the indices of groups corresponding to the lost columns are all distinct, we have $|B| = (w/m)q$. In this case, only the base equations are present, with no supplementary equations, and solving the base system requires reading relatively few data packets from other layers. If no additional cross-layer data packets need to be read, then the minimal theoretical number of packets read is

$$wq + (k-q)\frac{w}{m}q \;=\; (k+m-q)\frac{w}{m}q.$$

c) Upper bound: When overlaps occur in the indices of groups corresponding to the lost columns, both base and supplementary equations are present, and solving the system requires reading more data packets from other layers. If, across the w layers, the same-layer packets contained in $parity_1, \ldots, parity_m$ and the cross-layer packets contained in $parity_2, \ldots, parity_m$ are all involved, then

$$F_{\max} = w\left[(k-q) + (m-1)\frac{k}{m}\right] = w\left[\frac{(2m-1)k}{m} - q\right],$$

and the maximal theoretical number of packets read is

$$wq + w\left[\frac{(2m-1)k}{m} - q\right] \;=\; w\,\frac{(2m-1)k}{m}.$$

d) Structural Growth of Repair Bandwidth: The total number of packets read by LightMSR is $wq + F$, where wq parity packets provide the required linear equations and F counts the data packets read to construct the right-hand side.

As q increases, the required rank of the decoding system grows linearly as wq. When the union of involved groups B satisfies $|B| = (w/m)q$, the base equations already supply wq independent equations. However, overlaps among the q groups reduce $|B|$ below $(w/m)q$, creating an equation deficit

$$\Delta = wq - m|B| > 0,$$

which must be compensated by supplementary equations. The data packet reads F grow for two structural reasons:

First, F contains the same-layer known terms from all involved layers. Since F scales at least proportionally to $|B|$, and the expected size $\mathbb{E}[|B|]$ increases with q, the base component of F increases as q grows.

Second, when overlaps occur ($\Delta > 0$), supplementary equations are required. Each supplementary equation introduces additional same-layer known terms (and potentially cross-layer terms) that must be read. Because the required deficit Δ increases with q when overlaps become more likely, the supplementary component of F increases accordingly.

Therefore, as q increases, both the enlargement of $|B|$ and the increasing likelihood of overlap induced supplementary equations jointly cause a structural growth of F, leading to higher repair bandwidth. When q becomes close to m, the reconstruction process involves nearly all data packets, and the repair bandwidth correspondingly approaches saturation.

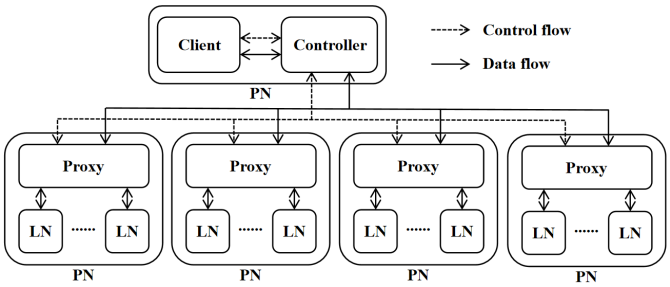


Fig. 3: Architecture of the experimental prototype. LN denotes a logical node, and PN denotes a physical node.

IV. Evaluation

In this section, we evaluate the repair and degraded read performance of LightMSR in an in-memory system built on Memcached 1.6.14. We compare LightMSR with wide-stripe RS, the most widely used erasure code, and with other MSR codes, namely Clay and HashTag. In addition, we evaluate normal read and write performance under YCSB workloads.

A. Experimental Setup

We implement LightMSR in C++, with ISA-L and Jerasure used for Gaussian elimination and matrix multiplication. The LightMSR algorithm contains approximately 1K SLoC. All codes are evaluated on the same experimental platform. Our experiments are conducted on five identical physical machines, each equipped with Intel Xeon Gold 5318Y CPUs and 128GB DDR4 memory. One machine hosts the client and the controller, while the other four machines form the storage cluster. We evaluate the system under both 10G TCP [4], [22] and 100G RDMA [9] network environments.

LightMSR requires Gaussian elimination and matrix multiplication, which are not specific to any CPU architecture. Our prototype uses ISA-L when its $GF(2^8)$ routines are applicable, because ISA-L is a widely used and highly optimized erasure coding library on Intel platforms. For configurations that require $GF(2^{16})$, which is beyond ISA-L's supported mode, we use Jerasure instead. Thus, ISA-L improves the absolute performance of our prototype on Intel CPUs, but the correctness and applicability of LightMSR do not depend on ISA-L.

Figure 3 illustrates the overall architecture. The client is responsible for issuing user read and write requests. All requests are forwarded to the controller, which serves as the centralized control plane of the prototype. The controller maintains global metadata, uniformly and randomly places data blocks across all logical nodes, coordinates request execution with storage side proxies, receives failure and repair completion notifications, and determines whether the system should operate in normal read mode or degraded read mode. This centralized design is reasonable in our testbed since all coding schemes are evaluated under the same control path.

On the storage side, each of the four physical machines runs one proxy process and $(k+m)/4$ Memcached instances, resulting in a total of $k+m$ Memcached instances in the storage cluster. Each Memcached instance emulates one logical node, and all system operations, including placement, read/write, failure injection, and repair, are performed at the logical-node granularity rather than the physical-node granularity [45].

Each proxy communicates with the controller and manages all Memcached instances on its local machine. Upon receiving requests from the controller, the proxy reads or writes the corresponding KV pairs from or to local Memcached instances. Data are stored as block-level KV pairs. Therefore, RS-based repair transfers entire blocks. For MSR-based repair, although a proxy reads a full block from the corresponding Memcached instance, it transmits only the required packets for recovery. This design is adopted because memory I/O is substantially faster and is not the dominant bottleneck in our in-memory setting. We use block-granularity storage, since packet-granularity placement would introduce additional metadata and storage management overhead.

In addition to serving requests, each proxy monitors the liveness of its local Memcached instances. When a Memcached instance fails, the corresponding proxy reports the failure to the controller, obtains metadata, and launches a new Memcached instance to replace the failed one for repair. After the controller receives a failure notification, it switches the system from normal read mode to degraded read mode so that subsequent requests can be served correctly during recovery. Once repair is completed, the responsible proxy reports completion to the controller, and the controller switches the system back to normal read mode. In our implementation, write encoding and degraded read decoding are performed at the controller, while repair decoding is performed at the responsible proxy.

B. Parameter Settings

The block size is the same across schemes within each experiment to ensure fairness, since each scheme is required to protect and recover the same block object. Since the w is intrinsic to each code construction, keeping the same block size necessarily results in different packet sizes. This difference is therefore part of the coding tradeoff being evaluated, rather than an artifact of the implementation. We compare the cost of recovering the same failed block, rather than enforcing the same repair granularity across codes.

In the comparison with RS codes, the k value is set to 16, 32, 64, 128, 192, and 256, respectively (typical wide-stripe k values). According to the mathematical model of LightMSR encoding and the definition of the minimum w, the corresponding w_{min} values are 8, 12, 12, 12, 12, and 16, respectively. To facilitate system management, the packet size is set to 4KB. Accordingly, for $w = 8$–16, the corresponding block sizes are in the range of 32KB to 64KB ($4w$KB).

In the comparison with other MSR codes, existing MSR implementations fail to support k values larger than 12 because the repair paths are hard-coded in advance under the wide-stripe setting ($m = 4$). When k exceeds the predefined range, data repair becomes infeasible. Therefore, k is fixed to 12 in this part of the evaluation. The w values for the Clay code

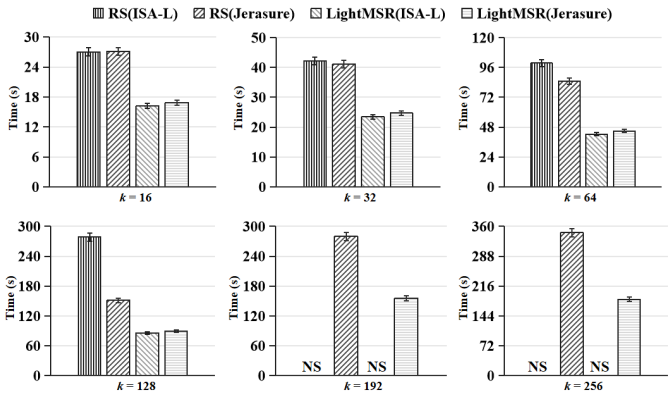


Fig. 4: Repair time of RS and LightMSR for a single-node failure under different k values over a 10G TCP network (NS indicates that ISA-L does not support code implementations with $k > 128$).

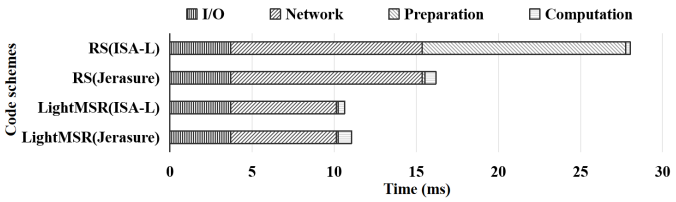


Fig. 5: Breakdown of the average degraded read latency for a single stripe in RS and LightMSR under a single-node failure with $k = 128$ over a 10G TCP network.

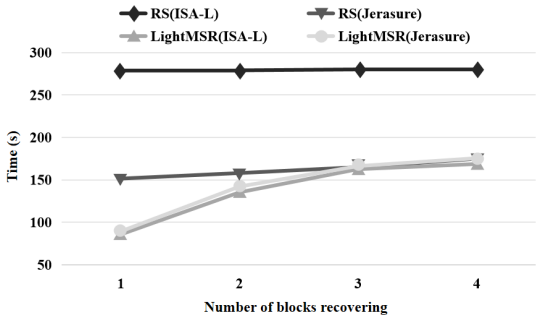


Fig. 6: Repair time of RS and LightMSR for different numbers of node failures with $k = 128$ over a 10G TCP network.

and HashTag code are determined by their respective mathematical constructions and are set to 256 and 64, respectively. LightMSR is evaluated with $w_{min} = 8$, and we additionally include $w = 16$ and $w = 64$ in the comparison as sensitivity tests. Since the w values vary significantly across different MSR codes, enforcing a fixed 4KB packet size for all schemes would lead to excessively large block sizes when w is large, which is impractical for in-memory systems. Therefore, we use $w = 64$ as a reference and set the packet size to 4KB under this configuration (256KB block size). For smaller w, the packet size remains a multiple of 4KB, while for larger w we relax this constraint to avoid overly large blocks.

Since the coefficient matrix Ψ for LightMSR is at least the $(mw) \times (k + \frac{k}{m})$ Cauchy matrix, the finite field $GF(2^n)$ must satisfy $2^n \geq mw + k + \frac{k}{m}$. However, the ISA-L library only supports computations over $GF(2^8)$; therefore, all test cases with $k > 128$ are implemented using the Jerasure library under the $GF(2^{16})$ arithmetic mode. In addition, all MSR codes employ the ISA-L library to accelerate matrix operations in the comparison for MSR codes.

C. Repair and Degraded Read Performance

Each Memcached instance is preloaded with 512MB of synthetic data using YCSB. We evaluate both the total repair time for failed logical nodes and the latency breakdown of degraded reads.

1) Comparison with RS:

a) Single-Node: Under a 10G TCP network, we measure the total time required for single-node repair. As shown in Figure 4, LightMSR significantly reduces the single-node repair time compared with RS, and this performance gap increases further as k grows.

As shown in Figure 5, to further break down the latency of single-stripe recovery, we measure the average degraded read latency of a single stripe and report the contribution of each part at $k = 128$. This is because degraded reads follow the same recovery subroutine as single-stripe repair, namely packet retrieval and linear equation solving.

The I/O part corresponds to the Get operations from Memcached instances, and network part represents read requests from controller. The preparation part mainly involves setting up the decoding matrices and scheduling data computation. For RS, this includes matrix inversion, while for LightMSR, it involves reduced row operation—both belonging to Gaussian elimination operations. The computation part corresponds to matrix multiplications.

Compared to Jerasure, ISA-L has fewer layers of software abstraction; thus, it performs less efficiently for Gaussian elimination operations (e.g., RS decoding preparation), but achieves higher efficiency in matrix multiplication. In contrast, Jerasure's additional software optimizations improve the efficiency of Gaussian elimination (e.g., the reduced row operations in LightMSR preparation) but decrease matrix multiplication performance. Consequently, RS implemented with ISA-L exhibits consistently higher preparation latency but lower computation latency in decoding.

Compared with RS codes, LightMSR also requires Gaussian elimination operations during the preparation part, which are analogous to the matrix inversion procedures used by RS. However, because the matrices involved in LightMSR are significantly smaller, its preparation latency is negligible compared with RS. On the other hand, LightMSR performs multiple small matrix multiplications, which lead to a slight increase in computation latency compared with RS. Overall, LightMSR achieves clear advantages in single-node repair, in terms of both decoding computation and repair bandwidth.

b) Multi-Node: As shown in Figure 6, we further measure the total time required to repair different failed nodes for both RS and LightMSR. Among all implementations, the ISA-

TABLE II: The percentage of LightMSR's average repair bandwidth relative to RS's repair bandwidth when recovering different numbers of data blocks.

(k, w)	1-block	2-block	3-block	4-block
$(16, 8)$	40.5%	79.1%	100.8%	101.0%
$(32, 12)$	40.4%	79.9%	100.3%	100.9%
$(64, 12)$	40.3%	80.1%	100.3%	100.8%
$(128, 12)$	40.3%	80.1%	99.6%	100.8%
$(192, 12)$	40.3%	80.1%	99.4%	100.6%
$(256, 16)$	40.2%	79.9%	99.4%	100.9%

L–based LightMSR consistently delivers the best performance and maintains a clear advantage when repairing one or two nodes. The Jerasure-based LightMSR performs slightly worse, but even in the worst case, it remains roughly on par with the Jerasure-based RS implementation. In contrast, the ISA-L–based RS shows the poorest performance, primarily due to its consistently high preparation latency.

As the number of failed nodes increases, the advantages of LightMSR diminish significantly. This is primarily because LightMSR no longer retains a bandwidth advantage during the repair process. In addition, the preparation phase of LightMSR mainly involves Gaussian elimination operations, and the matrix size depends on the value of w and the number of lost data blocks. As the number of failed blocks grows, the resulting increase in decoding computation further reduces LightMSR's advantage, leading to a noticeable increase in overall repair time.

We measured the average repair bandwidth of LightMSR when recovering different numbers of data blocks, expressed as a percentage of the bandwidth required by RS. The results in Table II show that, regardless of the value of k, LightMSR's bandwidth usage during single-block and two-block recovery consistently remains around 40% and 80%, respectively, of that of RS. Although large-q failures are possible in wide-stripe systems, their likelihood decreases as q increases. Furthermore, even for large q, LightMSR does not suffer additional performance degradation compared with RS, which preserves its practical value.

c) Memory Consumption: Figure 7 presents the memory usage of RS and LightMSR to recover different numbers of data blocks within a single stripe at $k = 128$. Since Jerasure is more extensively optimized for Gaussian elimination operations, it requires greater memory resources to support these optimizations. Consequently, the Jerasure-based RS implementation exhibits higher memory usage during decoding.

Although LightMSR involves a larger number of matrices for packets, each matrix is relatively small in size. Its dimensions depend only on w and the number of lost data blocks. As a result, LightMSR's memory consumption is lower than that of Jerasure-based RS, yet slightly higher than ISA-L–based RS. Furthermore, the memory usage of both RS and LightMSR remains largely unaffected by changes in the number of lost data blocks and shows stable growth as k increases.

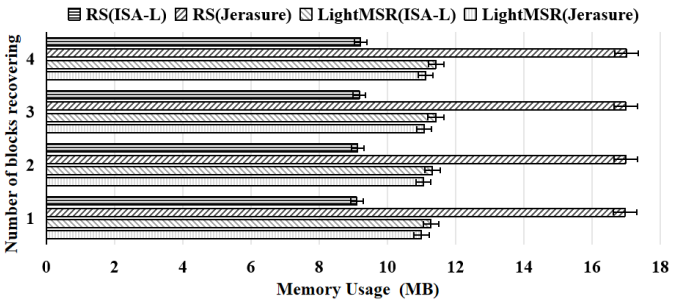


Fig. 7: The memory required by RS and LightMSR to recover different numbers of data blocks with $k = 128$.

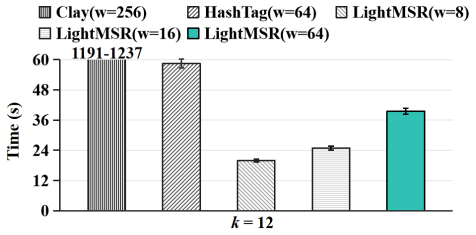


Fig. 8: Repair time of other MSR and LightMSR for a single-node failure with $k = 12$ over a 10G TCP network.

2) Comparison with Other MSR Codes:

a) Single-Node: Figures 8 and 9 compare LightMSR with Clay and HashTag codes over a 10G TCP network, where LightMSR is evaluated with varying w values and Clay/HashTag with fixed w value. Traditional MSR implementations lack an efficient algorithm for locating valid data packets during repair, and a matrix transformation algorithm that enables parallelized computation.

As a result, implementations of Clay and HashTag must explicitly schedule computation steps during the preparation part and execute them sequentially during computation part, both of which incur substantial time overhead. These costs increase further as w grows. Moreover, due to the intrinsic complexity of their algorithms, such codes are difficult to implement efficiently and are generally limited to small k values and single-block recovery scenarios.

The overall latency of Clay and HashTag is significantly higher than that of LightMSR. In particular, Clay codes with larger w values exhibit much higher I/O and network latencies, in addition to greater computational overhead. Even as w increases, LightMSR maintains substantially lower latency compared to HashTag with the same w, owing to its efficient packet location and matrix transformation algorithms that enable effective computation parallelization.

To further examine the reduction in repair bandwidth achieved by each MSR code, we measure the single-block repair bandwidth for all MSR codes. As shown in Table III, the Clay code achieves the lowest single-block repair bandwidth

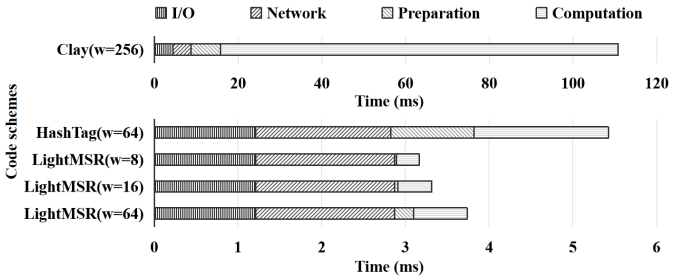


Fig. 9: Breakdown of the average degraded read latency for a single stripe in other MSR and LightMSR under a single-node failure with $k = 12$ over a 10G TCP network.

TABLE III: The percentage of single data block repair bandwidth of different MSR codes relative to RS repair bandwidth with $k = 12$.

Schemes	Clay	HashTag	LightMSR	LightMSR	LightMSR
w	256	64	8	16	64
Percentage	30.5%	38.8%	40.3%	40.3%	40.2%

under the given w value. However, its excessively large w value leads to very small packet sizes, which in turn causes frequent I/O and network transmission requests. This results in higher I/O and network latencies. Increasing the packet size to mitigate this issue would enlarge the overall block size, which is undesirable in in-memory systems.

Although the HashTag code shares a similar structural design with LightMSR, the latter employs a faster randomized generation strategy during encoding. While the single-block repair bandwidth of LightMSR is only marginally higher than that of HashTag, LightMSR's randomized construction ensures that its repair bandwidth remains stable across varying w values, demonstrating both efficiency and robustness.

b) Memory Consumption: Figure 10 presents the memory usage of other MSR and LightMSR to recover single-block within a single stripe at $k = 12$. The Clay code exhibits the highest memory consumption, primarily due to its large w value and the extensive data-scheduling operations required during single-block recovery. Compared with HashTag, LightMSR exhibits higher memory usage. This is because the core of LightMSR's algorithm relies on dynamically performing multiple parallel matrix operations, rather than following a hard-coded path with a fixed computation order.

In addition, the memory consumption of LightMSR does not fluctuate significantly as w varies. Although the matrix dimensions involved in the decoding computation change, the overall amount of data processed remains largely unchanged.

D. Extended Evaluation under 100G RDMA

To examine how LightMSR performs under a higher-speed network environment, we replace the 10G TCP network in Figure 4 with a 100G RDMA network. Since, in all previous experiments, changing the network environment only affects

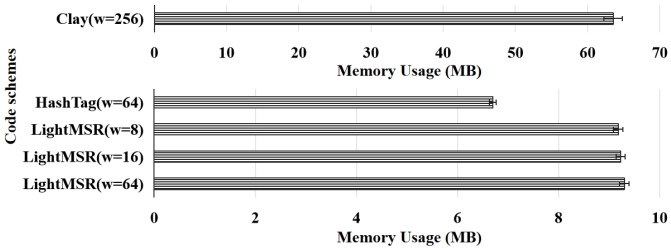


Fig. 10: The memory required by MSR codes with $k = 12$.

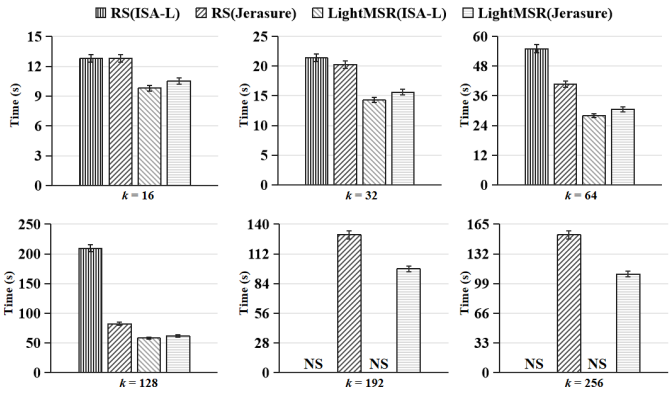


Fig. 11: Repair time of RS and LightMSR for a single-node failure under different k values over a 100G RDMA network (NS indicates that ISA-L does not support code implementations with $k > 128$).

the network latency while having negligible impact on other parts, we select the representative experiment in Figure 4 to demonstrate the influence of network speed.

As shown in Figure 11, LightMSR still maintains a clear advantage in single-node repair, and the gap between LightMSR and RS continues to widen as k increases. Although this advantage becomes smaller compared with Figure 4 due to the faster network, LightMSR still achieves better repair performance under a high-speed network.

As shown in Figure 12, compared with Figure 5, we observe that only the network latency exhibits a noticeable change for both RS and LightMSR. Due to the deployment of multiple logical nodes on a single physical node, together with the overhead of managing multiple Memcached instances, the network latency becomes lower than the I/O latency.

Although the repair network bandwidth is no longer a dominant bottleneck under faster network conditions, LightMSR still maintains a clear advantage in node repair performance, as it continues to reduce network latency and provides more efficient single-block repair computation.

E. Read and Write Performance

We use read-only and write-only YCSB workloads to evaluate the normal read and write throughput of each code. Since all codes considered in this paper are systematic, their normal read performance is identical. The measured throughput is shown in Figure 13. Specifically, $k = 12$ corresponds to

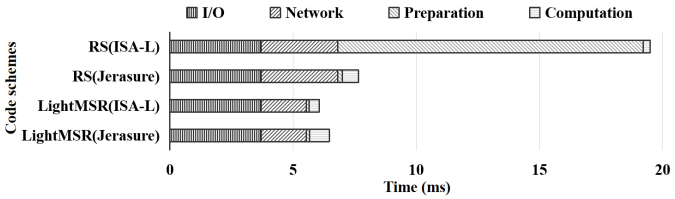


Fig. 12: Breakdown of the average degraded read latency for a single stripe in RS and LightMSR under a single-node failure with $k = 128$ over a 100G RDMA network.

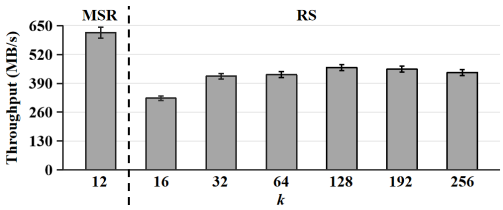


Fig. 13: Normal read throughput of all codes under different k values over a 10G TCP network.

the normal read throughput of other MSR and LightMSR measured under the experimental setup described in IV-C1, with the block size fixed at 256KB. By contrast, $k \geq 16$ corresponds to the normal read throughput of RS and LightMSR measured under the experimental setup in IV-C2, where the block size varies from 32KB to 64KB.

As shown in Figure 14, over a 10G TCP network, we measure the write throughput of RS and LightMSR across different k values. In the encoding preparation of both RS and LightMSR, the process does not involve the complex Gaussian elimination required in decoding preparation. For example, RS only needs to construct the Cauchy or Vandermonde matrix, while LightMSR only prepares a set of matrices derived from the generator matrix for subsequent computations. As a result, the encoding preparation latency of both RS and LightMSR is almost negligible. Moreover, along the write path, the I/O and network latencies of RS and LightMSR are similar, making matrix multiplication (Computation) the primary source of performance differences. Although LightMSR requires more matrix multiplications than RS, each multiplication involves relatively small matrices. Consequently, even as k increases, the gap between LightMSR and RS remains small. Furthermore, because ISA-L provides more efficient matrix multiplication, the ISA-L implementation of LightMSR achieves higher write throughput.

As shown in Figure 15, we further compare the normal write throughput of different MSR codes. Among all MSR codes, LightMSR achieves the best write performance. Although its throughput decreases slightly as w increases due to the larger number of packet operations, its lightweight encoding process and parallel matrix operation prevent the severe throughput degradation typically observed in traditional MSR codes.

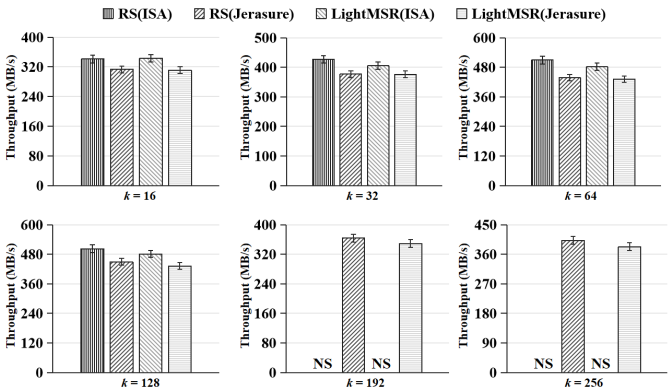


Fig. 14: Write throughput of RS and LightMSR under different k values over a 10G TCP network (NS indicates that ISA-L does not support code implementations with $k > 128$).

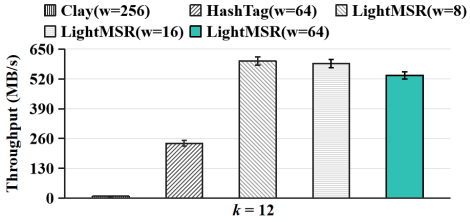


Fig. 15: Write throughput of other MSR and LightMSR with $k = 12$ over a 10G TCP network.

V. Discussion and Limitations

A. Large-q Failures and Repair Bandwidth

LightMSR is a matrix-based MSR design for wide-stripe settings. It introduces a refined feasible range of w, a lightweight encoding process, and an efficient repair algorithm, thereby reducing the design complexity of existing MSR schemes while preserving the repair bandwidth advantage of MSR for small-q failures.

However, this advantage degrades as q increases. This is a theoretical limitation of the current design. As more blocks fail, more unknown packets and linear equations are involved, while overlaps among packet groups reduce the amount of independent information available for reconstruction.

As discussed in Section III-D, different loss patterns may therefore lead to different repair bandwidth even for the same value of q. When $q = 3$ or $q = 4$, the repair process approaches near full sub-stripe recovery, so the repair bandwidth becomes close to saturation and the decoding cost also increases due to larger matrix operations.

Although large-q failures may occur in wide-stripe systems, their probability decreases as q increases. Moreover, in our prototype, logical blocks are uniformly and randomly placed across storage nodes. Therefore, the repair bandwidth variation caused by different group overlap patterns is averaged across many stripes during node repair, and its impact on total repair time is limited in practice for the same value of q. A future direction is to combine the group overlap properties of Λ with

placement strategies to further reduce the repair bandwidth for large-q failures.

B. Evaluation Scope

Our evaluation focuses on stable network environments where no additional foreground workload or external traffic is injected during repair. This setting is chosen to obtain controlled measurements of the algorithm itself. In real deployments, repair may coexist with foreground read/write traffic, bandwidth contention, packet loss, or straggler nodes, which are typically addressed by dedicated algorithms [2], [49]. In addition, following prior wide-stripe studies [45], our prototype deploys multiple logical nodes on each physical node. Therefore, this paper focuses on evaluating the benefit of the proposed algorithm under a stable logical-node-based system setting. Our prototype is intended to validate the coding algorithm itself rather than to model all production-level deployment factors. All schemes are evaluated on the same platform to ensure a fair comparison. In addition, since LightMSR uses a smaller value of w, its sensitivity to network fluctuations and related factors lies between that of RS and conventional MSR codes.

C. Specialization to $m = 4$

LightMSR in this paper is specialized to the practical wide-stripe setting with $m = 4$, which is commonly used in wide-stripe erasure-coded systems. This specialization keeps the encoding and repair procedures lightweight and avoids the additional complexity introduced by supporting arbitrary m. The matrix-based formulation and group-based construction are not inherently limited to $m = 4$, and can in principle be extended to other fixed m values for specialized deployments. Such extensions mainly require rebuilding the concrete algorithmic structure rather than changing the underlying matrix-based methodology. We leave this engineering extension as future work.

VI. Related work

Erasure coding is no longer limited to traditional disk-based storage systems, and has increasingly been introduced into in-memory systems to reduce redundancy. Recently, wide-stripe erasure coding [10], [14], [21], [28], [45], [46] has also been adopted in in-memory systems to further lower storage overhead [4], [22], [48]. However, this makes the problem of high repair bandwidth more severe. Taking the RS code as an example, the repair process requires reading k blocks. In wide-stripe settings, in order to minimize redundancy as much as possible under the same fault tolerance requirement, m is set to 4 while k becomes very large, which significantly degrades the repair performance.

Currently, there is limited research addressing the issue of excessive repair bandwidth in wide-stripe settings, as many existing optimization techniques become difficult to apply under such conditions [32]. Early RAIT [13] shares the high-level idea of stripe-based parity protection, but it targets tape archival reliability and transparent virtualization. Methodologically, RAIT constructs fixed vertical and diagonal parity stripes over tape data stripes and reconstructs erased stripes along these predefined parity lines. It does not address the repair bandwidth bottleneck caused by large k in wide-stripe in-memory storage, where efficient packet-level repair and low computation latency are critical.

Although existing systematic MSR schemes [16], [25], [27], [36], [38] preserve the same normal read performance as RS and can theoretically reduce repair bandwidth, they are difficult to apply in practice due to their high design complexity and the lack of complete and efficient repair algorithms in wide-stripe settings.

At present, only two approaches (LRC [14] and MBR [22]) attempt to optimize repair bandwidth in wide-stripe settings. However, both methods achieve bandwidth reduction at the cost of increased redundancy, aiming to strike a trade-off between repair bandwidth and storage overhead. This trade-off, however, contradicts the fundamental goal of wide-stripe, which is to minimize redundancy as much as possible. Therefore, it is meaningful to design an MSR scheme that can be practically deployed in wide-stripe in-memory systems.

VII. Conclusion

In this paper, we present LightMSR, a lightweight MSR code tailored for wide-stripe settings. It reduces the design complexity of MSR codes while preserving the repair bandwidth advantage of MSR in wide-stripe systems. We implement LightMSR with ISA-L and Jerasure in a Memcached-based in-memory system. Experimental results show that LightMSR substantially reduces repair bandwidth and achieves low repair latency compared with wide-stripe RS, while also maintaining low computation overhead. In addition, compared with existing MSR implementations, LightMSR provides significantly lower design complexity and better practical efficiency. Its write performance also remains competitive due to the lightweight encoding process and efficient matrix operations. Overall, LightMSR demonstrates that MSR codes can be made practical for wide-stripe in-memory systems through a lightweight matrix-based construction.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Key R&D Program of China No. 2023YFB4502703 and the National Natural Science Foundation of China under Grant No. U22A2027. The authors used LLM-based tools solely for language polishing and translation assistance.

References

[1] F. André, A. Kermarrec, E. L. Merrer, N. L. Scouarnec, G. Straub, and A. van Kempen, "Archiving Cold Data in Warehouses with Clustered Network Coding," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2014, pp. 21:1–21:14.

[2] Y. Cai, S. Lin, Z. Shen, J. Yang, and J. Shu, "ChameleonEC: exploiting tunability of erasure coding for low-interference repair," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 15–28.

[3] 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 Transactions on Computers*, vol. 63, no. 1, pp. 31–44, 2014.

[4] L. Cheng, Y. Hu, Z. Ke, J. Xu, Q. Yao, D. Feng, W. Wang, and W. Chen, "LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–15.

[5] L. Cheng, Y. Hu, and P. P. C. Lee, "Coupling Decentralized Key-Value Stores with Erasure Coding," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019, pp. 377–389.

[6] A. G. Dimakis, B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[7] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, p. 61–74.

[8] C. Gan, Y. Hu, L. Zhao, X. Zhao, P. Gong, and D. Feng, "Revisiting Network Coding for Warm Blob Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2025, pp. 139–154.

[9] J. Gao, J. Shu, B. Yan, Y. Zhang, and K. Huang, "Stripeless Data Placement for Erasure-Coded In-Memory Storage," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025, pp. 821–838.

[10] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen, "Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 233–248.

[11] Y. Hu, P. P. C. Lee, and K. W. Shum, "Analysis and Construction of Functional Regenerating Codes with Uncoded Repair for Distributed Storage Systems," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2013, pp. 2355–2363.

[12] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012, pp. 15–26.

[13] J. Hughes, C. Milligan, and J. Debiez, "High Performance RAIT," in *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST)*, 2002.

[14] S. Kadekodi, S. Silas, D. Clausen, and A. Merchant, "Practical Design Considerations for Wide Locally Recoverable Codes (LRCs)," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2023, pp. 1–16.

[15] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg, "On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018, pp. 865–877.

[16] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby, "HashTag Erasure Codes: From Theory to Practice," *IEEE Transactions on Big Data*, vol. 4, no. 4, pp. 516–529, 2018.

[17] Q. Li, L. Xu, Y. Li, M. Lyu, W. Wang, P. Zuo, and Y. Xu, "Enabling Efficient Erasure Coding in Disaggregated Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 1, pp. 154–168, 2024.

[18] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "BCStore: Bandwidth-Efficient In-Memory KV-Store with Batch Coding," in *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2017.

[19] X. Li, K. Cheng, K. Tang, P. P. C. Lee, Y. Hu, D. Feng, J. Li, and T. Wu, "ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2023, pp. 17–32.

[20] X. Li, R. Li, P. P. C. Lee, and Y. Hu, "OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019, pp. 331–344.

[21] X. Li, Z. Yang, J. Li, R. Li, P. P. C. Lee, Q. Huang, and Y. Hu, "Repair Pipelining for Erasure-Coded Storage: Algorithms and Evaluation," *ACM Transactions on Storage*, vol. 17, no. 2, pp. 1–29, 2021.

[22] X. Liu, Y. Hu, W. Wang, D. Feng, and H. Zhou, "Optimizing Encoding and Repair for Wide-Stripe Minimum Bandwidth Regenerating Codes in In-Memory Key-Value Stores," *Journal of Systems Architecture*, vol. 161, p. 103369, 2025.

[23] B. Ma, Y. Hu, D. Feng, R. Wu, and K. Zhang, "Repair I/O Optimization for Clay Codes via Gray-Code Based Sub-Chunk Reorganization in Ceph," in *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST)*, 2024.

[24] X. Niu, G. Zhang, Z. Li, and S. Cai, "DRBoost: Boosting Degraded Read Performance in MSR-Coded Storage Clusters," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2026.

[25] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic, "Opening the Chrysalis: On the Real Repair Performance of MSR Codes," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 81–94.

[26] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 401–417.

[27] K. V. 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 *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 81–94.

[28] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers," in *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM)*, 2014, pp. 331–342.

[29] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[30] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[31] S. E. Rouayheb and K. Ramchandran, "Fractional Repetition Codes for Repair in Distributed Storage Systems," in *Proceedings of the Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2010, pp. 1510–1517.

[32] Z. Shen, Y. Cai, K. Cheng, P. P. C. Lee, X. Li, Y. Hu, and J. Shu, "A Survey of the Past, Present, and Future of Erasure Coding for Storage Systems," *ACM Transactions on Storage*, vol. 21, no. 1, pp. 4:1–4:39, 2025.

[33] Z. Shen, X. Li, and P. P. C. Lee, "Fast Predictive Repair in Erasure-Coded Storage," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 556–567.

[34] M. Subramanian, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, S. Viswanathan, L. Tang, and S. Kumar, "F4: Facebook's Warm BLOB Storage System," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 383–398.

[35] I. Tamo and A. Barg, "A Family of Optimal Locally Recoverable Codes," *IEEE Transactions on Information Theory*, vol. 60, no. 8, pp. 4661–4676, 2014.

[36] I. Tamo, Z. Wang, and J. Bruck, "Zigzag Codes: MDS Array Codes with Optimal Rebuilding," *IEEE Transactions on Information Theory*, vol. 59, no. 3, pp. 1597–1616, 2013.

[37] K. Tang, K. Cheng, H. H. W. Chan, X. Li, P. P. C. Lee, Y. Hu, J. Li, and T. Wu, "Balancing Repair Bandwidth and Sub-Packetization in Erasure-Coded Storage via Elastic Transformation," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2023, pp. 1–10.

[38] M. Vajha, V. Ramkumar, B. Puranik, G. R. Kini, E. A. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi, "Clay Codes: Moulding MDS Codes to Yield an MSR Code," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018, pp. 139–154.

[39] H. Weatherspoon and J. Kubiatowicz, "Erasure Coding Vs. Replication: A Quantitative Comparison," in *Proceedings of the Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002, p. 328–338.

[40] S. Wu, Q. Du, P. P. C. Lee, Y. Li, and Y. Xu, "Optimal Data Placement for Stripe Merging in Locally Repairable Codes," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1669–1678.

[41] S. Wu, Z. Shen, P. P. C. Lee, Z. Bai, and Y. Xu, "Elastic Reed-Solomon Codes for Efficient Redundancy Transitioning in Distributed Key-Value Stores," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 670–685, 2024.

[42] J. Xia, J. Huang, X. Qin, Q. Cao, and C. Xie, "Revisiting Updating Schemes for Erasure-Coded In-Memory Stores," in *Proceedings of the International Conference on Networking, Architecture, and Storage (NAS)*, 2017, pp. 1–6.

[43] J. Xia, L. Luo, B. Sun, G. Cheng, and D. Guo, "Parallelized In-Network Aggregation for Failure Repair in Erasure-Coded Storage Systems," *IEEE/ACM Transactions on Networking*, vol. 32, no. 4, pp. 2888–2903, 2024.

[44] B. Xu, J. Huang, Q. Cao, X. Qin, and P. Xie, "F-Write: Fast RDMA-Supported Writes in Erasure-Coded In-Memory Clusters," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 817–826.

[45] G. Yang, H. Xue, Y. Gu, C. Wu, J. Li, M. Guo, S. Li, X. Xie, Y. Dong, and Y. Zhao, "XHR-Code: An Efficient Wide Stripe Erasure Code to Reduce Cross-Rack Overhead in Cloud Storage Systems," in *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, 2022, pp. 273–283.

[46] Q. Yao, Y. Hu, L. Cheng, P. P. C. Lee, D. Feng, W. Wang, and W. Chen, "StripeMerge: Efficient Wide-Stripe Generation for Large-Scale Erasure-Coded Storage," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 483–493.

[47] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee, "Erasure Coding for Small Objects in In-Memory KV Storage," in *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, 2017, pp. 1–12.

[48] Q. Yu, L. Wang, Y. Hu, Y. Xu, D. Feng, J. Fu, X. Zhu, Z. Yao, and W. Wei, "Boosting Multi-Block Repair in Cloud Storage Systems with Wide-Stripe Erasure Coding," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 279–289.

[49] Y. Zhang, X. Tu, L. Wang, Y. Hu, F. Wang, and Y. Wang, "FullRepair: Towards Optimal Repair Pipelining in Erasure-Coded Clustered Storage Systems," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2023, pp. 107–117.

[50] H. Zhou and D. Feng, "Make Updates Faster: A Fast Multi-Stripe Updates Framework in Erasure-Coded Storage Clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2025, pp. 2251–2265.

[51] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, "Carbink: Fault-Tolerant Far Memory," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 55–71.