

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

WiseCode: Breaking the Scalability Barriers of Wide-Stripe Vector Codes

Sijie Cai, Guangyan Zhang, and Xiao Niu, *Tsinghua University*

https://www.usenix.org/conference/osdi26/presentation/cai

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

WiseCode: Breaking the Scalability Barriers of Wide-Stripe Vector Codes

Sijie Cai, Guangyan Zhang*, Xiao Niu
Tsinghua University

Abstract

Wide-stripe erasure codes, with tens to over a hundred data chunks per stripe, offer high reliability at low storage overhead. Existing wide-stripe designs that are based on scalar codes (e.g., LRCs in Google and Azure) reduce *repair traffic* but increase *storage overhead*. Although vector codes are theoretically optimal in both metrics, they face severe scalability barriers in wide-stripe deployments.

We present WiseCode, *the first practical and scalable wide-stripe vector-coding approach* that achieves both efficient repair and ultra-low storage overhead. WiseCode overcomes three key scalability barriers through innovations in coding structure, coefficient selection, and coding algorithms. It introduces a template-unfold structure design that avoids sub-packetization blowup, a repetition-minimized search strategy that reduces coefficient search cost, and a two-stage coding algorithm that enables efficient encoding and decoding.

Evaluations on Ceph with ∼100-wide stripes and 1.04–1.06 storage overhead show that WiseCode increases repair throughput by 1.41×–2.18× compared to Google's UCLRCs at equal storage overhead, and also delivers higher throughput even at 2% lower storage overhead. WiseCode retains this advantage when combined with advanced repair-scheduling methods, consistently outperforming UCLRCs.

1 Introduction

Storage systems rely on redundancy mechanisms to tolerate routine device failures [1–4]. As data volumes grow exponentially [5], reducing storage overhead while preserving data reliability becomes increasingly important. In large-scale clusters managing exabytes of data, even a 1% storage overhead reduction saves millions of dollars in capital, operational, and energy costs [6]. This demand has driven the shift from replication to erasure codes [7–16] and now motivates *wide-stripe erasure codes*, or "wide codes" [6, 17, 18]. An (n,k) erasure code divides data into k data chunks and encodes them into $m = n-k$ parity chunks, forming a stripe with storage overhead of n/k (typically ≤ 1.5). Wide codes further reduce overhead by keeping m small while enlarging k. For example, moving from $(11,8)$ to $(106,100)$ reduces storage overhead from 1.375 to 1.06, without sacrificing reliability (§2.1).

Since data recovery is performed online, wide codes should quickly restore lost data [19–21] and minimize interference with foreground workloads [22–24]. Network and I/O resources are often the primary bottlenecks during recovery [24, 25], so reducing *repair traffic*—the amount of data accessed and transferred during recovery—is essential.

Achieving both *low repair traffic* and *ultra-low storage overhead* remains challenging in wide-code designs. Erasure codes fall into two categories: scalar and vector. Scalar codes, such as RS codes [18, 26] and Locally Recoverable Codes (LRCs) [6, 17, 27–30], operate at chunk granularity. RS codes satisfy the *Maximum Distance Separable (MDS)* property, achieving the optimal storage efficiency for a given reliability, but require reading k chunks to repair one; LRCs reduce repair traffic using *local parity* but increase storage overhead. Thus, **scalar codes cannot simultaneously optimize repair traffic and storage overhead.**

Vector codes [31–38], including Clay [34] and RS-ET [36], divide each chunk into α sub-chunks and organize them into α sub-stripes, where α is referred to as the *sub-packetization level*. Each sub-stripe defines a *coding equation* that specifies the relationship among its sub-chunks, allowing it to decode multiple sub-chunks independently. This enables repairing a lost chunk by decoding a small subset of sub-stripes, substantially reducing repair traffic while preserving the MDS property. As shown in Figure 1, implementing a vector code involves three key steps: (1) designing the *coding structure* that specifies sub-chunk variables in each sub-stripe's equation (with unknown coefficients), (2) selecting coefficients that ensure decodability under any 1 to m-chunk failures, and (3) developing efficient encoding/decoding algorithms.

In wide-stripe deployments (e.g., $n \approx 100$), vector codes face three **scalability barriers**:

1. *Sub-packetization blowup.* $(104, 100, \alpha)$ Clay codes require $\alpha = 4^{26}$ to achieve low repair traffic, making them

*Corresponding author: gyzh@tsinghua.edu.cn

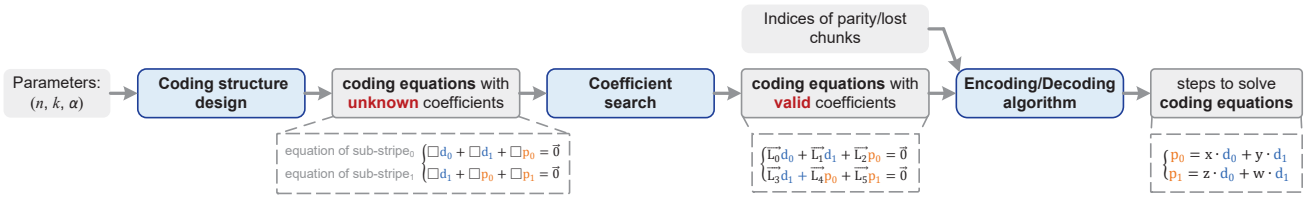


Figure 1: Three steps to realize vector codes—from coding structure design to coefficient search and coding algorithm.

impractical for deployments. RS-ET codes use smaller α but incur over 50% higher repair traffic than Clay codes.

2. *Prohibitive coefficient search.* For $(106, 100, 216)$ RS-ET codes, verifying one coefficient combination takes over 1,000 hours to check 1.7B failure cases. Moreover, many combinations must be tested before finding a valid one.

3. *High computational overhead.* Standard generator-matrix-based encoding achieves only 97 MB/s throughput for $(106, 100, 216)$ vector codes on our testbed, versus several GB/s for $(106, 100)$ scalar codes.

We propose WiseCode, *the first practical and scalable wide-stripe vector-coding approach* that achieves both low repair traffic and ultra-low storage overhead. WiseCode breaks the three scalability barriers via innovations in coding structure design, coefficient selection, and coding algorithms.

Template-unfold structure design. WiseCode takes a narrow MSR (Minimum Storage Regenerating [39]) stripe as a *template* and repeatedly instantiates it to form a wide-stripe structure while preserving the template's α and m. This allows each lost chunk to be recovered by reading sub-chunks from only a subset of sub-stripes. WiseCode selects the widest MSR template under a device-dependent α limit (higher on SSDs than HDDs for the same chunk size) to simultaneously avoid excessive data fragmentation and minimize instantiations that inflate repair traffic. This design achieves near-optimal repair traffic and minimum storage overhead without incurring sub-packetization blowup.

Repetition-minimized coefficient search. WiseCode reduces redundant work in coefficient search using two complementary techniques. *Divide-and-conquer verification* partitions chunks into disjoint groups and verifies decodability only for intra-group failure cases (i.e., all lost chunks lie within the same group), greatly reducing the verification scope while guaranteeing decodability for all failure cases. *Neighborhood-prioritized retry* generates a new candidate for retry by modifying only a small subset of coefficients in a rejected combination, preserving the decodability of most previously-verified failure cases. This significantly reduces per-candidate verification cost and accelerates retries, enabling n up to several hundred with $4 \le m \le 8$ over GF(2^{16}), whereas existing vector codes typically limit n to a few tens.

Two-stage encoding. Motivated by the observation that the generator matrix is much denser (up to 21×) than the sparse coefficients in the coding equations, WiseCode employs a *two-stage encoding* design. The *data aggregation* stage multiplies data by the sparse coefficient matrix to produce an intermediate vector; the *parity solving* stage then computes parity from the intermediate vector by solving the coding equations. WiseCode further optimizes both stages by extracting common operations during data aggregation and decomposing parity solving into computation-efficient steps using a divide-and-conquer strategy. This reduces multiplications from $O(km\alpha^2)$ to $km\alpha + O(m^3\alpha)$, cutting up to 94.6% of multiplications and improving throughput by 5.5×–22.4× in our experiments.

We integrated WiseCode into Ceph [8] as an erasure-code plugin. Evaluations on Ceph with stripe widths near 100 and storage overheads of 1.04–1.06 show that WiseCode delivers a strictly better tradeoff between repair performance and storage overhead than Google's UCLRCs [6], the state-of-the-art wide codes. At equal storage overhead, WiseCode increases repair throughput by 1.41×–2.18× compared to UCLRC; even with 2% lower storage overhead, it reduces repair traffic by 14.4% and improves repair throughput by 6.8%. When combined with the RepairBoost [21] repair-scheduling framework, WiseCode also increases repair throughput by 1.56×–2.38× compared to UCLRC, demonstrating consistent advantages under advanced scheduling.

2 Background and Challenges

2.1 Background

In an (n,k) erasure code, user data are divided into k equal-sized data chunks and encoded into $m = n-k$ parity chunks, forming an n-chunk *stripe* with storage overhead n/k. These n chunks are placed across independent failure domains (e.g., disks, hosts, racks) to tolerate multiple failures. A stripe with m parity chunks satisfies the MDS property [40] if any m lost chunks can be recovered. MDS codes have the minimum storage overhead for a given fault tolerance requirement.

Wide codes. Wide codes increase n while keeping m small (e.g., $n \ge 80$ and $m \le 8$ [6]) to reduce storage overhead without sacrificing reliability. For example, expanding RS codes

from $(11,8)$ to $(106,100)$ reduces storage overhead from 1.375 to 1.06, while increasing MTTDL (mean time to data loss, analyzed via a Markov model; see Appendix A.1) from 3.0×10^{11} to 5.4×10^{16} years. In large-scale storage clusters, degraded stripes are reported to be dominated by single-chunk failures (99.2% at Google [6], 98.08% at Facebook [22]). Moreover, modeling with a Binomial distribution [41, 42]—assuming an independent failure rate of 1/(4 years) and a repair rate of 1/(30 minutes) [17, 27, 28]—demonstrates that a stripe with $n = 100$ resides in a single-chunk failure state for over 99% of its total degraded duration. Therefore, existing wide-code designs (e.g., LRCs) primarily optimize single-chunk repair [6, 17].

Scalar and vector codes. Scalar codes treat each of the k data chunks as one variable and generate m coded variables, each corresponding to a parity chunk. Vector codes divide each chunk into α equal-sized sub-chunks and treat them as independent variables, increasing the number of variables in the encoding operation by a factor of α.

Limitations of wide-stripe scalar codes. Wide-stripe scalar codes (e.g., LRCs) reduce repair traffic by adding *local parities* on top of global parities, but this increases storage overhead. Consequently, they cannot achieve efficient repair and ultra-low storage overhead simultaneously.

2.2 Scalability Barriers of Vector Codes

In wide-stripe deployments, vector codes face three scalability barriers.

Sub-packetization blowup for minimal repair traffic. Minimizing repair traffic in vector codes comes at the cost of rapidly increasing α. MSR codes achieve optimal repair traffic [39] with $\alpha \geq m^{\lceil \frac{n}{m} \rceil}$ [43], causing severe and often impractical data fragmentation in wide stripes. For example, $(104,100,\alpha)$ Clay codes [34] require $\alpha = 4^{26}$. In practice, storage systems must cap α to limit I/O overhead from data fragmentation, which inevitably increases repair traffic. For example, when $\alpha = 64$, which corresponds to 64 MB chunks and 1 MB sub-chunks, RS-ET [36] and Hashtag+ [37] codes incur over 50% more repair traffic than the optimal bound. It is therefore challenging to design a coding structure that reduces repair traffic without incurring sub-packetization blowup.

Prohibitive search cost for valid coefficients. Two factors make coefficient search intractable: (1) *Expensive per-candidate verification*. A valid coefficient combination must ensure that every $m\alpha \times m\alpha$ coefficient matrix corresponding to any m-chunk failure case is invertible. In a $(106,100,216)$ RS-ET stripe, this requires checking 1.7 billion 1296×1296 matrices, each taking 2.4 ms on our testbed, totaling 1,130 hours. (2) *Frequent rejects*. Any failed invertibility check discards a candidate, and wide stripes have many independent failure cases, causing extremely high reject rates. With coefficients randomly selected from GF(2^{16}), $(76,72,\alpha)$ or wider RS-ET stripes fail to find any valid combination within 10 million trials in our experiments. The candidate space grows exponentially with the number of coefficients, making exhaustive enumeration infeasible.

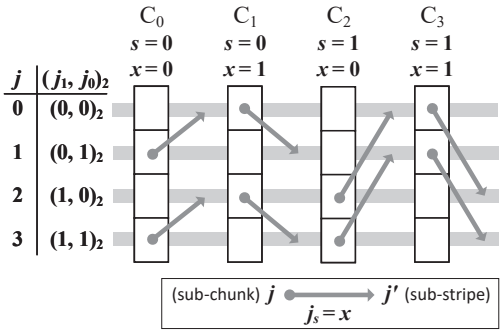


Figure 2: Coding structure of a $(4,2)$ MSR code with $\alpha = 4$. Columns correspond to chunks, each containing 4 sub-chunks. Rows correspond to sub-stripes, each annotated by index j. Rows depict Rule #1 and arrows depict Rule #2.

Excessive encoding cost using the standard method. Standard encoding, multiplying data by a generator matrix, works well for scalar codes [44, 45] but is too costly for vector codes. In the $(106,100,216)$ vector code (§5.1), each column of the generator matrix has up to 126 nonzero entries (vs. 6 in a $(106,100)$ scalar code), dramatically increasing the number of multiplications and lowering encoding throughput to 97 MB/s on our testbed (vs. several GB/s for the scalar code). This motivates developing encoding algorithms that restore computational efficiency.

2.3 MSR Coding Structure and Repair

We introduce the state-of-the-art MSR codes with the lowest sub-packetization levels $\alpha = m^{\lceil \frac{n}{m} \rceil}$ [34, 43, 46], which form the basis of WiseCode.

Coding structure. Each chunk (data or parity, determined by storage systems) is divided into α sub-chunks, and the stripe is organized into α sub-stripes. The sub-chunk to sub-stripe assignment follows two coding rules:

Rule #1. *Sub-Chunk$_j$ of each chunk belongs to Sub-Stripe$_j$ ($0 \leq j < \alpha$).*

Rule #2. *Sub-Chunk$_j$ of Chunk$_i$ also belongs to Sub-Stripe$_{j'}$ when $j' \neq j$. To compute j', express j in the base-m representation as $j = (\ldots, j_2, j_1, j_0)_m$. Let $s = \lfloor \frac{i}{m} \rfloor$, and $x = i \bmod m$. Then replace j_s with x to obtain $j' = (\ldots, j_{s+1}, x, j_{s-1}, \ldots)_m$.*

In Figure 2 for a $(4,2)$ MSR code, rows depict Rule #1 and arrows depict Rule #2. Take Sub-Chunk$_1$ of Chunk$_0$ as an example. Rule #1 assigns this sub-chunk to Sub-Stripe$_1$. Rule #2 sets $i = 0$, $j = (0,1)_2$, $s = 0$, and $x = 0$. This gives $j' = (j_1, x)_2 = (0,0)_2 \neq j$, so the sub-chunk also belongs to Sub-Stripe$_0$.

Coding equations and coefficients. Each sub-stripe defines a linear relation among its sub-chunks, expressed as a

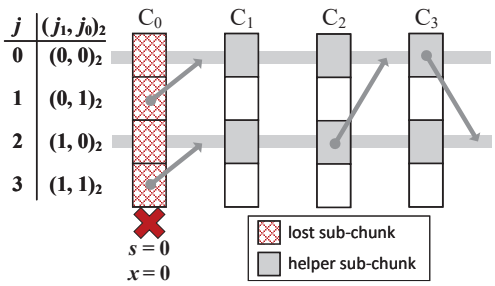


Figure 3: Repair pattern of Chunk_0 in a $(4,2)$ MSR code. Only Sub-Stripe_0 and Sub-Stripe_2 are decoded during repair.

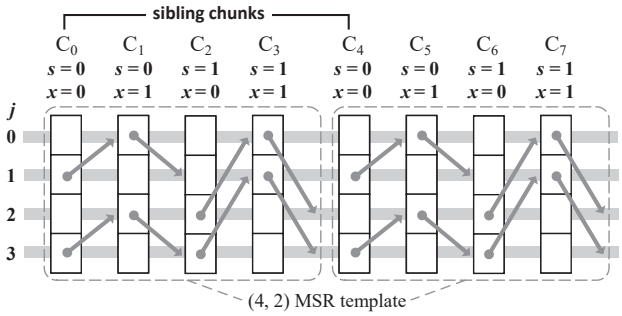


Figure 4: Constructing an $(8,6,4)$ WiseCode stripe using a $(4,2)$ MSR code as the template.

coding equation: $\vec{L}_0 c_0 + \vec{L}_1 c_1 + \cdots = \vec{0}$, where c_i represents a sub-chunk, and $\vec{L}_i$ is an m-element *coefficient vector* over a finite field (e.g., GF(2^8) [34]).

Single-failure repair pattern. Each sub-stripe can recover up to m lost sub-chunks. A lost chunk i is recovered by decoding α/m sub-stripes whose indices j satisfy $j_s = x$, where s and x are defined in Rule #2. Each such sub-stripe contains m lost sub-chunks. Each of the $n-1$ healthy chunks contributes one helper sub-chunk to each selected sub-stripe, incurring the total repair traffic (normalized by α) of $(n-1) \times \frac{\alpha}{m} \times \frac{1}{\alpha} = \frac{n-1}{m}$, which is optimal [39].

Figure 3 shows the repair pattern for Chunk_0 in the $(4,2)$ MSR code. With $s = 0$, and $x = 0$, Sub-Stripe_0 and Sub-Stripe_2 are the two sub-stripes whose indices satisfy $j_s = x$. Sub-Stripe_0 recovers Sub-Chunk_0 and Sub-Chunk_1 of Chunk_0, while Sub-Stripe_2 recovers the remaining two lost sub-chunks. In total, 6 healthy sub-chunks across these two sub-stripes are accessed as helper sub-chunks.

3 Template-Unfold Structure Design

This section presents the template-unfold structure design (§3.1), explains how it achieves near-optimal repair traffic for single-chunk failures (§3.2), shows how it reduces multi-chunk repair traffic (§3.3), and presents the dedicated chunk placement strategy (§3.4).

3.1 Code Construction via Template Unfolding

An (n,k,α) WiseCode constructs the coding structure in two steps: (1) It first selects a (n_{msr}, k_{msr}) MSR code ($n_{msr} < n$) as a *template*, which has the same parity count m and sub-packetization level α with the target coding structure; (2) It then *unfolds the template* by repeatedly instantiating it into multiple *instances* and concatenating sub-stripes with the same index across instances to scale the stripe width, while preserving m and α of the template.

Figure 4 illustrates the coding structure of an $(8,6,4)$ WiseCode where $m = 2$ and $\alpha = 4$, using a $(4,2)$ MSR code as the template. The template is instantiated twice, once for $\text{Chunk}_{0\sim3}$ and once for $\text{Chunk}_{4\sim7}$, with identical sub-chunk to sub-stripe assignments. We refer to chunks from different instances that occupy the same position in the template as *sibling chunks* (e.g., Chunk_0 and Chunk_4). Each chunks inherits the parameters s and x specified by Rule #2 (§2.3) from the template, and sibling chunks share identical s and x.

To generate the n chunks, the (n_{msr}, k_{msr}) template is instantiated $\lceil \frac{n}{n_{msr}} \rceil$ times—so each chunk has $\lceil \frac{n}{n_{msr}} \rceil - 1$ sibling chunks—producing a total of $\lceil \frac{n}{n_{msr}} \rceil \cdot n_{msr}$ chunks. WiseCode supports flexible n by retaining only the first n chunks and treating any remaining chunks as logical zero chunks, which are omitted from physical storage.

While n and k can take arbitrary integer values, α is discrete because the underlying (n_{msr}, k_{msr}) MSR code only supports α in the form of $\alpha = m^{\lceil \frac{n_{msr}}{m} \rceil}$. In practice, WiseCode imposes an upper bound on α by dividing the chunk size (e.g., 64 MB$\sim$1 GB [9, 11]) by an appropriate sub-chunk size (e.g., 1$\sim$4 MB on HDDs and 128 KB on SSDs [47]). WiseCode then selects the MSR template with the largest n_{msr} among those whose α remains within the bound, and sets $k_{msr} = n_{msr} - m$. This design mitigates the I/O overhead from data fragmentation and minimizes instantiations that inflate repair traffic (§3.2).

At this stage, (n,k,α) WiseCode has specified the sub-chunks in each of the α sub-stripes. Each sub-stripe defines a coding equation, where sub-chunks in this sub-stripe appear as variables and the coefficients are specified in Section 4.

3.2 Single-Chunk Failure Repair

An (n,k,α) WiseCode repairs a lost chunk by decoding α/m sub-stripes, each recovering m lost sub-chunks. Their indices j satisfy $j_s = x$, where s and x are the parameters of the lost chunk, inherited from the template (§3.1).

Figure 5 illustrates the repair pattern of Chunk_0 in the $(8,6,4)$ WiseCode. Both Sub-Stripe_0 and Sub-Stripe_2 contain two lost sub-chunks, and decoding them suffices to reconstruct all lost sub-chunks. Each healthy chunk, except Chunk_4, contributes only two helper sub-chunks, while Chunk_4 must

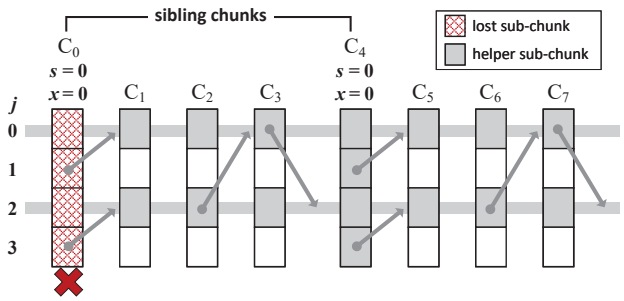


Figure 5: Repair pattern of Chunk_0 in the $(8,6,4)$ WiseCode. Only Sub-Stripe_0 and Sub-Stripe_2 are decoded during repair.

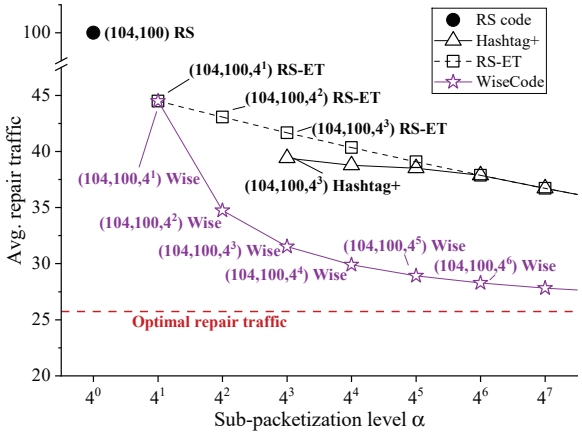


Figure 6: Trade-off between α and repair traffic for $(104,100,\alpha)$ WiseCode, HashTag+, and RS-ET codes.

contribute all 4 sub-chunks as helper sub-chunks. This is because Chunk_4 is a sibling chunk of Chunk_0, which means sub-stripes used to recover Chunk_0 also span the entire Chunk_4.

Generally, the repair pattern of Chunk_i in (n,k,α) WiseCode consists of two parts (proof in Appendix A.4):

❶ Non-sibling part: each healthy chunk that is not a sibling chunk of Chunk_i contributes 1 helper sub-chunk per sub-stripe, totally α/m helper sub-chunks whose indices i satisfy $i_s = x$, where s and x are the parameters of Chunk_i.

❷ Sibling part: each healthy chunk that is a sibling chunk of Chunk_i contributes m helper sub-chunks per sub-stripe. In total, all α sub-chunks serve as helper sub-chunks. □

Traffic analysis. Assuming Chunk_i has z sibling chunks, the repair traffic (normalized by α) is calculated as:

$$\left((n-1-z)\frac{\alpha}{m}+z\alpha\right)\cdot\frac{1}{\alpha}=\frac{n-1}{m}+\frac{m-1}{m}z$$

where the first term $\frac{n-1}{m}$ matches the optimal repair traffic [39], while the second term $\frac{m-1}{m}z$ reflects the overhead introduced by template unfolding.

Since each chunk has $z=\lceil\frac{n}{n_{msr}}\rceil-1$ sibling chunks (§3.1), increasing n_{msr} decreases z and thus reduces repair traffic. However, a larger n_{msr} also increases $\alpha=m^{\lceil\frac{n_{msr}}{m}\rceil}$, exacerbating data fragmentation. To balance this trade-off, WiseCode maximizes n_{msr} under a given upper bound of α (§3.1), thereby *achieving near-optimal repair traffic while mitigating data fragmentation*.

Figure 6 illustrates the trade-off between sub-packetization level (α) and repair traffic in $(104,100,\alpha)$ WiseCode under varied α. With a 64 MB chunk size and 1 MB sub-chunk size, $(104,100,64)$ WiseCode incurs only 22.3% higher repair traffic than the optimal, while with a 512 MB chunk size and 128 KB sub-chunk size, $(104,100,4^6)$ WiseCode incurs only 9.9% higher repair traffic.

Compared with two representative vector codes, HashTag+ [37] and RS-ET [36] codes, WiseCode converges to the optimal repair traffic more rapidly as α increases, achieving 19.3-25.3% lower repair traffic when $4^2\le\alpha\le4^7$ (noting that HashTag+ codes do not support $\alpha=m$ or $\alpha=m^2$). This advantage stems from coding design differences: HashTag+ requires decoding more sub-stripes, while RS-ET accesses more helper sub-chunks per sub-stripe, both of which increase repair traffic.

3.3 Multi-Chunk Failure Repair

Although multiple-chunk failures are rare, there is a growing concern that their likelihood increases in wide stripes in a certain degree [6]. When multiple-chunk failure occurs, WiseCode selects the *minimum subset of sub-stripes* to recover all lost sub-chunks. It uses a greedy strategy, iteratively selecting sub-stripes that maximize *recovery capability*—the number of lost sub-chunks they can recover—until all lost sub-chunks are recoverable.

In each iteration, WiseCode first calculates the recovery capability of the currently selected sub-stripes using a flow model. In this model, each sub-stripe sends up to m flows to the its lost sub-chunks, and any lost sub-chunks that receives at least one flow is considered recoverable. Thus, computing the recovery capability reduces to a max-flow problem. After obtaining the current recovery capability, WiseCode enumerate unselected sub-stripes and adds the sub-stripe that yields the largest increase in recovery capability.

In this way, only a subset of sub-stripes are selected for recovery, thereby significantly reducing the number of helper sub-chunks. For example, in the case of double-chunk failures, $(104,100,64)$ WiseCode incurs an average repair traffic of 66.68, compared to 100 in $(104,100)$ RS codes.

3.4 Fault-Tolerance and Placement Strategy

WiseCode designs its chunk placement policy in two steps: first, it enforces a hard durability constraint to guarantee data persistence against correlated failures; subject to this con-

straint, it then optimizes chunk co-location to minimize cross-rack network traffic.

Uniform fault-tolerance guarantee. Unlike LRCs—where local parities provide less protection than global ones and certain failure patterns involving no more than m chunks remain unrecoverable—WiseCode is designed to achieve the maximum fault-tolerance level, i.e., the MDS property. This property ensures WiseCode can recover from the loss of any m chunks (see §4), thereby provides a uniform fault-tolerance guarantee. This uniformity significantly simplifies chunk placement against correlated failures. For instance, to protect against rack-level correlated failures, a placement policy only needs to ensure that no more than m chunks of the same stripe are placed within a single rack, regardless of their specific chunk indices.

Rack-aware placement. Subject to the constraint that no more than m chunks reside within a single rack, WiseCode leverages its template-unfold design to minimize cross-rack traffic in hierarchical network topologies, where cross-rack bandwidth is typically much more constrained than intra-rack bandwidth [48]. Specifically, when repairing a lost chunk, sibling chunks contribute their entire data, while non-sibling chunks contribute only partial data. WiseCode exploits this asymmetry by grouping sibling chunks within the same rack whenever possible, thereby maximizing intra-rack data exchange and minimizing cross-rack transfers. For example, compared to random placement, rack-aware placement can save 18.9% cross-rack repair traffic for $(106,100,216)$ WiseCode.

4 Repetition-Minimized Coefficient Search

This section presents the repetition-minimized coefficient search, which combines two techniques: divide-and-conquer verification to reduce per-candidate verification cost (§4.1), and neighborhood-prioritized retry to minimize both the overhead of each retry and the number of retries (§4.2).

4.1 Divide-and-Conquer Verification

Given a candidate coefficient combination, WiseCode verifies the decodability of all failure cases in a *divide-and-conquer* manner. It first partitions chunks into disjoint groups and classifies all failure cases into two categories: *intra-group* failure cases where missing chunks lie within the same group, and *inter-group* failure cases where missing chunks span multiple groups. It then verifies decodability only for the intra-group ones, while safely skipping verification for the inter-group ones. This strategy is enabled by the coefficient design and the chunk-partitioning rule, as detailed below.

Coefficient design. In an (n,k,α) WiseCode, coefficients are generated from mn distinct parameters, denoted λ_0 to λ_{mn-1}. Each Chunk$_i$ is associated with m parameters, specifically λ_{im} to $\lambda_{(i+1)m-1}$. For Sub-Chunk$_j$ of Chunk$_i$, WiseCode selects the parameter $\lambda = \lambda_{im+j_s}$, where s and j_s are defined in Rule #2 (§2.3). The coefficient vector is then constructed as $\vec{\lambda} = [\lambda^0, \lambda^1, \cdots, \lambda^{m-1}]^T$ and applied to all variables corresponding to this sub-chunk in the coding equations.

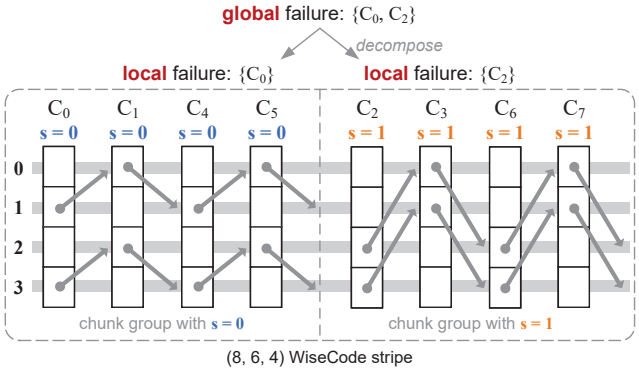


Figure 7: The chunk-partitioning rule groups chunks by s, decomposing a global failure into multiple local failures.

Chunk-partitioning rule. WiseCode partitions chunks by their parameters s, forming groups where all chunks share the same s value. Since $s = \lfloor i/m \rfloor$ for $0 \le i < n_{msr}$ (§2.3) takes $\lceil n_{msr}/m \rceil$ distinct values, the chunks are thus partitioned into $g = \lceil n_{msr}/m \rceil$ groups. As shown in Figure 7, in an $(8,6,4)$ WiseCode where $n_{msr} = 4$ and $m = 2$, chunks are partitioned into $g = 2$ groups corresponding to $s = 0$ and $s = 1$.

Below, we explain the correctness of divide-and-conquer verification. An inter-group failure case, referred to as a *global failure case*, can be decomposed into multiple *local failure cases*, each confined to a single group. As shown in Figure 7, in an $(8,6,4)$ WiseCode stripe, a global failure case $\{\text{Chunk}_0, \text{Chunk}_2\}$ is decomposed into two local failure cases, $\{\text{Chunk}_0\}$ and $\{\text{Chunk}_2\}$. We prove that global failure cases can be safely omitted from verification through Theorem 4.1 and Theorem 4.2.

Theorem 4.1. *A global failure case is decodable if and only if all corresponding local failure cases are decodable.*

Proof sketch. The proof follows a similar approach to prior theoretical work on MSR codes [49]. The key idea is to decompose the coefficient matrix of a global failure case into multiple sub-matrices, each corresponding to a local failure case, such that the overall matrix is invertible if and only if all sub-matrices are invertible (Appendix A.6). □

Theorem 4.2. *Every local failure is a subset of some m-chunk intra-group failure. If this m-chunk failure case is decodable, the local failure case is also decodable.*

Proof. Every local failure involves fewer than m lost chunks, and by adding chunk(s) from the same group we obtain the corresponding m-chunk intra-group failure. If that m-chunk failure is decodable, then any subset of its lost chunks—including the current local failure—is also decodable. □

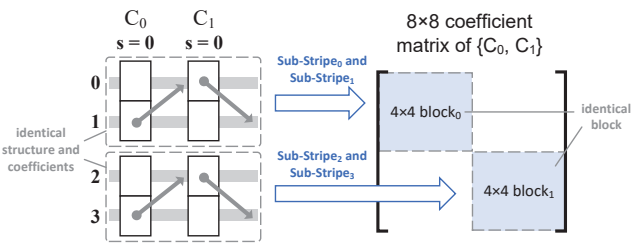


Figure 8: Coefficient matrices for chunks within a group exhibit a matrix-block-repetition pattern.

Consequently, WiseCode only needs to verify all m-chunk intra-group failure cases, totaling $g \times \binom{n/g}{m}$ cases—a small fraction of all $\binom{n}{m}$ possible failure cases. For example, in a $(106, 100, 216)$ WiseCode where $g = 3$, only 5.2 million failure cases ($\sim$0.3% of all failure cases) require verification.

Matrix size reduction. WiseCode also reduces the size of matrices to be verified according to a useful structural property under chunk partitioning: the coefficient matrix associated with chunks within a single group exhibits a *matrix-block-repetition pattern*. Specifically, it contains α/m identical diagonal matrix blocks, each corresponding to a set of m sub-stripes, with all sub-stripe sets sharing identical coding structures and coefficient assignments (Appendix A.5). As shown in Figure 8, the coefficient matrix of $\{\text{Chunk}_0, \text{Chunk}_1\}$ in the $s = 0$ group consists of two identical matrix blocks, one for Sub-Stripe$_0$ and Sub-Stripe$_1$, and another for the remaining sub-stripes. Therefore, WiseCode verifies invertibility by checking only a single matrix block rather than the entire matrix, reducing the computation cost to m/α of the original.

Overall, WiseCode significantly accelerates verification by reducing both the number of failure cases and the size of matrices to be verified. For example, in a $(106, 100, 216)$ stripe, WiseCode requires verifying only 5.2 million matrices of size 36×36, which can be completed within 3 minutes, achieving a speed-up of over $20,000\times$ compared to 1130 hours required by RS-ET code.

4.2 Neighborhood-Prioritized Retry

If a candidate coefficient combination fails verification, the search continues exploring new candidates until a valid one is found. To reduce retry overhead and accelerate convergence to valid solutions, WiseCode employs a *neighborhood-prioritized retry* strategy. Instead of resampling all coefficients, it updates only a small subset of coefficients while retaining the verified ones, generating a *neighborhood* of the rejected candidate. This localized adjustment preserves the decodability of most previously-verified failure cases, thereby avoiding redundant verification work.

To track which coefficients are retained, WiseCode treats each chunk's coefficients as a chunk-level *entry*, managed

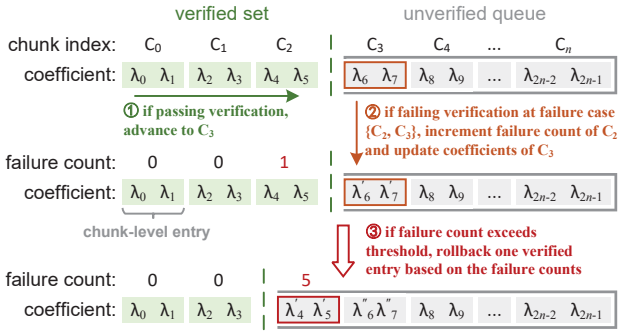


Figure 9: Three actions in each neighborhood-prioritized retry step: (1) advance, (2) update, and (3) rollback.

through a *verified set* and an *unverified queue*. A chunk is considered (un)verified if its coefficients are (un)verified. The unverified queue, initially containing all entries ordered by chunk index, is manipulated through *advance* and *rollback* operations described later. In each step, WiseCode updates only the coefficients of the entry at the front of the unverified queue, marking its corresponding chunk as the current one.

To determine which failure cases require verification, WiseCode groups chunks using the chunk-partitioning rule (§4.1) and identifies the group containing the current chunk. It then verifies only failure cases that involve the current chunk and any $m-1$ verified chunks within the same group. Failure cases involving only m verified chunks require no repeated verification since their coefficients remain unchanged. WiseCode also maintains a *failure count* for each verified entry, reflecting its contribution to failed verifications and guiding rollback decisions.

As shown in Figure 9, each search step operates on entries parameterized by λ (§4.1) and performs one of three actions:

1. **Advance**. If all required failure cases pass verification, WiseCode moves the current entry to the verified set and proceeds to the next unverified entry (or terminates if none remain);

2. **Update**. If verification fails, WiseCode increments the failure counts of involved verified entries and randomly re-samples the λ values of the current entry for retry;

3. **Rollback**. If any failure count exceeds a threshold, WiseCode randomly selects a verified entry (weighted by its failure count), updates its coefficients, moves it to the front of the unverified queue, and resets all failure counts.

Search overhead. WiseCode reduces both per-candidate verification cost and retry count, since failure cases composed entirely of verified chunks require no verification and never trigger retry. We conduct repeated experiments to compare the search overhead of the neighborhood-prioritized strategy with a naive baseline, *exhaustive enumeration*, which re-samples λ values for all entries and retry the verification of all failure cases whenever a candidate is rejected.

Table 1: Comparison of exhaustive enumeration (Baseline) and neighborhood-prioritized retry (Ours) in pre-candidate verification time, retry count, and total search time.

Stripe configuration	Verification time		# Retry		Total search time	
	Baseline	Ours	Baseline	Ours	Baseline	Ours
(104, 100, 16)	36.0 ms	2.49 ms	279	117	5.93 s	3.78 s
(104, 100, 64)	22.3 ms	0.78 ms	42	34	1.10 s	1.14 s
(105, 100, 25)	861 ms	66.0 ms	> 1.6e6	2047	> 24 h	131 s
(105, 100, 125)	819 ms	16.5 ms	> 1.6e6	312	> 24 h	21.0 s
(106, 100, 216)	2113 ms	163 ms	> 6.3e5	2403	> 24 h	341 s

Table 2: Maximum stripe width achievable from coefficient search over GF(2^{16}), for given m and α.

Code configuration		m=4	m=5	m=6	m=7	m=8
WiseCode	$\alpha = m$	232	80	48	35	27
	$\alpha = m^2$	464	160	96	70	54
	$\alpha = m^3$	696	240	144	105	81
HashTag+	$\alpha = m^3$	87	39	25	19	17
RS-ET	$\alpha = m$	72	46	30	28	24
	$\alpha = m^2$	65	40	34	25	26
	$\alpha = m^3$	68	45	31	20	15

Both strategies use 20 threads that verify different candidates in parallel. The neighborhood-prioritized retry synchronizes the candidate across threads whenever any thread advances or the failure count (aggregated across threads) of any verified entry exceeds the rollback threshold, which is set to 1000. All coefficients and λ parameters are defined over the finite field GF(2^{16})[1]. We measure the per-candidate verification time, retry count, and the total search time, averaged over 10 independent runs (Table 1). For $(104, 100, 16)$ and $(104, 100, 64)$ stripes, our method accelerates per-candidate verification by up to 28.6× and reduces retry count by up to 58%, while the total search time remains comparable due to additional synchronization overhead. For stripes with larger m—$(105, 100, 25)$, $(105, 100, 125)$, and $(106, 100, 216)$—our method finishes the search within 6 minutes, whereas the baseline fails to find valid solutions within 24 hours due to excessive retries.

Scalability in stripe width. We evaluate WiseCode's scalability by targeting a large stripe width of 1000 and measuring the maximum number of verified chunks—denoted as n^*—achieved with 20 threads within a 24-hour search period. At this point, the first n^* chunks have been verified, enabling any stripe of width n' ($n' \leq n^*$) to be constructed by the first n' chunks. For comparison, HashTag+ and RS-ET codes adopts a similar neighborhood-prioritized retry strategy, but without divide-and-conquer verification.

As shown in Table 2, WiseCode enables stripe widths up to several hundred with $4 \leq m \leq 8$. The achievable stripe width increases with α, because a higher α enlarges both n_{msr} and the group count $g = \lceil n_{msr}/m \rceil$, reducing intra-group failures and thereby accelerating the search. In contrast, HashTag+ and RS-ET are restricted to stripe widths of only a few tens, since the absence of divide-and-conquer verification forces them to check too many failure cases, significantly impeding the search process.

Existence of valid coefficients. Prior studies [35, 36] address the finite field size required to guarantee the existence of valid coefficients. For example, to ensure a valid coefficient assignment, a $(70, 66, 4)$ RS-ET code requires a field size $\geq 2(\binom{n-1}{k-1} - \binom{\lceil n/\alpha \rceil - 1}{\lceil k/\alpha \rceil - 1})$, which equals 1.7e6. However, as shown in Table 2, $(70, 66, 4)$ RS-ET code is successfully supported by GF(2^{16}) with a field size of 65536, suggesting that existing theoretical bounds are overly conservative. While determining the maximum stripe width supported by a given finite field remains an open problem, the results in Table 2 strongly suggest that the theoretical ceiling for WiseCode's scalability is substantially higher; we provide further empirical analysis in the Appendix A.2 to validate this inference.

5 Two-Stage Encoding Framework

This section presents the two-stage encoding framework that requires much fewer multiplications than standard generator matrix encoding (§5.1) and explains how it extracts common operations during *data aggregation* (§5.2) and decomposes *parity solving* into computation-efficient steps, followed by extending these optimizations to the decoding process (§5.3).

5.1 Design Overview

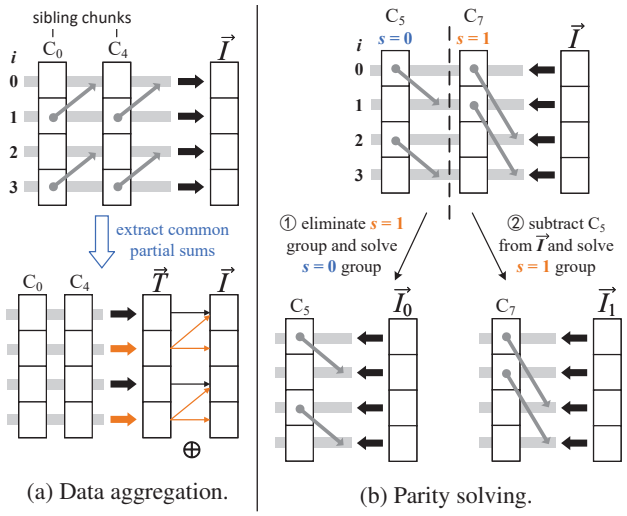
Once the parity chunks are designated, the coding equations can be expressed in matrix form as $C_D \times D + C_P \times P = \vec{0}$, where D and P denote the $k\alpha$ data sub-chunks and $m\alpha$ parity sub-chunks, and C_D and C_P are the corresponding $m\alpha \times k\alpha$ and $m\alpha \times m\alpha$ coefficient matrices, with C_P guaranteed invertible (§4). The standard encoding approach [45, 50] constructs the generator matrix $G = -{C_P}^{-1} \times C_D$ and computes $P = G \times D$.

However, the standard encoding approach is costly for WiseCode. In a $(106, 100, 216)$ WiseCode stripe, encoding with G requires 17–126 multiplications per data sub-chunk (depending on which chunks are designated as parity), while each coefficient vector in the coding equations contains only $m = 6$ elements. We make the following observation.

Observation 5.1. *The number of multiplications incurred by the nonzero entries in G far exceeds the number of coefficients in the coding equations.*

Motivated by this observation, WiseCode reduces multiplications—the dominant cost relative to additions—by multiplying each data sub-chunk only by its own coefficient vector and introducing a two-stage encoding framework:

[1]This choice balances coefficient design requirements and arithmetic complexity: GF(2^8) limits the stripe width to $256/m$ because coefficient design requires mn unique parameters (§4.1) but GF(2^8) provides only 256; GF(2^{32}) decreases multiplication throughput by 51.6% compared to GF(2^{16}).



(a) Data aggregation. (b) Parity solving.

Figure 10: Encoding scheduling: (a) extracting common data sums to eliminate redundant aggregation operations; (b) solving parity chunks using a divide-and-conquer strategy.

1. **Data aggregation**. Within each sub-stripe, data sub-chunks are multiplied by their m-element coefficient vectors and summed position-wise, producing an m-element aggregated-result vector. Concatenating the aggregated-result vectors from all α sub-stripes yields an $m\alpha$-element intermediate vector $\vec{I}$ that satisfies $\vec{I} = C_D \times D$.
2. **Parity solving**. After replacing $C_D \times D$ by $\vec{I}$, the parity sub-chunks P are then solved from the resulting coding equations $C_P \times P = -\vec{I}$.

This framework enables WiseCode to extract common operations across sub-stripes during data aggregation and follow a divide-and-conquer manner to process parity solving.

5.2 Extraction of Common Data Operations

During data aggregation, when a data sub-chunk contributes to multiple sub-stripes, it is multiplied by the same coefficient vector across those sub-stripes, as dictated by the coefficient selection (§4.1), yielding identical multiplication-result vectors. Moreover, sibling chunks follow the same coding structure, so sub-chunks with the same index across sibling chunks map to the same set of sub-stripes. Leveraging these properties, WiseCode extracts common partial sums from the multiplication-result vectors of sibling chunks, enabling their reuse across sub-stripes.

We illustrate the process of data aggregation using an $(8,6,4)$ WiseCode stripe (Figure 10a), where Chunk_0 and Chunk_4 are data chunks and form a pair of sibling chunks. Sub-chunks in each row are multiplied by their respective m-element coefficient vectors and summed to obtain an m-element common partial sum $\vec{T}$ for this row. The resulting $\vec{T}$ is then accumulated into the corresponding positions of $\vec{I}$ according to the sub-chunk to sub-stripe mapping. To minimize memory traffic, each element of $\vec{T}$ is added to $\vec{I}$ immediately after it is computed, avoiding temporary storage of $\vec{T}$ in memory. The remaining chunks are processed together with their sibling chunks (if any) in the same manner to complete the computation of $\vec{I}$.

Cost analysis. By extracting common partial sums, each of the $k\alpha$ data sub-chunks is multiplied by its m-element coefficient vector only once, leading to a total of $km\alpha$ multiplications. From the coding rules (§2.3), we observe that a fraction $(m-1)/m$ of all sub-chunks participate in two sub-stripes instead of one. Consequently, this method reduces the number of multiplications during data aggregation by $(m-1)/(2m-1)$.

5.3 Divide-and-Conquer Parity Solving

During parity solving, directly computing $P = -C_P^{-1} \times \vec{I}$ is prohibitively expensive because C_P^{-1} can be up to $10\times$ denser than C_P. Instead, WiseCode uses a divide-and-conquer strategy that operates on sub-matrices with sparse inverses. Parity chunks are partitioned into groups using the same chunk-partitioning rule as §4.1, which correspondingly partitions C_P into sub-matrices. As discussed in Section 4.1, each sub-matrix exhibits a matrix-block-repetition structure with nonzero entries confined to the matrix blocks, and its inverse preserves this sparsity[2]. WiseCode therefore performs matrix elimination and inversion at the sub-matrix granularity, using each sparse inverse to independently and efficiently solve the corresponding parity group.

Figure 10b illustrates this process for an $(8,6,4)$ WiseCode stripe. Chunk_5 and Chunk_7 are parity chunks and belong to groups with $s=0$ and $s=1$, respectively. To solve the $s=0$ group, WiseCode first eliminates the $s=1$ group via linear transformations (Appendix A.6), leaving only the $s=0$ sub-matrix and a transformed vector $\vec{I}_0$ (from $\vec{I}$). It then inverts this sub-matrix and multiplies it with $\vec{I}_0$ to obtain the $s=0$ group $\{\text{Chunk}_5\}$. After subtracting the contribution of Chunk_5 from $\vec{I}$ to produce $\vec{I}_1$, only the $s=1$ group remains unknowns. WiseCode applies the same procedure to solve the $s=1$ group $\{\text{Chunk}_7\}$.

In general, when parity chunks span multiple groups, WiseCode recursively eliminates groups until one remains, solves that group, and continues solving the remaining groups sequentially as the recursion unwinds.

Cost analysis. The divide-and-conquer parity solving incurs $O(m^3\alpha)$ multiplications (Appendix A.7), compared to $O(m^2\alpha^2)$ for directly computing $P = -C_P^{-1} \times \vec{I}$. Overall, the two-stage encoding process performs $km\alpha + O(m^3\alpha)$ multiplications, whereas the standard approach that directly computes $P = G \times D$ requires $O(km\alpha^2)$. WiseCode processes sub-chunks at a default 4 KB granularity (e.g., an 8 KB sub-chunk is processed as two 4 KB units), and therefore requires

[2]Since each matrix block appears independently and identically, we can invert them block-wise while preserving the repetition structure.

an additional $m\alpha\times$4KB of memory (within 5 MB in our experiments) to store the $m\alpha$-element intermediate vector $\vec{I}$ and other temporary results.

Extending to Decoding. During decoding, healthy chunks are treated as data chunks and aggregated using the same data-aggregation stage. Lost chunks are then partitioned into groups and recovered using the same divide-and-conquer parity-solving strategy.

6 Implementation

To demonstrate WiseCode's portability, we integrate it into Ceph [8, 51], a widely used open-source distributed storage system (§6.1). Because Ceph's native workflow is not adaptable to advanced repair scheduling, we also combine WiseCode with several representative scheduling methods to illustrate its compatibility with prior techniques (§6.2).

6.1 Ceph Erasure-Code Plugin

Although WiseCode introduces several coding-level designs, deploying it in an existing storage system requires only two essential operations: (1) identifying helper sub-chunks during repair, and (2) performing encoding or decoding using its dedicated algorithms. The coefficients obtained (§4) are stored in the system configuration, avoiding any runtime search.

To simplify integration with existing storage systems, we package WiseCode as an erasure-code plugin with two core functions: one identifies the required helper sub-chunks for given lost chunks, and the other performs encoding or decoding over available chunks. The finite field arithmetic for encoding and decoding is implemented with AVX-512 instructions [52], following GF-Complete [50]. We integrate this plugin into Ceph v17.2.5 with only ∼400 lines of C++ glue code, enabling Ceph's OSDs (Object Storage Daemons) to invoke the wrapped functions.

6.2 Integration with Repair Scheduling

Degraded read. We accelerate degraded reads using partial-parallel repair [19] and repair pipelining [20], two *intra-stripe scheduling* methods that select relay nodes to perform partial repairs[3] in parallel. With WiseCode, a node storing a non-sibling chunk of the lost chunk cannot serve as a relay because it would need to transmit a full-chunk partial result rather than a subset of sub-chunks (§3.2), increasing repair traffic. We therefore restrict relay selection to nodes storing sibling chunks, while keeping each method's original scheduling logic between the relays and the target node unchanged. Non-relay nodes simply distribute their helper data evenly across the relays and the target node.

[3]Partial repair produces a partial result using a subset of helper data (treating other helper data as zeros); summing all partial results recovers the lost chunk. WiseCode, being linear, supports this naturally.

Full-node repair. We accelerate full-node repair using RepairBoost [21], an *inter-stripe scheduling* method that reorders transmissions across stripes to improve network utilization and runs atop any intra-stripe scheduling method—partial-parallel repair, repair pipelining, or conventional repair (without relays). With WiseCode, RepairBoost only needs to account for the lower traffic contribution of non-sibling chunks than sibling chunks of the lost chunk, estimating network utilization using weighted rather than uniform traffic.

All methods are implemented in a ∼6.9K-line C++ prototype comprising a coordinator that issues repair tasks and multiple agents that execute repair operations, with gRPC [53] for inter-node communication.

7 Evaluation

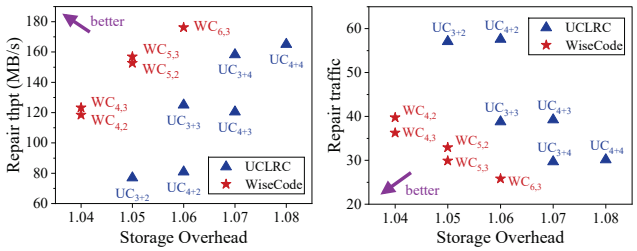
7.1 Methodology

We evaluate WiseCode against UCLRC (Uniform Cauchy LRC [6]), the state-of-the-art wide codes. We first use Ceph as the development platform, implementing both codes as erasure-code plugins for fairness, and examine offline and online repairs triggered by manually disabling one storage node. We measure repair throughput (repaired-data size divided by repair time), foreground I/O latency during online repair, and normal-write and degraded-read latencies without background repair. We then evaluate both codes with representative repair-scheduling techniques: RepairBoost [21] for full-node repair, and three strategies for single-chunk degraded reads—conventional repair (CR), partial-parallel repair (PPR) [19], and repair pipelining (RP) [20].

Code configurations. We configure UCLRC with (n,k,r,p), where r and p denote the numbers of global and local parities, to control storage overhead, repair traffic, and reliability (measured via MTTDL; see Appendix A.1). Both WiseCode and UCLRC use $k=100$, with remaining parameters varied to explore design trade-offs. Table 3 summarizes all configurations. UC_{r+p} denotes a $(100+r+p,100,r,p)$ UCLRC, and $\text{WC}_{m,b}$ denotes a $(100+m,100,m^b)$ WiseCode.

Testbed. Experiments run on a cluster of 161 ecs.hfg6.large instances [54] on Alibaba Cloud. Each instance has 2 vCPUs and 8 GB RAM. Among them, 120 serve as storage nodes, each with a 100 GB elastic SSD (∼170 MB/s bandwidth), one is the monitor, and 40 act as clients. All instances connect via a 1 Gbps network.

System and workload settings. On Ceph, we use 512 placement groups [55] for balanced chunk distribution. Storage is prefilled using YCSB [56] with 4000 records (1.46 TB). During online repair, each client issues YCSB requests with read:write ratios of 50:50 (YCSB-a), 100:0 (YCSB-c), and 0:100 (YCSB-w), as summarized in Table 4. We emulate light and heavy loads using 10 and 40 clients. For evaluations with repair scheduling, 1000 stripes are randomly placed across the cluster before repair. RP is configured with a 32 KB slice



(a) Offline repair throughput. (b) Measured repair traffic.

Figure 11: Repair throughput and measured repair traffic during offline repair on Ceph.

size following prior work [20].

Chunk-size settings are listed in Table 5. In Ceph experiments, UCLRC uses a 4 MB chunk size; we avoid larger chunks (e.g., 8 MB) because Ceph performs client I/O at full-stripe granularity, which becomes very large under wide stripes and can trigger Out-Of-Memory failures on Ceph's OSDs in our testbed, particularly when multiple requests target the same OSD concurrently. For repair-scheduling experiments, UCLRC follows prior work [20, 21] and uses 64 MB chunks. WiseCode derives its chunk size from the corresponding UCLRC configuration, rounding up to the nearest multiple of $\alpha \cdot 4$ KB to satisfy 4 KB sub-chunk alignment, and this adjustment does not bring any unfair performance advantage.

7.2 Overall Performance with Ceph

Offline repair. Figures 11a and 11b show offline repair throughput and repair traffic (total network traffic divided by size of repaired chunks) versus storage overhead, averaged over 5 independent runs. The results show that WiseCode achieves a strictly better throughput-overhead tradeoff than UCLRC. With storage overhead 1.06, $WC_{6,3}$ achieves 2.18× and 1.41× the repair throughput of UC_{4+2} and UC_{3+3}, respectively. With storage overhead 1.05, $WC_{5,3}$ outperforms UC_{3+2} by 2.04× in the repair throughput. These throughput gains stem from WiseCode's reduced repair traffic, which enables faster recovery under the same network bandwidth.

Multi-dimensional Pareto Improvement. Figure 12 illustrates the joint improvements in storage overhead, reliability, and repair throughput. Unlike MTTDL analysis in Table 3, which assumes full network bandwidth utilization during repair, the repair rate used in the reliability model for this figure is derived from the observed repair throughput and node capacity in our testbed. Results show that, with a consistent overhead of 1.05, $WC_{5,2}$ offers two orders of magnitude higher MTTDL and 1.98× higher repair throughput compared to UC_{3+2}. $WC_{6,3}$ outperforms UC_{4+4} by offering 2% lower overhead, 1.81× higher MTTDL, 14.4% lower repair traffic, and 6.8% higher repair throughput against UC_{4+4}. We further conduct an iso-MTTDL comparison by interpolating UCLRC results along log-linear curve (mapping log MTTDL to linear throughput) within the same overhead configurations. Our analysis shows that with the same MTTDL, $WC_{5,2}$ offers 1.53× higher repair throughput with 1% lower overhead, or 1.11× higher repair throughput with 2% lower overhead compared to UCLRC.

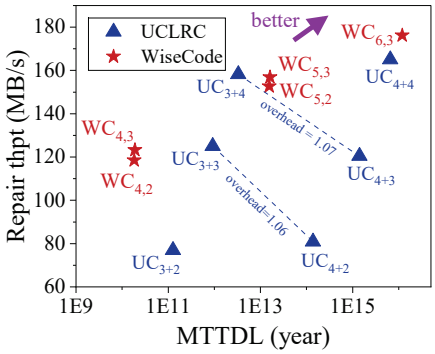


Figure 12: Joint comparison of storage overhead, MTTDL and repair throughput between WiseCode and UCLRC.

Online repair. Figure 13 plots average foreground I/O latency versus background repair throughput. At equal storage overhead, WiseCode improves both metrics: $WC_{6,3}$ delivers 36%–102% higher repair throughput and 11%–27% lower I/O latency than UC_{4+2}. Even against UC_{4+4}, which has 2% higher storage overhead than $WC_{6,3}$, $WC_{6,3}$ attains 27.7%–45.6% higher repair throughput with comparable I/O latency (within ±5%), except under YCSB-c with 40 clients, where it yields 38% higher throughput with 10% higher latency.

Normal-write and degraded-read latencies. Figure 14 shows normal-write (YCSB-w) and degraded-read (YCSB-c) latencies, each averaged over a 3-minute run. All codes exhibit similar performance (within ±5%), indicating that WiseCode introduces negligible read/write overhead, except a 6% latency increase for $WC_{5,3}$ (YCSB-c, 40 clients) and 7% for $WC_{6,3}$ (YCSB-c, 10 clients). These deviations stem from WiseCode's chunk-size rounding (Table 5), which causes Ceph to pad zeros and inflate I/O. In practice, this effect can be eliminated via precise chunking techniques [9, 27].

7.3 Performance with Repair Scheduling

Full-node repair. Figure 15 shows full-node repair throughput versus storage overhead, averaged over 5 independent runs. We present results using CR as the intra-stripe scheduling strategy for RepairBoost, as it consistently outperforms PPR and RP for both WiseCode and UCLRC in our experiments. The results show that WiseCode achieves a strictly better throughput-storage tradeoff than UCLRC, exhibiting the same performance advantages observed in the Ceph experiments. For example, with storage overhead 1.06, $WC_{6,3}$ achieves 2.38× and 1.56× the repair throughput of UC_{4+2} and UC_{3+3}, respectively. Even against UC_{4+4}, $WC_{6,3}$ achieves 13.6%

Table 3: Evaluated codes with their storage overhead, theoretical repair traffic for a single-chunk failure, and MTTDL. UC_{r+p} denotes a $(100+r+p, 100, r, p)$ UCLRC, and $WC_{m,b}$ denotes a $(100+m, 100, m^b)$ WiseCode.

Code	UC_{3+2}	UC_{3+3}	UC_{3+4}	UC_{4+2}	UC_{4+3}	UC_{4+4}	$WC_{4,2}$	$WC_{4,3}$	$WC_{5,2}$	$WC_{5,3}$	$WC_{6,3}$
Storage overhead	1.05	1.06	1.07	1.06	1.07	1.08	1.04	1.04	1.05	1.05	1.06
Repair traffic	51.50	34.33	25.75	52.00	34.67	26.00	34.75	31.52	28.42	25.60	21.59
MTTDL (years)	2.51×10^{10}	1.69×10^{11}	6.38×10^{11}	2.59×10^{13}	2.64×10^{14}	1.20×10^{15}	3.47×10^{9}	3.82×10^{9}	2.83×10^{12}	3.14×10^{12}	2.46×10^{15}

Table 4: Workloads used in the Ceph experiments.

Workload	R/W ratio	# Records	# KV pairs	Value size	Total
YCSB-a	50%/50%				
YCSB-c	100%/0%	4000	400	10^6 B	1.46 TB
YCSB-w	0%/100%				

Table 5: Chunk sizes used in the two sets of experiments.

Chunk size (KB)	UCLRCs	$WC_{4,2}$	$WC_{4,3}$	$WC_{5,2}$	$WC_{5,3}$	$WC_{6,3}$
Ceph	4096	4096	4096	4100	4500	4320
Repair-scheduling	65536	65536	65536	66000	66000	65664

higher repair throughput, despite 2% lower storage overhead.

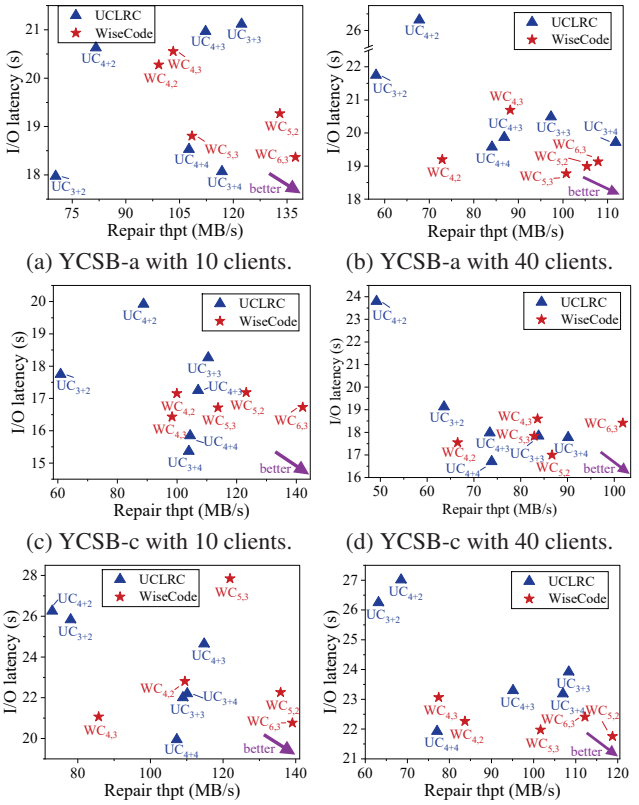
Single-chunk degraded read. Figure 17 shows single-chunk degraded-read latency, averaged over 10 independent runs. The results show that WiseCode's improvements vary across the three methods. For example, at equal storage overhead, WiseCode reduces degraded-read latency by 32%–55% compared to UCLRC under CR, due to its 37%–58% lower repair traffic. Under PPR, WiseCode and UCLRC achieve comparable latencies. In contrast, under RP, WiseCode incurs 117%–200% higher latency than UCLRC because the relay-selection constraints limit scheduling flexibility (§6.2).

The potential increase in pipelining latency represents an acceptable tradeoff for two primary reasons. First, this overhead only impacts foreground degraded reads, while WiseCode significantly enhances background repair throughput, which shortens the degraded window and directly improves MTTDL. Second, the performance impact is minimal given the rarity of failure states. Modeling with either Binomial distribution or Markov chain indicates that a stripe resides in degraded states for less than 1% of its total lifetime. This translates to a negligible impact on aggregate read performance, especially for warm and cold storage where accesses are infrequent and latency-insensitive [57].

7.4 Ablation Study of Two-Stage Coding

We measure WiseCode's computational overhead by encoding and decoding a stripe from k in-memory chunks using a single thread. Coding throughput is defined as the stripe size divided by the coding time, averaged over 10 independent runs, with chunk sizes matching the Ceph experiments.

We evaluate four variants to isolate each optimization: (1) **baseline**: generator-matrix encoding; (2) **+framework**: two-stage encoding framework only; (3) **+stage-1 opt**: framework plus data-aggregation optimization; (4) **+full opts**: all optimizations enabled (parity-solving optimization added).



(a) YCSB-a with 10 clients. (b) YCSB-a with 40 clients.

(c) YCSB-c with 10 clients. (d) YCSB-c with 40 clients.

(e) YCSB-w with 10 clients. (f) YCSB-w with 40 clients.

Figure 13: Foreground I/O latency and background repair throughput during online repair on Ceph.

As shown in Figure 16, the two-stage framework and the data-aggregation optimization increase throughput by 2.6×–3.0× and 2.0×–5.0×, respectively. The parity-solving optimization provides an additional 1.5×–1.8× speedups for $WC_{5,3}$ and $WC_{6,3}$. Overall, the two-stage encoding approach achieves 5.5×–22.4× throughput over the baseline.

7.5 Applicability to Diverse Hardware

Applicability to HDDs. While our cloud testbed uses SSDs, as in production wide-stripe deployments such as VastData [58], we also evaluate WiseCode's applicability to HDDs, which remain prevalent for their lower cost per byte [59]. We examine HDD read bandwidth during repair, where read operations dominate (95%–97%) but may be slowed by data fragmentation incurred by sub-packetization.

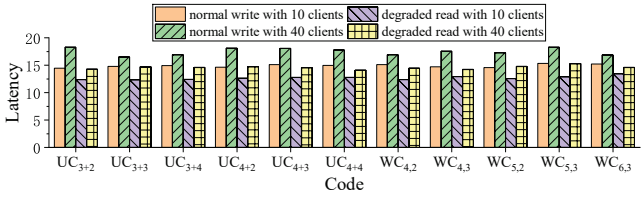


Figure 14: Normal-write and degraded-read latencies on Ceph.

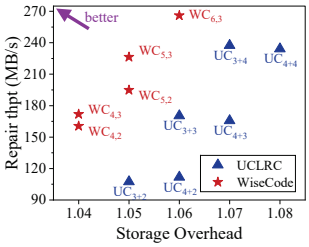


Figure 15: Full-node repair throughput with RepairBoost.

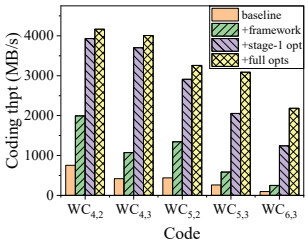


Figure 16: Coding throughput.

For example, during the repair process of $WC_{4,2}$, each helper chunk exhibits one of three read patterns with different percentages: repeatedly reading 1 sub-chunk and skipping 3 (44.2%), reading 4 consecutive sub-chunks (44.2%), or reading all 16 sub-chunks (11.6%).

We emulate these read patterns on a 7200 RPM HDD using *fio* [60] (iodepth = 8), measure per-pattern read bandwidth, and compute the percentage-weighted average, as shown in Figure 18. The results show that fragmentation reduces read bandwidth, but WiseCode still match raw network bandwidth with practical chunk sizes (e.g., 12 MB or 64 MB), ensuring that HDDs are not the bottleneck in the read-transfer pipeline [21]. Under the 1 Gbps (∼125 MB/s) network in our testbed, $WC_{4,2}$ requires 12 MB or larger chunks, and $WC_{6,3}$—despite its higher sub-packetization level—requires 64 MB or larger, both easily satisfied by the 64 MB–1 GB chunk sizes used in production [9, 11].

Applicability to faster network. Production nodes typically pair high-bandwidth network (e.g. 10+ Gbps) with high-density disk arrays (e.g., 36–216 HDDs [61]) to maximize network utilization. For instance, the Amazon D3.8xlarge instance [62] pairs 25 Gbps network with 24 HDDs, while the Tencent Cloud DA4m.32XLARGE512 instance [63] pairs 100 Gbps (50 Gbps full-duplex) network with 48 HDDs. For these big-data-oriented instances, WiseCode's repair improvement remains highly applicable because the aggregate storage bandwidth typically exceeds the network capacity. Specifically, as WiseCode achieves over 130 MB/s per disk (with 64 MB chunks), a disk array with 24 HDDs can provide over 25 Gbps of aggregate I/O, which matches the 25 Gbps network and does not become a bottleneck. Furthermore, given its high coding throughput (> 2000 MB/s), WiseCode can process 10+ Gbps network traffic using only a single dedicated CPU thread without incurring computational bottleneck.

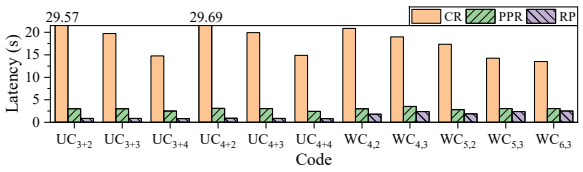


Figure 17: Single-chunk degraded-read latency with repair scheduling.

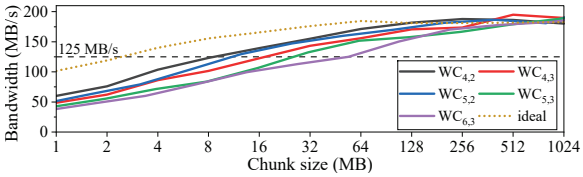


Figure 18: Average HDD read bandwidth during repair.

8 Related Work

Wide codes. Most wide-stripe solutions are based on LRCs [6, 17, 27, 29, 30], which reduce repair traffic but sacrifice the MDS property. WiseCode is the first wide-stripe vector code that achieves near-optimal repair traffic while preserving the MDS property. VAST [58] designs a wide-stripe locally decodable code, but its design details remain undisclosed.

Vector codes. Clay codes [34], Butterfly codes [32], and other MSR codes suffer from sub-packetization blowup in wide stripes. Zigzag codes [64] alleviate this only for $m = 2$, offering insufficient reliability in wide stripes. HashTag+ [37] and RS-ET [36] codes avoid sub-packetization blowup but incur higher repair traffic than WiseCode and face the scalability barrier in coefficient search. WiseCode cannot yet support the flexible sub-packetization levels of RS-ET codes due to limitations of current MSR templates; future work will explore new template designs to enable this.

Repair scheduling. Prior work [19–21, 24] optimizes repair transmission schedules to better utilize network resources and accelerate repair. WiseCode is complementary: it reduces repair traffic itself and can be combined with these scheduling techniques through appropriate adaptions, as discussed in Section 6.

9 Conclusion

By breaking three scalability barriers, we present WiseCode, the first practical and scalable wide-stripe vector-coding approach that achieves near-optimal repair traffic and ultra-low storage overhead (e.g., 1.04–1.06) in large-scale storage clusters. Experimental results show that WiseCode delivers a strictly better tradeoff between repair performance and storage overhead than Google's UCLRCs.

Acknowledgments

We sincerely thank our shepherd for helping us improve the paper. We are also grateful to all reviewers of this paper for their helpful comments and feedback. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–11, 2016.

[2] Bianca Schroeder and Garth A Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *ACM Transactions on Storage (TOS)*, 3(3):8–es, 2007.

[3] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, pages 17–29, 2007.

[4] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 137–149, 2020.

[5] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 16, 2018.

[6] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, 2023.

[7] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2025.

[8] Ceph Community. Ceph Git Repository. https://github.com/ceph/ceph, 2025.

[9] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, et al. What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 277–291, 2024.

[10] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: performance-oriented evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, 2023.

[11] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, 2014.

[12] Su Zhou, Erci Xu, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, et al. SMRSTORE: A storage engine for cloud object storage on HM-SMR drives. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 395–408, 2023.

[13] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[14] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, 2022.

[15] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, et al. Fisc: a large-scale cloud-native-oriented file system. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 231–246, 2023.

[16] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick PC Lee. ELECT: Enabling erasure coding tiering for {LSM-tree-based} storage. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 293–310, 2024.

[17] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *FAST*, pages 233–248, 2021.

[18] Meng Wang, Jiajun Mao, Rajdeep Rana, John Bent, Serkay Olmez, Anjus George, Garrett Wilson Ransom, Jun Li, and Haryadi S Gunawi. Design considerations and analysis of multi-level erasure coding in large-scale

data centers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[19] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the eleventh European conference on computer systems*, pages 1–16, 2016.

[20] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. Repair pipelining for Erasure-Coded storage. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 567–579, 2017.

[21] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick PC Lee, and Jiwu Shu. Boosting Full-Node Repair in Erasure-Coded Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 641–655, 2021.

[22] Korlakai Vinayak Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[23] Zhufan Wang, Guangyan Zhang, Yang Wang, Qinglin Yang, and Jiaji Zhu. Dayu: Fast and low-interference data recovery in very-large storage systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 993–1008, 2019.

[24] Yuhui Cai, Shiyao Lin, Zhirong Shen, Jiahui Yang, and Jiwu Shu. Chameleonec: Exploiting tunability of erasure coding for low-interference repair. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–28. IEEE, 2025.

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

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

[27] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[28] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.

[29] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014.

[30] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. *ACM Transactions on Storage (TOS)*, 16(2):1–32, 2020.

[31] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 81–94, 2015.

[32] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *14th USENIX conference on file and storage technologies (FAST 16)*, pages 81–94, 2016.

[33] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, volume 21, 2012.

[34] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 139–154, 2018.

[35] Katina Kralevska, Danilo Gligoroski, Rune E Jensen, and Harald Øverby. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data (TBD)*, 4(4):516–529, 2017.

[36] Kaicheng Tang, Keyun Cheng, Helen H. W. Chan, Xiaolu Li, Patrick P. C. Lee, Yuchong Hu, Jie Li, and Ting-Yi Wu. Balancing repair bandwidth and subpacketization in erasure-coded storage via elastic transformation. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[37] Katina Kralevska and Danilo Gligoroski. An explicit construction of systematic mds codes with small subpacketization for all-node repair. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing,*

16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pages 1080–1084. IEEE, 2018.

[38] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, and Dan Feng. Revisiting network coding for warm blob storage. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 139–154, 2025.

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

[40] R. Singleton. Maximum distanceq-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, 1964.

[41] John E Angus. On computing mtbf for a k-out-of-n: G repairable system. *IEEE Transactions on Reliability*, 37(3):312–313, 2002.

[42] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. Tiger:{Disk-Adaptive} redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, 2022.

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

[44] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[45] Intel, Inc. Intelligent Storage Acceleration Library. https://github.com/intel/isa-l, 2025.

[46] Min Ye and Alexander Barg. Explicit constructions of optimal-access mds codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory*, 63(10):6307–6317, 2017.

[47] Jonggyu Park and Young Ik Eom. FragPicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, pages 280–294, 2021.

[48] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[49] Guodong Li, Ningning Wang, Sihuang Hu, and Min Ye. MSR codes with linear field size and smallest sub-packetization for any number of helper nodes. *IEEE Transactions on Information Theory*, 2024.

[50] James S Plank, Ethan L Miller, Kevin M Greenan, Benjamin A Arnold, John A Burnum, Adam W Disney, and Allen C McBride. GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic Version 1.02, 2014.

[51] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[52] Intel, Inc. AVX-512 Overview. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. 2025.

[53] gRPC Authors. gRPC, A high performance, open source universal RPC framework. https://grpc.io/, 2025.

[54] Alibaba Cloud Company. User guide of Instance families with high clock speeds (hf series). https://www.alibabacloud.com/help/en/ecs/user-guide/instance-families-with-high-clock-speeds, 2025.

[55] Ceph authors and contributors. Placement Groups in Ceph. https://docs.ceph.com/en/quincy/rados/operations/placement-groups/, 2025.

[56] YCSB Community. Yahoo! Cloud Serving Benchmark (YCSB) Git Repository. https://github.com/brianfrankcooper/YCSB, 2025.

[57] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R Ganger, and KV Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 330–346, 2024.

[58] VastData, Inc. Breaking Resiliency Trade-offs With Locally Decodable Erasure Codes. https://vastdata.com/blog/breaking-resiliency-trade-offs-with-locally-decodable-erasure-codes, 2019.

[59] Sanjith Athlur, Timothy Kim, Saurabh Kadekodi, Francisco Maturana, Xavier Ramos, Arif Merchant, KV Rashmi, and Gregory R Ganger. Okapi: Decoupling data striping and redundancy grouping in cluster file systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 897–914, 2025.

[60] Jens Axboe. FIO - Flexible I/O Tester. https://github.com/axboe/fio, 2025.

[61] Chunqiang Tang. Meta's hyperscale infrastructure: Overview and insights. *Communications of the ACM*, 68(2):52–63, 2025.

[62] Amazon Web Service. Amazon EC2 D3 and D3en Instances. https://aws.amazon.com/cn/ec2/instance-types/d3/, 2026.

[63] Tencent Cloud. Big data DA4m instance specifications. https://www.tencentcloud.com/document/product/213/59879, 2026.

[64] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory*, 59(3):1597–1616, 2012.

[65] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, 2010.

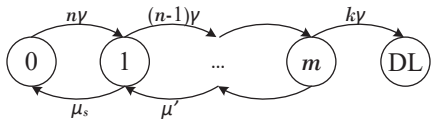
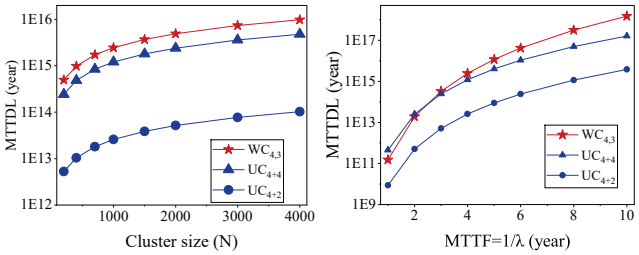


Figure 19: Markov model used for reliability analysis.



(a) Varying cluster size N. (b) Varying MTTF $1/\lambda$.

Figure 20: Sensitivity analysis on MTTDL.

A Appendix

A.1 MTTDL and Markov Model

We evaluate data reliability using the mean-time-to-data-loss (MTTDL) metric, modeled via continuous-time Markov chains following prior works [6, 27]. As shown in Figure 19, each circles corresponds to a health states of a stripe: state i indicates that i chunk(s) are lost, while the Data Loss (DL) state represents unrecoverable data loss. A stripe transitions between adjacent states at specified rates, and MTTDL is defined as the expected time for the stripe to transition from state 0 to the DL state.

The state transition rate from state i to state $i+1$ is $(n-i)\gamma$, where γ is the failure rate of each node and n is the stripe width. A stripe enters the DL state once fewer than k healthy chunks remain. For UCLRC, local parity does not fully protect the stripe: when $i \geq r$, an additional failure may directly cause data loss with probability p_i, which is empirically measured via simulation [6]. The transition rate μ_s from state 1 back to state 0 is $\mu_s = \frac{1}{T_s}$, where $T_s = \frac{CS}{\varepsilon(N-1)B}$ is the mean time to repair a single-failure stripe. Here, S is the node capacity, C is the repair traffic normalized by chunk size, CS is the total network traffic during repair, N is the cluster size, B is the per-node network bandwidth, and ε is the fraction of network bandwidth allocated to repair. The term $\varepsilon(N-1)B$ represents the total available bandwidth assuming the repair traffic is evenly distributed across nodes. Since multi-chunk failures are rare and prioritized, the repair rate μ', i.e. the transition rate from state $i(>1)$ to state $i-1$, is basically dominated by $\mu' = \frac{1}{T'}$, where T' denotes the time to detect multi-failure stripes and trigger repair [6, 27].

We configure the parameters as follows: the mean-time-to-failure (MTTF) $1/\gamma = 4$ years, $N = 1000$ nodes, $S = 16$ TB, $B = 10$ Gbps, $T' = 30$ minutes, $\varepsilon = 0.1$. The MTTDL results of evaluated codes in our experiments are shown in Table 3. Figure 20 further illustrates the sensitivity of MTTDL to variations in N and $\frac{1}{\gamma}$ for $WC_{6,3}$, UC_{4+2}, and UC_{4+4}. The results indicate that while the MTTDL of both WiseCode and UCLRC scales at a similar rate with N, WiseCode exhibits a faster growth rate as $1/\gamma$ increases. Although $WC_{4,3}$ underperforms UC_{4+4} under an extremely low MTTF ($\leq$2 years), WiseCode demonstrates superior reliability in more realistic MTTF ranges (3–5 years) [17, 28, 65].

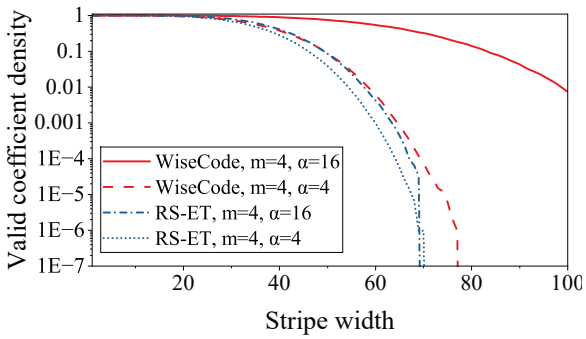


Figure 21: Density of valid coefficient combinations.

A.2 Existence of Valid Coefficients

As Table 2 summarizes the maximum stripe width achieved in our experiments, we believe that these results suggest that the theoretical ceiling for WiseCode's scalability is substantially higher. To validate this, we randomly choose up to 1,000,000 coefficient combination from GF(2^{16}) for RS-ET codes and WiseCode, calculating the density of valid assignments for each. And we observe that WiseCode has a higher probability to find valid coefficient under stripe-width configurations. As shown in Figure 21, the density of valid combinations for RS-ET codes with $m = 4$ decreases precipitously when the stripe width approaches 70. While WiseCode with $m = 4$ and $\alpha = 4$ exhibits a slightly better density curve, the density improves significantly as α increases. This improvement is driven by divide-and-conquer verification, which effectively reduces the number of coefficient constraints (§4.1) and thus render less coefficient combinations invalid.

A.3 Revisiting the Construction of WiseCode

Section 3 illustrates the template-based construction of WiseCode. Here we offer a straightforward matrix-based construction to simplify the discussions and proofs. We focus on how to directly obtain the parity-check matrix P, i.e., the entire coefficient matrix of coding equations, such that the coding equations can be collectively defined as $P \times C = \vec{0}$, where C represents all chunks.

In an (n,k,α) WiseCode built atop the (n_{msr},k_{msr}) MSR template, we have $m = n-k = n_{msr}-k_{msr}$. Define $b = n_{msr}/m$, then $\alpha = m^b$. If the $m \times 1$ coefficient vector of each sub-chunk is considered as a element, the parity-check matrix (i.e., coeffi-

cient matrix) P corresponding to all n chunks can be regarded as a $\alpha \times n\alpha$ matrix, where each of the n chunks corresponds to a $\alpha \times \alpha$ sub-parity-check-matrix (*SPCM*), i.e., Chunk_i corresponds to $SPCM_i$. Basically, each row of $SPCM_i$ corresponds to a sub-stripe, and each column corresponds to a sub-chunk. The elements of $SPCM_i$ are given as follows.

As discussed in Section 3.1, Chunk_i's parameters x and s are inherited from the template and are originally defined in Rule #2 (§2.3). The position of each element in the *SPCM* is denoted by the coordinate (r,c) starting from the top left corner, r for row and c for column. Given the base-m representation of c as $c = (c_{b-1}, c_{b-2}, \ldots, c_0)_m$, we define $c(x,s) = (c_{b-1}, \ldots, c_{s+1}, x, c_{s-1}, \ldots, c_0)_m$. We also define $\vec{\lambda} = [1, \lambda, \lambda^2, \ldots, \lambda^{m-1}]^T$. With m parameters λ_{im} to $\lambda_{(i+1)m-1}$ as specified in Section 4, the elements in the $SPCM_i$ of chunk_i are:

- $\overrightarrow{\lambda_{im+c_s}}$, if $c = r$ or $c(x,s) = r$,
- $\vec{0}$, otherwise.

We note that the conditions "$c = r$" and "$c(x,s) = r$" match the Rule #1 and Rule #2 in Section 2.3, respectively.

Lemma A.1. *Row_r of $SPCM_i$ contains 1 or m non-zero elements: If $r_s = x$, Row_r contains m non-zero elements at columns with indices $\{r(t,s) \mid 0 \le t < m\}$, with the element values specified as $\{\overrightarrow{\lambda_{im+t}} \mid 0 \le t < m\}$. If $r_s \neq x$, Row_r contains one non-zero element only at Column_r, with the element value specified as λ_{im+r_s}.*

A.4 WiseCode Repair Pattern

We prove the correctness of WiseCode repair pattern shown in Section 3.2. The key is to answer two questions: (1) For the lost chunk i, which sub-stripes are selected for decoding; (2) In a helper chunk i', which sub-chunks belong to the selected sub-stripes.

Answer for question (1). With the parameters x and s of chunk i, the α/m sub-stripes whose indices j satisfy $j_s = x$ are selected for decoding. According to Lemma A.1, each Row_j in the $SPCM_i$ contains m non-zero elements at columns with the indices $\{j(t,s) \mid 0 \le t < m\}$, therefore the Sub-Stripe_j can recover the m corresponding sub-chunks by solving the equation on Row_j. It is obvious that for any two different selected sub-stripes, their j vary at some digit j_p ($p \neq s$), therefore the two sets of $\{j(t,s) \mid 0 \le t < m\}$ have no intersection, and traversing all selected α/m sub-stripes can recover all α lost sub-chunks.

Answer for question (2). If $\text{Chunk}_{i'}$ is a sibling chunk of Chunk_i, it is obvious that it needs to contribute all α sub-chunks as helper sub-chunks, because $SPCM_{i'}$ is identical to $SPCM_i$ regarding the positions of non-zero elements, and these positions span all columns (sub-chunks).

If $\text{Chunk}_{i'}$ is not a sibling chunk of Chunk_i, we prove that in $SPCM_{i'}$, any non-zero element of Row_j only occurs within columns whose indices c satisfy $c_s = x$, i.e., having the same value range as the indices of the selected α/m rows (sub-stripes). Let the parameters of $\text{Chunk}_{i'}$ to be s' and x', then we discuss the position of non-zero elements in each Row_j based on Lemma A.1. If $j_{s'} = x'$, then the Column_c having a non-zero elements satisfies $c \in \{j(t,s') \mid 0 \le t < m\}$. In this case we have $s' \neq s$, otherwise $s' = s$ and $x' = j_{s'} = j_s = x$, which contradicts the assumption that $\text{Chunk}_{i'}$ is not a sibling chunk of Chunk_i. Therefore, any $c \in \{j(t,s') \mid 0 \le t < m\}$ satisfies $c_s = j_s = x$ because the sth digit of $j(t,s')$ is identical to j. If $j_{s'} \neq x'$, then the Column_c having a non-zero elements satisfies $c = j$, which also means $c_s = x$ because $j_s = x$. In conclusion, any non-zero element of Row_j only occurs within columns whose indices c satisfy $c_s = x$, which means that the indices c of helper sub-chunks also satisfy $c_s = x$ since each column corresponds to a sub-chunk with the same index.

A.5 Matrix-block-repetition pattern

Recall that each element of $SPCM_i$ corresponds to a coefficient vector of a sub-chunk, and there are at most m non-zero elements in each row of $SPCM_i$ (Lemman A.1). The m^b sub-chunks of chunk_i can be partitioned into m^{b-1} mutually disjoint subsets, each containing m sub-chunks, such that any sub-stripe only contains sub-chunks within a single subset. Using the sub-chunk index to describe each subset, sub-chunk_c belongs to the subset $\{c(t,s) \mid 0 \le t < m\}$, and these m sub-chunks sequentially corresponds to coefficients $\{\overrightarrow{\lambda_{im+t}} \mid 0 \le t < m\}$. The indices j of sub-stripes related to this subset of sub-chunks satisfy $j \in \{c(t,s) \mid 0 \le t < m\}$, i.e., having the same value range of the sub-chunk indices. Therefore, $SPCM_i$ of size $m^b \times m^b$ can be partitioned into m^{b-1} small matrices of size $m \times m$, each corresponding to one sub-chunk subset and its sub-stripe subset.

These small matrices have identical elements and cover all non-zero elements of $SPCM_i$. We denote each of these small matrices as elementary-parity-check-matrix (EPCM), and therefore $SPCM_i$ consists of m^{b-1} $EPCM_i$. An $EPCM_0$ in the case of $m = 4$ is as follows:

$$EPCM_0 = \begin{bmatrix} \overrightarrow{\lambda_0} & \overrightarrow{\lambda_1} & \overrightarrow{\lambda_2} & \overrightarrow{\lambda_3} \\ & \overrightarrow{\lambda_1} & & \\ & & \overrightarrow{\lambda_2} & \\ & & & \overrightarrow{\lambda_3} \end{bmatrix} \quad (1)$$

Corresponding to chunk partitioning rule in Section 4.1, concatenating SPCMs of chunks within group_s produces grouped-$SPCM_s$ ($GSPCM_s$). For each SPCM within group_s, there is an EPCM with the rows and columns indexed within the same value range $\{c(t,s) \mid 0 \le t < m\}$, and these EPCMs can be also concatenated together to produce grouped-$EPCM_s$ ($GEPCM_s$). Therefore, assuming group_s include m_s chunks, $GSPCM_s$ can be partitioned into m^{b-1} identical $GEPCM_s$, exhibiting the matrix-block-repetition pattern.

A.6 The Parity Solving and Matrix Elimination

Here are the mathematical details of Section 5.3, and we also prove Theorem 4.1 along this process. The objective of matrix elimination is to eliminate *GSPCM* of other groups, such that computing each sub-chunks within group$_s$ only needs to invert $GSPCM_s$. As $GSPCM_s$ exhibits the matrix-block-repetition pattern, the inversion of $GSPCM_s$ is further reduced to the inversion of $GEPCM_s$.

Details of matrix elimination. Define $\vec{\lambda}^{[h,t)} = [\lambda^h, \lambda^{h+1}, \ldots, \lambda^{t-1}]^T$, i.e., performing head and tail truncation on the original $\vec{\lambda}$. Then we define that substituting each element $\vec{\lambda}$ by $\vec{\lambda}^{[h,t)}$ in $GSPCM_s$ and $GEPCM_s$ produces $GSPCM_s^{[h,t)}$ and $GEPCM_s^{[h,t)}$, respectively. Now we start to separate $GSPCM_0$ from the other GSPCMs and eliminate $GSPCM_0$. Note that we may logically permute matrix rows between steps to facilitate the demonstration, which has no actual impacts on computations. Without loss of generality, we assume that the m chunks to be computed are parity chunks with indices $\{p_1, \ldots, p_m\}$, while any m other chunks follow the same procedure to perform divide-and-conquer solving. We start with the entire coefficient matrix $[P_{p_1}, \ldots, P_{p_m}]$, corresponding to parity chunks $\{C_{p_1}, \ldots, C_{p_m}\}$. The matrix elimination consists of two key operations: *row transformation* and *sub-chunk transformation*. Equation 2 demonstrates the row transformation process, with the row transformation matrix RT_0 described later:

$$
\begin{aligned}
&[P_{p_1}, \ldots, P_{p_m}] \\
&= [GSPCM_0^{[0,m)} \,|\, GSPCM_1^{[0,m)} \,|\, \cdots \,|\, GSPCM_{b-1}^{[0,m)}] \\
&\quad \text{(Group } SPCM\text{s within each group}_s \ (0 \le s \le b-1).) \\
&= \left[\begin{array}{c|c|c|c} GSPCM_0^{[0,m_0)} & GSPCM_1^{[0,m_0)} & \cdots & GSPCM_{b-1}^{[0,m_0)} \\ GSPCM_0^{[m_0,m)} & GSPCM_1^{[m_0,m)} & \cdots & GSPCM_{b-1}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Define } m_s \text{ as the number of parity chunks in group}_s.) \\
&= \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ GSPCM_0^{[m_0,m)} & GSPCM_{[1,b)}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Focus on only group}_0.) \\
&\Rightarrow \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ 0 & RT_0 \times GSPCM_{[1,b)}^{[0,m_0)} + GSPCM_{[1,b)}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Eliminate } GSPCM_0^{[m_0,m)} \text{ using } RT_0 \times GSPCM_0^{[0,m_0)}. \\
&\quad RT_0 \text{ represents row transformations to eliminate group}_0.) \\
&= \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ 0 & A \end{array}\right] \\
&\quad \text{(Denote the bottom-right part by } A.)
\end{aligned}
\tag{2}
$$

Note 1 for proof of Theorem 4.1. Now we have decomposed the coefficient matrix $[P_{p_1}, \ldots, P_{p_m}]$ into $GSPCM_0^{[0,m_0)}$ and A, such that the coefficient matrix is invertible if and only if $GSPCM_0^{[0,m_0)}$ and A are invertible. The remaining of the proof lies in proving that A is equivalent to $GSPCM_{[1,b)}^{[0,m-m_0)}$ regarding the invertibility (See Note 2). If we can establish this equivalence, then we can continuously decompose $GSPCM_s^{[0,m_s)}$ from $GSPCM_{[1,b)}^{[0,m-m_0)}$, such that $[P_{p_1}, \ldots, P_{p_m}]$ is invertible if and only if all $GSPCM_s^{[0,m)}$ are invertible, matching Theorem 4.1.

Row transformation matrix. Since $GSPCM_0$ can be partitioned into m^{b-1} identical $GEPCM_s$, RT_0 can also be partitioned into m^{b-1} identical elementary-RT_0 (ERT_0), each eliminating $GEPCM_0^{[m_0,m)}$ using $ERT_0 \times GEPCM_0^{[0,m_0)}$, i.e., $ERT_0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m)}$. According to linear algebra, $ERT_0 = GEPCM_0^{[m_0,m)} \times (GEPCM_0^{[0,m_0)})^{-1}$. Each ERT_0 is of size $(m-m_0)m \times mm_0$ and one can verify that it has at most $(mm_0 + m - m_0)m_0(m-m_0)$ non-zero coefficients.

Given the original parity-check matrix equation:

$$
\begin{aligned}
&[P_{p_1}, \ldots, P_{p_m}] \times \{\text{all parity chunks}\} = \vec{R} \\
&\text{i.e., } GSPCM_{[0,b)}^{[0,m)} \times \{\text{parity chunks in group}_{[0,b)}\} = \vec{R}
\end{aligned}
$$

RT_0 also transforms the intermediate results $\vec{R}$ (see §5.3) of size ml to $\vec{R}'$ of size $(m-m_0)l$. That is,

$$
A \times \{\text{parity chunks in group}_{[1,b)}\} = \vec{R}' \tag{3}
$$

where $A = RT_0 \times GSPCM_{[1,r)}^{[0,m_0)} + GSPCM_{[1,r)}^{[m_0,m)}$.

Sub-chunk transformations. To maintain the original coding coefficients of the remaining parity sub-chunks (i.e., to transform A back to $GSPCM_{[1,b)}^{[0,m-m_0)}$), we adopt sub-chunk transformations (ST_0) such that

$$
GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0 \times \{\text{parity chunks in group}_{[1,b)}\} = \vec{R}' \tag{4}
$$

Comparing Equation (3) and (4), we have

$$
RT_0 \times GSPCM_{[1,b)}^{[0,m_0)} + GSPCM_{[1,b)}^{[m_0,m)} = GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0 \tag{5}
$$

To determine the required ST_0, we analyze the left side of Equation 5 at the granularity of ERT_0. As each ERT_0 performs the row transformation on a $GEPCM_0$, it also transforms coefficients of parity sub-chunks within other groups. We take group$_1$ for an example. In $GSPCM_1$, we select the m rows corresponding to ERT_0 and select the m sub-chunks with the same indices as these rows. One can verify that the coefficient matrix of the m selected sub-chunks within the m selected rows is a $m \times m$ diagonal matrix, whose elements are identical coefficient vectors. The coefficient vector can be

determined as in Appendix A.3, and here we assume its value to be $\vec{\lambda}$, such that coefficient matrix, denoted by $Diag_\lambda$, can be written as follows:

$$Diag_\lambda = \begin{bmatrix} \vec{\lambda} & & & \\ & \vec{\lambda} & & \\ & & \ddots & \\ & & & \vec{\lambda} \end{bmatrix}$$

Therefore, for the m selected sub-chunks and m selected rows, Equation 5 turns into

$$ERT_0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m)} = Diag_\lambda^{[0,m-m_0)} \times EST_{0,\lambda} \tag{6}$$

where $EST_{0,\lambda}$ denotes an elementary-ST_0 corresponding to $Diag_\lambda$. Recall that $ERT_0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m)}$, we define ERT_0^t $(0 \le t < m-m_0)$ to be the sub-matrix of ERT_0 that satisfies

$$\begin{aligned} &ERT_0^t \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0+t,m_0+t+1)} \\ \Rightarrow &ERT_0^0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m_0+1)} \\ \Rightarrow &ERT_0^0 \times GEPCM_0^{[t,m_0+t)} = GEPCM_0^{[m_0+t,m_0+t+1)} \\ \Rightarrow &ERT_0^t \times GEPCM_0^{[0,m_0)} = ERT_0^0 \times GEPCM_0^{[t,m_0+t)} \end{aligned}$$

That is to say, performing ERT_0^t on $\vec{\lambda}^{[0,m_0)}$ is equivalent to performing ERT_0^0 on $\vec{\lambda}^{[t,m_0+t)}$. Therefore, we have

$$\begin{aligned} &ERT_0^t \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0+t,m_0+t+1)} \\ =&ERT_0^0 \times Diag_\lambda^{[t,m_0+t)} + Diag_\lambda^{[m_0+t,m_0+t+1)} \\ =&(ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \times Diag_\lambda^{[t,t+1)} \\ =&Diag_\lambda^{[t,t+1)} \times (ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \end{aligned}$$

traversing t from 0 to $m-m_0$, we have

$$\begin{aligned} &ERT_0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m)} \\ =&Diag_\lambda^{[0,m-m_0)} \times (ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \end{aligned} \tag{7}$$

comparing Equation 6 and Equation 7 we have $EST_{0,\lambda} = ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}$. And sub-chunk transformation ST_0 is determined by traversing all ERT_0 and getting all $EST_{0,\lambda}$. Now we successfully eliminate group$_0$ and maintain the original coefficient matrix (with necessary truncation) of sub-chunks in other groups. Each $EST_{0,\lambda}$ is of size $m \times m$, and one can verify that $(EST_{0,\lambda})^{-1}$ has at most $mm_0 + m - m_0$ non-zero elements.

Note 2 for proof of Theorem 4.1. Now we have given the form of $EST_{0,\lambda}$. Recall that $A = GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0$, then A is equavilent to $GSPCM_{[1,b)}^{[0,m-m_0)}$ regarding invertibility as long as each $EST_{0,\lambda}$ is invertible. One can refer to [49] for the proof of a similar problem in the MSR-code version.

A.7 The Computational Overhead of Parity Solving and Matrix Elimination

We count the number of multiplications imposed by eliminating group$_s$. Denote the number of remaining chunks after eliminating group$_s$ as r_s. Firstly, each of m^{b-1} ERT_s imposes at most $(mm_s + m - m_s)m_s r_s$ multiplications on $\vec{R}$, totaling $(mm_s + m - m_s)m_s r_s m^{b-1}$. Secondly, eliminating group$_s$ introduces sub-chunk transformations ESTs that need to be inverted, imposing at most $mm_s + m - m_s$ multiplications for each of $r_s m^{b-1}$ ESTs (m^{b-1} ESTs for each of r_0 remaining chunks), totaling $(mm_s + m - m_s)r_s m^{b-1}$. Thirdly, when the remaining $r_s m^b$ chunks are solved, substituting them into the equation of group$_s$ takes m_s multiplications for each sub-chunk, totaling $m_s r_s m^b$. Finally, we need to solve the following equation for group$_s$:

$$GSPCM_s^{[0,m_s)} \times \{\text{parity chunks in group}_s\} = \vec{U}$$

where $\vec{U}$ is the remaining intermediate results after substituting solved sub-chunks. One can verify that each $(GEPCM_s^{[0,m_s)})^{-1}$ has at most $(mm_s + m - m_s)(m_s)^2$ non-zero coefficients, totaling $(mm_s + m - m_s)(m_s)^2 m^{b-1}$ for $(GSPCM_s^{[0,m_s)})^{-1}$, and the number of multiplications for solving this equation does not exceed this number. Note that $\sum m_s = m$, $r_s + m_s \le m$, and $mm_s + m - m_s \le m^2$. It can be easily verified that the total multiplication number is $O(m^{b+3}) = O(m^3\alpha)$.