Repair I/O Optimization for Clay Codes via Gray-Code Based Sub-Chunk Reorganization in Ceph

Baijian Ma
Huazhong University of Science and Technology
Wuhan, Hubei, China
mabaijian@hust.edu.cn

Yuchong Hu
Huazhong University of Science and Technology
Wuhan, Hubei, China
Shenzhen Research Institute of
Huazhong University of Science and Technology
Shenzhen, Guangdong, China
yuchonghu@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
Wuhan, Hubei, China
dfeng@hust.edu.cn

Ray Wu
Inspur
Jinan, Shandong, China
wuruizhen@inspur.com

Kevin Zhang
Inspur
Jinan, Shandong, China
zhangkai_bj@inspur.com

***Abstract*—Erasure coding provides high fault tolerance with low storage redundancy, albeit at the expense of excessive repair bandwidth. Minimum-Storage Regenerating (MSR) codes employ sub-chunk partitioning to minimize repair bandwidth with the lowest storage redundancy. Clay codes stand as state-of-the-art *access-optimal* MSR code (i.e., the amount of data read from disk equals that of network transmission). However, it has an issue of significant non-sequential I/Os during repair, making disk read a bottleneck for repair performance. Some studies explore the adoption of a larger chunk size to alleviate I/O overhead, yet the effect is limited.**

We find that adjusting the order of sub-chunks on the disk contributes to enhancing I/O continuity during repair, and we theoretically prove the existence of a minimum *overall* I/O amount. Furthermore, we propose G-Clay, employing a Gray code sequence as the sub-chunk arrangement for Clay codes, transforming non-sequential disk access into sequential access with optimal I/O efficiency. We design read and write schemes based on Ceph and optimize additional overhead. Experiments indicate that G-Clay, compared to Clay, optimizes *overall* single-chunk repair time by 43.6%.

I. Introduction

Erasure codes are widely employed in modern storage systems [16], [44], offering higher reliability with lower redundancy compared to replication schemes. Among erasure codes, Reed-Solomon (RS) codes are particularly popular in production due to their simple structure and Maximum Distance Separable (MDS) property (i.e., fault tolerance equals the additional storage overhead) [35]. Systems like Azure [16], Google [12], Facebook [25], etc., widely employ RS codes. However, RS codes suffer from high repair overhead, requiring the reading and transmission of k times the data for repairing a lost chunk in a storage system with k data chunks and m parity chunks.

Various repair-friendly codes have been proposed in the literature to reduce repair bandwidth, such as locally repairable codes [16], [19], [36], piggybacking codes [33], and regenerating codes [9]. Among these, the Minimum Storage Regenerating (MSR) code maintains MDS property similar to RS and achieves the lowest repair bandwidth. Several theoretical MSR constructs have been developed, including NCCloud [15], PM-RBT [31], Butterfly [27], and Clay codes [42]. We define the amount of data read from disks by helping nodes as the *repair-read volume* and the transmitted data volume as *repair bandwidth*. For example, with $k = 10$, $m = 4$ MSR code, the *repair bandwidth* is only 32.5% of RS code, showing a substantial bandwidth reduction. Early MSR codes require *repair-read volume* greater than *repair bandwidth*, but Clay codes possess the *access-optimal* property, ensuring these two are equal and minimized (i.e., helper nodes directly transmit locally read data). Additionally, as the theoretically state-of-the-art MSR code, Clay exhibits advantages like flexible parameter selection, low computational overhead, and serves as a plugin in Ceph [1].

Despite the bandwidth advantages, we observe disk performance becoming the bottleneck for Clay code repairs in modern data centers due to the sub-packetization of MSR. As illustrated in Figure 1, we define the single-chunk repair time as the sum of disk read, network, and computation for $k = 16$, $m = 2$ Clay code using 2MB chunks. We employ mature SIMD computation libraries for the decoding calculation [23], [30] (green portion), and configure a 20Gbps RDMA network, capable of supporting up to 100Gbps [11], [17] (yellow portion). Thus *disk read time becomes the bottleneck* (blue portion).

The leftmost bar in Figure 1 represents the repair time of RS code, where each chunk exhibits identical performance. In

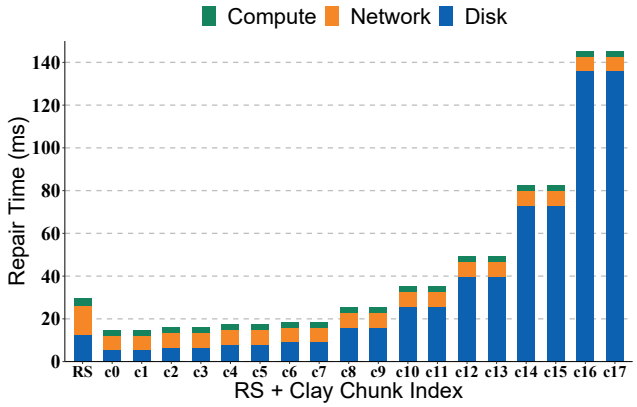


Figure 1. Comparison of single-chunk repair time between RS and Clay Code ($k = 16, m = 2$) on HDD. $c0 \sim c17$ represent the repair time for the 18 chunks of Clay code, respectively.

contrast, the repair process for Clay codes involves helping nodes transmit a subset of sub-chunks within a chunk. Due to its unique coupled structure (see §II-B), the sub-chunk access pattern varies based on the repaired chunk's position, demonstrating different degrees of non-sequentiality. Therefore we provide the repair time for all chunks of Clay (use $c0 \sim c17$ in Figure 1 to represent each chunk). For repairing $c0$, the other 17 helper nodes need to transmit the first half of the sub-chunks within the chunk, requiring only one I/O. This amount is 2 for $c2$ or $c3$, 4 for $c4$ or $c5$, and for $c17$, accessing 256 discontinuous sub-chunks leads to 256 I/Os, resulting in significantly high disk read time, surpassing even RS repair time, as depicted in Figure 1.

A direct approach involves reading the entire chunk into memory in a single I/O, filtering a subset of sub-chunks, and then transmitting them. However, this naive method increases system overhead, violating the *access-optimal* property. In §V, we discuss the limitations of this approach. Some studies attempt to mitigate non-sequential I/O overhead by utilizing larger chunk sizes while maintaining the minimum *repair-read volume* [1], [39], [42]. However, the sub-packetization level of *access-optimal* MSR codes (i.e., the number of sub-chunks within a single chunk) has been proven to be exponential [5]. For moderately large k and m, even with the largest chunk sizes allowed by system configurations, it is not effective in reducing I/O overhead (see §V). Furthermore, if small chunks are unavoidable, there is no existing research addressing the disk read efficiency issues for Clay codes.

Our insight is that, while maintaining the same *repair-read volume*, reorganize the positions of sub-chunks to improve the continuity of sub-chunk access. Placing two required scattered sub-chunks together on the disk allows for the subsequent repair to read them in a single I/O (see §II-E). The key challenge here is how to carefully select the order of sub-chunks (with a total of $\alpha!$ possible permutations) to achieve the best repair I/O performance. We consider the scenario where adjusting the sub-chunks for better continuity in the repair of one chunk might aggravate non-sequential I/O overhead for the repair of another chunk. We model the I/O problem of Clay codes, define optimization objectives, and through various attempts and comprehensive theoretical analysis, ultimately demonstrate that using **Gray codes** achieves the lowest number of I/O operations (see §III-D).

We propose G-Clay, an I/O optimization scheme that uses Gray code sequence as the sub-chunk arrangement for Clay codes, significantly enhancing the continuity of sub-chunk access during single-chunk repairs. Our contributions are as follows:

- We highlight the severe non-sequential I/O problem in Clay codes, demonstrating that disk read performance becomes a bottleneck for Clay code repairs.
- We model the I/O issues of Clay codes, demonstrate the optimality of using Gray code sequence for sub-chunk arrangement, and analyze optimization bounds.
- We implement the G-Clay plugin based on Ceph, design read and write processes incorporating sub-chunk reorganization, and optimize additional overhead. We propose a non-cyclic Gray code generation method, expanding the selection space for sub-chunk arrangements. The source code is available at: https://github.com/YuchongHu/G-Clay.
- We demonstrate the optimization effect of G-Clay with varying chunk sizes through experiments on different storage media. Results indicate that, compared to Clay, G-Clay optimizes *overall* disk read time by 47.3%, *overall* single-chunk repair time by 43.6%, with negligible additional overhead.

II. Background and Motivation

We introduce the background details of erasure codes and MSR codes. We discuss the pairwise coupling structure of Clay codes and the issue of excessive I/O operations during single-chunk repair. We review the existing solutions and propose our motivation.

A. Erasure Coding and MSR Codes

Erasure codes are commonly used as a redundancy scheme in modern storage systems [32], [44]. One popular type is Reed-Solomon (RS) codes, where divides a fixed-size object into k data chunks and computed with m parity chunks, distributed across $n = k + m$ failure domains (e.g., nodes). This provides strong reliability with smaller storage overhead, allowing for data recovery with k chunks even in the presence of up to m chunks loss.

However, the repair overhead of erasure codes is substantial. Compared to replication-based schemes, repairing a chunk incurs k times more disk reads and bandwidth amplification. Existing studies have explored various optimization approaches for RS code repair. Two main directions have emerged: the first involves system-level optimizations such as PPR [24] and repair pipeline [21], which distribute repair traffic across all nodes. However, these approaches do not effectively reduce *repair bandwidth*. The second approach involves designing new coding structures, such as locally repairable codes [16], [28], [36], where repairing a data chunk within a group requires accessing fewer nodes and less data, but at the cost of increased storage overhead. Network coding [9] encompass Minimum-Storage

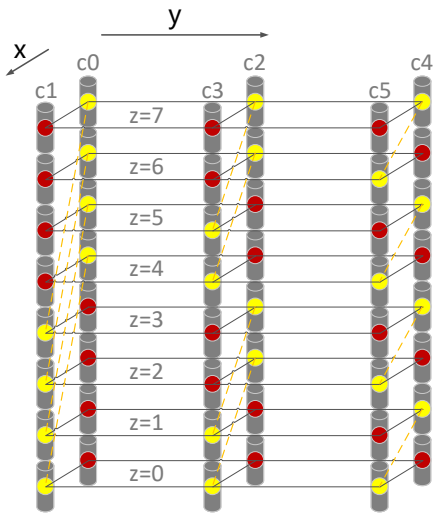


Figure 2. The pairwise coupling structure of sub-chunks in $k = 4, m = 2$ Clay code.

Regenerating (MSR) and Minimum-Bandwidth Regenerating (MBR) codes. MBR codes require storing more data, while MSR codes satisfy the MDS property and minimize *repair bandwidth*, making them popular alternatives to RS codes.

In the past decade, MSR codes have seen significant theoretical advancements [15], [20], [34], [46], however, they each have limitations. NCCloud [15] requires $m = 2$ and is a functional code, PM-RBT [31] requires $n \geq 2k - 1$, and Butterfly [27] is applicable only to $m = 2$ with optimizations limited to data chunk repair. The recently proposed Clay code [42] is the closest MSR code to perfection in theory and has found practical applications [1].

B. Clay Code

Clay code is one of MSR codes with various desirable properties, achieving the lowest *repair bandwidth* while minimizing storage cost. Compared to other MSR codes, Clay codes offer advantages such as flexible parameter selection, smaller sub-packetization level, lower computational complexity, and equivalent repair efficiency for data and parity chunks. Clay codes have been implemented as an erasure coding plugin in Ceph, establishing the practical superiority over other theoretically MSR codes.

The core idea of Clay codes is sub-packetization, with parameter set $\{(k, m), d, \alpha, \beta\}$, where k represents the number of data chunks, m represents the number of parity chunks, and the total number $n = k + m$. d denotes the number of helper nodes participating in data transmission during repair, considering only single-chunk repair with all other nodes involved, i.e., $d = n - 1$. α represents the sub-packetization level, indicating the number of sub-chunks within a chunk. Assuming n is divisible by m, then $\alpha = m^{\frac{n}{m}}$. β represents the number of sub-chunks transmitted by each helper node during single-chunk repair, $\beta = \frac{\alpha}{m}$.

The construction of Clay codes is based on pairwise coupling between sub-chunks across multiple stacked layers. Figure 2 illustrates the construction details of a Clay code with $k = 4$ and $m = 2$, where each chunk can be divided into eight sub-chunks. $c0, c1, c2, c3$ represent the four data chunks, while $c4, c5$ represent the two parity chunks, with each chunk distributed to a separate node. Nodes can be represented using coordinates (x, y), with m nodes on each y-plane, thus satisfying $0 \leq x < m, 0 \leq y < \frac{n}{m}$. The coordinates for $c0$ to $c5$ are as follows: $(0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (1, 2)$.

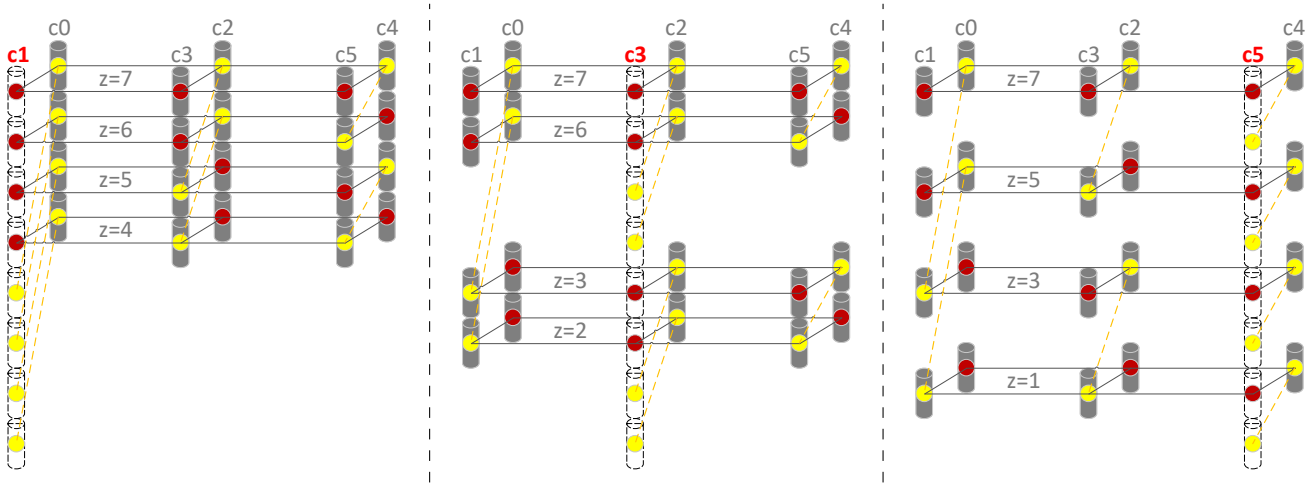
A set of sub-chunks with the same position from n different chunks is defined as a layer. In Figure 2, there are eight layers, from $z = 0$ to $z = 7$. z can be represented using $\frac{n}{m}$ coordinates in m base. For example, $z = 6$ can be represented as $(1, 1, 0)$, where $z_0 = z_1 = 1, z_2 = 0$. Uncoupled sub-chunks are represented by red cylinders in Figure 2, satisfying $x = z_y$. Paired coupled sub-chunks are represented by yellow cylinders and connected by yellow dashed lines, where the values of x and z_y are exchanged. Half of the eight sub-chunks in each chunk remain uncoupled (this ratio depends on m). Clay codes achieve encoding and decoding through the transformation of coupled and uncoupled structures, and based on the data mapping of different layer sub-chunks, single-chunk repair can be achieved with a smaller amount of data [42].

When repairing a chunk in Figure 2, the red points determine the four layers involved, and the other $n - 1$ helper nodes transmit the corresponding four sub-chunks of those layers. Compared to RS codes, which require transmitting $4 * 8 = 32$ sub-chunks from four nodes during single-chunk repair, Clay codes only require transmitting $5 * 4 = 20$ sub-chunks from five nodes, reducing the amount of disk read and bandwidth. The decode process involves pairwise reverse transform (PRT), MDS decode, and pairwise forward transform (PFT) [42].

C. Excessive Non-Sequential Reads in Clay

Although Clay codes have the theoretically lowest sub-packetization level in *access-optimal* MSR codes, they still exhibit exponential behavior [5]. In practical storage systems, without increasing the *repair-read volume*, the repair process for certain nodes inevitably introduces a significant amount of non-sequential I/Os determined by the theoretical coding structure. In this paper, we focus on the issue of single-chunk repair, as it accounts for the majority of chunk losses [18], [38].

As shown in Figure 3, for a Clay code with $k = 4$ and $m = 2$, the missing chunk are represented by dashed lines, and the uncoupled red points determine the layers to be accessed from the helper nodes. For instance, to repair $c1$, the remaining five helper nodes need to read sub-chunks at positions 4, 5, 6, and 7, which are then transmitted over the network to the substitute node for decoding calculations. Clearly, this requires one sequential I/O. When repairing $c3$, the remaining five nodes need to read sub-chunks at positions 2, 3, 6, and 7, resulting in two reads. Similarly, for repairing $c5$, the remaining nodes need to read sub-chunks at positions 1, 3, 5, and 7, but none of the sub-chunks are colocated, resulting in four reads. The I/O access patterns for repairing $c0$, $c2$, and $c4$ are similar to that of $c1$, $c3$, and $c5$, respectively.



(a) Repair c1 (b) Repair c3 (c) Repair c5

Figure 3. Illustration of sub-chunks access pattern for single-chunk repair in Clay$(4, 2)$. Dashed bars represent missing chunks.

In practical storage systems, larger parameter values for k are often used [4], [6], [16], [25], such as $k = 16$ and $m = 2$. In this case, $\alpha = 512$ and $\beta = 256$, meaning that in the **worst-case** scenario (repairing the last two chunks), since all the target sub-chunks stored in helper nodes are not contiguous, similar to Figure 3(c), a separate I/O operation is required to retrieve each sub-chunk, resulting in a total of 256 disk reads.

Such a high number of I/O operations required for single-chunk repair will be an unbearable overhead in storage system. Figure 1 illustrates this severe issue, where the chunk size is 2MB, and the sub-chunk size is 4KB. Repairing $c0$ or $c1$ represents the best-case scenario, requiring only one disk read, Clay codes outperform RS codes in terms of repair performance. Unfortunately, as the index increases, the number of disk reads doubles, resulting in increasingly longer repair time. When repairing $c17$, the disk read time is 10x of RS, and the repair time is 4.9x of RS.

D. Increasing Chunk Size to Mitigate I/O Overhead

Researchers have proposed some solutions in addressing fragmented reads. Vajha [42] suggests that using a stripe size of 64MB for parameters $k = 16$ and $m = 4$ can eliminate read amplification on SSDs, as the SSD page size is 4KB, thereby making the *repair-read volume* equal to the theoretical minimum. However, even with a sub-chunk size of 4KB, as shown in Figure 1, the worst-case disk read time increases sharply. The Ceph documentation [1] also points out that for better disk I/O performance, it is recommended to use a larger stripe size for larger k and m.

Geometric partitioning [39] employs a hybrid strategy to mitigate the impact of fragmented reads in Clay codes. It adopts a bucket-based stripe layout, dividing objects into a geometric sequence. Smaller chunks use RS encoding to eliminate fragmented reads while larger chunks utilize Clay encoding to improve disk efficiency and reduce *repair bandwidth*.

The essence of these approaches is to use larger chunks. In fact, for HDD, a disk read operation includes disk rotation, head seeking, and data reading, while SSD, built with flash memory chips, can directly access data without mechanical movement, resulting in faster access speed and higher throughput. Additionally, a single *read()* operation is often accompanied by the overhead of system calls. Therefore, reading more contiguous data can undoubtedly improve performance. However, using large chunks is not general (i.e., small chunk sizes have to be used for small objects), and larger chunks increase system overhead and lead to read amplification in partial reads. Furthermore, when applying Clay codes, moderate to large parameters are typically used, and the sub-packetization level remains relatively high. Therefore, after striping and sub-chunk partitioning, the size of a sub-chunk for a larger object (e.g., 64MB [12]) is at KB level. In the worst-case repair scenario based on HDDs, each read request involves a small amount of data, and the accumulated seek time is significant [26]. Even for SSDs, minimizing the number of sub-chunk reads is preferred.

Finding a solution that optimizes read time while maintaining the *access-optimal* property of Clay codes has become an urgent problem. In this paper, we propose a novel system-level optimization.

E. Motivation

Since the performance issue of Clay code repair is mainly caused by excessive disk reads, is there a way to reduce the number? A natural idea is: for repairing $c5$, originally we need to access sub-chunks $1, 3, 5, 7$ (see Figure 3), which are all non-contiguous and require 4 I/O operations. Is it possible to place these sub-chunks together to read the required ones with fewer I/O?

The motivation diagram is shown in Figure 4, where the arrangement of sub-chunks follows the order $\{0, 1, 3, 2, 6, 7, 5, 4\}$. This implies that the pairwise coupling during encoding follows the normal order $\{0, 1, 2, 3, 4, 5, 6, 7\}$, but when the chunks are written to disk, the internal sub-chunks are reorganized in the modified order. In this way, when repairing $c5$, we still need sub-chunks $1, 3, 5, 7$, but their arrangement has changed. Sub-chunks $1, 3$ are placed together, allowing both to be read

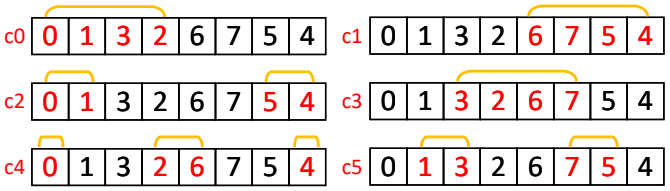


Figure 4. Motivating example for Clay$(4, 2)$. The eight sub-chunks of each chunk are reorganized in the specified order. Yellow brackets representing the positions of sub-chunks that ***should be read from the other five nodes*** for repairing that node.

in a single I/O operation, and the same goes for sub-chunks $7, 5$. As a result, when repairing $c5$, the remaining 5 helper nodes only need to perform two I/O operations each, compared to the original 4, resulting in a 50% reduction. Besides $c5$, similar benefits can be achieved for repairing other nodes. For repairing $c4$, the number of I/Os is reduced from 4 to 3, for repairing $c3$, it is reduced from 2 to 1. The numbers of I/Os for repairing $c0$, $c1$, and $c2$ remain the same as before, with no increase.

We need to point out that not every arrangement of sub-chunks can achieve optimization effects. Taking $\{2, 5, 0, 7, 4, 3, 6, 1\}$ as an example, the numbers of I/O operations for repairing $c0$ to $c5$ are: 4, 3, 3, 3, 4, 4, resulting in a significant increase. How to choose the appropriate order of sub-chunks is precisely the core problem we aim to solve.

III. Analysis of Sub-chunks Optimal Arrangement

We define the *overall* number of I/O operations and explore generating the optimal arrangement using a brute force method. We demonstrate that Gray code can achieve optimality and provide the optimization upper bound.

A. Definition and Objective

An intuitive observation is that after reorganizing the sub-chunks, the goal is to minimize the number of I/O operations for repairing a single chunk. Different orders of sub-chunks will result in different I/O distributions, and it is one-sided to only consider the I/O improvement that a certain chunk gains from the order. Here, we consider the problem from an *overall* perspective. Therefore, we define the *overall* number of I/O operations, denoted as S_{total}, to measure the quality of a sub-chunk order. For convenience, we assume that $d = n - 1$ and that n is divisible by m.

Let S_i represent the number of I/O operations for repairing the ith node, where $0 \leq i < n$. Then, we have:

$$S_{total} = \sum_{i=0}^{n-1} S_i \tag{1}$$

We denote the S_{total} for the sub-chunks arranged in sequential order as S_{seq}. In this case, S_i is dependent on the position i of the node and forms an approximately geometric progression as i increases. For example, when $m = 4$, the sequence of S_i starting from $i = 0$ is $1, 1, 1, 1, 4, 4, 4, 4, 16, 16, 16, 16$.... We can generalize S_i as:

$$S_i = m^{\lfloor i/m \rfloor} \tag{2}$$

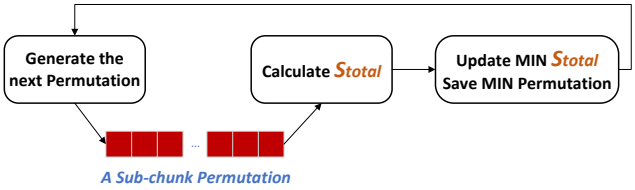


Figure 5. Finding all optimal permutations using brute force.

We can calculate the value of S_{seq} as:

$$S_{seq} = \frac{m(m^{\frac{n}{m}} - 1)}{m - 1} \tag{3}$$

Our objective is to minimize S_{total} as much as possible. For the original sequential order of sub-chunks, when $m = 2$, we have $S_i = 2^{\frac{i}{2}}$ and $S_{seq} = 2^{\frac{k}{2}+2} - 2$, where $k = 16$ results in $S_{seq} = 1022$. Such a high number of sub-chunk reads has a significantly negative impact on repair performance.

B. Brute Force

For a given set of parameters defining a Clay code, the number of possible sub-chunk orders is fixed. We can determine the order with the minimum S_{total} by exhaustively enumerating all possible permutations. As illustrated in Figure 5, we start with an initial sequential order and generate the next permutation based on current one until we have exhausted all permutations. For each generated sub-chunk permutation, we calculate S_{total}, and continuously update the min value.

The brute force enumeration guarantees finding all permutations with the minimum S_{total}, but it suffers from high complexity. For example, consider the parameters $k = 4, m = 2$, resulting in $\alpha = 8$. There are a total of $8! = 40320$ possible permutations of sub-chunks. It can be quickly determined that there are 144 permutations with the minimum $S_{total} = 10$. However, when $k = 6, m = 2$ with $\alpha = 16$, the number of possible permutations exceeds twenty trillion. As k increases, the size of solution space becomes $\alpha!$, making it practically infeasible to compute the minimum S_{total}.

We need to address the following questions: Is there an efficient algorithm to obtain an optimal sub-chunk arrangement with the minimum S_{total}? Alternatively, can we derive a theoretical lower bound for S_{total} within $\alpha!$ permutations?

C. Introduction to Gray Code

Gray code [2], [7], [10], [14] is widely used in electrical field, refers to a coding scheme where adjacent code words differ in only one position. Its initial purpose is to prevent false intermediate states during switch transitions. For example, when transitioning from 3 to 4, using natural binary representation would require a conversion from 011 to 100, indicating that all three voltages have to flip. However, this is not an atomic process, meaning the three switch operations may not be synchronized. This problem can be resolved by changing only one switch at a time. For instance, representing 3 as 011 and 4 as 010 in binary, the transition process only requires a single pulse. This is precisely the definition of Gray code: an arrangement of code words where any two words are different,

and adjacent code words differ by only one symbol (also known as *single-distance*).

Gray code can also exist in non-binary form [14]. For example, consider a 3-ary Gray code with two digits, where the underlined positions indicate changes compared to the previous code word:

$$\{00, 0\underline{1}, 0\underline{2}, \underline{1}2, 1\underline{1}, 1\underline{0}, \underline{2}0, 2\underline{1}, 2\underline{2}\}$$

D. Optimality of Gray Code

We have found that there exists a theoretical lower bound for S_{total}, achieving it represents a Gray code sequence (Theorem 2).

First, we define the concept of bit flipping. Given a sub-chunk arrangement, we write it as a sequence of digits in base m. Bit flip is defined as the position of a $value$ that differs from the previous $value$ encountered while traversing from top to bottom. Table I shows the bit flips for the sequential sub-chunk order of $k = 4, m = 2$ Clay code. Each bit flip is indicated in bold, and the total number of bit flips at each position is provided at the bottom. Here, we distinguish the concept of *cyclic* [10]. The $value$ in the first row always represents one bit flip, even if it is the same as the $value$ in the last row (implying that if the required sub-chunks are at first and last positions, also requiring two I/O operations).

In §II-B, we mention that the sub-chunk index z can be represented by $\frac{n}{m}$ m-base digits, as the three binary digits shown in Table I. When $d = n - 1$ and n is divisible by m, the n nodes of Clay code are organized into $\frac{n}{m}$ y-planes, each plane containing m nodes. Therefore, $0 \leq y < \frac{n}{m}$, and a node's y-coordinate can also represent the y-th digit (or column) in the m-base table (e.g., in Table I, $y = 0, 1, 2$ corresponds to positions $Bit0, Bit1, Bit2$, respectively).

We use $BitFlip_{total}$ to denote the total number of bit flips and $BitFlip_y$ to denote the number of flips in the y-th column. Thus, we have the following equation:

$$BitFlip_{total} = \sum_{y=0}^{\frac{n}{m}-1} BitFlip_y \tag{4}$$

Lemma 1. *The number of flips in y-th column of the m-base table is equal to the sum of ranges with the same value in y-th column.*

Proof: In y-th column, there are z values, each taking a range of $0 \leq value < m$. Each bit flip from top to bottom represents encountering a different range of values (the first bit flip represents the first $value$ range). Therefore, the sum of bit flips in a column is equal to the sum of ranges. □

We illustrate this with Table I, for instance, in $Bit1$ position ($y = 1$), there are a total of 4 bit flips (indicated in bold), each flip represents encountering a different *continuous* range of 0 or 1. Thus, these two quantities are equivalent.

Lemma 2. *The number of flips in y-th column of the m-base table is equal to the sum of all node's single-chunk repair I/O operations in the y-plane. With nodes represented by the coordinates x and y, we have:*

Table I
BIT FLIPS OF SEQUENTIAL ORDER IN CLAY$(4, 2)$

Sub-chunk Index	Bit0	Bit1	Bit2
0	**0**	**0**	**0**
1	0	0	**1**
2	0	**1**	**0**
3	0	1	**1**
4	**1**	**0**	**0**
5	1	0	**1**
6	1	**1**	**0**
7	1	1	**1**
Sum = 14	*2*	*4*	*8*

Table II
BIT FLIPS OF GRAY CODE ORDER IN CLAY$(4, 2)$

Sub-chunk Index	Bit0	Bit1	Bit2
0	**0**	**0**	**0**
1	0	0	**1**
3	0	**1**	1
2	0	1	**0**
6	**1**	1	0
7	1	1	**1**
5	1	**0**	1
4	1	0	**0**
Sum = 10	*2*	*3*	*5*

$$BitFlip_y = \sum_{x=0}^{m-1} S_{(x,y)} \tag{5}$$

Proof: For a given $y = Y$ plane, the uncoupled sub-chunks of an internal node (x, Y) satisfy $x = z_Y$, which means that the positions of required sub-chunks that other helper nodes read and transfer should satisfy this equation. The arrangement of sub-chunks on disks follows the order of the m-base table, and the sub-chunks satisfying $x = z_Y$ corresponding to all rows in the table's Y-th column with a $value$ of x. Each separated range of $value = x$ represents one I/O operation. The sum of I/O operations for all nodes with $0 \leq x < m$ in the Y-plane is equal to the sum of ranges for all $0 \leq value < m$ in the Y-th column of the table. According to Lemma 1, it is also equal to the sum of bit flips. □

For example, in Table I, the number of bit flips in the $Bit2$ column is 8, and the I/O amount of repairing the two nodes $c4$ and $c5$ with $y = 2$ is 4 each (see Figure 2), which adds up to 8.

Theorem 1: *For any Clay code, the overall number of I/O operations is equal to the total sum of bit flips in the m-base table organized in sub-chunks order. We have:*

$$S_{total} = BitFlip_{total} \tag{6}$$

Proof: Based on Lemma 2, summing both sides over all y planes, according to Equation (1) and (4), the proof holds. □

Lemma 3. *In the m-base table of a Clay code order, the lower bound for the total number of bit flips only exists when there is exactly one bit flip between every two adjacent code words.*

Proof: Each z value is unique, which means that there is at least one differing bit between every two adjacent z values. Therefore, if we can ensure that there is only one bit flip between adjacent code words, it implies the minimum total number of flips. □

Theorem 2. *For any Clay code, the minimum overall number of I/O operations can only be achieved by using a Gray code sequence as the sub-chunks order. Denoting the lowest S_{total} as S_{min}, we obtain:*

$$S_{min} = m^{\frac{n}{m}} + \frac{n}{m} - 1 \tag{7}$$

Proof: Based on Lemma 3 and the definition of Gray code in §III-C, it can be concluded that the Gray code sequence achieves the lower bound of bit flip amount. Furthermore, according to Theorem 1, the Gray code sequence minimizes the *overall* number of I/O operations. In this case, S_{min} can be calculated as the sum of $\frac{n}{m}$ flips in the first row of the m-base table, and one flip per row for the remaining $m^{\frac{n}{m}} - 1$ rows. □

Table II provides the bit flips for a Gray code sequence $\{0, 1, 3, 2, 6, 7, 5, 4\}$ when $k = 4$ and $m = 2$, with $S_{min} = 10$. The repair I/O patterns of this order is shown in Figure 4.

E. Applying Gray Codes

We refer to Clay codes with sub-chunks reorganized in Gray code order as **G-Clay**. Based on Equation (3) and (7), we can calculate that for $k = 16$ and $m = 2$, $S_{seq} = 1022$ and $S_{min} = 520$, which represents an optimization of nearly half I/O amount.

There are multiple methods to generate Gray codes, and different Gray codes have the same minimum S_{total}, albeit the distribution of I/O amounts can vary. This may impact the repair performance of practical storage systems. Here, we introduce two Gray code generation algorithms: Binary Reflected Gray Codes (*BRGC*) [13] and Balanced Gray Codes (*BalanceGC*) [7]. *BRGC* is a classic method that generates Gray code using mirror symmetry, while *BalanceGC* can be generated using partitioning and Hamiltonian cycle [40], with the characteristic of nearly equal digit flips at each position. Taking $k = 6$ and $m = 2$ as an example, with $\alpha = 16$ sub-chunks, the Gray codes generated by these two methods are as follows ($SeqOrder$ representing the original sequence):

$$\begin{aligned} SeqOrder &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\} \\ BRGC &= \{0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8\} \\ BalanceGC &= \{1, 3, 7, 5, 4, 12, 14, 6, 2, 10, 11, 15, 13, 9, 8, 0\} \end{aligned}$$

Table III provides the distribution of I/O amounts for these three sub-chunk orders. It can be observed that although both *BRGC* and *BalanceGC* achieve the optimal sum of 19 I/O operations, the distribution in *BalanceGC* is more uniform. While *BRGC* has a difference of up to 4, which implies that there can be significant variations in disk read time when repairing chunks with different indices, i.e., non-uniformity.

It is worth mentioning that when it is possible to manually control how chunks are distributed across each node (unlike Ceph's CRUSH algorithm [45]), it is feasible to allocate indices requiring fewer I/Os for repair to nodes that are more prone to failure. This way, G-Clay demonstrates excellent disk access performance for frequent repairs on that node.

Table III
COMPARISON OF I/O AMOUNTS FOR DIFFERENT ORDER IN CLAY$(6, 2)$

Repaired Chunk Index	SeqOrder	BRGC	BalanceGC
0	1	1	3
1	1	1	2
2	2	2	3
3	2	1	2
4	4	3	3
5	4	2	2
6	8	5	2
7	8	4	2
Sum of I/O amounts	*30*	*19*	*19*

There can be additional overhead while writing Clay-coded data to disk with a well-designed sub-chunk order. For instance, when reading the object normally, the sub-chunks are retrieved from disk of the order different from sequential, requiring restoration to the original sequence. However, as we will discuss later, this overhead can be optimized to a low level, making it acceptable (see §IV-C).

F. Optimization Upper Bound

Here we prove that the reduction ratio of I/O operations S_{min} compared to S_{seq} has an upper bound, determined by m. Dividing Equation (7) by Equation (3), we obtain:

$$\frac{S_{min}}{S_{seq}} = \frac{n(m-1)}{m^2(m^{\frac{n}{m}} - 1)} + \frac{m-1}{m} \tag{8}$$

As n increases with a fixed m (i.e., more data chunks), the left side of Equation (8) approaches 0, converging to a constant:

$$\lim_{n \to +\infty} \frac{S_{min}}{S_{seq}} = \frac{m-1}{m} \tag{9}$$

Therefore, the ratio of the reduction in S_{min} compared to S_{seq} is:

$$\lim_{n \to +\infty} \frac{S_{seq} - S_{min}}{S_{seq}} = \frac{1}{m} \tag{10}$$

Typically $m \geq 2$ is required in MSR codes. When $m = 2$, the theoretical upper bound for I/O optimization is achieved at 50%. Figure 6 illustrates the converging trend of the reduction ratio applying Gray codes as $digits$ increases, where $digits$ represents the number of digits in z, equal to $\frac{n}{m}$. Clearly, as the number of data chunks increases, the I/O optimization ratio becomes higher, while it quickly approaches the theoretical upper bound, For example, when $m = 2$, $k = 16$ achieves a ratio of 49.1%, while for $m = 4$, $k = 16$ achieves 24.6%, and for $m = 5$, $k = 15$ achieves 19.5%, without imposing a large value of k.

IV. DESIGN AND IMPLEMENTATION

We design G-Clay, the enhanced Clay code aims to improve I/O efficiency. We outline the system architecture, and propose an algorithm for generating non-cyclic Gray codes. We optimize the additional overhead of encoding and reading.

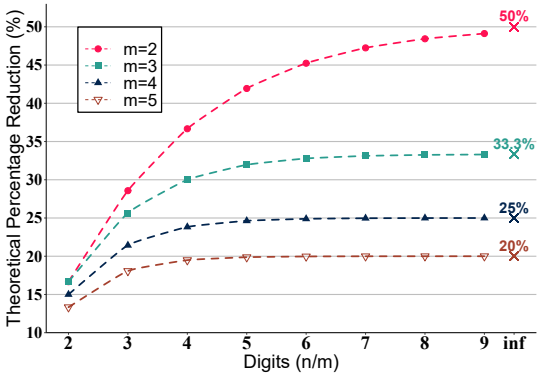


Figure 6. Theoretical optimization upper bound with different m of applying Gray codes.

A. Architecture

We design G-Clay based on Ceph. Ceph is a popular open-source distributed unified storage system that supports object, file, and block storage applications [44], based on the underlying Rados object store. Ceph eliminates the need for metadata maintenance on a single node through CRUSH algorithm [45], making it highly suitable for distributed scalability with massive data. Each OSD (Object Storage Daemon) serves as a backend daemon responsible for storing actual data, typically managing one storage device (HDD or SSD), and acts as an individual storage node (i.e., failure domain is a node). Ceph adopts BlueStore as the backend storage engine, replacing the old file system-based FileStore. BlueStore allows direct data writes to block devices, eliminating the additional overhead of file systems. It also integrates data compression and checksum functionalities more effectively. Additionally, Ceph has been optimized for modern hardware, such as SSDs, resulting in improved performance.

Figure 7 illustrates the architecture of G-Clay. First, outside the Ceph system, we generate a Gray code in advance to serve as the sub-chunk reorganization scheme. Generally, using *BRGC* yields satisfactory results. When writing an object to Rados, it is initially mapped to a Placement Group (PG) based on its object ID, determining the backend OSD daemons to store the chunks after divided into erasure coding stripes. Each PG has a primary OSD, denoted as p-OSD, responsible for dividing the object into k equal-sized chunks, performing encoding to generate m parity chunks, and transmitting the remaining $k+m-1$ chunks to other OSDs within the same PG.

Before the chunks are persisted to disk, we need to perform the sub-chunk reorganization. As depicted in Figure 7, assuming there are four sub-chunks and the desired order is $\{0,1,3,2\}$, all chunks need to be rearranged accordingly. This operation takes place in memory and involves simple copying, resulting in fast execution.

Once all sub-chunks within a chunk have been persisted in the Gray code sequence, the benefits of improved read performance with increased continuity can be realized when repairing a stripe.

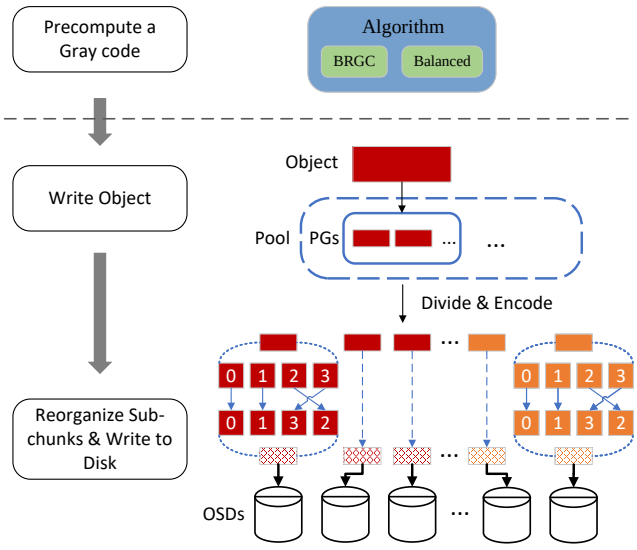


Figure 7. Architecture of G-Clay based on Ceph.

B. An Algorithm for Generating Non-Cyclic Gray Codes

Most existing methods for generating Gray codes require ensuring *cyclic* property or obtain only one result [13], [14]. This limitation hinders the selection of suitable Gray codes. Recall that sub-chunks order does not need to be *cyclic* (§III-D). Therefore, given the challenges posed in this paper, it is necessary to design a new algorithm for generating non-cyclic Gray codes.

Figure 8 illustrates the algorithm, which offers the advantage of a significantly larger solution space, facilitating easier selection. Firstly, for 4 sub-chunks with $k=2$ and $m=2$, there are a total of $4!=24$ permutations. By using a brute-force method, we find that there are 8 permutations satisfying $S_{total}=S_{min}=5$, as shown in Figure 8 for $k=2$. Here, we demonstrate how to recursively obtain a portion of the optimal permutations for $k=4$ based on the results for $k=2$. For example, considering the first permutation $\{0,1,3,2\}$, the last digit is 2. Based on the condition of starting with 2, we can select two permutations: $\{2,0,1,3\}$ and $\{2,3,1,0\}$. Adding 4 (the number of sub-chunks for $k=2$) to these two permutations gives $\{6,4,5,7\}$ and $\{6,7,5,4\}$, respectively.

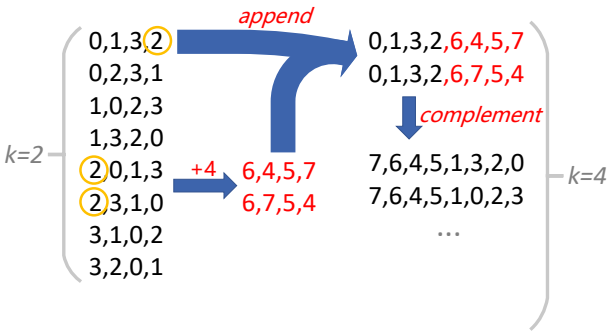


Figure 8. An example of generating non-cyclic Gray codes.

We then append them to the end of $\{0, 1, 3, 2\}$, resulting in two optimal permutations for $k = 4$: $\{0, 1, 3, 2, 6, 4, 5, 7\}$ and $\{0, 1, 3, 2, 6, 7, 5, 4\}$. Furthermore, applying the complement operation with respect to 8 (the number of sub-chunks for $k = 4$) yields $\{7, 6, 4, 5, 1, 3, 2, 0\}$.

In this way, each permutation for $k = 2$ can determine four optimal permutations for $k = 4$, resulting in a total of 32 results. However, this is not the entire solutions for $k = 4$ but a subset (total is 144). Nevertheless, by continuing to recursively iterate from $k = 4$ to $k = 6$, $k = 8$, and so on, the resulting set becomes extremely large, enabling a more deliberate selection of Gray codes that meet the desired I/O distribution.

Moreover, we can start the iteration from the 144 optimal permutations for $k = 4$ and move forward, resulting in a richer solution space. The correctness is similar to the proof of *BRGC*. For example, consider the sequence $\{0, 1, 3, 2, 6, 4, 5, 7\}$, the first four and last four digits are individually Gray codes. Besides, the middle two digits, 2 and 6, differ only in the most significant bit being 0 or 1. Therefore, the entire sequence is a Gray code.

C. Parallel for Reading

We can optimize the overhead of reorganizing sub-chunks for normal reads by using multithread, similar for encoding phase. Firstly, we distribute the overhead of sub-chunk reorganization evenly among the participating OSDs, allowing them to execute in parallel instead of single-point bottleneck. As shown in Figure 9, for a given OSD, we first read a single chunk belonging to the stripe from persistent storage (e.g., HDD or SSD) and obtain a set of sub-chunks organized in a Gray code.

The traditional approach involves using a single thread to process α sub-chunks and restore them to the initial order of $0, 1, 2, 3, 4....$ However, this process can be accelerated using multiple threads. For example, as shown in Figure 9, we use thread 1 to process the restoration of the first four sub-chunks and thread 2 to process the restoration of the next four sub-chunks, and so on. Because the position of each sub-chunk is unique, there are no conflicts between threads, eliminating the need for thread locks. Once finished, the original chunk is sent to the p-OSD. The p-OSD then receives all the original

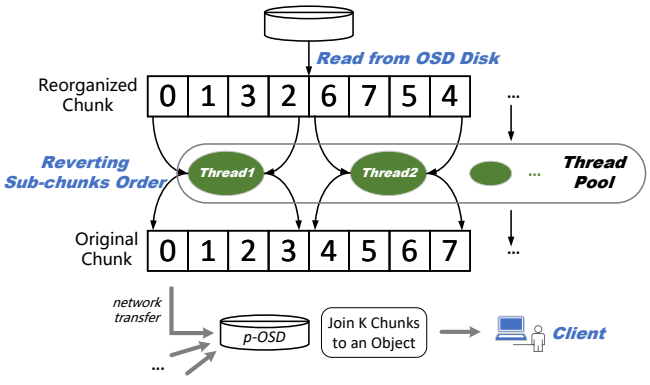


Figure 9. Parallel sub-chunks order restoration during reading.

chunks, concatenates them into a complete object, and sends to the client.

Depending on the machine performance and α, we can set different number of threads. For instance, on a 4-core machine with 512 sub-chunks, we can run 8 threads simultaneously, each responsible for restoring a smaller range (64 each).

We provide clarification that the introduction of sub-chunk reorganization does not impact data integrity checks in Ceph. 1) At the chunk level, the chunks sent to OSD are as original. OSD generates and stores the checksum based on the original chunk, then reorganizes the sub-chunks. When reading the chunk, the checksum is verified on the restored chunk. 2) At the stripe level, writing and reading objects involve the original data chunks, following the same process as before.

D. Implementation in Ceph

We implement the G-Clay plugin on Ceph (Jerasure 2.0 as the RS component library to leverage Intel SIMD instructions [29]), along with several Gray code generation algorithms, about 2k lines of C++ code. G-Clay plugin requires the addition of a class member vector called $order$, which represents the chosen Gray code sequence. During plugin initialization, we hardcode the constant $order$, which means that the Gray code used remains unchanged throughout the system's runtime.

The most critical modification involves altering the *minimum_to_decode()* function. It serves as a interface exposed by the erasure coding module. It determines which sub-chunks need to be read from other nodes when Clay code needs to repair a chunk at a specific index. The function returns a list of tuples in the form $< pos, count >$, indicating that $count$ sub-chunks need to be read from position pos. The more tuples in the list, the more I/O operations are required.

With G-Clay, we need to modify the range of sub-chunks to be read based on the existing $order$. For example, for Clay$(4, 2)$, when repairing the 5-th chunk, the original code requires the other five nodes to access sub-chunks at positions $1, 3, 5, 7$, resulting in a list of tuples: $[< 1, 1 >, < 3, 1 >, < 5, 1 >, < 7, 1 >]$. However, with G-Clay and the $order$ $\{0, 1, 3, 2, 6, 7, 5, 4\}$, the list is modified to $[< 1, 2 >, < 5, 2 >]$, which instructs the *ECBackend* module to read sub-chunks at positions $1, 2$ (corresponding to the original sub-chunks $1, 3$) and sub-chunks at positions $5, 6$ (corresponding to the original sub-chunks $7, 5$) in fewer I/O operations. In the end, BlueStore reads the disk in a *DIRECT* manner. It is important to note that BlueStore does not have a built-in prefetch mechanism, so it does not transparently read additional unnecessary sub-chunks.

During single-chunk repair, it may seem that we encounter a problem: the help data read and transmitted to the p-OSD is reorganized based on $order$, do we need to restore sub-chunks? Actually it does not. We can slightly modify the decoding calculation. For example, when we need the original Clay's sub-chunk at position i, we directly map it to the sub-chunk at position j in the help data, which actually holds the original i-th sub-chunk. Similarly, a mapping is required when specifying the destination sub-chunk positions. So the decoding process incurs no additional overhead.

V. Evaluation

We conduct experiments on different Gray codes and parameters across various storage media. Results reveal the benefits of the sub-chunk reorganization scheme in reducing non-sequential I/Os, while demonstrating that the side effects can be minimal.

A. Experiment Setting

We conduct tests on a local cluster comprising 18 nodes, each node deploying an OSD daemon for data storage, and there are 3 monitors and 1 manager, with each daemon coexisting on a single OSD node. We perform direct read and write operations on the underlying objects using *rados* command. Each node is equipped with a Intel Xeon CPU of 3.3GHz (supporting SSE and AVX), 8GB of memory, and both a 1TB 7200RPM HDD and a 500GB NVMe SSD. By binding an OSD to either HDD or SSD, we are able to evaluate the optimization effects on different storage devices (for each experiment, a consistent set of HDD or SSD is used). We use Ubuntu 18.04 as the operating system and Ceph version 15.2.17.

Servers are connected via a 25Gbps RoCEv2 network, we configure the bandwidth using Linux *tc* command. To measure the time taken for a single chunk repair, we first write 1000 objects of the same size to a single PG. We then manually take one machine offline to initiate the automatic node repair process within the cluster. All experiments are run 10 times, and the average repair time are calculated. Ceph provides logging capabilities, so we add additional logging codes to the G-Clay plugin and the *ECBackend* module. We set a certain log level and after the repair process concludes, analyze the result logs (located in *systemd journal* [3]). By correlating specific behaviors with timestamps, we can determine the read time, network transmission time, computation time, and sub-chunk reorganization time.

To evaluate the algorithm's effectiveness on large objects, we need to set *osd_recovery_max_chunk* to a sufficiently large value. Additionally, when varying the chunk size, we can adjust the *stripe_unit* parameter in the *erasure-code-profile*. Before each experiment, we thoroughly clean up the previous pool and data, and set a new *erasure-code-profile* to avoid any impact on the results due to old stripes.

B. Codes Evaluated

For the most focused parameters with $k = 16$ and $m = 2$, we compare four different Gray codes with the original order. The main one used is *BRGC*, which has the effect of almost halving the number of I/O operations required for each chunk's repair (see Table IV). We also employ the Gray code generation methods mentioned in §IV-B, randomly selecting two of them, naming as *Special1* and *Special2*. We also generate balanced Gray codes as [40]. Table IV presents a comparison of these Gray codes in terms of I/O amounts for repairing each chunk.

When $m = 2$, since the repairing I/O amounts for chunks at odd positions are very close to those at adjacent even positions, for the sake of clarity, *we only plot the disk read time for odd chunks*. However, the reduction ratio is with respect to the total of all chunks.

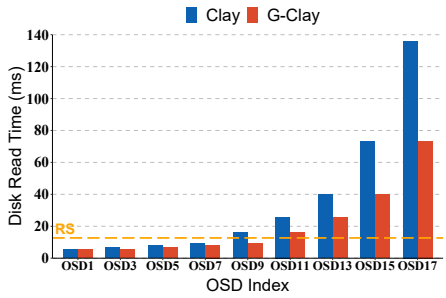
Table IV
Comparison of I/O amounts for different order in Clay$(16, 2)$

Index[a]	SeqOrder	BRGC	Special1	Special2	BalanceGC
0	1	1	1	1	29
1	1	1	1	1	28
2	2	2	2	2	29
3	2	1	1	1	28
4	4	3	3	3	29
5	4	2	2	2	28
6	8	5	5	5	30
7	8	4	4	4	29
8	16	9	9	9	30
9	16	8	8	8	29
10	32	17	17	17	30
11	32	16	16	16	29
12	64	33	33	72	30
13	64	32	32	71	29
14	128	65	117	66	29
15	128	64	117	65	28
16	256	129	76	89	28
17	256	128	76	88	28
Sum	*1022*	*520*	*520*	*520*	*520*

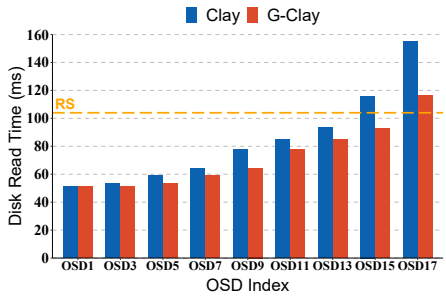
[a]Repaired Chunk Index.

C. Experiments

Experiment 1 (Optimization Effects of $BRGC$ on HDD): In §I, we discuss how the disk read time of repairing becomes a bottleneck for Clay$(16, 2)$. Here, we employ $BRGC$ as the sub-chunk reorganization scheme and measure the repair time. Figure 10(a) only presents a comparison of the disk read time since the overhead for G-Clay network transmission and decoding computation is the same as Clay. We use a yellow

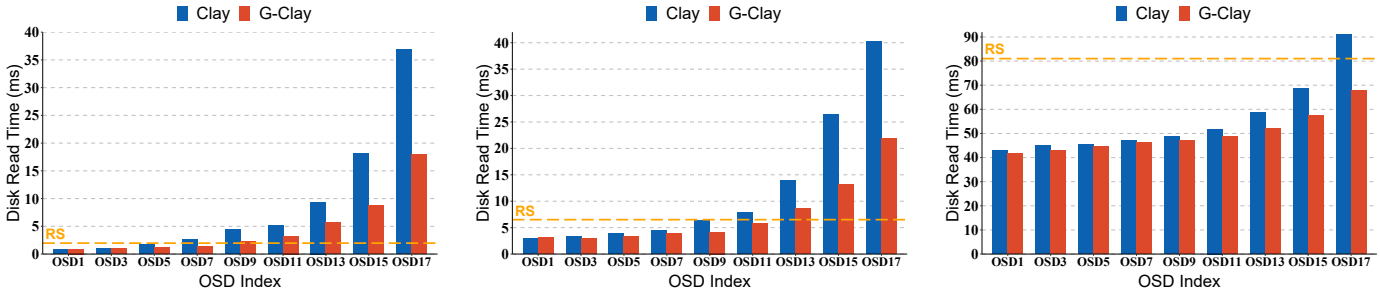


(a) HDD-2MB



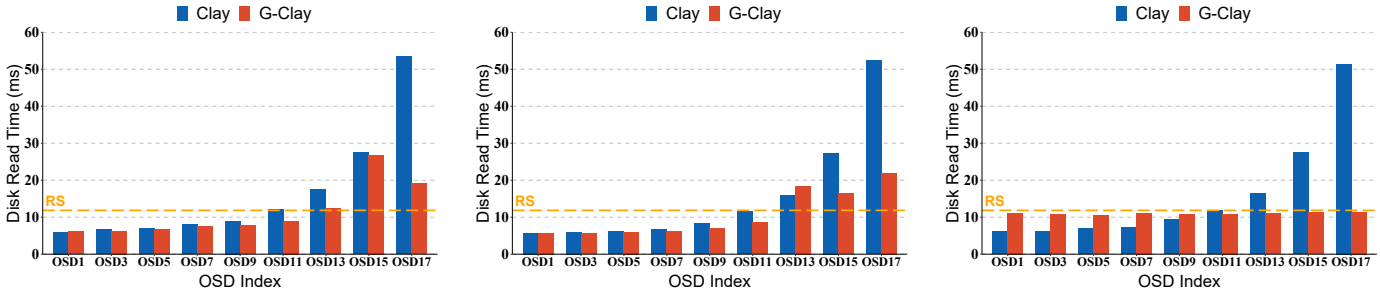
(b) HDD-16MB

Figure 10. Experiment 1: Effects of G-Clay$(16, 2)$ on HDD using chunk sizes of 2MB and 16MB (yellow dashed line representing RS reads an entire chunk).



(a) SSD-2MB (b) SSD-16MB (c) SSD-256MB

Figure 11. Experiment 2: Effects of G-Clay$(16, 2)$ on SSD using chunk sizes of 2MB, 16MB and 256MB.



(a) Special1-32MB (b) Special2-32MB (c) Balance-32MB

Figure 12. Experiment 3: Effects of G-Clay$(16, 2)$ on SSD using other Gray codes.

dashed line to represent the time taken to read an entire chunk of RS code (subsequent experiments follow the same pattern).

We observe that G-Clay reduces the reading time for repairing almost each chunk, with savings reaching close to 50% in the worst-case, resulting in an *overall* disk read time reduction of 40.2%. Correspondingly, the *overall* repair time (i.e., the sum of repair time for all chunks) reduced by 37% compared to Clay in Figure 1.

We also measure the repair performance for 16MB chunks on HDD. As shown in Figure 10(b), when the chunk size increases to 16MB, the additional I/O operations do not significantly affect the reading time due to the low throughput of HDD media. Thus, G-Clay, as a solution aimed at reducing the number of I/O, achieves modest optimization effects of 12.9%.

Experiment 2 (Optimization Effects of $BRGC$ on SSD): Here we explore optimization when using a faster storage medium. We utilize an NVMe SSD with read speed of 3GB/s, which is a significant improvement compared to the 150MB/s of the HDD mentioned above. The SSD also performs excellently in terms of IOPS and other aspects. Figure 11(a) illustrates the significant optimization effect of G-Clay compared to Clay in terms of disk read time when the chunk size is 2MB, resulting in a reduction ratio of 45.6%. As the chunk size gradually increases, the optimization effect diminishes, but at a slower rate compared to HDD. For instance, Figure 11(b) shows that even with a chunk size of 16MB, the SSD can still achieve a 35.4% optimization. Moreover, Figure 11(c) indicates that with a chunk size of 256MB, i.e., each sub-chunk being 512KB, the reduction ratio is 11.7%.

We can use empirical deduction to explain the reasons of the significant performance differences between HDD and SSD. Defining a disk read as the sum of two parts: *preparation delay* and *storage media read*. For high-end storage devices like SSDs, both the parts are fast. Regardless of reading large or small chunks, the *preparation delay* remains the same, while the time for *storage media read* increases with larger chunk sizes. It becomes apparent that when the chunk size is small, reducing the number of disk reads yields significant optimization due to the *preparation delay* and *storage media read* being relatively equal. However, as the chunk size increases, the proportion of *storage media read* gradually rises, while the *preparation delay* represents a negligible portion. Thus, larger chunks degrading the advantages of G-Clay.

Additionally, it is challenging to determine whether the trade-off on reading more data to reduce I/O operations genuinely reduces disk read time (e.g., read an entire chunk and transfer required sub-chunks). It leads to reading a substantial amount of unnecessary data (e.g., when $m = 4$, only $\frac{1}{4}$ of sub-chunks are needed), which indicates that *repair-read volume* can not remain minimal. In practical storage systems, disk performance is affected not only by the medium itself but also by factors such as machine load and I/O queues [26]. In some cloud environments, block device may not be locally attached to the OS, and additional read data is translated to network overhead [43], leading to increased repair time.

In short, our algorithm effectively reduces the number of I/O operations and improves disk efficiency without increasing the *repair-read volume*.

Experiment 3 (Optimization Effects of Different Gray Codes): Based on Table IV, it can be observed that different

Gray codes have the same $S_{min} = 520$, but the distribution differs. We compare different Gray codes using 32MB chunk size. The impact of $BRGC$ has already been discussed, and Figures 12(a)(b) demonstrate the optimization effects of $Special1$ and $Special2$, respectively. We can see that the disk read time is proportional to I/O amount. For example, the I/O operations for repairing OSD17 is reduced from 256 to 76 in the case of $Special1$, showing a significant optimization effect in Figures 12(a). Balanced Gray codes, on the other hand, exhibit the characteristic that the I/O amounts are more uniform. Figure 12(c) validates this observation, and the optimized disk read time are all lower than that of RS reading an entire chunk. Therefore, balanced Gray codes are suitable for clusters that require uniformity in repair operations.

Despite the different distribution of I/O amounts, all three Gray codes mentioned above achieve approximately a 30% optimization in disk read time compared to the original Clay order. This validates our empirical analysis, which focused on reducing the number of *preparation delay*.

Experiment 4 (Case of $m > 2$): We further evaluate the actual disk read time for $k = 9$ and $m = 3$ to provide more comprehensive evidence for the benefits. Figure 13 illustrates the disk read time required for repairing each OSD when sub-chunk size is 4KB, i.e., the chunk size is 324KB. Unlike the case of $m = 2$, where the reduction in I/O operations per chunk is nearly halved, we plot the results for all OSDs in this scenario. According to Equation (8), the theoretical reduction ratio should be 30%, while the actual results achieve 26.5%. The original I/O amounts for the last three chunks are 27 for each. After applying $BRGC$ as the sub-chunk arrangement, the I/O amounts for OSD9 and OSD11 decreased to 14, while OSD10 remains 27 without any optimization. This demonstrates that as m increases, the optimization effect indeed diminishes.

Experiment 5 (Optimization Effects of Different k and Chunk Sizes): We evaluate the *overall* disk read time (sum of read time for all chunk's repair) using various k with fixed $m = 2$. Due to the different I/O distributions, we focus on the collective disk read time rather than a individual chunk. The results are depicted in Figure 14. Firstly, we observe that as k increases, the optimization effect becomes more pronounced, validating our theoretical derivation in §III-F. For instance, when $k = 16$ and the sub-chunk size is 2KB (i.e., chunk size is 1MB), achieving an optimization ratio of 47.3%, which is very close to its theoretical upper bound of 50%. While the optimization ratio decreases to 24.4% when $k = 4$.

We also find that as the sub-chunk size increases, the optimization effect gradually diminishes, reaching its lowest point at 512KB sub-chunks, consistently around 10%. This is because the *preparation delay* already constitutes a small proportion, limiting the impact of optimizing this part.

However, we rarely use such large sub-chunk size of 512KB when k is medium or large in practical scenarios (e.g., HDFS block size is 64MB [8]), which indicates that G-Clay can bring an *overall* I/O performance gain of over 25% in most cases.

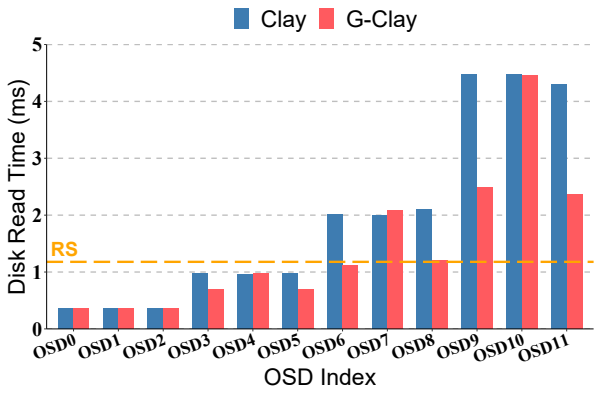


Figure 13. Experiment 4: Effects of G-Clay$(9, 3)$ on SSD using $BRGC$ and chunk size of 324KB.

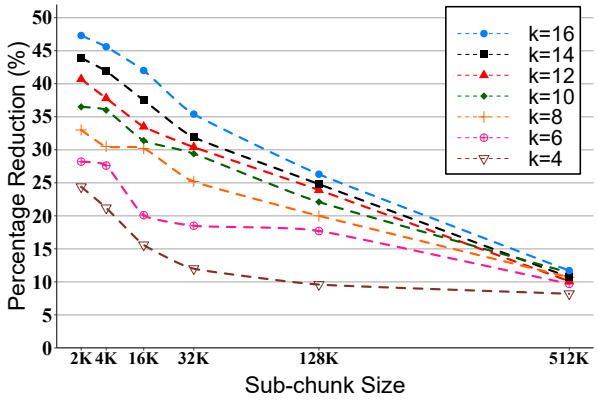


Figure 14. Experiment 5: Reduction ratio of *overall* disk read time for G-Clay$(k, 2)$ on SSD varies with sub-chunk sizes.

Experiment 6 (Encoding Time and Reading Time): Sub-chunk reorganization improves the single-chunk repair performance at the cost of encoding time and normal reading time, as it introduces an additional sub-chunk reorganizing process. Figure 15 presents an evaluation of encoding time and reading time for 2MB chunks with $k = 16, m = 2$ on HDD. G-Clay demonstrates that p-OSD completes the sub-chunk reorganization operation for all chunks using a single thread, the encoding time increases by 2.3% and the reading time increases by 3.2%. This indicates that the overhead of sub-chunk reorganization is already relatively low. After applying the optimizations mentioned in §IV-C, as shown for G-Clay-Opt in Figure 15, the additional overhead for encoding and reading is only 0.04% and 0.06% respectively, which is almost negligible.

We attribute this to the fact that the sub-chunk reorganization primarily involves a small amount of memory copying, rather than slow disk reads or network transfers, and parallelism has increased the speed. In addition, the normal reading process already incurs additional calculations such as chunk hash verification, which helps offset the overhead.

We need to point out that encoding is a one-time task, and normal reading is optimized to have minimal overhead, it is worthwhile to balance the routine single-chunk repairs.

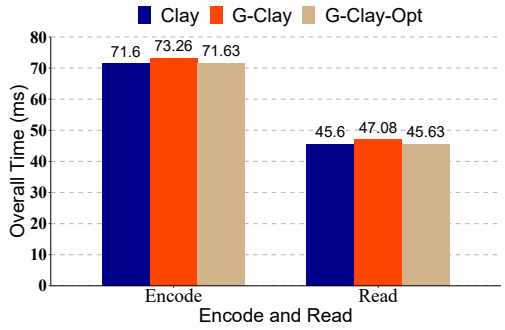


Figure 15. Experiment 6: Comparison of encode and read time for Clay, G-Clay and G-Clay-Opt.

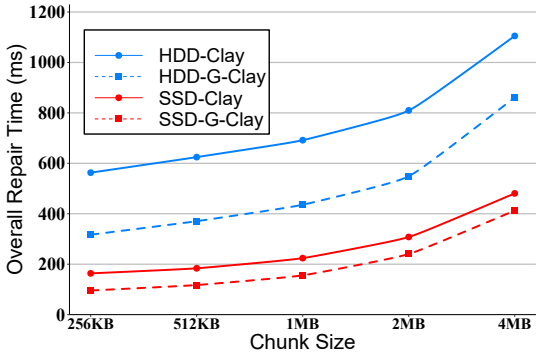


Figure 16. Experiment 7: *Overall* repair time (sum of repair time for 18 chunks) of Clay and G-Clay ($k = 16, m = 2$) on HDD and SSD with varying chunk sizes.

Experiment 7 (Optimization of *Overall* Repair Time): The optimization of G-Clay for disk reads translates into a significant reduction in repair time. Similar to §III-A, we continue to consider the optimization of repair time from an *overall* perspective, defining the *overall* repair time as the sum of single-chunk repair time for all $k + m$ chunks. Figure 16 illustrates the comparison before and after optimization on HDD and SSD.

We configure the bandwidth to 20Gbps within the cluster and employ Intel Streaming SIMD Extensions [23] (SSE) instructions to accelerate calculations. We observe that G-Clay exhibits significant optimization effects on *overall* repair time, especially for smaller chunk sizes. For instance, with a chunk size of 256KB, G-Clay achieves a 43.6% improvement on HDD and a 41.5% improvement on SSD. However, this optimization ratio gradually diminishes as the chunk size increases. For example, with a chunk size of 4MB, the *overall* repair time optimization on HDD reduces to 22%. This can be attributed to the fact that larger chunk sizes enhance the disk efficiency through better sub-chunk continuity, while network transmission time and computation time gradually occupy a larger proportion, thereby degrading the advantages of G-Clay.

G-Clay demonstrates more pronounced optimization effects on HDD since disk read performance is usually the bottleneck in such cases. Even for high-performance SSDs, adopting small chunks can still result in a significant number of non-sequential I/Os becoming a bottleneck for repairs. In this regard, G-Clay demonstrates outstanding I/O optimization.

VI. Related Work

Increasing Chunk Size. Researchers have proposed solutions to improve I/O efficiency by increasing the chunk size. Vajha suggests using 64MB chunks [42], and larger stripe sizes are also recommended in the Ceph documentation [1]. Geometric partitioning presents a system-level optimization based on a hybrid strategy, where small chunks use RS encoding and large chunks employ Clay encoding [39]. However, these methods cannot optimize for small chunks. In contrast, our G-Clay can reduce the number of I/O operations regardless of chunk size.

Parallel Repair. ParaRC [22] parallelizes the repair process of MSR codes to reduce the maximum repair load of individual nodes. However, this comes at the cost of increased *repair bandwidth*, which is no longer optimal for Clay codes.

Elastic Transformation. Elastic transformation [41] allows the conversion of a base code into a code with lower *repair bandwidth*, which has smaller sub-packetization level than Clay, thus limiting non-sequential I/O overhead. However, the *repair bandwidth* remains higher than that of Clay.

Hop-and-Couple. This method is initially proposed in Hitchhiker [33] and has been applied to vector codes like Clay. It involves dividing a chunk into α (sub-packetization level) sub-chunks. Instead of operating on individual bytes, the encoding calculations are performed on sub-chunks composed of a group of bytes. This should serve as a fundamental method for partitioning sub-chunks in vector code implementations, but there is still disk inefficiency when accessing non-sequentially placed sub-chunks.

Gray Code. Gray code is a classic encoding technique that solves practical problems in various domains, including electrical engineering [2], [13]. Gray code is defined as a sequence where adjacent code words differ by only one digit. There are several ways to generate Gray codes, with the most typical being Binary Reflected Gray Code ($BRGC$) [13]. Additionally, there are balanced Gray codes [7], non-Boolean Gray codes [14], and other theoretical advancements [37], [47]. We now extend the application of Gray code to MSR codes in the realm of storage, enriching its connotation.

VII. Future Work

In this paper, we only consider single-chunk repair, and we will explore how to improve the I/O efficiency of multi-chunk repair for Clay codes. Furthermore, we will study how to design the arrangement of sub-chunks within adjacent stripes during full-node repair to achieve better I/O contiguity. It may be worth considering relaxing the access-optimal property of Clay codes by reading some additional sub-chunks to reduce the number of I/O operations. We will explore I/O optimization for regenerating codes in more complex scenarios, such as involving background operations, SSD aging, storage device heterogeneity, etc. We also plan to implement G-Clay on different storage platforms to provide a more comprehensive report on repair performance improvements.

VIII. Conclusions

We propose G-Clay, an optimal sub-chunk reorganization scheme that using Gray code to reduce non-sequential I/O overhead and improve disk efficiency for single-chunk repairs. We prove that employing Gray code is necessary to achieve the minimum *overall* number of I/O operations. We analyze the optimization limits and provide a Gray code generation algorithm. We implement the G-Clay plugin based on Ceph and conduct experiments to validate the improved repair performance with minimal side effects. We believe that the system-level optimizations presented in this paper can assist in mitigating the practical challenges of Clay codes.

Acknowledgment

We thank our shepherd, James Hughes, and the anonymous reviewers for their valuable feedback. This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No.2021YFB0301400), National Natural Science Foundation of China (No.62272185), the Key Research and Development Program of Hubei Province (No.2021BAA189) and Key Laboratory of Information Storage System Ministry of Education of China. The corresponding author is Yuchong Hu.

References

[1] Ceph erasure-code-clay. https://docs.ceph.com/en/quincy/rados/operations/erasure-code-clay.

[2] Gray code. https://en.wikipedia.org/wiki/Gray_code.

[3] journalctl manual. https://man7.org/linux/man-pages/man1/journalctl.1.html.

[4] B. Beach. Backblaze Vaults. Zettabyte-scale cloud storage architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture, 2017.

[5] S. Balaji and P. V. Kumar. A tight lower bound on the sub-packetization level of optimal-access msr and mds codes. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 2381–2385. IEEE, 2018.

[6] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 351–365, 2014.

[7] G. S. Bhat and C. D. Savage. Balanced gray codes. *the electronic journal of combinatorics*, 3(1):R25, 1996.

[8] D. Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

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

[10] R. W. Doran. The gray code. Technical report, Citeseer, 2007.

[11] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, et al. When cloud storage meets {rdma}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[13] F. Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

[14] D.-J. Guan. Generalized gray codes with applications. In *PROC NATL SCI COUNC REPUB CHINA PART A PHYS SCI ENG*, volume 22, pages 841–848. Citeseer, 1998.

[15] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. Nccloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, volume 21, 2012.

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[18] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*, page 20, 2012.

[19] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. *ACM Transactions on Storage (TOS)*, 16(2):1–32, 2020.

[20] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data*, 4(4):516–529, 2017.

[21] R. Li, X. Li, P. P. Lee, and Q. Huang. Repair pipelining for {Erasure-Coded} storage. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 567–579, 2017.

[22] X. Li, K. Cheng, K. Tang, P. P. Lee, Y. Hu, D. Feng, J. Li, and T.-Y. Wu. {ParaRC}: Embracing {Sub-Packetization} for repair parallelization in {MSR-Coded} storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 17–32, 2023.

[23] C. Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23:1–21, 2011.

[24] S. Mitra, R. Panta, M.-R. Ra, and S. 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.

[25] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm {BLOB} storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, 2014.

[26] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 1–15, 2012.

[27] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of {MSR} codes. In *14th USENIX conference on file and storage technologies (FAST 16)*, pages 81–94, 2016.

[28] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, 2014.

[29] J. S. Plank and K. M. Greenan. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. *Technical Report UT-EECS-14–721, University of Tennessee, Tech. Rep.*, 2014.

[30] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast galois field arithmetic using intel simd instructions. In *FAST*, pages 299–306, 2013.

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

[32] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[33] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A" hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 331–342, 2014.

[34] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011.

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

[36] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. *arXiv preprint arXiv:1301.3791*, 2013.

[37] C. Savage. A survey of combinatorial gray codes. *SIAM review*, 39(4):605–629, 1997.

[38] B. Schroeder and G. 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.

[39] Y. Shan, K. Chen, T. Gong, L. Zhou, T. Zhou, and Y. Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In

Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 457–471, 2021.

[40] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 875–892, 2016.

[41] K. Tang, K. Cheng, H. H. Chan, X. Li, P. P. Lee, Y. Hu, J. Li, and T.-Y. Wu. Balancing repair bandwidth and sub-packetization in erasure-coded storage via elastic transformation. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2023.

[42] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay codes: Moulding {MDS} codes to yield an {MSR} code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 139–154, 2018.

[43] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the VLDB Endowment*, 3(1-2):506–514, 2010.

[44] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.

[45] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 122–es, 2006.

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

[47] K. Zeger and A. Gersho. Pseudo-gray coding. *IEEE Transactions on communications*, 38(12):2147–2158, 1990.