2015 IEEE 34th Symposium on Reliable Distributed Systems

Seek-Efficient I/O Optimization in Single Failure Recovery for XOR-Coded Storage Systems

Zhirong Shen, Jiwu Shu *, Yingxun Fu
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
zhirong.shen2601@gmail.com, shujw@tsinghua.edu.cn, fu-yx10@mails.tsinghua.edu.cn

***Abstract*—Erasure coding provides an effective means for storage systems to protect against disk failures with low redundancy. One important objective for erasure-coded storage systems is to speed up single disk failure recovery. Previous studies reduce the amount of read data for recovery by reading only a small subset of data, but their approaches often incur high disk seeks, which may negate the resulting recovery performance. We propose *SIOR*, a seek-efficient I/O recovery algorithm for single failure. SIOR carefully balances the trade-off between the amount of read data and the number of disk seeks by considering the data layout at the multi-stripe level. It then greedily determines the data to read for recovery using Tabu search. Experiments show that SIOR achieves similar performance to the brute-force enumeration method while keeping high search efficiency. Also, SIOR reduces 31.8%∼65.1% of disk seeks during recovery and provides up to 186.8% recovery speed improvement, when compared to a state-of-the-art greedy recovery approach.**

I. Introduction

Disk failures are commonplace in large-scale distributed storage systems [18], and hence protecting data with redundancy is critical to provide fault tolerance guarantees. Given that replication has high storage overhead and aggravates the operational cost of storage, *erasure coding* is increasingly adopted in commercial storage systems (e.g., Azure [10] and Facebook [15]). It takes original information as input and produces redundant information. The original information and redundant information connected by an erasure code form a ***stripe***. A storage system organizes data in different stripes, each of which is independently operated by erasure coding.

Although erasure coding tolerates multiple disk failures, single disk failures dominate over 90% of failure events in practice [11], [15], [18]. Conventional recovery for single (disk) failures often needs to read a considerable amount of surviving data. To reduce the amount of read data during single failure recovery, researchers propose a spate of solutions [11], [12], [19], [21]–[23], which can be categorized as follows:

1) New constructions of RAID-6 codes which tolerate double failures, such as HDP Code [21], and HV Code [19], which read less data for recovery than existing RAID-6 codes.
2) Optimization techniques for specific codes, such as RDP Code (by [22]) and X-Code (by [24]), which provide a provably lower bound of the amount of read data.
3) General optimization techniques for any erasure codes [11], [25].

Although these studies effectively reduce the amount of read data for recovery, they achieve this by reading and correlating different portions of a stripe to reconstruct the failed data. This leads to additional *disk seeks*,[1] which can negate the actual recovery performance. A disk seek needs to move the mechanic disk head to the position where the accessed data resides, and its overhead is generally larger than that of transferring the same size of accessed data. However, most existing studies do not take into account the disk seek overhead in their recovery solutions. Some studies mitigate the disk seek overhead by assuming large-size I/O units (e.g., 16MB [5], [11], [12], [19], [25]), yet traditional file systems often operate on small-size disk blocks (e.g., 4KB in EXT3 [4]). How to balance the trade-off of different block sizes between normal I/O performance and recovery performance remains unexplored.

In this paper, we explore the seek-efficient I/O optimization problem of single failure recovery for any XOR-based erasure code. Our observation is that previous studies only focus on recovery at the *single* stripe level, yet storage systems are usually configured in stripes [7]. Thus, by collectively examining the data layouts of *multiple* stripes, we can reduce the number of disk seeks in recovery. To this end, we propose *SIOR*, a multi-stripe single failure recovery scheme that not only reduces the amount of read data, but also minimizes the number of disk seeks. *SIOR is suitable for the storage systems (e.g., [9], [20]) that are sensitive to both repair traffic and recovery I/O for the data reconstruction.*

SIOR can be decomposed into two stages. In the first stage, SIOR finds an initial recovery solution that is able to rebuild the lost data at the multi-stripe level and satisfies the requirements on the data reduction. In the second stage, SIOR utilizes Tabu Search [6] and iteratively obtains another solution that gains fewer disk seeks without violating the requirement on the

* Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).
This work was supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the State Key Laboratory of High-end Server and Storage Technology (Grant No. 2014HSSA02).

[1] We use the term "seeks" to collectively refer to seeks and rotations [17].

1060-9857/15 $31.00 © 2015 IEEE
DOI 10.1109/SRDS.2015.20

data reduction. Besides, SIOR also proposes a *filling method* that selectively reads unrequested data to further decrease disk seeks. Therefore, *SIOR fully lightens the load on disks during single failure recovery, in terms of the amount of read data and the number of disk seeks.* To our best knowledge, it is the ***first*** work to simultaneously consider the optimization of these two metrics for single failure reconstruction.

In summary, we make the following contributions.

1) We formulate the seek-efficient I/O optimization problem for single failure recovery. The formulation not only minimizes the number of disk seeks, but also reduces the amount of read data.
2) We make use of Tabu search [6] and propose a general greedy algorithm named SIOR for any XOR-based erasure code. SIOR iteratively selects a recovery solution with fewer disk seeks, while preserving the reduction on the read data for recovery. SIOR also proposes the filling method to reduce disk seeks.
3) We implement SIOR over a real storage system constructed over several representative XOR-based erasure codes. Evaluations shows that SIOR significantly decreases the search time when compared with the brute-force enumeration method. Moreover, SIOR decreases 31.8%∼65.1% disk seeks during the recovery, and provides up to 186.8% recovery speed improvement, when compared with Zpacr [25], a state-of-the-art greedy recovery approach to minimize the amount of read data.

The rest of this paper is organized as follows. Section II introduces the research background. The problem formulation is provided in Section III. We present the detailed design of SIOR in Section IV. After that, we evaluate the performance of SIOR in Section V and conclude our work in Section VI.

II. Background

A. Basics

One typical class of erasure codes is **M**aximum **D**istance **S**eparable (MDS) codes [2], which reach the optimal storage efficiency. MDS codes are typically configured by two parameters k and m: an (k, m) MDS code transforms k pieces of original data and produces another m redundant pieces, such that *any* k out of $k+m$ pieces are sufficient to reconstruct the original data. The $k + m$ dependent pieces collectively form a ***stripe*** and are distributed to $k + m$ disks. In this work, we focus on XOR-based MDS codes, in which encoding/decoding operations are based on XOR operations only.

The coded storage system is also the composition of many independent stripes. Figure 1 shows the logical structure of an XOR-coded storage system with 3 stripes. Meanwhile, we denote the maximum set of information in a stripe that stored on the same disk as ***strip***. In XOR-coded storage systems, a strip is partitioned into w elements with equal-size, where ***element*** is the basic operated unit (e.g., byte or block). For example, in Figure 1, each stripe has 6 strips and every strip has 4 elements.

To clarify the internal connection among elements in a stripe, we present the layout of RDP Code [3] over $p + 1$ disks ($p = 5$) as an example in Figure 2. Table I also lists the notations and their descriptions in this clarification. A stripe usually consists of ***data elements*** and ***parity elements***. Data elements contain the original data information, while parity elements keep the redundant information. For example, $E_{1,1}$ is a data element and $E_{1,5}$ is a parity element in Figure 2(a).

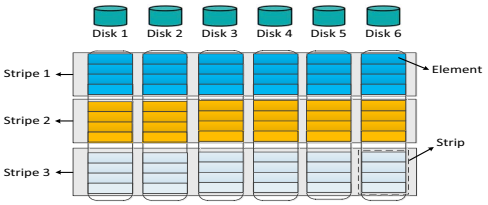


Fig. 1. The relationship among disk, stripe, strip, and element in an XOR-Coded storage system.

TABLE I
Background Notations and Descriptions

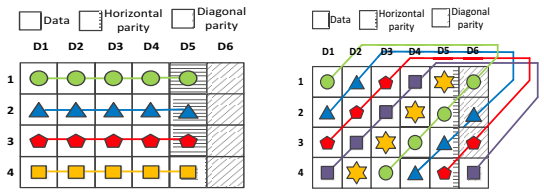
Notations	Descriptions
p	prime number to configure the stripe size
n	number of disks in an XOR-coded storage system
k, m	number of data disks, number of parity disks
w	number of elements in a strip
D_j	the j-th disk
$E_{i,j}$	element at the i-th row and j-th disk
$\oplus$	XOR operation
$\sum\{E_{i,j}\}$	sum of XOR operation among the elements $\{E_{i,j}\}$

To generate the parity element, a subgroup of data elements will be selected to perform the encoding operation. For example, RDP Code needs to calculate "horizontal parity elements" and "diagonal parity elements". The horizontal parity element $E_{1,5} := \sum_{i=1}^{4} E_{1,i} := E_{1,1} \oplus \cdots \oplus E_{1,4}$ in Figure 2(a), and the diagonal parity element $E_{1,6} := E_{1,1} \oplus E_{4,3} \oplus E_{3,4} \oplus E_{2,5}$ as shown in Figure 2(b). We also denote the parity generation relationship as the ***parity chain*** when it is used to repair lost elements. For example, $E_{1,1}$ can be recovered by two parity chains in Figure 2, i.e., $\{E_{1,1}, E_{1,2}, \cdots, E_{1,5}\}$ that generates the horizontal parity element $E_{1,5}$ and $\{E_{1,1}, E_{4,3}, E_{3,4}, E_{2,5}, E_{1,6}\}$ that produces the diagonal parity element $E_{1,6}$.

B. XOR-based Erasure Codes

XOR-based erasure codes calculate the parity elements by just performing XOR operations. This kind of erasure codes has better encoding/decoding/update efficiency, compared to the codes that are based on complicated operations in the finite field, such as Reed-Solomon Code [16]. As shown in Figure 2, RDP Code [3] is a typical XOR-based erasure code.

To generally represent any XOR-based erasure code, one can arrange the data elements residing on disks into a *data element vector* and utilize a *generator matrix* [13], [14] which is actually a binary matrix, to simulate the encoding procedure as the multiplication between the generator matrix and the data element vector. For example, Figure 3 illustrates the generator matrix of RDP Code ($p = 5$) in a stripe.



(a) The Horizontal Parity Chain. (b) The Diagonal Parity Chain.

Fig. 2. The layout of RDP Code with $p+1$ disks ($p=5$) in a stripe. $\{E_{1,1}, \cdots, E_{1,5}\}$ is a horizontal parity chain and $\{E_{1,1}, E_{4,3}, E_{3,4}, E_{2,5}, E_{1,6}\}$ is a diagonal parity chain.

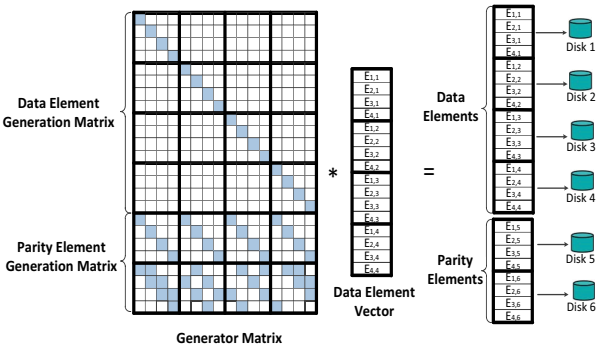


Fig. 3. The generator matrix for RDP Code (p=5) in a stripe. The filling cell in the generator matrix means "1", while the blank cell means "0".

C. Single Failure Problem

For single failure recovery, most previous works propose to accelerate the repair process by reducing the number of elements that need to be read from the surviving disks.

For RDP Code, Xiang et al. [22] firstly propose to repair the corrupted elements by selectively using the diagonal parity chains and the horizontal parity chains, so that the optimal solution with the fewest retrieved elements will be obtained. Figure 4 shows three recovery schemes to rebuild Disk 1 in two stripes for RDP Code. To repair the lost data elements in the Stripe 1, the first 2 schemes (i.e., Figure 4(a) and Figure 4(b)), which only use horizontal parity chains and diagonal parity chains respectively during the recovery, require 16 elements. By mixing the two kinds of parity chains, the recovery solution of Stripe 1 in Figure 4(c) only needs 12 elements. Following this inspiration, Xu et al. [24] investigate the optimal recovery for single failure in X-Code [23].

To get the minimum number of read elements during single failure recovery for any XOR-based erasure code, Khan et al. [11] propose to enumerate all the parity chains for each failed element based on the generator matrix and convert the possible parity chains into a directed and weighted graph. Therefore, the optimization of single failure recovery can be transformed to the problem of finding the shortest path in the converted graph. Besides retrieving the fewest elements, Luo et al. [12] also discover that the solution that evenly reads elements from surviving disks can achieve a faster recovery. Fu et al. [5] also study the load-balancing problem at multi-stripe setting. However, all of the above solutions are NP-Hard and thus are not capable of supporting the on-line reconstruction.

Aiming at this shortcoming, Zhu et al. [25] greedily search the solution with a near-minimum number of read elements for any XOR-based erasure code. In the meantime, many erasure codes [10], [19], [21] also consider the performance of single failure recovery in their code designs. However, these methods are only suitable for the recovery in a single stripe.

D. Remaining Problems in Single Failure Recovery

Although intensive attempts have been made on the single failure recovery, two restrictions are still remained.

The Increasing Number of Disk Seeks: Most prior works [5], [11], [12], [19], [21], [22], [25], [26] only focus on decreasing the elements to be read during the recovery, and neglect the increasing number of disk seeks. For example, in Figure 4(a), it reads 32 elements from the surviving disks, and causes 5 seek operations. Figure 4(c) only reads 24 elements, but produces 17 seek operations. Disk seek is expensive especially when the size of an I/O unit is small. Thus, the optimization on disk seeks should be carefully studied for single failure recovery.

Recovery in Multiple Stripes: Most previous studies [11], [12], [19], [21], [22], [26] also only consider the recovery in a single stripe, generally because their problems are stripe-independent and the stripe-level solutions they propose can be directly applied in multiple stripes.

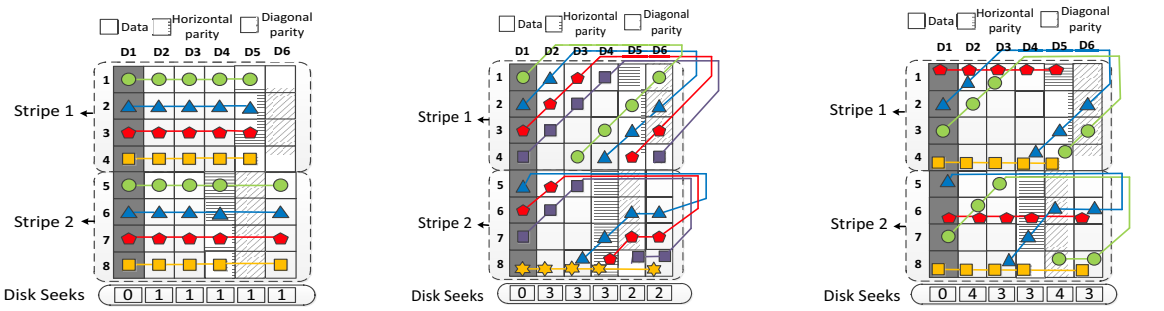
However, the optimization of disk seeks is more complicated, as the number of disk seeks usually correlates with the positions of read elements among stripes. For example, in Figure 4(b), the two elements $E_{4,3}$ (in Stripe 1) and $E_{5,3}$ (in Stripe 2) can be read directly, while it needs an extra seek operation to retrieve $E_{3,2}$ (in Stripe 1) and $E_{5,2}$ (in Stripe 2). Therefore, it is more practical to consider this problem at the multi-stripe setting.

In practice, as the disk hosting parity elements will usually serve the heavy updates when the associated data elements are written, multiple stripes are usually organized by the stripe rotation [7] for load balancing. In this paper, we rotate the stripes by gradually shifting the elements to left when the stripe identity increases. Take Figure 4 as an example, parity elements of Stripe 1 are placed at D_5 and D_6, and parity elements of Stripe 2 will be shifted to D_4 and D_5.

III. Problem Formulation and Motivations

Based on the above problems, we thus pose the following question: Can we decrease disk seeks during the single failure recovery at the multi-stripe level, while still reducing the amount of read data? One typical scenario is: *Given an expected number of retrieved elements, minimize the number of disk seeks at the multi-stripe level.*

We can formulate the problem as follows: suppose an XOR-coded storage system consists of n disks $\{D_1, \cdots, D_n\}$. Given a corrupted disk D_i $(1 \leq i \leq n)$ and the expected number of retrieved elements M, suppose the recovery solution reads m_j elements from disk D_j $(1 \leq j \neq i \leq n)$ and consequently causes o_j seek operations. Our objective is to find the recovery solution that minimizes the number of seek operations and



(a) The Conventional Solution with Horizontal Parity Chains only. It needs 32 elements and causes 5 seek operations.

(b) The Solution with Diagonal Parity Chains in Stripe 1 and Hybrid Chains in Stripe 2. It reads 29 elements and causes 13 seek operations.

(c) An Optimal Solution (2 horizontal parity chains and 2 diagonal parity chains in each stripe). It fetches 24 elements and incurs 17 seek operations.

Fig. 4. The three recovery solutions to repair Disk 1 for RDP Code ($p = 5$) in two rotated stripes. It indicates that the data reduction in single failure recovery may increase the seek operations.

also retrieves no more than M elements. This objective can be formulated as follows:

$$\text{Minimize} \sum_{1 \le j \ne i \le n} o_j \tag{1}$$

subject to

$$\sum_{1 \le j \ne i \le n} m_j \le M \tag{2}$$

Our objective is an optimization problem of disk seeks with a restriction on the number of read elements. To achieve this objective, the *enumeration method* that exhaustively tests all possible parity chains for each lost element will cause an extremely high computation complexity. For example, if the disk D_0 fails in X-Code [23], suppose there are s stripes and the prime number is p. Then there are $(p-2)s$ data elements on each disk and each lost data element can be recovered by at least 2 parity chains (according to the property of RAID-6), so the search space is larger than $2^{(p-2)s}$. Pre-storing the optimal recovery solution cannot always retain its optimality especially under the environment with constant changing factors, for example, the changing available network bandwidth in a heterogeneous environment [26].

Therefore, a better idea is to propose a greedy algorithm that can efficiently pursue a near-optimal solution. To this end, we propose to partition the search procedure into two stages. In the first stage, we find an "initial solution" that satisfies the restriction (2). As discussed above, extensive studies have been made to minimize the number of required elements in a single stripe, such as Zpacr [25]. Therefore, we can apply them for each stripe in turn and get an initial solution. In the second stage, we can further perform a deeper optimization based on the initial solution, by trying every other possible parity chain for each lost element and greedily selecting the best replacement in each iteration.

IV. The Detailed Design of SIOR

Based on the above motivation, we propose our greedy algorithm called SIOR. SIOR can be decomposed into "Initial Solution Selection" and "Initial Solution Optimization". The used symbols and descriptions in SIOR are listed in Table II.

Algorithm 1: Initial Solution Selection

Input: s: number of stripes
M: expected number of read elements in s stripes
Output: $\mathbb{R}_{ini}$: initial solution

```
for i = 1 to s do
    Initialize R_i for the i-th stripe
Build a recovery solution R_ini := (R_1, ..., R_s)
for i = 1 to s do
    Apply Zpacr [25] to R_i and get R̄_i
    Set R_ini := (R̄_1, ..., R̄_i, R_{i+1}, ..., R_s)
    Get the element distribution D_ini of R_ini
    if e(D_ini) ≤ M then
        Return R_ini
Return false
```

A. Initial Solution Selection

To obtain an initial solution, for each stripe, we can make use of existing stripe-level[2] methodologies [11], [25] to minimize the number of read elements for recovery, until the number of retrieved elements accumulated in all the stripes satisfies the restriction (2). Two widely adopted methodologies for any XOR-based erasure code are considered, i.e., the enumeration scheme [11] that is NP-Hard, and the greedy algorithm Zpacr [25] that finds the near-optimal solution in polynomial time. To efficiently provide an on-line recovery solution, we choose Zpacr [25] to serve as the cornerstone in Initial Solution Selection. The detailed procedure is presented in Algorithm 1.

Algorithm Details: At the beginning, we first initiate a solution $\mathcal{R}_i$ ($1 \le i \le s$) for the i-th stripe (step 1∼2), where $\mathcal{R}_i$ is composed of parity elements. It indicates that the lost elements in the i-th stripe can be recovered by using

[2]The stripe-level means that these methodologies can only optimize the number of retrieved elements for a single stripe.

TABLE II
THE USED SYMBOLS AND DESCRIPTIONS IN SIOR

Symbols	Descriptions
Defined in Algorithm 1	
M	expected number of read elements for recovery
$\mathbb{R}_{ini}$	initial recovery solution of s stripes
$\mathcal{R}_i$	recovery solution for the i-th stripe before optimization
$\overline{\mathcal{R}}_i$	recovery solution for the i-th stripe after optimization
s	number of stripes
$\mathbb{D}$	distribution of read elements before filling
$e(\cdot)$	function to calculate the number of read elements
Defined in Algorithm 2	
$\mathcal{L}$	Tabu list
$\mathbb{F}$	distribution of read elements after filling
$\mathbb{F}_{opt}$	optimal recovery solution found by SIOR
C_x	candidate parity chain for the lost element x
S_x	selected parity chain for the lost element x
$\mathcal{A}$	set of candidate solutions in an iteration
$q(\cdot)$	function to calculate the number of disk seeks

the parity chains that associate with these parity elements in $\mathcal{R}_i$. For example, $\mathcal{R}_1 := \{E_{1,5}, \cdots, E_{4,5}\}$ in the Stripe 1 of Figure 4(a), where the corrupted element $E_{j,1}$ ($1 \leq j \leq 4$) can be recovered by using the horizontal parity chain that generates $E_{j,5}$, i.e., $E_{j,1} = E_{j,2} \oplus E_{j,3} \oplus E_{j,4} \oplus E_{j,5}$.

In this paper, a recovery solution is ***valid*** if it satisfies the restriction (2) on the number of retrieved elements. We construct an initial solution $\mathbb{R}_{ini}$ which includes the recovery solutions for s stripes (step 3). Then for the i-th stripe, we take $\mathcal{R}_i$ as input, apply Zpacr [25] for optimization, and get a near-optimal recovery solution $\overline{\mathcal{R}}_i$ (step 5). We replace $\mathcal{R}_i$ with $\overline{\mathcal{R}}_i$ in $\mathbb{R}_{ini}$ and get an updated initial solution $\mathbb{R}_{ini} := (\overline{\mathcal{R}}_1, \cdots, \overline{\mathcal{R}}_i, \mathcal{R}_{i+1}, \cdots, \mathcal{R}_s)$ (step 6). Suppose $\mathbb{D}_{ini}$ denotes the element distribution of $\mathbb{R}_{ini}$ and $e(\mathbb{D}_{ini})$ represents the number of retrieved elements in $\mathbb{D}_{ini}$ (step 7). Once the total number of retrieved elements $e(\mathbb{D}_{ini})$ is no more than the expected number of read elements M, then $\mathbb{R}_{ini}$ will be returned as a valid initial solution (step 8∼9). Otherwise, the algorithm will return false (step 10).

An Example: We take Figure 4 as an example, where $s = 2$, and M is set as 24 in this example. We use a parity element to denote its associated parity chain. Algorithm 1 first generates a solution $\mathbb{R}_{ini} := (\mathcal{R}_1, \mathcal{R}_2)$ and its element distribution $\mathbb{D}_{ini}$ is shown in Figure 4(a), where $\mathcal{R}_1 := \{E_{1,5}, \cdots, E_{4,5}\}$ and $\mathcal{R}_2 := \{E_{5,4}, \cdots, E_{8,4}\}$. It reads 32 elements. After performing the optimization, we obtain the initial solution $\mathbb{R}_{ini} := (\overline{\mathcal{R}}_1, \overline{\mathcal{R}}_2)$ and its distribution $\mathbb{D}_{ini}$ is shown in Figure 4(c), where $\overline{\mathcal{R}}_1 := \{E_{1,5}, E_{2,6}, E_{3,6}, E_{4,5}\}$ and $\overline{\mathcal{R}}_2 := \{E_{6,5}, E_{6,4}, E_{8,5}, E_{8,4}\}$. The initial solution retrieves 24 elements and satisfies the requirement (2).

B. Initial Solution Optimization

Although the initial solution satisfies the restriction (2), it may cause many disk seeks. In this section, we accordingly propose Algorithm 2 based on Tabu Search [6] to optimize the number of disk seeks. Tabu Search effectively avoids sticking in suboptimal search regions when compared to other local search methods like hill-climbing algorithm [25]. It specifies a search rule and keeps a structure called a Tabu list. For a visited solution, once it violates the rule, it will be recorded in the Tabu list and passed over for a period of time. Before explicitly presenting Algorithm 2, we first give a rough sketch about its main idea.

Main idea:

1) ***Simplified recovery model***. To iteratively make the initial solution approach the optimal one, an important step is to try other parity chains to repair lost elements and then execute a greedy selection. For example, in the solution of Figure 4(a), the lost data element $E_{1,1}$ is recovered by the horizontal parity chain that generates $E_{1,5}$. We can get another recovery solution by simply selecting another parity chain that $E_{1,1}$ involves, i.e., the diagonal parity chain that produces $E_{1,6}$. Some previous works, such as [11] and [12], adopt the enumeration method to exhaustively try every possible parity chain according to the generator matrix. Unfortunately, this enumeration method is NP-Hard.

Given the need for efficiency, we choose a simplified recovery model instead. We repair lost elements by simply using parity chains that encode the parity elements. For example, we only consider the horizontal/diagonal parity chains for data recovery in RDP Code, as it only has these two kinds of parity elements.

2) ***The filling procedure***. Algorithm 2 may get a solution requiring less than M element retrievals. Given the distribution of requested elements, we first define the concept of *interval* that is the core reason that causes disk seeks.

Definition 1. *If two sequentially requested elements are not physically contiguous, we call the maximum set of unrequested elements between them as an "interval".*

We take the solution illustrated in Figure 5(b) as an example. $E_{4,2}$ and $E_{6,2}$ are two elements that are sequentially requested, and $E_{5,2}$ is an element that is not needed between them. $E_{5,2}$ is called an interval since it takes an extra seek operation for the disk to read $E_{4,2}$ and $E_{6,2}$. Similarly, $\{E_{2,3}, E_{3,3}\}$ forms an interval between the requested elements $E_{1,3}$ and $E_{4,3}$. The ***filling procedure*** reads all the unrequested elements in the selected intervals, so as to reduce the disk seeks. For example, the solution in Figure 5(b) needs 15 disk seeks. As a comparison, the solution in Figure 5(c) not only reads the same requested elements in Figure 5(b), but also reads $E_{5,2}$ and $E_{7,3}$, and then the number of disk seeks reduces to 13. We can observe that filling an interval by reading the elements in it will decrease a seek operation.

Therefore, given a valid recovery solution $\mathbb{R}$ and its element distribution $\mathbb{D}$, suppose the number of retrieved elements in $\mathbb{D}$ is $e(\mathbb{D})$. To decrease the disk seeks, we can fill the intervals by retrieving no more than $M - e(\mathbb{D})$ elements that are unrequested. Moreover, we also give the Theorem 1, which points out how to reduce the maximum number of disk seeks when given a constant number of filled elements.

Theorem 1. *The filling procedure that fills the intervals in*

Algorithm 2: Initial Solution Optimization

Input: $\mathcal{L}$: Tabu list
t: number of iterations
$\mathbb{R}_{ini}$: initial recovery solution
M: expected number of read elements to repair s stripes
Output: $\mathbb{F}_{opt}$: recovery solution found by SIOR

1 Set $\mathcal{L} \leftarrow \emptyset$, $\mathbb{R} \leftarrow \mathbb{R}_{ini}$, $q(\mathbb{F}_{opt}) \leftarrow \infty$, $\mathcal{A} \leftarrow \emptyset$
2 **for** *each iteration* **do**
3 **for** *each lost element* x **do**
4 **for** *each candidate parity chain* C_x **do**
5 $\mathbb{R}' \leftarrow \mathbb{R} - S_x + C_x$
6 **if** $\mathbb{R}'$ *is not valid* **then**
7 Reuse Initial Solution Selection to optimize it
8 Get the element distribution $\mathbb{D}'$
9 Run Filling Intervals to $\mathbb{D}'$ and obtain $\mathbb{F}'$
10 Calculate the caused disk seeks $q(\mathbb{F}')$
11 **if** $q(\mathbb{F}') \notin \mathcal{L}$ **then**
12 Record $\{\mathbb{R}', \mathbb{F}'\}$ in $\mathcal{A}$
13 Select $\{\mathbb{R}_{min}, \mathbb{F}_{min}\}$ from $\mathcal{A}$
14 ▷ record the near-optimal scheme
15 **if** $q(\mathbb{F}_{min}) \leq q(\mathbb{F}_{opt})$ **then**
16 $q(\mathbb{F}_{opt}) \leftarrow q(\mathbb{F}_{min})$
17 $\mathbb{F}_{opt} \leftarrow \mathbb{F}_{min}$
18 ▷ update the Tabu list
19 **if** $\mathcal{L}$ *is full* **then**
20 Evict the oldest value out of $\mathcal{L}$
21 Append $q(\mathbb{F}_{min})$ to $\mathcal{L}$
22 Update $\mathbb{R} \leftarrow \mathbb{R}_{min}$, set $\mathcal{A} \leftarrow \emptyset$
23 Return $\mathbb{F}_{opt}$
24 **Procedure** Filling Intervals ()
25 Calculate $e(\mathbb{D}')$ and set $\Delta \leftarrow M - e(\mathbb{D}')$
26 Fill Δ elements to intervals from smallest to largest in size, and obtain $\mathbb{F}'$
28 **return** $\mathbb{F}'$;

the order of smallest to largest in size, [3] *can decrease the maximum number of seek operations.*

Proof: As discussed above, filling an interval can reduce a seek operation, therefore the more filled intervals will cause the more reductions on disk seeks. When given a constant number of available elements for filling, the method that fills the intervals in the order of smallest to largest in size can decrease the maximum number of intervals, and thus introduce the most reduction of seek operations. ■

3) ***The maintenance of a Tabu list***. Algorithm 2 always maintains a Tabu list $\mathcal{L}$ throughout the search, which keeps the number of seek operations of the selected solutions in recent iterations. In next iterations, Algorithm 2 will ignore the solutions that need the same disk seeks recorded in $\mathcal{L}$, and choose the one with the least disk seeks among remaining candidates. This design can effectively avoid the search process sticking to a suboptimal solution.

Algorithm Details: In Algorithm 2, we first initialize a Tabu list $\mathcal{L}$ as an empty set and set the current optimal number of disk seeks $q(\mathbb{F}_{opt})$ as infinity (step 1). In a new iteration, given the current recovery solution $\mathbb{R}$, for each lost element x, we replace the current selected parity chain S_x with each candidate parity chain C_x, and construct another recovery solution $\mathbb{R}'$ (step 2∼5). If $\mathbb{R}'$ retrieves more than M elements, then it will be optimized to be a valid one by reusing Algorithm 1 (step 6∼7). Given $\mathbb{R}'$, we can have the distribution of retrieved elements, which is denoted as $\mathbb{D}'$ (step 8). We perform the procedure of Filling Intervals to $\mathbb{D}'$, get the element distribution $\mathbb{F}'$ after filling procedure, and calculate the number of seek operations $q(\mathbb{F}')$ (step 9∼10). If $q(\mathbb{F}')$ is not kept in the Tabu list, then we record the tuple $\{\mathbb{R}', \mathbb{F}'\}$ in the candidate set $\mathcal{A}$ (step 11∼12).

After testing every candidate parity chain for each lost element, we select the tuple $\{\mathbb{R}_{min}, \mathbb{F}_{min}\}$ from $\mathcal{A}$, where $\mathbb{F}_{min}$ has the fewest disk seeks among the solutions in $\mathcal{A}$ (step 13). We compare $q(\mathbb{F}_{min})$ with the optimal number of seek requests $q(\mathbb{F}_{opt})$ currently found, and record $\mathbb{F}_{min}$ and $q(\mathbb{F}_{min})$ if $\mathbb{F}_{min}$ brings fewer seek operations than $\mathbb{F}_{opt}$ (step 15∼17). Finally, we append $q(\mathbb{F}_{min})$ in the Tabu list $\mathcal{L}$ (step 21) and select $\mathbb{R}_{min}$ for the next iteration (step 22). $q(\mathbb{F}_{opt})$ will be iteratively reduced with Algorithm 2 proceeds.

For the procedure of Filling Intervals, given the element distribution $\mathbb{D}'$, we first calculate the number of read elements $e(\mathbb{D}')$ and further get the number of unrequested elements Δ that can be read (step 25). To fill the maximum number of intervals according to Theorem 1, we sort the intervals and fill the intervals from smallest to largest in size (step 26). The procedure finally returns the filled distribution $\mathbb{F}'$ (step 28).

An Example: The current recovery solution in Figure 4(c) is $\mathbb{R} := (\mathcal{R}_1, \mathcal{R}_2)$, where $\mathcal{R}_1 := \{E_{1,5}, E_{2,6}, E_{3,6}, E_{4,5}\}$ and $\mathcal{R}_2 := \{E_{6,5}, E_{6,4}, E_{8,5}, E_{8,4}\}$. This solution requires 24 elements and causes 17 disk seeks. The element distribution of $\mathbb{R}$ is shown in Figure 5(a).

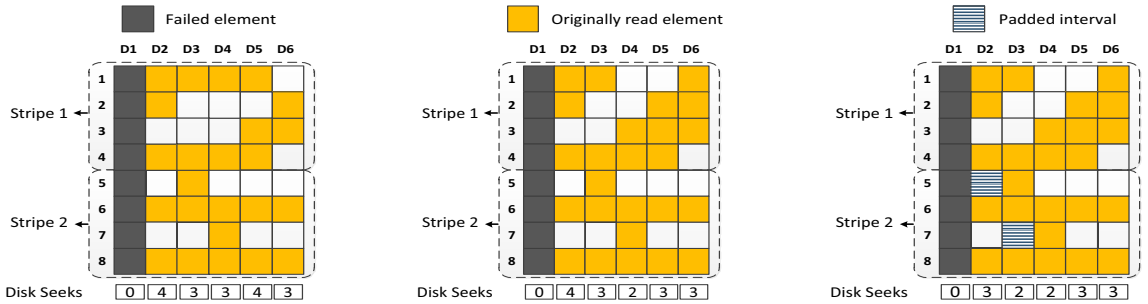
We set $M = 27$ and assume $\mathcal{L} = \{14\}$ which is updated in previous iterations. The current parity chain for $E_{1,1}$ (i.e., element x in Algorithm 2) in $\mathbb{R}$ is the horizontal parity chain led by $E_{1,5}$ (i.e., S_x). We replace it with another parity chain that $E_{1,1}$ involves, i.e., the diagonal parity chain led by $E_{1,6}$ (i.e., C_x), and construct a candidate solution $\mathbb{R}' = (\mathcal{R}'_1, \mathcal{R}_2)$, where $\mathcal{R}'_1 := \{E_{1,6}, E_{2,6}, E_{3,6}, E_{4,5}\}$.

We can obtain a new element distribution $\mathbb{D}'$ as shown in Figure 5(b). $\mathbb{D}'$ is a valid distribution because it requires 25 elements ($\leq M = 27$). We then calculate the number of filling elements (i.e., $\Delta = 2$), perform the filling procedure (i.e., read unrequested data elements $E_{5,2}$ and $E_{7,3}$), and obtain the element distribution after filling (i.e., $\mathbb{F}'$) as shown in Figure 5(c). The number of disk seeks in $\mathbb{F}'$ (i.e., $q(\mathbb{F}')$) is 13. Since $13 \notin \mathcal{L}$, then we record the tuples $\{\mathbb{R}', \mathbb{F}'\}$ in $\mathcal{A}$.

After testing all the possible candidate parity chains for every lost element $\{E_{1,1}, E_{2,1}, \cdots, E_{8,1}\}$, we then perform the greedy selection in step 13∼22.

Complexity Analysis: We will respectively analyze the complexity of Algorithm 1 and Algorithm 2. The descriptions of symbols can be reviewed in Table I and Table II.

[3]The size of an interval is denoted by the number of elements it includes.



(a) Element Distribution Before Replacement. It needs 24 elements and causes 17 seek operations.

(b) Element Distribution After Replacement. It needs 25 elements and causes 15 seek operations.

(c) Element Distribution After Filling. It needs 27 elements and causes 13 seek operations.

Fig. 5. An example for a replacement in Algorithm 2

Since the complexity of Zpacr is $O(mw^3)$ [25] and Algorithm 1 will invoke it for at most s times, the complexity of Algorithm 1 is $O(smw^3)$.

Before giving the complexity of Algorithm 2, we first analyze the complexity of Filling Intervals Procedure. The total number of intervals is no more than $(n-1)sw$, where $(n-1)$ is the number of surviving disks and $(n-1)sw$ is the total number of surviving elements. Therefore, the sorting complexity is $O(nsw\log(nsw))$ when employing the sorting algorithms, such as Quicksort [8]. The filling operation (step 26 in Algorithm 2) scans at most $(n-1)sw$ elements. Thus the complexity of Filling Intervals Procedure is $O(nsw\log(nsw))$.

In Algorithm 2, for each parity replacement, Algorithm 2 may reuse Algorithm 1 and invoke the filling procedure. Since our simplified recovery model only considers the parity chains that generate parity elements, there are at most mw parity chains in a stripe. Each iteration will try at most smw replacements for s stripes, so the total number of replacements after t iterations is at most $tsmw$. Based on the above analysis, in Algorithm 2, the complexity of filling operations is $O(tnms^2w^2\log(nsw))$, while the complexity of reusing Algorithm 1 is $O(ts^2m^2w^4)$. Therefore, the complexity of Algorithm 2 is $O(ts^2m^2w^4)+O(tnms^2w^2\log(nsw))$.

V. Performance Evaluation

We conduct a series of intensive tests to evaluate the performance of SIOR. We choose the stripe-level greedy algorithm Zpacr [25] as the reference, since it also works for any XOR-based erasure code and only optimizes the number of read elements without considering the optimization on disk seeks. Therefore, the comparison can fairly represent the advantage of SIOR. We select four typical coding schemes, i.e., RDP Code (over $p+1$ disks, where p is a prime number), X-Code (over p disks), STAR Code (over $p+3$ disks) and CRS Code. The features of these four coding schemes are shown in Table III, where n is the number of disks in a stripe and m is the tolerated number of disk failures.

Evaluation Environment: We choose n from 5 to 15. This range covers typical system configurations of many well-known storage systems [1], [10]. The test is run on a Linux server with a X5472 processor and 8GB memory. The operating system is SUSE Linux Enterprise Server and the filesystem is EXT3. The deployed disk array consists of 15 Seagate/Savvio 10K.3 SAS disks, each of which has 300GB storage capability and 10,000 rmp. We organize the disks in the JBOD (just a bunch of disks) mode and each disk is independently handled as a node. The machine and the disk array are connected by a Fiber cable with the bandwidth of 800MB/sec. The codes are realized by Jerasure 1.2 [14], a widely-used library to realize erasure coding storage systems.

TABLE III
Four Representative Coding Schemes (p is a prime number)

Coding Scheme	n	k	m
RDP Code	$p+1$	$p-1$	2
X-Code	p	$p-2$	2
STAR Code	$p+3$	p	3
CRS Code	general pairs of (k,m)		

Evaluation Metrics: Suppose the storage system consists of n disks $\{D_1,\cdots,D_n\}$. When D_i fails, a recovery solution causes o_j disk seeks to D_j $(1\le j\ne i\le n)$, and takes the time T_j to retrieve the required elements from D_j. We concern the following metrics:

1) ***Seek load***. This metric denotes the averaged number of seek operations that a surviving disk serves during the reconstruction and is defined as follows:

$$\text{seek load} = \frac{\sum_{1\le j\ne i\le n} o_j}{n-1} \tag{3}$$

2) ***Recovery speed***. According to the principle of parallel technology, the recovery time is determined by the disk that takes the longest time. Therefore, the recovery time is $Max\{T_j|1\le j\ne i\le n\}$. The recovery speed can then be defined as follows:

$$\text{recovery speed} = \frac{\text{size of all the lost elements}}{\text{recovery time}} \tag{4}$$

Evaluation Method: For each coding scheme, we generate the data elements and perform the encoding by producing

parity elements. These elements are then dispersed according to the layout of that code and the stripe rotation principle. The element size is set as 4KB and the stripe number is set as 100. We then destroy the elements on a random selected disk, and trigger SIOR to generate the recovery solution over these 100 stripes. We also employ Zpacr [25] to compute the recovery solution for each stripe. Finally, we repair the lost elements based on the solutions found by SIOR and Zpacr respectively.

A. Impact of Number of Iterations

In this experiment, we evaluate the performance on seek load and recovery speed when the number of iterations increases. We select $p = 11$, so that the number of disks n constructed over RDP Code, X-Code and STAR Code will be 12, 11 and 13, respectively. For CRS Code, we choose the parameter $(k = 8, m = 4)$, so that the number of disks $n = k+m = 12$. Like in Zpacr [25], We also select $w = 6$ for CRS Code. For each code, SIOR reads 5% more elements than Zpacr. To clearly illustrate the gap of recovery speed between SIOR and Zpacr, we normalize the recovery speed of Zpacr as 1. The test results are shown in Figure 6 and Figure 7.

Seek Load: Figure 6 demonstrates that SIOR can significantly reduce the seek load for various kinds of codes, when compared to Zpacr [25]. For RDP Code, SIOR cuts down up to 65.1% of seek operations for each disk during the reconstruction. For X-Code, SIOR decreases up to 44.2% of seek operations. For STAR Code and CRS Code, SIOR removes up to 64.7% and 52.4% of seek operations, respectively.

Moreover, with the increasing of iterations, the number of optimized seek operations first sharply decreases and then becomes stable. This is because the gained reduction on seek operations in each iteration generally becomes smaller when the sought solution is closer to the optimal one.

Recovery Speed: Figure 7 shows SIOR greatly improves the recovery speed. For RDP Code, the recovery speed of SIOR is 144.9% faster than that of Zpacr. For CRS Code and X-Code, SIOR improves the recovery speed by up to 97.0% and 33.0% respectively compared to Zpacr [25].

B. Scalability

In this test, we evaluate the scalability of SIOR in terms of seek load and the recovery speed when the system scale expands. The number of iterations is set as 400. In the evaluation of each code, SIOR reads 5% more elements than Zpacr. To clearly illustrate the gap of recovery speed between SIOR and Zpacr, we normalize the recovery speed of Zpacr as 1. Results are shown in Figure 8 and Figure 9.

Seek Load: Figure 8 indicates that SIOR keeps its advantage to optimize disk seeks under different scales of the storage systems. Take RDP Code as an example, SIOR decreases about 31.8% of seek operations loaded on each disk when $p = 5$, and this reduction increases to 62.8% when $p = 13$.

Moreover, SIOR widens the benefit on disk seeks reduction when the scale of the storage system expands. Take STAR Code as an example, the saving of seek operations brought by SIOR increases from 37.7% ($p = 5$) to 64.7% ($p = 11$).

TABLE IV
THE ACCURACY AND EFFICIENCY OF SIOR

Num. of Stripe	Accuracy	Time (SIOR)	Time (Enumeration)
RDP Code ($p = 11$)			
1	1.00	0.04 sec	0.04 sec
2	1.00	0.12 sec	7.52 sec
3	1.00	0.21 sec	3h 17min 30sec
X-Code ($p = 11$)			
1	1.05	0.03 sec	0.04 sec
2	1.08	0.15 sec	26.04 sec
3	1.02	0.27 sec	20h 4min 8sec
STAR Code ($p = 11$)			
1	1.00	0.21 sec	0.22 sec
2	1.09	0.29 sec	7h 1min 10sec

Recovery Speed: Figure 9 confirms that SIOR keeps its capability for recovery speedup under different system scales, as SIOR still behaves well to reduce the seek operations when the system scale expands. For example, for RDP Code, SIOR accelerates the data recovery by 59.6% when $p = 5$ and this improvement reaches 186.8% when $p = 13$.

C. Accuracy and Efficiency

We then evaluate the accuracy and efficiency of SIOR. Accuracy is used to denote the closeness between the optimal solutions sought by SIOR and the Enumeration respectively.

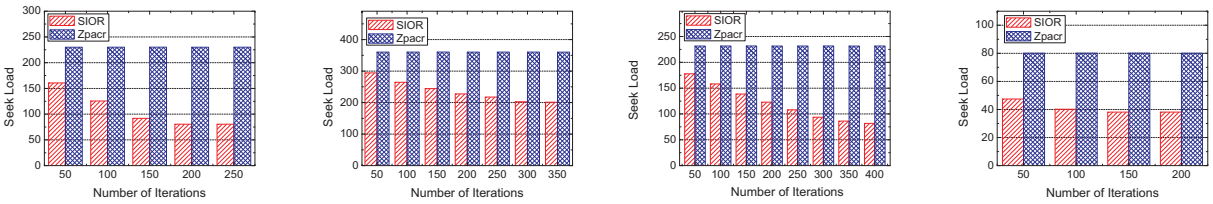
$$\text{accuracy} = \frac{\text{the least disk seeks in SIOR}}{\text{the least disk seeks in Enumeration}} \quad (5)$$

We select RDP Code ($p = 11$), X-Code ($p = 11$), and STAR Code ($p = 11$), and vary the number of stripes. In the evaluation of each code, both SIOR and Enumeration method read 5% more elements than Zpacr. We run SIOR and Enumeration method respectively, calculate the search accuracy, and record the search latency in Table IV.

Table IV indicates that SIOR is accurate and efficient when compared with Enumeration. SIOR obtains the solution with the same minimum number of disk seeks compared to Enumeration in RDP Code, and causes no more than 8% extra seek operations when compared with Enumeration in X-Code. On the aspect of efficiency, SIOR greatly decreases the search latency. For example, SIOR only needs 0.21 seconds to find the solution with the least disk seeks for RDP code when the number of stripes is 3. On the contrary, the search time of Enumeration exponentially enlarges when the number of stripes increases. For example, for RDP Code, Enumeration requires more than 3 hours to find the solution with minimum seek operations when there are only 3 stripes.

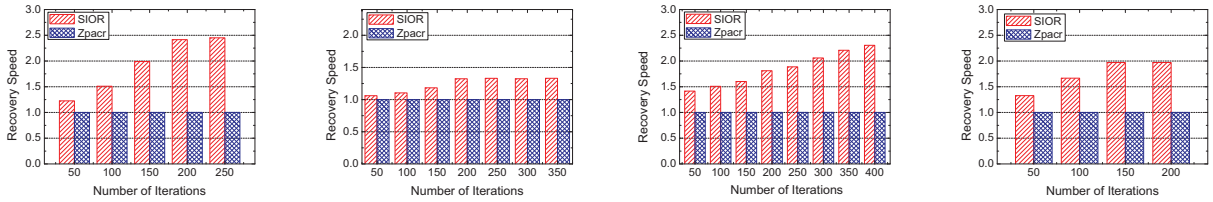
D. Data Reduction

We also test the capability of SIOR to reduce the mount of read data for recovery. We select $p = 11$ for RDP Code, X-Code, and STAR Code. We also use CRS Code with the parameters $(k = 8, m = 4, w = 6)$. In the evaluation of each code, SIOR reads 5% more elements than Zpacr. The comparison results among Zpacr, SIOR, and the conventional method (i.e., without hybrid parity chains) are normalized in Table V. Table V indicates that both Zpacr and SIOR



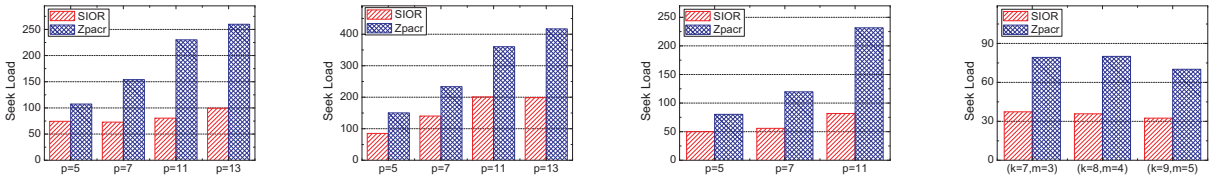
(a) RDP Code, $p = 11$ (b) X-Code, $p = 11$ (c) STAR Code, $p = 11$ (d) CRS Code ($k = 8, m = 4$)

Fig. 6. The seek load under different numbers of iterations. The smaller value means the lighter load on disks.



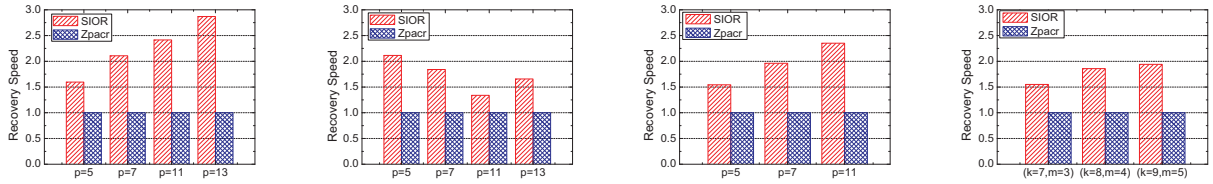
(a) RDP Code, $p = 11$ (b) X-Code, $p = 11$ (c) STAR Code, $p = 11$ (d) CRS Code ($k = 8, m = 4$)

Fig. 7. The recovery speed under different numbers of iterations. The larger value means the faster recovery.



(a) RDP Code (b) X-Code (c) STAR Code (d) CRS Code

Fig. 8. The seek load under different number of disks. The smaller value means the lighter load on disks.



(a) RDP Code (b) X-Code (c) STAR Code (d) CRS Code

Fig. 9. The recovery speed under different number of disks. The larger value means the faster recovery.

TABLE V
COMPARISON ON DATA REDUCTION

Codes	Zpacr	SIOR	Conventional Method
RDP Code	0.77	0.81	1.00
X-Code	0.72	0.76	1.00
STAR Code	0.72	0.76	1.00
CRS Code	0.87	0.91	1.00

impressively decrease the amount of read data compared to the conventional method. For example, SIOR reduces 19% unnecessary retrieved elements compared with the conventional method. Meanwhile, SIOR only retrieves 5% more elements compared with Zpacr, and this ratio can also be further adjusted according to the administrator's requirements.

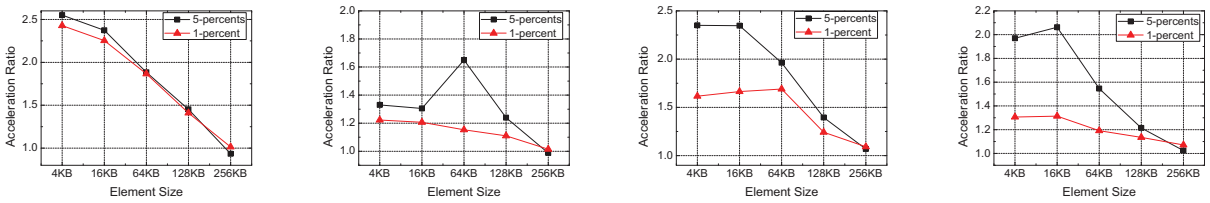
E. Impact of Element Size

To investigate how the element size affects the benefit of SIOR, we compare SIOR with Zpacr in terms of recovery speed under different element sizes. We first define the concept of **acceleration ratio** in Equation (6).

$$\text{acceleration ratio} = \frac{\text{recovery speed of SIOR}}{\text{recovery speed of Zpacr}} \tag{6}$$

Obviously, when the acceleration ratio is larger, SIOR can achieve a faster data reconstruction when compared with Zpacr [25]. With respect to the selection of M, we mainly consider two cases for each code, i.e., $M_1 = (1+1\%) \times C_{min}$ and $M_5 = (1+5\%) \times C_{min}$, where C_{min} is the least number of required elements found by Zpacr [25]. We then vary the element size from 4KB to 256KB, and calculate the acceleration ratio under these two selections of M (i.e., $M = M_1$ and $M = M_5$). The results are presented in Figure 10.

First, the advantage of SIOR will eliminate when the element size increases. For example, when the element size is 4KB and $M = (1 + 5\%) \times C_{min}$, the acceleration ratio of RDP Code is 2.6, meaning that the recovery speed of SIOR is 2.6 times faster than that of Zpacr. When the element size



(a) RDP Code (b) X-Code (c) STAR Code (d) CRS Code

Fig. 10. The acceleration rate under different element sizes.

increases to 256KB, SIOR reaches the same recovery speed with Zpacr. This test also suggests that SIOR is more suitable to be deployed in the environment with a small element size (i.e., smaller than 256KB). This is because the influence of seek time declines when the size of I/O unit expands.

Second, the number of read elements also affects the performance of SIOR. When the element size is small, selecting $M = M_5$ gains more performance improvement, compared with the chosen of $M = M_1$. This is because when $M = M_5$, SIOR has a larger search space to find the solution with fewer disk seeks. However, when the element size increases, the effect of disk seek reduction becomes insignificant and the increasing size of retrieved elements slows down the reconstruction in reverse. Thus the gained benefit when $M = M_5$ will reduce, compared to the case when $M = M_1$.

F. Summary

These tests show that both Zpacr and SIOR are suitable for storage systems that are easily restricted by the repair traffic, such as [9] and [20]. Compared with Zpacr, SIOR not only sustains the advantage of data reduction, but also decreases the disk seeks to reach the faster data reconstruction.

VI. Conclusion

In this paper, we propose a greedy algorithm called SIOR based on Tabu search to optimize both the number of disk seeks and the amount of read data for recovery. The algorithm includes two stages. The first stage makes use of an existing algorithm that optimizes the number of retrieved elements for each stripe, and gets an initial solution. The second stage optimizes the initial solution and iteratively approaches the optimal recovery solution. Finally, the evaluation indicates that SIOR reduces 31.8%∼65.1% of disk seeks and improves the recovery speed by up to 186.8% during the recovery.

References

[1] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling. Hdfs raid. In *Hadoop User Group Meeting*, 2010.

[2] Y. Cassuto and J. Bruck. Cyclic lowest density mds array codes. *Information Theory, IEEE Transactions on*, 55(4):1721–1729, 2009.

[3] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of USENIX FAST*, 2004.

[4] EXT3. http://www.en.wikipedia.org/wiki/ext3.

[5] Y. Fu, J. Shu, and X. Luo. A stack-based single disk failure recovery scheme for erasure coded storage systems. In *Proc. of IEEE SRDS*, 2014.

[6] F. Glover and M. Laguna. *Tabu search*. Springer, 1999.

[7] J. Gray, B. Horst, and M. Walker. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. In *Proc. of VLDB*, 1990.

[8] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[9] 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 *USENIX FAST*, 2012.

[10] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX ATC*, 2012.

[11] 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 *Proc. of USENIX FAST*, 2012.

[12] X. Luo and J. Shu. Load-balanced recovery schemes for single-disk failure in storage systems with any erasure code. In *Proc. of IEEE ICPP*, 2013.

[13] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *In Proc. of USENIX FAST*, 2009.

[14] J. S. Plank, S. Simmerman, and C. 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.

[15] K. 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 *Proc. of ACM SIGCOMM*. ACM, 2014.

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

[17] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[18] B. Schroeder and G. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *Proc. of USENIX FAST*, 2007.

[19] Z. Shen and J. Shu. Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems. In *Proc. of IEEE/IFIP DSN*, 2014.

[20] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.

[21] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie. Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6. In *Proc. of IEEE/IFIP DSN*, 2011.

[22] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *Proc. of ACM SIGMETRICS*, 2010.

[23] L. Xu and J. Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[24] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui. Single disk failure recovery for x-code-based parallel storage systems. *IEEE Trans. on Computers*, 2013.

[25] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu. On the speedup of single-disk failure recovery in xor-coded storage systems: Theory and practice. In *Proc. of IEEE MSST*, 2012.

[26] Y. Zhu, P. P. Lee, L. Xiang, Y. Xu, and L. Gao. A cost-based heterogeneous recovery scheme for distributed storage systems with raid-6 codes. In *Proc. of IEEE/IFIP DSN*, 2012.