PITR: An Efficient Single-failure Recovery Scheme for PIT-Coded Cloud Storage Systems

Peng Li, Jiaxiang Dong, Xueda Liu, Gang Wang, Zhongwei Li, Xiaoguang Liu
Nankai-Baidu Joint Lab, College of Computer and Control Engineering
Nankai University, Tianjin, China
Email: {lipeng, dongjx, liuxd, wgzwp, lizhongwei, liuxg}@nbjl.nankai.edu.cn

***Abstract*—Storage systems are transitioning to the use of erasure coding instead of replication, for its offering interesting trade-offs between efficiency, reliability, and storage overhead. However, it results in high read latency and long recovery time when corrupted blocks are erasure coded. In this paper, we design a parity independent array codes (PIT), a variation of STAR code, which is triple fault tolerant and nearly space-optimal, and also propose an efficient single-failure recovery scheme (PITR) for them to mitigate the problem. PITR aims to reduce the amount of disk reads and network transfers to reconstruct the unavailable block when single failure happens. In addition, we present a "shortened" version of PIT (SPIT) to further reduce the recovery cost. In this way, less disk I/O and network resources are used, thereby reducing the recovery time and resulting in a high system reliability and availability. Specifically, experimental results show a factor of as much as 40.0% and roughly more than 34.5% reduction of disk and network traffic over the naive recovery scheme on all cases. SPIT codes further achieve a factor of 7.1% performance improvement in average over PIT codes.**

Paper Type: Regular Paper

I. Introduction

In recent years, with the size of modern storage systems growing to an unprecedented scale, drive failures have become more of the norm rather than the exception [8], [29]. To ensure data safety, traditional cloud storage systems [8], [33] employ 3-way replication, and thus have a very high extra overhead of 2 (or 200%). Erasure coding outperforms replication in storage efficiency, but has to suffer high recovery cost for unavailable data.

An erasure code is defined by two parameters, k and m, where k data blocks are used to determine m additional parity blocks so that any k-subset of the $k+m$ blocks are sufficient to recover original k data blocks. The $k+m$ blocks consist of a *code stripe* and are usually distributed among different nodes for failure isolation in cloud storage systems. We use $\mathrm{EC}(k,m)$ to represent an erasure code with parameters k and m. We define the *overhead* as m/k (typically around 0.5), which we want to minimize to reduce storage costs.

We define a *corrupted* block to be any block which is unavailable, possibly due to a node or drive failure, a checksum error, or something else. When failure occurs, a request for the corrupted data block will require fetching k blocks (belonging to the same code stripe as the corrupted block) from multiple remote nodes to reconstruct the missing data, called a *degraded read*. This read requires large amounts of network traffic and disk I/O, and thus negatively impacts the system's availability significantly.

In addition, when failure occurs, the system is said to be in *degraded mode* and a resource-consuming *reconstruction* (or *recovery*) process is triggered to recover the corrupted data on the failed drive or node, which will be stored on other available drives. Modern systems often conduct online failure recovery where the reconstruction process will compete for system resources with user requests. Moreover, in degraded mode the system is vulnerable to additional, simultaneous failures which may lead to data loss and reduce the system's reliability.

Reducing the amount of data required for reconstruction will save much network traffic and disk I/O, therefore reducing the recovery costs when failure occurs. This improves both system's availability and reliability, motivating the study of methods in this paper which mitigate these disadvantages.

Though modern cloud storage systems [1], [3], [15] employ erasure codes which tolerate multiple simultaneous failures, single node or disk failure (or single-failure in short) occurs far more frequently than simultaneous failures. Particularly, single-failures occupy 99.75% of recovery cases in real deployment [30]. Thus, we seek codes which are both (a) tolerant to multiple failures, and (b) efficient at reconstruction in the case of a single-failure.

Let M denote the size in bits of whole data object. It has been theoretically proved in [6] that communicating (and reading) less than M bits are sufficient to repair a single erasure, compared to exact M bits using naive repair. Many solutions have been proposed to reduce the amount of data during single failure recovery, and can be categorized as follows:

- Construction of new erasure codes, such as Hitchhiker [26], LRCs [11], [28] and Hv code [31], which use less network bandwidth and disk I/O than existing codes.
- Optimizations for specific codes, such as RDP [37], EVENODD, X-code and STAR [34], which prove a lower bound of the amount of required data by using multiple parities for recovery.

We continue the research of reducing the amount of data reads during single failure recovery for STAR code in this paper. Though the range $(\frac{2M}{3}, \frac{13M}{18})$ of required data for single-failure recovery in STAR code has been derived in [16], [34], [39], we figure out that it is a *two-stage* recovery process and works not so efficiently as this range shows (see Section II for details). Thus, we propose a variated construction of STAR

code to eliminate this inefficiency. In addition, we detailedly analyze the effects that *code shortening* has on our modified code. Shortening of a code is first mentioned in [4], [5], [34], but they either serve a different purpose or do not consider the effects elaborately.

In this paper, we design a kind of parity-independent array code with triple parities, *PIT code*, which is based on STAR code and is nearly space-optimal. While it is tolerant to 3 erasures, we focus on improving single data block recovery in a PIT code stripe and present an efficient single recovery scheme, *PITR*, with the aim of reducing data traffic during reconstruction. In addition, we propose a "shortened" version of PIT code, *SPIT*, and deploy PITR into SPIT to further reduce the recovery costs of single-failure scenario.

PITR is different from previous work in several ways:

- The two-stage recovery for STAR code negatively impact the recovery performance, which is not considered in traditional optimizations [16], [34]. PITR overcomes the deficiency by proposing PIT code, a variation of STAR. PIT also brings side benefits such as better encoding and updating performance.
- Code shortening is first applied to a code, aiming to overcome parameter limitation, but in PITR it works to further reduce the amount of data reads for single-failure recovery.
- Though general approaches have been proposed in [14], [39], we propose our optimized approaches specially for PIT code, which works more naturally and time-efficiently.

The remainder of the paper is structed as follows. We motived the PITR in Section II and describe PIT code and PITR recovery strategy in detail in Section III. We illustrate two different implementations of PITR in Section IV-C and shortening strategy in Section V. We evaluate them in Section VI and conclude our paper and future work in Section VIII.

II. Motivation

In this section, we first briefly describe STAR code [12] and illustrate the problem of typical two-stage single-failure recovery scheme for STAR. Then we give intuitions about our improvements.

A. STAR Code

STAR code is one of erasure codes, which tolerates any 3 failures with 3 additional parity blocks and is thus *Maximum Distance Separable* (MDS) [18]. Besides MDS property, the STAR code is computationally efficient in encoding and decoding, where only XOR operations are required.

The STAR encoding takes a $(p-1)\times(p+3)$ array $A=(a_{i,j})$, where p is prime. The columns of A are indexed by $\mathbb{Z}_p \cup \{p, p+1, p+2\}$, where each column corresponds to a *block*, either data or parity block, which split into (data or parity) *units*. Here $\mathbb{Z}_x$ denotes $\{0,1,\ldots,x-1\}$. Unit $a_{i,j}$ $(0 \le i \le p-2, 0 \le j \le p+2)$ represents unit i in column j. STAR code is one popular two-dimensional array code, where failures mean column erasures, and STAR code can tolerate any 3 columns erasure.

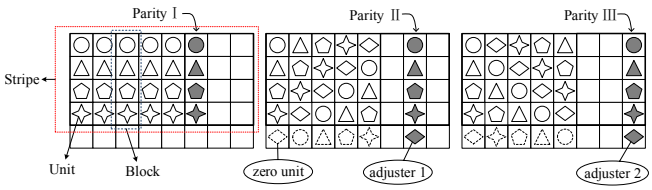

Fig. 1: The STAR encoding with $p=5$.

In STAR, the first p columns are data blocks and the last three columns are parity blocks. The parity units in column p are generated by XORing the data units in the same row while the parity units in columns $p+1$ and $p+2$ are computed differently and are described as follows: (1) An imaginary row $p-1$ with *zero* values is added to this array. (2) We compute the corresponding parity units in columns $p+1$ and $p+2$ by XORing the data units along the same diagonal with slop 1 and -1 respectively, which are marked with different shapes in Fig.1. (3) Units $a_{p-1,p+1}$ and $a_{p-1,p+2}$ become non-zero and are called *adjusters*. Then *adjuster complementary* is performed to eliminate the adjusters by adding (XOR addition) them to all the units in column p and $p+1$ respectively. Refer to [12] for more details.

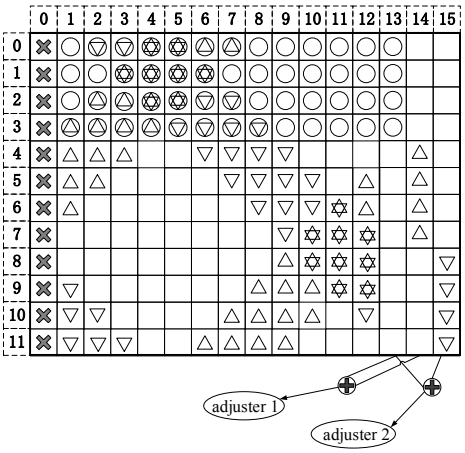

Fig. 2: Two-stage recovery with $p=13$ and block 0 failed for STAR. The circles indicate that the row units are used to perform recovery; the upward-triangles and downward-triangles indicate that the diagonal units with slop 1 and -1 are used to perform recovery respectively.

B. Two-stage Single Failure Recovery for STAR

When single failure occurs, the units in the corrupted column can be recovered by row or diagonal parities. Wang et al. [34] proposed an optimized single failure recovery scheme for STAR code, which is shown in Fig.2. It divides the corrupted units into three parts averagely and recovers them by utilizing parity units in columns p, $p+1$ and $p+2$ respectively.

Once diagonal parities are involved in the recovery, as in Fig.2, the decoding process will be divided into two stages: (1) calculating two adjusters by XORing column p with column $p+1$ and $p+2$ respectively, (2) fetching the required units and performing decoding operations. That's why we call it two-stage recovery. This proposed scheme gives an upper bound $(\frac{13}{18}p^2+\frac{17}{19}p-\frac{47}{18})\frac{M}{p(p-1)} \approx \frac{13M}{18}$ for any single data block recovery.

However, a question remains unanswered when deploying STAR into cloud storage systems: does the upper bound $\frac{13M}{18}$ work efficiently as it shows? Our answer is "no". The additional stage 1 requires $3(p-1)$ disk unit reads, $3(p-2)$ unit XOR operations and 3 unit transfers, which increases the complexity for recovery and thus negatively impacts the performance. With this reason, we propose a simple code based on STAR, called PIT code, to address this problem by storing the two adjusters rather calculating them, which will be presented detailedly in Section III.

C. Code Shortening

The shortening of a code is first mentioned in [4], [5], aiming to break the limitation of prime parameter. Then in [34], a "shortened" version of $(4,2)$ EVENODD [4] code with $p=3$ is presented, which follows a different aim and makes it possible to transfer 3 units to repair the single failure. EVENODD code uses the exact same encoding rules as STAR for the first two parity columns, and does not have the third parity column. Both previous works motivate the study of code shortening.

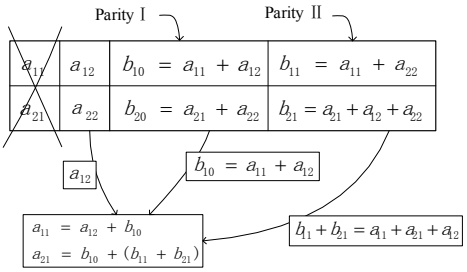

Fig. 3: Repair of a $(4,2)$ EVENODD code with $p=3$ if the first column is erased, where three units are transmitted [34].

We take an example of repairing column 0 for a shortened version of EVENODD code (i.e., column 2 is shortened) from [34], as is illustrated in Fig.3. One difference is that it requires linear combinations before transferring the units (i.e. $b_{11}+b_{21}$), which is called *regenerating*. The behind principle about regenerating has been theoretically analyzed and proved in [6], which is out of the scope of this paper. In this paper, we mainly study how code shortening impacts the recovery performance and whether we can find a better shortened version of PIT code.

III. PIT Code

The PIT code consists of p data blocks and 3 parity blocks as STAR code, and can be described by a $p \times (p+3)$ array $A=(a_{i,j})$, where $0 \le i \le p-1, 0 \le j \le p+2$. We exclude the units $a_{p-1,j}$ for $j \in \mathbb{Z}_p$. We use $\mathrm{PIT}(p)$ to represent PIT code with prime p.

A. PIT Code Encoding

The PIT code uses the same encoding rules of STAR code for the first parity column (i.e. column p). The units in columns $p+1$ and $p+2$ are computed very similar to STAR code except that adjuster complementary is removed (refer to Section II-A for STAR encoding).

In short, PIT code stores the two adjusters $a_{p-1,p+1}$ and $a_{p-1,p+2}$ in the array, rather than removes them by adjuster complementary. Algebraically, the encoding of three parity columns, indexed by p, $p+2$, and $p+2$, can be represented as for all $i \in \mathbb{Z}_p$:

$$a_{i,p} = \bigoplus_{j\in\mathbb{Z}_p} a_{i,j}, \quad \text{provided } i \neq p-1, \tag{1}$$

$$a_{i,p+1} = \bigoplus_{j\in\mathbb{Z}_p} a_{i-j,j}, \text{ provided } i-j \neq p-1, \tag{2}$$

$$a_{i,p+2} = \bigoplus_{j\in\mathbb{Z}_p} a_{i+j,j}, \text{ provided } i+j \neq p-1, \tag{3}$$

Giving the $3p-1$ linear equations, these are illustrated in Fig. 4.

B. PIT Code Decoding

PIT code is different from STAR code in column $p+1$ and $p+2$. It can be easily converted to STAR by performing adjuster complementary. Therefore, the decoding algorithm for STAR code is suitable for PIT code, i.e., when columns failure occurs, we first transform the columns $p+1$ and $p+2$ (if available) to that of STAR code and then perform STAR decoding. This also acts as a simple proof that PIT code can tolerate any 3 erasures as STAR code.

In fact, when digging into the decoding process of STAR for two or three erasures, the adjuster(s) is to be calculated first in most of erasure cases except column p is an erasure. For single erasure, any data unit $a_{i,j}$ can be recovered in three distinct ways (using one of (1), (2), or (3)) by XORing $p-1$ other units, where the optimized algorithm will be illustrated in Section IV.

It should be noted that the method proposed in [34] for STAR needs to calculate two adjusters. The overhead is presented in Section II-B, which is eliminated in PIT code owing to the storing of two adjusters. Thus, PIT code will accelerate the decoding procedure.

C. PIT Code Updating

The updating of PIT code is the same as STAR, except that we want to update such a unit $a_{i,j}$ that it involves in the adjuster construction, i.e., i and j satisfy $i+j=p-1$ or $i-j=p-1$.

PIT code brings side benefit of better unit-updating performance than STAR. When a data unit $a_{i,j}$ need to be rewritten in PIT, only 4 unit reads and writes (i.e., one original $a_{i,j}$ and 3 corresponding parity units) are enough. But, this is not true in STAR if $a_{i,j}$ is the exact unit involving in the adjuster construction, which requires $p+2$ unit reads and writes.

D. PIT Code in Cloud Storage System

When a file is uploaded to a cloud storage system using $\mathrm{PIT}(p)$, the file is first divided into multiple data blocks of a fixed size (e.g. 64MB) with each data block containing $p-1$ data units. Then for each group of p data blocks, the system computes the 3 parity blocks as per (1), (2), and (3), and the

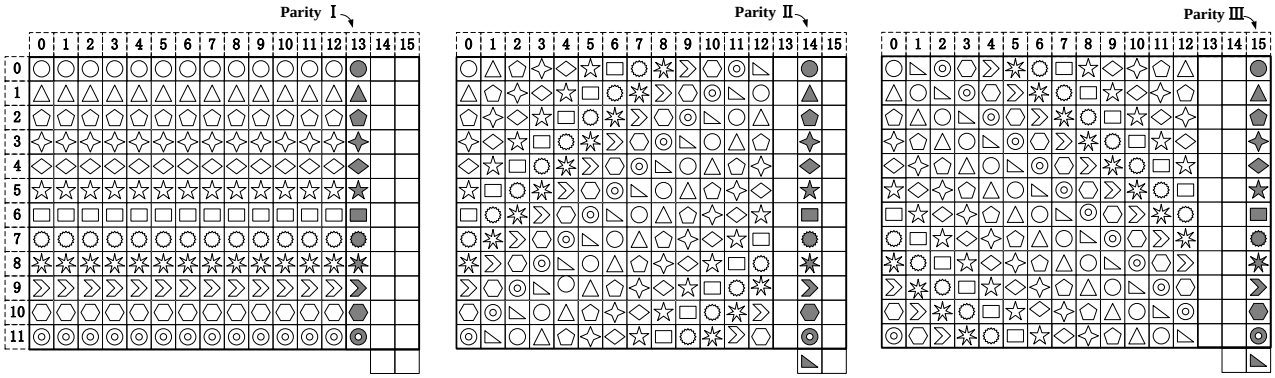

Fig. 4: Illustrating how parity units are determined in $\mathrm{PIT}(13)$. In every matrix, the units marked by the same shape constitute a parity group. The left, middle, and right matrices correspond to equations (1), (2), and (3) respectively.

$p+3$ blocks are randomly distributed over $p+3$ distinct storage nodes. For the group without enough p data blocks, extra *zero blocks* are added into this group, which assists the encoding.

The storage overhead of $\mathrm{PIT}(p)$ is

$$\frac{\text{no. parity units}}{\text{no. data units}} = \frac{3p-1}{p(p-1)} \sim \frac{3}{p},$$

which has asymptotically the same overhead as $\mathrm{MDS}(p,3)$ codes, which has overhead $3/p$.

The *upward-diagonal parity block* (block with indice $p+1$, or block $p+1$ in short) and *downward-diagonal parity block* (block $p+2$) both have p units, which differs from the block size to the other types of blocks, i.e., data blocks (blocks $0,1,\ldots,p-1$) and the *horizontal parity block* (block p) which all have $p-1$ units. While we have differing block sizes, since the blocks are stored randomly distributed, the nodes end up with approximately balanced storage requirements.

IV. Single Failure Recovery in PIT code

When a single-failure occurs, the units in the corrupted data blocks are recovered by XORing the remaining horizontal, upward-diagonal, or downward-diagonal units, which, by design, would have been distributed to other healthy nodes. The units in the available blocks need to be read from their corresponding disks and then sent to a central repair node for recovery, consuming network and disk traffic. Importantly, when recovering a block due to failure, we can reduce the number of reads by choosing to recover its corrupted units in a way that reads available units that reuses the units used to recover other corrupted units.

A. Problem Formulation

In this paper, we aim to address the problem: in a PIT coded storage systems, can we minimize the amount of data traffic for single data block recovery? Though there are many aspects needed to be taken into consideration and comparison (e.g. the system workload, available network bandwidth, disk seeks, and decoding time, etc.) for the recovery performance of single-failure, we focus only on one metric of minimizing the cost of network traffic and disk I/O. Note that if the parity block is corrupted, all M data bits are required to recover this parity block by encoding operation, which is not considered in this paper.

Under some strategy ST for deciding how to recover a corrupted data block q, we define the *recovery array* $c=(c_{i,j})$ as the $p\times(p+3)$ array where $c_{i,j}$ is the number of times unit $a_{i,j}$ will be used in recovering block q under the strategy ST. A recovery array c necessarily has $c_{i,j}=0$ when $j=q$ and when $i=p-1$ and $j\in\mathbb{Z}_p\cup p$. Recovery arrays c indicate the units which are used for to recover block q. We illustrate recovery arrays with crosses indicating erased units, and circles, upward-pointing triangles, and downward-pointing triangles respectively indicating when that unit is used in recovery via equations (1), (2), and (3). Essentially, the more 0's there are in the recovery arrays, the better the strategy.

For $q\in\mathbb{Z}_p$, we use COST_q to denote the number of units required to be read in order to recover the $p-1$ units of data block q. Thus, COST_q can be described by

$$\mathrm{COST}_q := \sum_{i=0}^{p-1}\sum_{j=0}^{p+2} f_{i,j}. \tag{4}$$

where $f_{i,j}$ satisfies

$$f_{i,j} := \begin{cases} 1, & \text{if } c_{i,j}\neq 0,\\ 0, & \text{otherwise.}\end{cases} \tag{5}$$

Thus, under some recovery strategy ST, the average number of units required for any corrupted data block is

$$\mathrm{COST}_{\text{avg}}[\mathrm{ST}] = \mathrm{COST}_{\text{avg}} := \frac{1}{p}\sum_{q=0}^{p-1}\mathrm{COST}_q. \tag{6}$$

We will seek to minimize $\mathrm{COST}_{\text{avg}}$.

It should be noted the seeking to minimize $\mathrm{COST}_{\text{avg}}$ is only an once-off task, meaning that for the given coding parameters and corrupted block, the optimal recovery array is fixed. We can simple compute and store the recovery solutions *in advance* for all the corrupted possibilities, to eliminate this seeking overhead.

B. Saving Ratio Metrics

In a storage system using PIT, to recover a corrupted data block, a simple way is to use the surviving data blocks and the horizontal parity block, which we call *horizontal recovery* strategy (HOR). HOR reads exactly $p(p-1)$ units (i.e., M

bits) to recover from any single block failure, so e.g. with $p = 13$, we have $\mathrm{COST}_q = 156$ for all $q \in \mathbb{Z}_p$. Note only one parity block is involved in the recovery process, which is not efficient.

Previous works [34], [37] have shown that using more parity groups generally reduces the number of units required for single-failure. For $q \in \mathbb{Z}_p$, We define the q-th *saving ratio* $\mathrm{SR}_q = \mathrm{SR}_q[\mathrm{ST}]$ for a strategy ST to be the ratio of number of units involved in recovering block q vs. HOR, i.e.,

$$\mathrm{SR}_q[\mathrm{ST}] := \frac{\mathrm{COST}_q[\mathrm{HOR}] - \mathrm{COST}_q[\mathrm{ST}]}{\mathrm{COST}_q[\mathrm{HOR}]}. \tag{7}$$

For a given strategy ST, we pay more attention on the average single-failure repair performance, so we also define

$$\mathrm{SR}_{\mathrm{ST}} = \frac{\mathrm{COST}_{\mathrm{avg}}[\mathrm{HOR}] - \mathrm{COST}_{\mathrm{avg}}[\mathrm{ST}]}{\mathrm{COST}_{\mathrm{avg}}[\mathrm{HOR}]}. \tag{8}$$

Minimizing (6) is equivalent to maximizing (8).

C. Optimal and Near-optimal Recovery

When a data block q is corrupted, a strategy ST determines how we recover the units $a_{i,q}$, for $i \in \mathbb{Z}_p \setminus \{p-1\}$, using horizontal, upward-diagonal, or downward-diagonal parities. This results in an ordered composition (H, U, D) of $\mathbb{Z}_p \setminus \{p-1\}$, with H, U, and D respectively containing the indices i of the corrupted unit for which horizontal, upward-diagonal, or downward-diagonal parities are used. An ordered composition of (H, U, D) determines one recovery array.

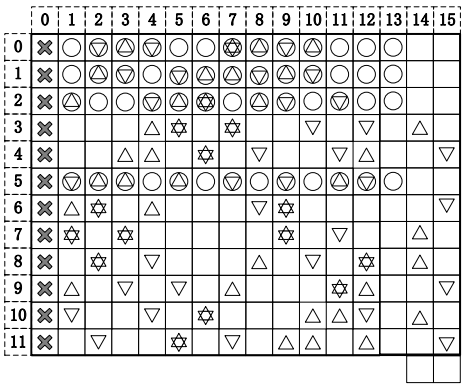

Fig. 5: The PITR recovery array for block 0 in $\mathrm{PIT}(13)$.

We use PIT recovery (PITR) to represent our proposed strategy. A PITR recovery array used to recover block 0 for $\mathrm{PIT}(13)$ is shown in Fig.5. We have $H = \{0, 1, 2, 5\}$, $U = \{3, 7, 8, 10\}$, and $D = \{4, 6, 9, 11\}$, giving the ordered composition (H, U, D) of $\{0, 1, \ldots, 11\}$. In this case, we have $\mathrm{COST}_0[\mathrm{PITR}] = 103$, which is less than $\mathrm{COST}_0[\mathrm{HOR}] = 156$. Hence $\mathrm{SR}_0[\mathrm{PITR}] = (156 - 103)/156 = 34.0\%$ for $p = 13$. It is noted that this saving ratio breaks the theoretical limit of 33.3% for STAR code in [16], owing to the use of PIT code.

Obviously, there are a number of ways to repair the corrupted units, which motivates the exploration of better recovery arrays for $\mathrm{PIT}(p)$ in the following sections.

Enumerating Recovery Arrays for Small p: Enumeration recovery is such an approach to reach the optimal result by enumerating all possibilities, which has been studied in [13], [14], [39] for general XOR-based erasure codes.

Here, we briefly introduce our enumerating method (PITR_e) for PIT code. We use $\mathrm{COST}_q = \mathrm{COST}_q(H, U, D)$ as the number of units read in recovering block q. We aim to seek such a composition of (H, U, D), which minimizes COST_q. This is a classical *balls-into-bins* problem with $p-1$ balls and 3 bins, which leads to 3^{p-1} possibilities. Computing COST_q given (H, U, D) can be done by computing the recovery array and counting the number of non-empty cells, requiring $O(p^2)$ steps. Obviously, the number of recovery arrays grows *exponentially* with p and it is a NP-hard problem [13] to achieve the optimal recovery array that minimizes COST_q.

Therefore, PITR_e will inevitably become computationally infeasible for large p. In addition, data recovery may be provided by third-parity companies like DataRecovery [2], which makes the optimal recovery array can not be computed in advance. Therefore we consider time-efficient heuristic techniques to find the close-optimal recovery array.

Generating Good Recovery Arrays for Larger p: In [39], a computationally efficient replace recovery algorithm is proposed to seek better solutions of single-failure recovery for any XOR-based erasure codes. Here, we follow the similar idea but design a more efficient greedy switching method (PITR_g) specially for PIT code, aiming to find a reasonable, locally optimal recovery array, suitable for larger primes, $p \geq 31$ say.

Given some ordered composition (H, U, D) of $\{0, 1, \ldots, p-2\}$, the switch we consider moves i from whichever set in $\{H, U, D\}$ it belongs to, to another set in $\{H, U, D\}$.

Algorithm 1 Greedy Switching Algorithm

Input: Prime p and corrupted block q
Output: Ordered composition (H, U, D) of $\{0, 1, \ldots, p-2\}$

```
1: Initialize the sets H = U = D = ∅
2: for i = 0 to p − 2 do
3:     s[0] = (H ∪ {i}, U, D)
4:     s[1] = (H, U ∪ {i}, D)
5:     s[2] = (H, U, D ∪ {i})
6:     (H, U, D) = arg min_{s[j]} COST_q(s[j])
7: end for
8: f = true
9: for i = 0 to p − 2 do
10:    r = COST_q(H, U, D)
11:    H = H \ {i}; U = U \ {i}; D = D \ {i}
12:    s[0] = (H ∪ {i}, U, D)
13:    s[1] = (H, U ∪ {i}, D)
14:    s[2] = (H, U, D ∪ {i})
15:    (H, U, D) = arg min_{s[j]} COST_q(s[j])
16:    if COST_q(H, U, D) < r then
17:        f = false
18:    end if
19: end for
20: Repeat Lines 8-19 until f == true
21: return (H, U, D)
```

Algorithm 1 shows the details of this greedy method. We start with $H = U = D = \emptyset$ and sequentially adds indices from

$\{0, 1, \ldots, p-2\}$ to either H, U, or D, depending on which introduces the fewest reads in recovering block q (Lines 1-7). If some switch reduces the number of reads which would be made in recovering block q, we apply it (Line 6), and do so repeatedly until no such switch exists (Lines 8-20). By finite descent, the algorithm will terminate. This results in an ordered composition (H, U, D), which achieves the local minimum number of reads.

The method proposed in [39] is for general XOR-based codes and thus there will be many obviously invalid parity switches, which introduces unnecessary computation. We follow a different rule with $H = U = D = \emptyset$, and repeatedly add or switch indices into that set who introduces the fewest number of unit reads, which ensures each step is valid and each switch is greedily performed.

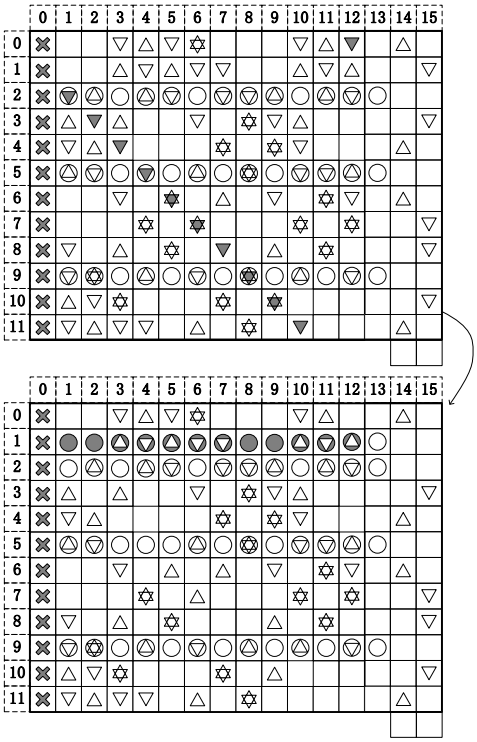

Fig. 6: An example of a switch: we use the horizontal parity (top) instead of the downward-diagonal parity (bottom) to recovery unit $a_{1,0}$ of $\mathrm{PIT}(13)$. That is, we move 1 from D to H.

Fig. 6 illustrates a switch process that might occur. In this example, we begin with $H = \{2, 5, 9\}$, $U = \{0, 4, 6, 11\}$, and $D = \{1, 3, 7, 8, 10\}$, which is obtained by Lines 1-7 of Algorithm 1. This composition results in the cost of 104 units to recover block 0. When the second loop runs (Lines 8-19) for the first time, it moves 1 from D to H, giving $H = \{1, 2, 5, 9\}$, $U = \{0, 4, 6, 11\}$, and $D = \{3, 7, 8, 10\}$, which would result in 103 units. We execute the loop again but it does not give any reduction on the number of unit read and leads to a local minimum result of 103 units. This local optimum is indeed the global optimum as realized in Fig.5.

We now analyze the time complexity to search the ordered composition (H, U, D). Line 6 takes $O(p)$ time to determine how many new units are required for the recovery of unit i. Since COST_q is smaller than or equal to p^2 and is decreased after each switch round (Lines 8-19), Line 20 will repeat for at most p^2 times in total. Therefore, the overall time complexity of Algorithm 1 is $O(p^4)$, which is polynomial with p.

In Algorithm 1, the corrupted units are examined sequentially in each switch round (Lines 8-19). Here more sophisticated strategy can be designed, e.g. at each switch round, the switch that results in the fewest units read among all the feasible switches is performed.

V. SPIT Codes

Thus far, the discussion of PIT implies the number of blocks in one code stripe is fixed at $p + 3$ for any prime p. In fact, we can use a *shortening* technology suggested in [4], [5] to break this prime limit. We select a large p (e.g. $p \geq 31$) for PIT code and then delete some data blocks from the code stripe. We use s to denote the number of deleted blocks and focus solely on deleting the last s data blocks (i.e. blocks from $p - s$ to $p - 1$).

Practically, we assume the deleted blocks are filled with zeros. They are involved in the construction of PIT code but do not affect the content of the parity blocks. We call the code constructed by this shortening technology *shortened PIT code* (SPIT). We use $\mathrm{SPIT}(p, s)$ to denote the code transformed from $\mathrm{PIT}(p)$ by deleting the last s data blocks. Moreover, because it simply treats the data on deleted blocks as zeros, it inherits the original code's fault tolerance.

In addition, a shortened version of EVENODD is proposed in [34] (see Section II-C), which shows possible benefit that code shortening has on single-failure recovery. To our knowledge, there is no other work caring about how code shortening affects single-failure recovery. The enumerating method PITR_e and the heuristic method PITR_g (see Section IV-C) proposed for PIT codes can be easily applied to SPIT codes for our purpose.

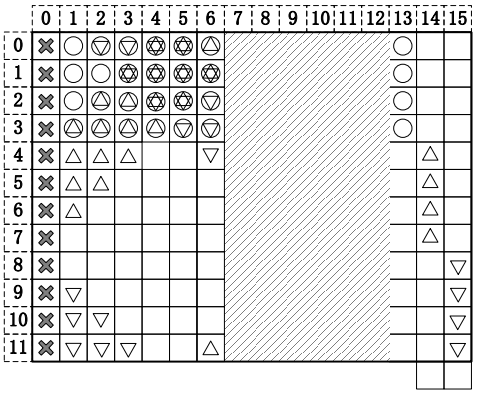

Fig. 7: The PITR recovery array for $\mathrm{SPIT}(13, 6)$, where six data blocks (blocks 7-12) are deleted and imagined to contain all 0s.

With $s > 0$ and $q \in \{0, 1, \ldots, p-1-s\}$ for $\mathrm{SPIT}(p, s)$, the average number of units required for any corrupted data block q under some strategy ST is

$$\mathrm{COST}_{\mathrm{avg}}[\mathrm{ST}] := \frac{1}{p-s} \sum_{q=0}^{p-1-s} \mathrm{COST}_q, \text{ if } s > 0. \quad (9)$$

The PITR_e recovery array for $\mathrm{SPIT}(13,6)$ with block 0 corrupted is shown in Fig.7. We have $H = \{0,1,2,3\}$, $U = \{4,5,6,7\}$ and $D = \{8,9,10,11\}$ for corrupted block 0. It requires reading 50 units from disk (along with the accompanying network transfers, etc.) while HOR needs to read $12 \times 7 = 84$ units, which implies $\mathrm{SR}_0[\mathrm{PITR}] = 1 - 50/84 \simeq 40.5\%$, which is an improvement upon $\mathrm{SR}_0[\mathrm{PITR}] \simeq 34.0\%$ as in Fig.5.

We can see that the shortening technology works on two sides:

1) It improves the *parameter flexibility*. Various choices of p and s for SPIT give different erasure codes $\mathrm{EC}(k,3)$ where $k = p - s$. In this way, SPIT expands the PIT code parameter space.
2) It improves the *recovery performance*. As Fig.7 and Fig.5 show, PITR can achieve higher saving ratio on $\mathrm{SPIT}(p,s)$ than on $\mathrm{PIT}(p)$. Moreover, we will show that $\mathrm{SPIT}(k+s,s)$ has better recovery performance than $\mathrm{PIT}(k)$.

With fixed p, we can trade-off storage overhead with recovery cost by tuning s; given fixed k, we can seek for the p with minimum recovery cost. Moreover, the shortening technology eliminates the prime limit.

The above discussion raises an interesting and important question: given k, which pair (p,s) (meeting $p = k+s$ and p is a prime) achieves the best $\mathrm{SR}_{\mathrm{PITR}}$? This question is aroused for practical need in cloud storage systems and we explore this experimentally in the next section.

VI. Evaluation

We conduct extensive experiments to evaluate PITR (i.e., PITR_e and PITR_g) on PIT codes and SPIT codes. The parameter p varies from 5 to 67 in our experiments, which is set according to real storage systems [1], [3], [11], [15], [23]. Unless otherwise specified, we use PITR_e for small p and PITR_g for larger p respectively.

A. Enumeration vs. Greedy Generating

a) Computing Time: We first examine the computing time of both enumerating and greedy switching methods. We conduct the evaluation on a Linux desktop server, which is configured with CentOS 6.5, quad-core Intel Core i7-3770 CPU @ 3.40GHz with hyper-threading enabled, and 32GB memory.

Table I shows the average time of enumerating and greedy switching methods for different p for PIT code. The second column "*Enum*" lists the time taken by PITR_e and the third column "*Greedy*" shows the time taken by PITR_g. We can see from the table that the enumerating time approximately follows the exponential growth according to 3^p. For $p = 29$, it takes more than 27 days to find the optimal recovery array, so we can estimate the computation time will be about 250 days for $\mathrm{PIT}(31)$. In contrast, the time incurred by greedy switching method is ignorable compared to enumeration. For $p = 31$, the computation time is only around 0.4 ms. We have also verified that for very large $p = 997$, the computation time is less than 3 s, which can satisfy any kind of requirement for practical use.

p	Enum	Greedy	Enum/Greedy
7	4.1 ms	18 us	228.3
11	6.7 ms	33 us	203.2
13	30.5 ms	51 us	597.4
17	2.0 s	98 us	2.0×10^4
19	16.9 s	162 us	10.4×10^4
23	22.3 m	326 us	4.1×10^6
29	27.5 d	276 us	8.6×10^9
31	—	379 us	—

TABLE I: The time taken by the enumerating and greedy switching methods for PIT with p ranging from 7 to 31.

b) Performance: Fig.8 shows the average cost, i.e. $\mathrm{COST}_{\mathrm{avg}}$ achieved by PITR_e and PITR_g respectively as the parameter p varies from 5 to 29 for PIT.

We can see that the performance of PITR_g are very close to PITR_e for all different p, differing by a factor of 0 to 0.9%. In fact, PITR_g achieves exactly the same COST_{avg} as PITR_g except at one point, e.g., $p = 19$. It indicates the effectiveness of the greedy switching method for small p.

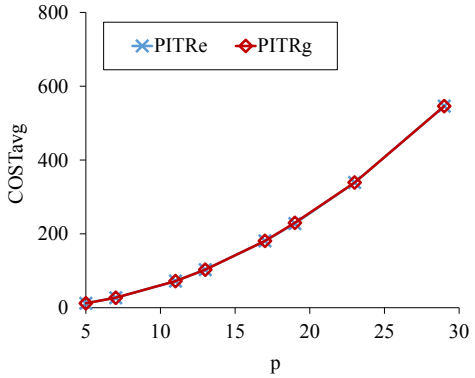

Fig. 8: Average number of units required for single data block failure recovery by using PITR_e and PITR_g respectively as prime p varies from 5 to 29.

When we compare the recovery performance of different recovery strategies and different coding schemes, simply focusing on $\mathrm{COST}_{\mathrm{avg}}$ is not enough: (1) as p increases, $\mathrm{COST}_{\mathrm{avg}}$ continues to increases, which will eventually fail to reflect the actual performance improvement intuitively; (2) if the shortening technology is applied, we cannot equate lower $\mathrm{COST}_{\mathrm{avg}}$ with higher recovery performance any longer. So we will use a more reasonable metric - saving ratio (SR, see Section IV-B) - in the following sections.

B. PIT Code vs. SPIT Code

In this section, we evaluate the saving ratios for PITR with different parameters and finally will show how SPIT code is superior over PIT code.

For $\mathrm{SPIT}(p,s)$ with s data blocks deleted, we only consider the cases where $p - s \geq 4$ because in cloud storage systems [1], [3], [11], a code stripe generally consists of more than or equal to 4 data blocks. We define $r = \frac{s}{p}$ as the shortening proportion.

To explore how r impacts the PITR performance as p varies, we run the experiments in three cases: (1) $r = 0$ (PIT) where no data blocks are deleted; (2) $r = \frac{1}{3}$, where a small proportion of data blocks of each code stripe is deleted; (3) $r = \frac{2}{3}$, where a large proportion of data blocks of each code stripe is deleted.

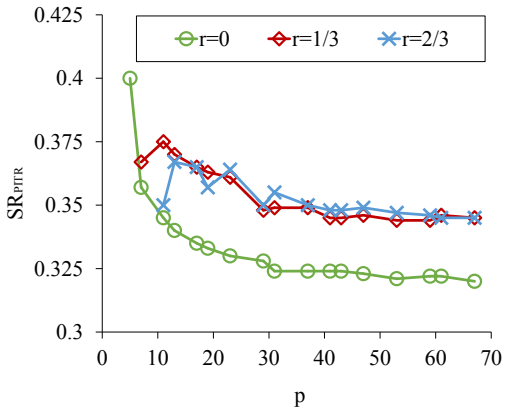

Fig. 9: SR of PITR with $r = 0$, $\frac{1}{3}$, and $\frac{2}{3}$ as prime p varies from 5 to 67.

The experimental results are shown in Fig.9. We learn that there is no a universal optimal shortening proportion r for all p and a specific r may leads to quite different performance changes for different p. For example with $r = 0$, SR varies from 32.0% to 40.0% as p varies. When p is very small (e.g. $p = 5$), the best PITR performance is achieved with $r = 0$. This is because that when p is small and $r > 0$, too few data blocks in one code stripe are left to overlap more units to yield a low single-failure recovery cost. As p increases, the saving rate of regular PIT code decreases rapidly. PITR with $r = 1/3$ achieves the highest SR when $7 \leq p < 23$ while PITR with $r = 2/3$ does best for $23 \leq p \leq 53$. As p continues to increase, PITR with $r = 1/3$ achieves nearly the same SR as that with $r = 2/3$ and their performance is stable. The SR is about 3 percent higher than that with $r = 0$ for large p, which highlights the benefit of shortening strategy.

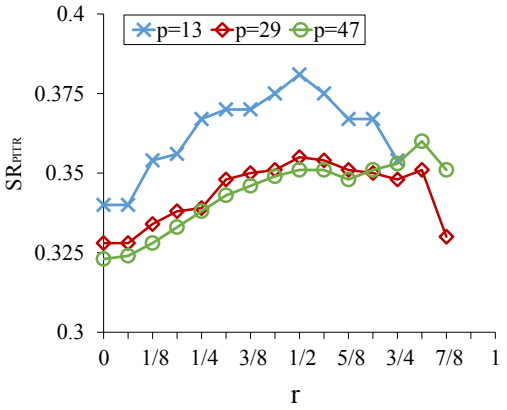

Fig. 10: $\mathrm{SR}_{\mathrm{PITR}}$ with $p = 13$, 29, and 47 as r varies, where $0 \leq r < 1$ and its step size is $\frac{1}{16}$.

We also explore how r impacts SR with the given p in detail and use $p = 13$, 29 and 47 for experiments, which represent the small, middle and large stripe sizes respectively.

The results are illustrated in Fig.10. Some observations are drawn from the figure: (1) In the first stage where $r < \frac{1}{2}$, all of the three SR curves climbs quickly as r increases; (2) As r continues to increase, SR for $p = 13$ decreases sharply while those for $p = 29$ and $p = 47$ also have a similar decreasing trend. This is mainly because that only a few data blocks in one code stripe are left when $p = 13$ and $r > \frac{1}{2}$; (3) Varying r impacts the SR significantly for the given p. For example, SR for $p = 13$ experiences a factor of at most 12.1% difference by changing r.

C. Optimal SPIT Codes

We have shown in Fig.10 that both p and r would impact SR performance. Different combinations of p and r (or s) result in an average of around 10% performance gap on SR. In practical cloud storage systems, k (or $p - s - 1$) is generally fixed since the user usually requires a specific storage overhead. This gives us a hint to explore the optimal p (or s) to achieve the best recovery performance for the given k.

For $4 \leq k \leq 67$, We have enumerated all of the feasible $\mathrm{SPIT}(p, s)$ as p varies from 5 to 67 with $p - s = k$. Table II shows the SR over the naive HOR method and corresponding (p, s) for SPIT and PIT codes. The first Column "k" denotes the physical number of data blocks in each code stripe. For SPIT, different combinations (p, s) lead to the same k (i.e., the same storage overhead) but diverse SR, and we present the maximum and minimum SR and corresponding (p, s) combinations in the second column "*Maximum*" and the third column "*Minimum*" respectively. Whereas, the PIT codes with $k = 4, 6, 9, 12$ and 44 have no results because k should be prime.

	SPIT				PIT	
	Maximum		Minimum		Original	
k	SR	(p, s)	SR	(p, s)	SR	(p, s)
4	37.5%	(5, 1), (7, 3)	32.3%	(59, 55)	—	—
5	40.0%	(5, 0)	33.3%	(37, 32)	40.0%	(5, 0)
6	38.9%	(7, 1)	34.7%	(61, 55)	—	—
7	38.1%	(13, 6)	35.2%	(61, 54)	35.7%	(7, 0)
9	37.5%	(17, 8)	35.3%	(41, 32)	—	—
11	37.2%	(23, 12)	34.5%	(11, 0)	34.5%	(11, 0)
12	36.7%	(23, 11)	34.7%	(37, 25)	—	—
13	36.7%	(23, 10)	34.0%	(13, 0)	34.0%	(13, 0)
17	35.6%	(23, 6)	33.5%	(17, 0)	33.5%	(17, 0)
23	35.2%	(47, 24)	33.0%	(23, 0)	33.0%	(23, 0)
31	35.0%	(53, 22)	32.4%	(31, 0)	32.4%	(31, 0)
44	34.5%	(67, 23)	32.4%	(47, 3)	—	—

TABLE II: The performance improvement with various k of PITR, compared to HOR method.

Some conclusions are drawn from the table: (1) By deploying PITR, both SPIT and PIT achieves the SR performance by

a factor of over 32.0% for different stripe sizes, showing the benefit of PITR; (2) The greatest improvement comes from small values of k, which are most frequent cases in popular storage system [11], [28], [33]; (3) As k increases, the SR for PIT decreases by around 19 percent while the maximum performance for SPIT decreases by around 13 percent and finally tends to be stable (around 35.0%), which illustrates the effectiveness of shortening technology; (4) SPIT shows a roughly of more than 34.5% performance improvement over HOR at any k, which is at most 15% better than another recovery strategy presented in [16] for STAR code, and roughly 7.1% better in average than PIT code; (5) Some k can not be achieved for PIT because of the prime limit (e.g. $k = 6$) and SPIT overcomes the limit of prime p and extends the parameter space.

To deploy our codes into cloud storage systems like HDFS-RAID [3], we first select an appropriate k, according to the storage overhead. For example, with $k = 6$, we choose SPIT code with $(p, s) = (7, 1)$ to encode the 6 data blocks in the code stripe. We also store all the optimal and near-optimal recovery arrays generated by PITR for any corrupted block in advance. When disk or node failure happens in this system, blocks on it become corrupted. For one given code stripe with single data block q corrupted, we use the pre-stored recovery array for recovery. In this way, it saves 38.9% network traffic and units read compared to naive HOR method to recover the single-failure.

D. PIT and SPIT vs. STAR

In this section, we show the comparison of PIT and SPIT codes with STAR code. In [34], [39], a lower bound (i.e., $\frac{2}{3}p^2 - p$) and an upper bound (i.e., $\frac{13}{18}p^2 + \frac{17}{9}p - \frac{47}{18}$) of the nubmer of units required for single-failure in STAR code are theoretically proved respectively. We use STAR_{low} and STAR_{up} to denote the lower and upper bound, and STAR_{opt} to denote the optimal curve. We run the experiments for PIT and SPIT codes as p varies from 5 to 67.

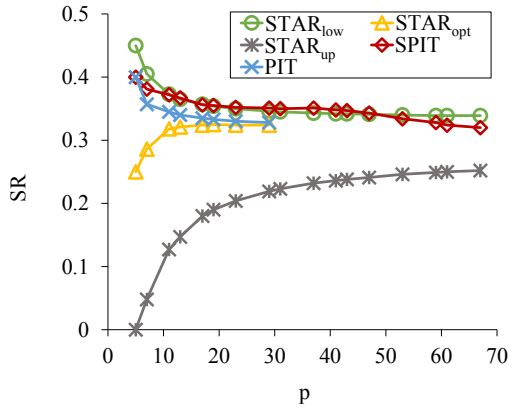

Fig. 11: SR of STAR_{low}, STAR_{opt}, STAR_{up}, PIT and SPIT over HOR as prime p varies from 5 to 67.

Fig.11 plots the SR of STAR_{low}, STAR_{opt}, STAR_{up}, PIT and SPIT, where the horizontal axis k is the number of left data blocks in one code stripe. We use PITR_e for PIT and STAR_{opt} as p (i.e. k) varies from 5 to 29. For SPIT, the optimal SRs are achieved by enumerating all combinations of (p, s), meeting $k = p - s$.

We can draw some conclusions from the figure:

- The upper bound is not tight, with an average of 23.3% SR gap between STAR_{up} and STAR_{opt}. As p increases, SR for STAR_{up} increases sharply and should finally be stable at $1 - \frac{13}{18} = 27.8\%$ according to the upper bound equation. By contrast, SR for STAR_{low} will be stable at $1 - \frac{2}{3} = 33.3\%$.
- SPIT performs better than both STAR and PIT. Fig.11 illustrates a higher SR for SPIT than the optimal case for STAR and PIT. Specially, SPIT shows roughly a factor of 20% improvement over STAR, and 6.1% improvement over PIT in average.

VII. Related Work

Extensive effort has been put into reducing the recovery costs of erasure coding in various ways, e.g. including new erasure coding methods such as Hitchhiker Code [26], Weaver Code [9], Hover Code [10], LRCs [11], [28], Rotated RS Code [14] and Regenerating Codes [22], [27]; hybrid methods such as AutoRAID [35], DiskReduce [7], HDFS-RAID [3] and HACFS [36]; parallel hardware and/or software methods such as PPR [19], CRR [21] and CRS [24]; proactive fault tolerance methods, such as RAIDShield [17], ProCode [20] and Fatman [25].

The frequency of single disk failure is much higher than simultaneous failures [30], which arouses researchers' attention. Some researchers design new non-MDS codes which use extra storage to reduce single-failure recovery costs. For example, a new class of codes called LRCs are proposed in [11] and [28], which stores additional local parities for subgroups of blocks to reduce recovery costs. Aside from not being storage efficient, they require Galois field multiplication operations for encoding and decoding. In this paper, the efficient single-failure recovery is achieved by using PIT code, which is nearly storage optimal and requires only XOR operations.

Some researchers focus on optimization techniques for specific XOR-based MDS codes. For example, [16], [34], [37], [38] theoretically prove the lower/upper bound of the amount of read data for some popular array codes like RDP, X-code, EVENODD, STAR and TP. The techniques to reduce the amount of I/Os for general XOR-based erasure codes are proposed in [14], which finds the optimal recovery by constructing and searching the weighted graph and in [39], which presents a time-efficient greedy search algorithm. We follow the similar idea but design PITR specially for PIT and SPIT code, which works efficient.

There are also researchers [32] taking disk seeks into consideration, given the fact that disk seek may negate recovery performance. We do not use disk seeks as our metrics, because in large-scale cloud storage systems a large block size (i.e. 64MB) is commonly deployed, where the number of units read (and transferring) dominates the recovery performance.

VIII. Conclusion and Future Work

In this paper, we have shown the additional overhead of STAR during single-failure recovery. To address this issue, we have designed PIT code, a variation of STAR, and proposed

an efficient single-failure recovery scheme for PIT. We also deploy a shortening strategy for PIT code to further optimize the recovery cost. Experimental results show that our proposed methods decrease the recovery cost significantly compared to existing state-of-art strategies by sacrificing a bit more storage resources.

In the future research, we consider to deploy PIT code and PITR into a distributed cluster like HDFS-RAID [3], aiming to evaluate a more realistic recovery performance.

REFERENCES

[1] "Colossus, successor to google file system," http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf, [Online; accessed 14-September-2015].

[2] "DataRecovery," http://www.datarecovery.com/.

[3] "HDFS RAID," http://wiki.apache.org/hadoop/HDFS-RAID, [Online; accessed 14-September-2015].

[4] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 192–202, 1995.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *FAST-2004: 3rd Usenix Conference on File and Storage Technologies*, 2004.

[6] 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*, vol. 56, no. 9, pp. 4539–4551, 2010.

[7] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 6–10.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[9] J. L. Hafner, "Weaver codes: Highly fault tolerant erasure codes for storage systems." in *FAST*, vol. 5, 2005, pp. 16–16.

[10] ——, "Hover erasure codes for disk arrays," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE, 2006, pp. 217–226.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure coding in windows azure storage." in *USENIX Annual Technical Conference*, 2012, pp. 15–26.

[12] C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 889–901, 2008.

[13] O. Khan, R. C. Burns, J. S. Plank, and C. Huang, "In search of i/o-optimal recovery from disk failures." in *HotStorage*, 2011.

[14] 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*, 2012, p. 20.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[16] F. W. L. Qiu and C. Li, "Eds: A novel scheme for boosting single-disk failure recovery of triple-erasure-correcting code storage systems." vol. 36, 10 2013, in Chinese.

[17] A. Ma, F. Douglis, G. Lu, D. Sawyer, S. Chandra, and W. Hsu, "Raidshield: characterizing, monitoring, and proactively protecting against disk failures," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, 2015, pp. 241–256.

[18] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[19] 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*. ACM, 2016, p. 30.

[20] R. J. S. G. W. Z. L. P. Li, J. Li and X. Liu, "Procode: A proactive erasure coding scheme for cloud storage systems," in *Proceedings of 35th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2016, pp. 219–228.

[21] R. J. S. G. W. Z. L. X. L. P. Li, X. Jin, "Parallelizing degraded read for erasure coded storage systems using collective communications," in *TrustCom/BigDataSE/ISPA*.

[22] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple regenerating codes: Network coding for cloud storage," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2801–2805.

[23] J. S. Plank and M. G. Thomason, "A practical analysis of low-density parity-check erasure codes for wide-area storage applications," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 115–124.

[24] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*. IEEE, 2006, pp. 173–180.

[25] A. Qin, D. Hu, J. Liu, W. Yang, and D. Tan, "Fatman: Cost-saving and reliable archival storage based on volunteer resources," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1748–1753, 2014.

[26] K. V. Rashmi, N. B. Shah, D. GU, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhikers guide to fast and efficient data reconstruction in erasure-coded data centers." in *Proc. ACM SIGCOMM*, 2014.

[27] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[28] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proc. VLDB Endowment*, 2013.

[29] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" in *FAST*, vol. 7, 2007, pp. 1–16.

[30] ——, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" in *FAST*, vol. 7, 2007, pp. 1–16.

[31] Z. Shen and J. Shu, "Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 550–561.

[32] Z. Shen, J. Shu, P. P. Lee, and Y. Fu, "Seek-efficient i/o optimization in single failure recovery for xor-coded storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 877–890, 2017.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[34] Z. Wang, A. G. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*. IEEE, 2010, pp. 1905–1909.

[35] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The hp autoraid hierarchical storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 108–136, 1996.

[36] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 213–226.

[37] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in rdp code storage systems," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, 2010, pp. 119–130.

[38] S. Xu, R. Li, P. P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. C. Lui, "Single disk failure recovery for x-code-based parallel storage systems," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 995–1007, 2014.

[39] 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 *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–12.