Exploiting Parallelism of Disk Failure Recovery via Partial Stripe Repair for an Erasure-Coded High-Density Storage Server

Lin Wang[†], Yuchong Hu[†], Qian Du[†], Dan Feng[†], Ray Wu[§], Ingo He[§], Kevin Zhang[§]
[†]Huazhong University of Science and Technology [§]Inspur
{wanglin2021, yuchonghu, durant_qian, dfeng}@hust.edu.cn {wuruizhen, heyingbj, zhangkai_bj}@inspur.com

ABSTRACT

High-density storage servers (HDSSes), which pack many disks into single servers, are currently used in data centers to save costs (power, cooling, etc). Erasure coding, which stripes data and provides high availability guarantees, is also commonly deployed in data centers at lower cost than replication. However, when applying erasure coding to a single HDSS, we find that erasure coding's state-of-the-art studies that improve repair performance in parallel mainly use multiple servers' sufficient footprint, which is yet quite limited in the single HDSS, thus leading to a memory-competition issue for disk failure recovery.

In this paper, for a single HDSS, we analyze its disk failure recovery's parallelism which exists within each stripe (intra-stripe) and between stripes (inter-stripe), observe that the intra-stripe and inter-stripe parallelisms are mutually restrictive, and explore how they affect the disk failure recovery time. Based on the observations, we propose, for the HDSS, partial stripe repair (HD-PSR) schemes which exploit parallelism in both active and passive ways for single-disk recovery. We further propose a cooperative repair strategy to improve multi-disk recovery performance. We prototype HD-PSR and show via Amazon EC2 experiments that the recovery time of a single-disk failure and a multi-disk failure can be reduced by up to 71.7% and 52.5%, respectively, over existing erasure-coded repair scheme in high-density storage.

KEYWORDS

High-density storage server, Erasure coding, Disk failure recovery

ACM Reference Format:
Lin Wang[†], Yuchong Hu[†], Qian Du[†], Dan Feng[†], Ray Wu[§], Ingo He[§], Kevin Zhang[§]. 2022. Exploiting Parallelism of Disk Failure Recovery via Partial Stripe Repair for an Erasure-Coded High-Density Storage Server. In *51st International Conference on Parallel Processing (ICPP '22), August29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545014

1 INTRODUCTION

As data center storage expands at scale, its power and cooling costs have increased significantly due to its expanding physical space. To save the costs, high-density storage servers (HDSSes) have been often deployed to reduce the physical data center space by packing many disks into single servers and thus improving power and cooling efficiency. Recent studies [1] show that a data center based on HDSSes can save millions of dollars per year. Since an HDSS is composed of many disks (e.g., A Dell PowerEdge server can support up to 192 drives [4]), it is critical for it to provide availability guarantees against disk failures within the server.

To maintain data availability, there are two major redundancy methods: replication and erasure coding. The former that replicates data into multiple copies and stores them across nodes incurs high redundancy. In contrast, the latter encodes every k original chunks into n coded chunks to form a *stripe* such that any failed chunk can be repaired by any k chunks of the stripe, and costs a much lower redundancy than replication while maintaining the same availability [33]. Therefore, erasure coding is widely deployed in production (e.g., HDFS [29], Ceph [34], Swift [2] and QFS [23]). In fact, many HDSS products have deployed erasure coding, e.g., DELL's HDSS [4] leverages RAID-like erasure codes to handle its disk failures in the server.

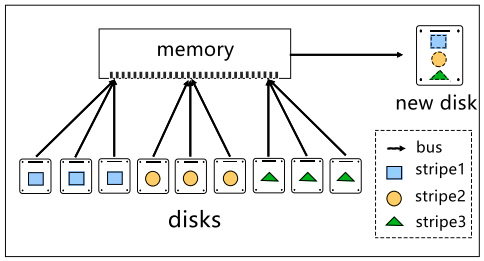
However, when deploying erasure coding under an HDSS, it is challenging to perform disk failure recovery efficiently due to limited memory of the server. There are two reasons stated below:

- When repairing each lost chunk of the failed disk, a traditional (low-density) RAID-based storage server typically reads k chunks of each stripe from disks to memory (we call *full stripe repair*); as the number of the traditional storage server's disks is often not much larger than the size of stripe, the server basically cannot simultaneously support multiple full stripe repairs. In contrast, as illustrated in Figure 1(a), an HDSS that owns dozens of disks allows multiple full stripe repairs to be performed at the same time, thus incurring *memory competition* among these stripes.
- Most erasure-coded state-of-the-art repair schemes focus on distributed storage with limited network resources, as illustrated in Figure 1(b). Many of them [5, 18, 21, 36] aim to trade sufficient computation and memory resources of distributed servers for repairing the lost chunk of a stripe efficiently, in a way that schedules the repair traffic over servers such that each involved server can handle partial repair of the stripe in parallel (we call *partial stripe repair*). Unfortunately, a single HDSS has limited memory resources, so the current partial-stripe-repair based parallelism cannot simply work.

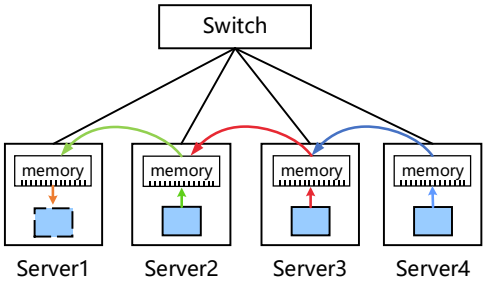
Based on the above reasons for the disk failure recovery challenge in an erasure-coded HDSS, we pose the following question: *When reconstructing each chunk of the failed disk from each stripe in an erasure-coded HDSS, how to handle the memory competition among stripes by exploiting the parallelism with the help of partial stripe repair*?

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545014



(a) Memory competition in an erasure-coded high-density storage server



(b) Bandwidth competition in erasure-coded distributed storage systems

Figure 1: The challenges of deploying erasure coding to a high-density storage server and distributed storage.

In this paper, we propose a new *partial stripe repair* scheme for an HDSS (HD-PSR) to accelerate disk recovery repair by exploiting parallelism within each stripe (intra-stripe) and between stripes (inter-stripe). Our contributions are as follows:

- We propose the concept of intra-stripe parallelism (P_a) and inter-stripe parallelism (P_r) and observe that intra-stripe parallelism and inter-stripe parallelism mutually restrict each other and both affect single-disk repair in an HDSS. This motivates us to develop hybrid schemes which exploit both intra-stripe parallelism and inter-stripe parallelism to minimize the repair time.
- We design three algorithms for HD-PSR. The first two are active algorithms that obtain the transfer speed of disks through active testing: (i) HD-PSR-AP, which minimizes the transfer time and (ii) HD-PSR-AS, a slower-first algorithm that reduces the time complexity compared to HD-PSR-AP. We also design a passive algorithm: (iii) HD-PSR-PA that obtains the transfer speed of disks through passive adjustment.
- We prototype HD-PSR and evaluate it via Amazon EC2 experiments. The results show that HD-PSR can reduce the single-disk repair time and the multi-disk repair time by up to 71.7% and 52.5%, respectively. Our prototype is opensourced at: https://github.com/YuchongHu/hdpsr.

2 BACKGROUND AND MOTIVATION

We introduce the basic concepts of erasure coding (§2.1), describe the parallelism in erasure-coded repair for distributed storage (§2.2), state the memory-competition issue in erasure-coded repair for an HDSS (§2.3), and motivate the idea of partial stripe repair design for an HDSS via examples (§2.4).

2.1 Basics of Erasure Coding

There are various forms of erasure codes and Reed-Solomon (RS) Codes [31] are the most widely used one, so we focus on RS codes in this paper. Two parameters are configured in RS codes which are n and k ($n > k$), implying that the system encodes k fixed-size chunks into n chunks, which form as a *stripe* including the original k *data chunks* and $m = n - k$ *parity chunks*. RS codes are Maximum Distance Separable (MDS), which means that any chunk can be reconstructed by any k surviving chunks out of its stripe. Therefore, (n, k) RS codes can tolerate $m = n - k$ failures. Mathematically, the m parity chunks are generated by a linear combination of the k data chunks based on the arithmetic of Galois Field GF(2^w) in w-bit word [27]. Assuming that the k data chunks are $D_1, D_2, \ldots, D_k$, and the m parity chunks are $P_1, P_2, \ldots, P_m$. Then,

$$P_j = \sum_{i=1}^{k} \alpha_{ij} D_i, \tag{1}$$

where α_{ij} is the encoding parameter in encoding matrices. Based on Equation (1), any chunk reconstruction can be performed through a linear addition. For example, if D_i is failed, we can reconstruct it by a linear combination:

$$D_i = \frac{1}{\alpha_{ij}}(P_j + \sum_{t=1}^{i-1} \alpha_{tj} D_j + \sum_{t=i+1}^{k} \alpha_{tj} D_j). \tag{2}$$

Based on Equation (2), the conventional RAID-based storage server [26] repairs a failed chunk by reading the k available chunks of its stripe from k disks at a time into memory, which we call *full stripe repair*, or FSR for short. FSR will be the baseline algorithm in this paper for disk failure recovery in a high-density storage server.

2.2 Parallelisms in Erasure-Coded Repair for Distributed Storage

We survey the erasure coding's state-of-the-art studies that improve repair performance in parallel for distributed storage systems, and classify them into two types of parallelism.

2.2.1 *Intra-stripe parallelism for distributed storage.*

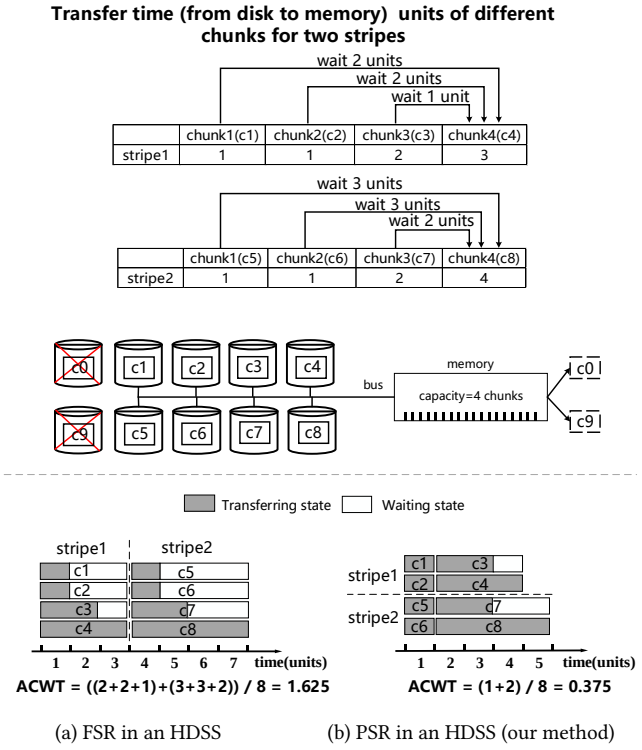
Equation (2) shows the linearity of erasure-coded repair, which enables a single-chunk repair operation to do additions in any order, so the single-chunk repair can be performed in parallel within its stripe over distributed servers, which we call *intra-stripe parallelism for distributed storage*.

Based on the above intra-stripe parallelism for distributed storage, extensive repair schemes that accelerate the single-chunk repair have been proposed, including PPR [21], RP [18], PPT [5], SMFRepair [36], etc. They all decompose the repair operations into sub-operations distributed to multiple servers for parallel computation to improve single-chunk repair performance.

2.2.2 *Inter-stripe parallelism for distributed storage.*

In distributed storage, different server nodes can transfer data simultaneously, so for a single-node failure, multiple single-chunk repairs can be performed in parallel, which we call *inter-stripe parallelism for distributed storage*.

Based on inter-stripe parallelism for distributed storage, recent state-of-the-art schemes that accelerate the single-node repair have



(a) FSR in an HDSS (b) PSR in an HDSS (our method)

Figure 2: A motivation example of how our PSR based method outperforms FSR. We set $k = 4$ and the memory capacity is four chunks. There are two stripes $stripe_1$ (containing four available chunks c_1, c_2, c_3 and c_4) and $stripe_2$ (containing four available chunks c_5, c_6, c_7 and c_8), and each stripe needs to repair one lost chunk. The table shows the transfer time (i.e., the time units for data chunks to be transferred from disk to memory) and the waiting time of chunks for each stripe. The grey bar indicates the transferring state of a chunk, which is being transferred into the memory, and the white bar indicates the waiting state of a chunk, which is waiting for the slower chunks.

been proposed, including ECWide [13], RepairBoost [19], etc. They accelerate the single-node repair via considering the heterogenous load of each node to balance the repair traffic among concurrent multiple state-of-the-art single-chunk repairs schemes (like PPR and RP).

2.2.3 Partial stripe repair for distributed storage.
All the above erasure-coded parallelled repair state-of-the-arts are based on repair traffic scheduling by paralleling the repair operation over distributed servers, each of which utilizes its memory to do partial repair of the stripe, which we call *partial stripe repair*, or PSR for short.

Note that PSR is only designed for distributed storage so far. However, existing studies for an HDSS only use FSR, and no study has focused on how to exploit intra-stripe and inter-stripe parallelisms via PSR for an HDSS. Therefore, we will detail the memory competition issue incurred by FSR in the HDSS in §2.3, and then illustrate how to leverage PSR to parallel disk failure recovery in the HDSS in §2.4

2.3 Memory Competition for an HDSS with FSR

When applying FSR in an HDSS, as shown in Figure 1(a), we find that multiple simultaneous FSR's stripes will compete for memory. More importantly, FSR implies that its retrieval time of k chunks of a stripe from disks to memory is determined by the slowest chunk who has the slowest transfer speed from disk to memory, so the other chunks that are read into memory prior to the slowest chunk have to enter the waiting state, thus degrading the disk recovery performance.

Average chunk waiting time (ACWT): Specifically, we first define the time interval during the waiting state of a chunk as its waiting time. Then, to repair a single failed disk's all lost chunks, we define the average of all available chunks' waiting time as *average chunk waiting time* for the single-disk recovery, denoted by $ACWT$. Clearly, $ACWT$ implies the efficiency of disk recovery performance.

We observe that FSR largely increases $ACWT$, since for each stripe, before the its slowest chunk is read into memory, all the others of the stripe have to wait in the memory, as illustrated in Figure 2(a), $stripe_2$ has to wait until $stripe_1$ finishes its single-chunk repair. It lowers the memory utilization significantly and thus impairing the disk recovery performance.

2.4 A Motivation Example based on PSR

As stated in §2.3, FSR causes a large $ACWT$, which impairs the performance of disk recovery. Figure 2(a) shows that the total time of repairing two lost chunks of the failed disk is 7 time units, and $ACWT = 1.625$, which can be calculated by dividing the sum of all the waiting time of all available chunks (i.e., (2+2+1)+(3+3+2)=13 time units) by the total number of available chunks (i.e., 8 chunks).

In contrast, we extend the concept of PSR in §2.2.3 in an HDSS scenario, which allows each stripe involved in the disk failure recovery to provide part of its chunks into the HDSS's memory, and compute the partial result of the repaired chunk based on Equation (2). For example, Figure 2(b) shows the idea of our method. We read 2 chunks of each stripe into memory at a time, instead of reading all of them (i.e., FSR). Then compared to FSR, the total transfer time of PSR is reduced to 5 and $ACWT$ of PSR is is reduced to $(1 + 2)/8 = 0.375$, since for all eight chunks, only two chunks c_3 and c_7 need to wait one and two time units, respectively.

Nevertheless, it is still challenging to take full advantages of PSR to exploit intra-stripe and inter-stripe parallelisms, meaning that how to parallel the partial stripe repairs to minimize the total repair time still needs more analysis, which will be specified in §3.

3 OBSERVATIONS AND ANALYSIS

Based on the definitions of intra-stripe and inter-stripe parallelisms for distributed storage in §2, we re-define them for an HDSS in §3.1, show our observations in §3.2, and give analysis as well as our main idea in §3.3.

3.1 Parallelism for an HDSS

3.1.1 Intra-stripe parallelism for an HDSS.
In §2.2.1, the intra-stripe parallelism for distributed storage shows

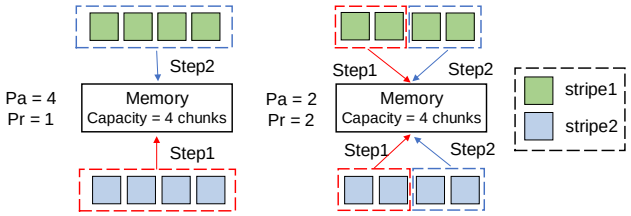
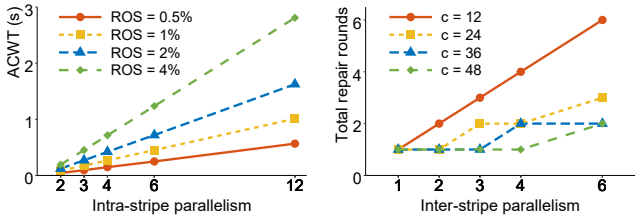


Figure 3: Observation 1: Illustration on mutually restrictive relationship between intra-stripe parallelism and inter-stripe parallelism.



(a) Observation 2 (b) Observation 3

Figure 4: Observations 2 and 3: P_a and $ACWT$ are positively correlated. P_r and the total repair rounds are positively correlated.

that a single-chunk repair operation can be paralleled over distributed servers such that each server performs PSR in parallel. Different from it, the intra-stripe parallelism for an HDSS is that to repair a single-chunk of a stripe, the HDSS's memory can read multiple chunks of a stripe in parallel to accelerate the single-chunk repair. We define the number of chunks within the stripe read from disks to memory in parallel as *intra-stripe parallelism degree*, denoted by P_a.

For example, in Figure 2, for FSR, $stripe_1$ allows all its four chunks $c_1, \ldots, c_4$ to be read into memory in parallel during time units 1-3, and then $stripe_2$ also parallels its four chunks into memory during time units 4-7, which means that $P_a = 4$ all the time; for PSR, $stripe_1$ only allows two chunks c_1 and c_2 to be read into memory during time unit 1, while $stripe_2$ only allows two chunks c_5 and c_6 to be read into memory, so $P_a = 2$.

3.1.2 *Inter-stripe parallelism for an HDSS.*

In §2.2.2, the inter-stripe parallelism for distributed storage means that for a single node recovery, multiple single-chunk repairs can be performed in parallel. Different from it, the inter-stripe parallelism for an HDSS is for a single-disk recovery, the HDSS's memory can read multiple stripes' chunks in parallel so as to accelerate the single-disk recovery. We define the number of stripes which are being read into memory in parallel as *inter-stripe parallelism degree*, denoted by P_r.

For example, in Figure 2, for FSR, $stripe_1$ occupies the memory during time units 1-3 and $stripe_2$ does during time units 4-7, which means the memory handles only one stripe all the time, so $P_r = 1$; for PSR, $stripe_1$ and $stripe_2$ can be read into memory in parallel, so $P_r = 2$.

Table 1: Descriptions of notation.

Notation	Description
s	number of stripes to be repaired
c	capacity of memory in chunks
P_a	intra-stripe parallelism degree
P_r	inter-stripe parallelism degree
$ACWT$	average chunk waiting time
TR	total repair rounds
$L_{s\times k}$	2D array of transfer time for chunks
$L[i][j]$	transfer time of j^{th} chunk of i^{th} stripe of $L_{s\times k}$
L_s	1D array of transfer time for stripes
L_i	transfer time of the i^{th} stripe of L_s
T	total transfer time for disk recovery

3.2 Observations

Observation 1 (There is a mutually restrictive relationship between these two parallelisms): Due to the limited memory of HDSS, all stripes will be queued into memory. If P_a is high which means the memory reads more chunks within a stripe at a time, then P_r is low as memory can process fewer stripes at the same time, and vice verse (i.e., if P_a is low, then P_r is high). Mathematically, if we define the memory capacity in chunks as c and assume each stripe has the same P_a, then the rigorous relationship between P_a and P_r is

$$P_a = \lceil c/P_r \rceil \tag{3}$$

Figure 3 illustrates the mutually restrictive relationship between these two parallelisms. We assume the memory can only contain 4 chunks (i.e., $c = 4$). If we set P_a as 4, then P_r is $\lceil c/P_a \rceil = 1$. If we set P_a as 2, then P_r is $\lceil c/P_a \rceil = 2$.

Observation 2 (P_a and $ACWT$ are positively correlated): For further observation, we denote the array of transfer time of chunks by a two-dimensional $L_{s\times k}$, where $L[i][j]$ denotes the transfer time of the j^{th} chunk of the i^{th} stripe, and generate some data sets for $L_{s\times k}$ that fits the normal distribution with the mean equal to 2 and the variance equal to 4. We also set $s = 100$, $k = 12$, and $c = 12$. We define ROS as the ratio of the slow chunks to the total number of chunks and set it as 2%, 5%, 8%, and 10%. Then we calculate $ACWT$ for different P_a and P_r, respectively. Figure 4(a) shows the relationship between $ACWT$ and P_a.

Figure 4(a) shows that $ACWT$ increases with P_a. The reason is that a higher P_a indicates that more chunks of a stripe will be read into memory at a time, so more chunks have to wait for the slow chunks of the stripe, thus increasing the $ACWT$. In addition, we figure out that when ROS increases, $ACWT$ also increases, meaning that the more the slow chunks, the more time wasted waiting for them.

Observation 3 (P_r and the total repair rounds (TR) are positively correlated): We define that a stripe's P_a chunks read into memory as a *repair round*, and then define the number of rounds it takes to repair a stripe as the *total repair rounds* of the stripes, denoted by TR, which can be computed as k/P_a. For example, we set $k = 6$ and $P_a = 2$, and then a stripe will take 3 total repair rounds to complete the repair. Figure 4(b) plots the relationship

between the total repair rounds and P_r. An increase in the total repair rounds affects the single-chunk repair and thus influences the repair time of single-disk.

3.3 Analysis

Observation 1 shows that an increase in one parallelism necessarily leads to a decrease in another parallelism. Therefore, we need to consider both parallelisms together instead of simply setting one of them to the maximum (like full stripe repair).

Observation 2 shows that the conventional method sets P_a as the maximum value k simply. Based on the above two observations, the conventional method results in the $ACWT$ value being the largest, thus leading to the worst memory utilization. Therefore, we propose the ideology of partial stripe repair to make appropriate adjustments to P_a and Pr, so that the memory utilization can be improved, thereby accelerating the repair speed.

Observation 3 shows that although a larger P_r decreases the $ACWT$ significantly, it results in larger total repair rounds, which also has the potential to make the overall repair time longer. Thus we cannot naively set P_r to the maximum, instead we should design more sophisticated algorithms to select the reasonable P_r.

Main idea: Based on the observations described above, our idea is to exploit both intra-stripe parallelism (P_a) and inter-stripe parallelism (P_r) to reduce the $ACWT$ while making TR not too large, thus improving the repair performance of disk failures based on PSR in an HDSS, which is called HD-PSR and will be specified in §4.

4 HD-PSR DESIGN

We first provide definitions for disk failure recovery in an HDSS (§4.1), propose PSR based single-disk repair algorithms in an active way (§4.2) and a passive way (§4.3), and propose a cooperative multi-disk repair scheme (§4.4).

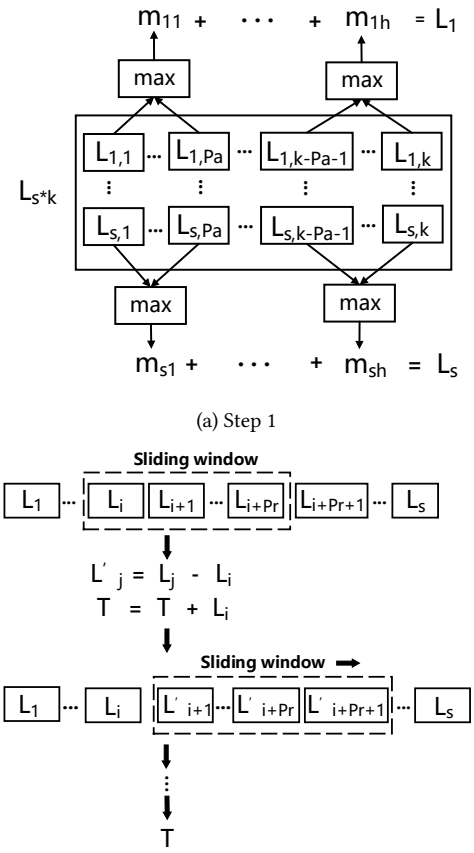
4.1 Definitions

We first define notation in Table 1 used in this section. First, the notation c, P_a, P_r, $L_{s\times k}$, $ACWT$, and TR have been described in §2 and §3. For the other notation, s is the number of stripes to be repaired in the failed disk. L_s is a one-dimensional array with s elements reduced from $L_{s\times k}$, where L_i denotes the transfer time of the i^{th} stripe. T means the total transfer time of all of the s stripes.

Based on the main idea in §3.3, we design three algorithms that fully utilize both intra-stripe parallelism and inter-stripe parallelism to accelerate disk repair in the HDSS.

4.2 Active Partial Stripe Repair

First, we propose active PSR that determines P_a and P_r by actively obtaining $L_{s\times k}$ and operating on it. To obtain the $L_{s\times k}$, we need to actively test the transfer speed of disks in advance. The test method is that we first read a small amount of data on a certain disk (e.g., 1KiB of data) and measure the read time. Divide the amount of data by the read time to get the current transfer speed of the disk. Then we divide the chunk size by the transfer speed of disks to obtain the $L_{s\times k}$.



(a) Step 1

(b) Step 2

Figure 5: An example of twice dimensionality reduction algorithm

4.2.1 Active Preliminary Algorithm: HD-PSR-AP.

Main idea: The specific objective of HD-PSR-AP is to obtain a reasonable P_a (or P_r) that can minimize the total transfer time T based on the input $L_{s\times k}$. Since the algorithm input $L_{s\times k}$ is a two-dimensional array and the algorithm output T is a value, our algorithm needs to reduce the dimensionality of the $L_{s\times k}$ twice to get T. We elaborate the twice dimensionality reduction algorithm to simulate the transfer process of stripes repair and compute the total transfer time T.

Details: The algorithm is specified by the following steps, the first two of which are illustrated in Figure 5.

Step 1 (First dimensionality reduction based on P_a):

From the parallel property in PSR, it is known that the time for P_a chunks of a stripe to be transferred is determined by the slowest chunk. For example, the transfer time of the first P_a chunks of the first stripe is the maximum value of $L[0][0:P_a]$. According to P_a, it is obvious that a stripe requires $\lceil n/k \rceil$ repair rounds to be repaired. The time required for the i^{th} read of the j^{th} stripe is the maximum value of $L[j][i\times P_a : (i+1)\times P_a]$. Thus, the transfer time of the j^{th}

Algorithm 1 HD-PSR-AP

Input: $L_{s\times k}$, a sorted two-dimensional array of transfer time of all chunks, and c, the memory capacity in chunks.
Output: P_a, the intra-stripe parallelisms chosen by HD-PSR-AP that can minimize the total transfer time.

```
1:  Pa ← 0, Tmin ← MAXFLOAT, Ls ← ∅
2:  for i = 2 to k do
3:      vPa = i, vPr = ⌈c/vPa⌉
4:      for j = 1 to s do
5:          t = 0
6:          for m = 1 to ⌈k/vPa⌉ do
7:              Max = max(L[m][m × vPa : (m + 1) × vPa])
8:              L = L + Max
9:          end for
10:         Ls = Ls + L
11:     end for
12:     T ← 0
13:     for j = 1 to s − vPr do
14:         Min = min(Ls[j : j + vPr])
15:         for m = j to j + vPr do
16:             Ls[m] = Ls[m] − Min
17:         end for
18:         T = T + Min
19:     end for
20:     if Tmin > T then
21:         Tmin = T
22:         Pa = vPa
23:     end if
24: end for
25: return Pa
```

stripe can be formulated as:

$$L_j = \sum_{i=0}^{\lceil k/P_a \rceil} \max_{i\times P_a \le f < (i+1)\times P_a} L[j][f] \tag{4}$$

Furthermore, in order to easily obtain the maximum value of $L[j][i\times P_a : (i+1)\times P_a]$, we can sort each row of $L_{s\times k}$ in ascending order in advance. Thus, the maximum value of $L[j][i \times P_a : (i + 1) \times P_a]$ is equivalent to $L[j][(i + 1) \times P_a - 1]$. In this way, the 2D array $L_{s\times k}$ is dimensionally reduced to a 1D array L_s containing s elements. The i^{th} element $L[i]$ represents the total transfer time of i^{th} stripe.

Step 2 (Second dimensionality reduction based on P_r):

P_r indicates that only P_r stripes can be processed in memory at the same time. We consider the process of stripes being transferred as follows. First, the memory will partition itself into P_r intervals, and each interval is responsible for keeping a stripe. After the stripe kept in an interval is repaired, the interval will select the next stripe from the stripes waiting queue to continue repairing. Finally, the repair ends when both the memory and stripes waiting queue are empty.

According to the above procedure, we simulate memory with a sliding window of size P_r and L_s simulates the stripes waiting queue. We initialize the value of T to 0. The first repaired stripe in memory is the stripe corresponding to the smallest value in the sliding window. For the convenience of obtaining the smallest L_i in sliding window, we sort the L_s array in ascending order. Then if the sliding window is at $L[i : j]$, the minimum value of $L[i : j]$ is $L[i]$. All values in the sliding window are subtracted each time from $L[i]$. Then we add the minimum value with T, and the sliding window slides back one value. Repeat the above procedure until the sliding window reaches the last value of L_s. Finally, we add T to the maximum value in sliding window to obtain the total transfer time, then the algorithm finishes.

Step 3 (Compute the final result by traversing):

Based on the above algorithm, we can calculate the total transfer time T for any given $L_{s\times k}$ and P_a(or P_r). Therefore, we can simply traverse P_a from 2 to k to calculate T based on the twice dimensionality reduction algorithm each time. Ultimately, the minimum T is the smallest transfer time, and we can obtain P_a and P_r that achieve the minimum T.

Algorithm: Algorithm 1 shows the algorithmic details of HD-PSR-AP. We traverse P_a from 2 to k to find the minimal transfer time T (lines 2-3). For each stripe, we need $\lceil k/vP_a \rceil$ repair rounds (line 6). We get the transfer time Max of each repair round and sum them as L(line 7-8). Then we add this element to the L_s array (line 10). We obtain the 1D array L_s until all stripes have been dimensionally reduced. Then we define a sliding window whose size is vP_r. The sliding window starts at the first vP_r elements of L_s and chooses the minimum value Min in the sliding window (line 14). And we let the remaining elements in the sliding window subtract Min (lines 15-17), and then let T add Min (line 18). Repeat the above process until the sliding window is empty. Finally, we select the P_a when T takes the minimum value in the $k-1$ cycles (lines 20-23).

Time complexity: For the first step, we sort each row of $L_{s\times k}$. Since there are k elements in each row, the time complexity of each sorting is $O(k \times log(k))$. $L_{s\times k}$ has a total of s rows, so the time complexity of sorting is $O(s\times k\times log(k))$. Then, all the $\lceil n/Pa \rceil$ maximum values need to be located in each row, and the time complexity of determining the maximum value each time is $O(1)$ (because every row has been sorted). Therefore, the time complexity of first dimensionality reduction for each row is $O(\lceil n/Pa \rceil)$ and the first dimensionality reduction of $L_{s\times k}$ is $O(s \times \lceil n/Pa \rceil)$. Thus, the overall time complexity of the first dimensionality reduction is $O(s \times k \times log(k)) + O(s \times \lceil n/Pa \rceil)$.

For the second step, we sort the L_s array at first, and the time complexity is $O(s\times log(s))$. Then, a total of $s-Pr$ cycles are required to find the minimum value within the sliding window each time (time complexity of searching for minimum value is $O(1)$). Hence, the overall time complexity of the second dimensionality reduction is $O(s \times log(s)) + O(s - Pr)$.

For the third step, we traverse the above procedure $k-1$ times to obtain the minimized result. Overall, the time complexity of HD-PSR-AP algorithm is $O(s \times k \times log(k) \times (k-1)) + O(s \times \lceil k/Pa \rceil \times (k-1)) + O(s \times log(s) \times (k-1)) + O((s - Pr) \times (k-1))$. If there are enormous stripes to be repaired (i.e., s is very large) and the k is relatively small, the time complexity is $O(s \times log(s) \times k)$. If the k is large (e.g., $k = 128$ [13]) and the number of stripes to be repaired is small, the time complexity is $O(s \times k^2 \times log(k))$.

Note that although HD-PSR-AP can obtain the optimal transfer time T, but its time complexity is quite high which incurs high

Algorithm 2 HD-PSR-AS

Input: $L_{s\times k}$, a two-dimensional array of transfer time of all chunks, and c, the memory capacity in chunks.
Output: P_a, the intra-stripe parallelisms chosen by HD-PSR-AS that can minimize the total transfer time.

```
1:  P_a ← 0
2:  for i = 0 to s do
3:      sp ← 0, fp ← 1, slow ← 0
4:      while fp < k do
5:          if L[i][fp] is a slower then
6:              swap(L[i][fp], L[i][sp])
7:              sp = sp + 1
8:              slow = slow + 1
9:          end if
10:         fp = fp + 1
11:     end while
12:     P_a = max(P_a, slow)
13: end for
14: P_a = max(min(P_a, k/2), 2)
15: return P_a
```

algorithm running time, so it may not be acceptable for fast disk recovery performance. This motivates us to design a heuristic algorithm that decreases the time complexity, which is proposed in §4.2.2.

4.2.2 Active Slower-First Algorithm: HD-PSR-AS.
Due to the high time complexity of HD-PSR-AP, we propose another solution called active slower-first PSR (HD-PSR-AS) to reduce the time complexity.

Main idea: We state that the essential reason for slow transfer speed and poor memory utilization is that the slow chunks (*slowers*) drag down the fast chunks ($fasters$). $Fasters$ are read into memory earlier, thus having to wait for *slowers*. This results in wasted memory space occupied by the $fasters$ during the waiting state. Therefore, the key insight of HD-PSR-AS is to try not to let $fasters$ be encumbered by *slowers*. To this end, HD-PSR-AS tries to arrange the transfer of $fasters$ and *lowers* separately. In this way, it can reduce the negative effect of *slowers* on $fasters$.

According to this idea, the primary point of HD-PSR-AS is that for the i^{th} stripe, we count the number of *slowers*, termed as $slow_i$, and let the maximum value among all $slow_i (0 \le i < s)$ be P_a. For practical use, we will take the minimum of P_a and $k/2$ to ensure that the value of P_r will not be too small, and we will also take the minimum of P_a and 2 to guarantee the value of P_r will not be too large. Therefore, P_a can be expressed as:

$$P_a = \max(\min(\max_{0\le i<s} slow_i, k/2), 2) \tag{5}$$

After determining the value of P_a, we arrange the *slowers* together by move operation and begin to transfer according to P_a.

Algorithm: Algorithm 2 shows the algorithmic details of HD-PSR-AS. Systematically, we use fast and slow pointers algorithm to count the number of *slowers* of each stripe and move the *slowers* together. The method is as follows: for the i^{th} stripe, assuming the number of *slowers* is $slow_i$, $slow_i$ will be initialized to 0 (line 3).

Algorithm 3 HD-PSR-PA

Input: D, the disk distribution of a stripe.
Output: P_{a1}, the first round intra-stripe parallelism, and P_{a2}, the second round intra-stripe parallelism.

```
1: P_a1 ← k, P_a2 ← 0
2: for i = 1 to |D| do
3:     if D[i] is slow then
4:         P_a1 = P_a1 + 1
5:         P_a2 = P_a2 - 1
6:     end if
7: end for
8: return P_a1, P_a2
```

First, we set slow pointer direct at the first chunk of i^{th} stripe and fast pointer point to the next chunk (line 3). Secondly, fast pointer judges whether the chunk it points to is a *slower* (line 5). If so, the chunk pointed by the slow pointer and the chunk pointed by the fast pointer are exchanged (line 6). The value of $slow_i$ is incremented by 1 (line 8). Fast pointer and slow pointer move backward one chunk at the same time (line 7 and line 10). If not, faster pointer moves backward one chunk and slow pointer remains stationary (line 10). The algorithm completes until the fast pointer has judged the last chunk. After the above process, all *slowers* are moved to the front of the i^{th} stripe. $slow_i$ also records the number of *slowers* in the stripe. Finally, to avoid P_a too small or too large, we take the minimum value of P_a and $k/2$ and then take the maximum value of P_a and 2 (line 14).

Time complexity: For each stripe, it only needs to loop k times to count the number of *slowers*. There are s stripes in total, thus the value of P_a can be confirmed by looping $s \times k$ times. Therefore, the time complexity of HD-PSR-AS is $O(s \times k)$. Obviously, HD-PSR-AS may not necessarily obtain the minimizing result, but can significantly reduce the time complexity.

4.3 Passive Partial Stripe Repair

Active PSR can determine P_a and P_r by utilizing the $L_{s\times k}$. Therefore, both HD-PSR-AP and HD-PSR-AS have to obtain $L_{s\times k}$ through active testing. However, active testing the transfer speed of the disks will occupy the CPU and memory resources. Thus, it is necessary for CPU/memory constrained cases to design a passive PSR based repair scheme that does not have to actively detect current disk transfer speed.

Main Idea: We further propose a passive algorithm HD-PSR-PA that will obtain the transfer speed of the disks passively. HD-PSR-PA adopts the conventional repair (i.e., FSR) by default and sets a timer when reading a chunk. When the time to read a chunk exceeds a certain threshold, HD-PSR-PA will mark the disk as *slow*. If a *slow* disk is encountered while repairing a stripe, HD-PSR-PA will reconstruct the stripe in two repair rounds. In the first round, HD-PSR-PA reads all chunks on the *slow* disks to memory. In the second round, HD-PSR-PA reads other chunks to memory. This will achieve the purpose of not letting the slow chunks drag down the fast chunks.

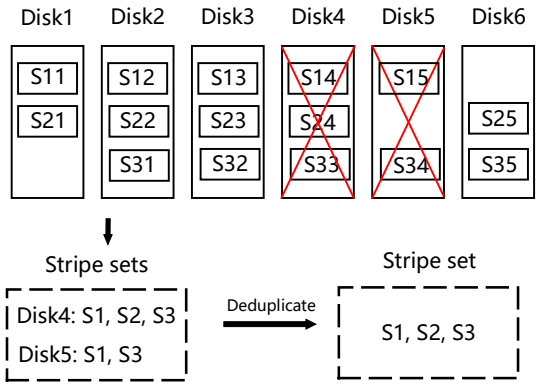


Figure 6: Example of multi-disk failures.

Algorithm: Algorithm 3 shows the algorithmic details of HD-PSR-PA. HD-PSR-PA will utilize FSR by default. In other words, HD-PSR-PA will assign the first round P_a (P_{a1}) to k and the second round P_a to 0 initially (line 1). However, if HD-PSR-PA finds that the stripe's disk distribution contains s *slow* disks, then it will adjust P_{a1} to $P_{a1} - s$ and P_{a2} to s (lines 2-6).

Time complexity: For each stripe, we need to traverse each disk of the stripe to obtain the number of *slow* disks. Thus the overall time complexity of HD-PSR-PA is $O(s \times k)$.

4.4 Cooperative Multi-Disk Repair

We propose a cooperative multi-disk repair scheme for multiple disk failures. We aim to reduce repetitive computations and extra disk I/Os which occur if the multiple disk failures make some stripes lose multiple chunks.

If we simply repair each failed disk individually until all the failed disks are reconstructed (we call this *naive* method), there will be some unnecessary redundant computations and disk I/Os. Figure 6 shows an example of multi-disk failures in an HDSS. We assume that $(n, k) = (5, 3)$ and the number of disks is six. There are three stripes in the server S_1, S_2 and S_3, and $S_{i,j}$ denotes the j^{th} chunk of the i^{th} stripe. We can see the three stripes' distributions on disks. We assume that $Disk_4$ and $Disk_5$ fail. If we adopt the naive method, then we need to repair $Disk_4$ and $Disk_5$ separately, which will cause 15 chunks to be read. Specifically, we first repair $Disk_4$ by reading 9 chunks, composed of all the 8 chunks of $Disk_1$, $Disk_2$ and $Disk_3$, and the last chunk S_{25} from $Disk_6$; we then repair $Disk_5$ by reading 6 chunks, composed of S_{11}, S_{12}, S_{13} and S_{31}, S_{32}, S_{35}. However, we find that the repair of $stripe_1$ and $stripe_3$ is repeated twice and we only need to read 9 chunks to repair $Disk_4$ and $Disk_5$.

Thus, we propose a cooperative multi-disk repair scheme to minimize redundant computations and disk I/Os. We create an array called *stripe set* for each disk that stores the collection of stripes kept on that disk. Figure 6 shows that when we detect that some disks fail, we collect the *stripe sets* of those disks together. Then we join these sets together and remove the duplicate elements. Then we have the minimum number of stripes that need to be repaired to reconstruct all the failed disks. Then we use HD-PSR to repair each of the stripes and write back the failed chunks to backup disks respectively.

5 EVALUATION

We describe HD-PSR's implementation (§5.1), provide the experiment setup and configurations (§5.2), and show the experiment results (§5.3).

5.1 Implementation

We implement HD-PSR as a prototype designed for the HDSS with about 4000 SLoCs in Golang on Linux. Specifically, HD-PSR mainly includes the following two modules.

Encoding Module: The encoding module (about 220 SLoCs) implements (n, k) codes based on RS codes. We implement the coding operations based on the RS coding module called *reedsolomon* [3], which is implemented in pure Golang. The encoding module mainly utilizes two *reedsolomon* APIs: *Encoder.Split*, which splits the raw data into k equal-sized splices, and *Encoder.Encode*, which encodes the split data into n splices, including k data splices and $m = n - k$ parity splices.

Repair Module: The repair module (about 1300 SLoCs) implements three algorithms, including FSR (about 230 SLoCs), HD-PSR-AP (about 430 SLoCs), HD-PSR-AS (about 280 SLoCs), and HD-PSR-PA (about 270 SLoCs). To show HD-PSR's performance improvements, we implement HD-PSR based on the *reedsolomon* API: *Encoder.reconstruct*, which can reconstruct any failed chunk using k surviving chunks. To enable PSR, we extend *reedsolomon* and implement an API called *Encoder.RecoverWithSomeShards*, which can repair intermediate chunks based on the partial chunks of the stripe.

5.2 Experiment Setup

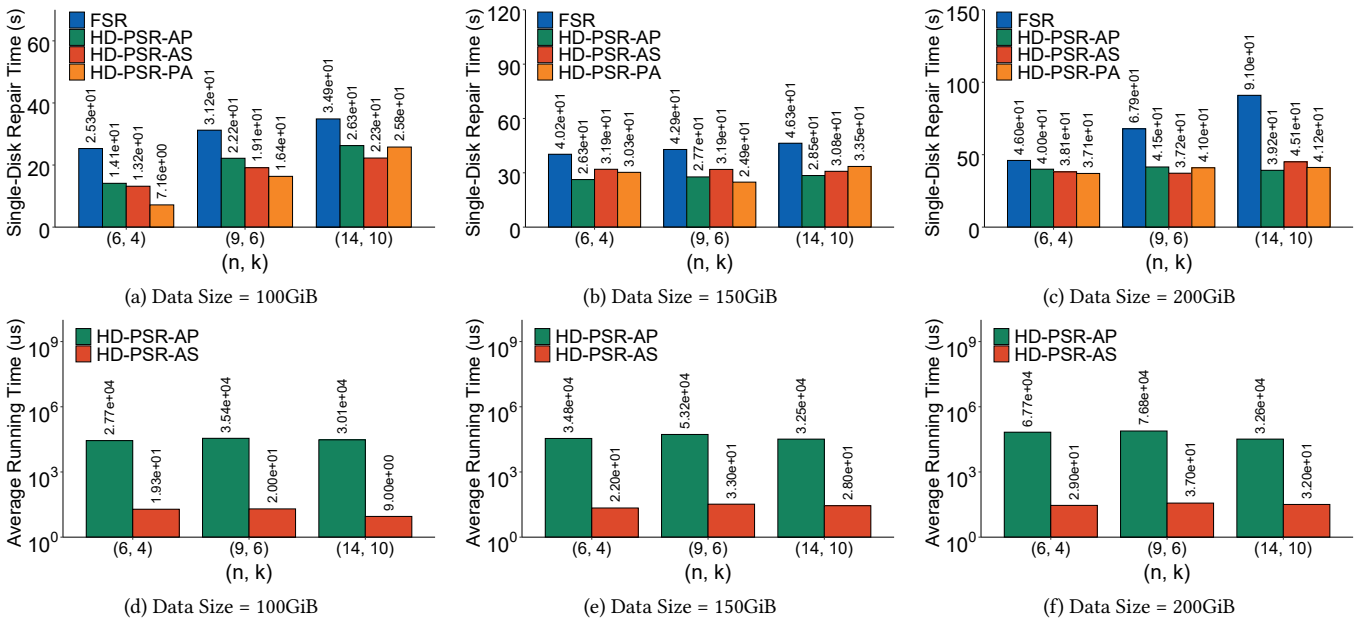
We conduct our experiments on Amazon EC2 with a d3en.12xlarge instance in US East (North Virginia) region. The instance is equipped with 36 3.7TiB SATA disks, and runs Ubuntu 16.04. We create 36 directories on the root directory and mount the 36 disks on these directories.

We set the configurable parameters (n, k) of RS codes to the following values: $(6, 4)$ (a widely-deployed RAID6 setting), $(9, 6)$ (deployed in QFS [23]), $(14, 10)$ (deployed in Facebook [22]). We set the chunk size as 64MiB by default in coding like GFS [12].

We conduct our experiments under different amount of data to be repaired. We measure the time required for FSR, HD-PSR-AP, HD-PSR-AS, and HD-PSR-PA to repair disks, each of which has capacities of 100GiB, 150GiB, and 200GiB, respectively. We report average results of each experiment over five runs.

5.3 Experiments

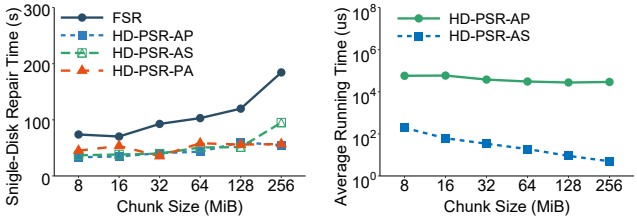
Experiment 1 (Overall single-disk repair time versus (n, k)): We measure the overall single-disk repair time consumed by FSR, HD-PSR-AP, HD-PSR-AS and HD-PSR-PA with different size of data under different parameters (n, k) respectively. Figure 7(a)-(c) show the overall single-disk repair time for FSR, HD-PSR-AP, HD-PSR-AS and HD-PSR-PA. Overall, all of HD-PSR-AP, HD-PSR-AS and HD-PSR-PA have less single-disk repair time than FSR. For



(a) Data Size = 100GiB (b) Data Size = 150GiB (c) Data Size = 200GiB

(d) Data Size = 100GiB (e) Data Size = 150GiB (f) Data Size = 200GiB

Figure 7: Experiment 1-2: Overall single-disk repair performance and average algorithm running time of FSR, HD-PSR-AP, HD-PSR-AS **and** HD-PSR-PA **for different** (n, k) **under different size of the failed disk.**



(a) Overall repair time (b) Average running time

Figure 8: Experiment 3-4: The overall repair time and the average algorithm running time versus chunk size.

example, when (n, k) = (14, 10) and the size of data of the failed disk is 200GiB, compared to FSR, HD-PSR-AP and HD-PSR-AS can reduce the single-disk repair time by up to 56.9% and 50.46%, respectively. Note that HD-PSR-PA can reduce the single-disk repair time by up to 71.7%, which performs the best, when $(n, k) = (6, 4)$ and the size of data of the failed disk is 100GiB.

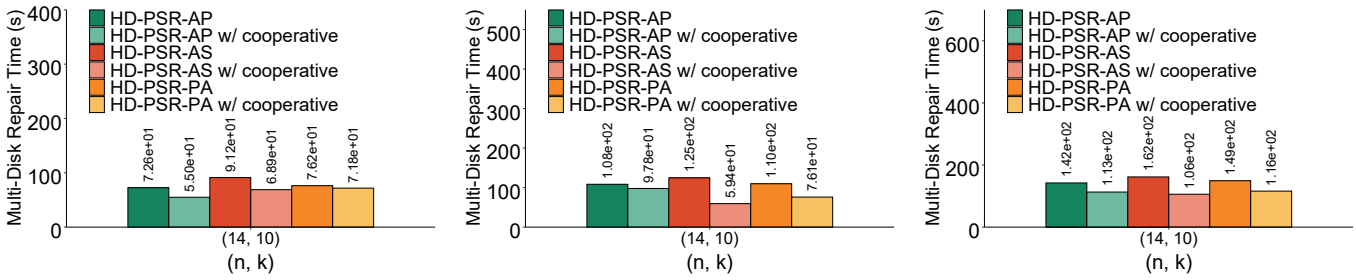
We can see that the single-disk repair time increases rapidly when k becomes larger. In contrast, the single-disk repair time growth of HD-PSR-AP, HD-PSR-AS and HD-PSR-PA is relatively slow. The reason is that an increase in k causes more chunks to be in the waiting state in the memory, resulting in a higher $ACWT$ and lower memory utilization (see §3). On the other hand, Figure 7(a)-(c) show that the larger the k is, the greater the reduction in HD-PSR for the overall repair time we have. The reason is that the larger k makes the higher $ACWT$ of FSR and the lower memory utilization, which leaves more potential for improvement in HD-PSR.

Experiment 2 (Average algorithm's running time versus (n, k)): We measure the algorithm's running time of HD-PSR-AP and HD-PSR-AS under different parameters (n, k) and different number of stripes. Since we set the chunk size as 64MiB, the number of stripes will grow with the increasing data size. The algorithm's running time is the time required to derive P_a. Here, we only need to measure the algorithm running time of HD-PSR-AP and HD-PSR-AS, because HD-PSR-PA that is based on a passive way does not need to derive P_a in advance (see §4.3), so the algorithm's running time of HD-PSR-PA is 0.

Note that although HD-PSR-PA has no running time, its overall repair time may not always the smallest (e.g., $(n, k) = (14, 10)$ and the size of data of the failed disk is 100GiB), since its passive way may not show the current disk status as well as the other two active ways.

Figure 7(d)-(f) show that the average running time of HD-PSR-AP and HD-PSR-AS are not in the same order of magnitude. Specifically, HD-PSR-AS reduces the total running time by an average of 98% compared to HD-PSR-AP. The result is consistent with their time complexities, i.e., $O(s \times log(s) \times k)$ for HD-PSR-AP and $O(s \times k)$ for HD-PSR-AS. Therefore, HD-PSR-AS is much more practical than HD-PSR-AP in repairing tremendous number of stripes.

Experiment 3 (Overall single-disk repair time versus chunk size): We measure the overall single-disk repair time versus different chunk sizes. We set the size of the failed disk as 200GiB and the (n, k) as (9, 6). We vary the chunk size ranging from 8MiB to 256MiB. Figure 8(a) shows the overall repair time of FSR, HD-PSR-AP, HD-PSR-AS and HD-PSR-PA. The overall repair time grows with the increasing chunk size. We can see that all of HD-PSR-AP,



(a) Number of Failed Disks = 2 (b) Number of Failed Disks = 3 (c) Number of Failed Disks = 4

Figure 9: Experiment 5: Overall multi-disk repair time of FSR, HD-PSR-AP, HD-PSR-AS and HD-PSR-PA under different (n, k) for different number of failed disks.

HD-PSR-AS and HD-PSR-PA maintain the advantages in different situations.

Experiment 4 (Average algorithm running time versus chunk size): Similar to Experiment 2, we only measure the average algorithm running time of generating the P_a for HD-PSR-AP and HD-PSR-AS, without HD-PSR-PA. Figure 8(b) compares the average running time of HD-PSR-AP and HD-PSR-AS and the chunk size is varied from 8MiB to 256MiB. We set the size of data to be repaired as 200GiB. We can see that the average running time decreases when the chunk size becomes larger. The reason is that we set the size of data to be repaired to a fixed value. Then the number of stripes will become smaller when the chunk size becomes larger. This will cause the factor s in the time complexity to become larger, which makes the running time of HD-PSR-AP and HD-PSR-AS longer.

Experiment 5 (Overall multi-disk repair time versus (n, k)): We measure the overall multi-disk repair time for HD-PSR without and within cooperative multi-disk repair under $(n, k) = (14, 10)$. We set the size of data of each failed disk as 200GiB. We can see from Figure 9 that all of HD-PSR-AP, HD-PSR-AS and HD-PSR-PA with the cooperative multi-disk repair effectively reduce the overall multi-disk repair time compared to those without cooperative multi-disk repair. Specifically, compared to the schemes without the cooperative multi-disk repair, the reduction of multi-disk repair time of cooperative HD-PSR-AP can be up to 24.2% (when the number of failed disks is two), that of cooperative HD-PSR-AS can be up to 52.5% (when the number of failed disks is three), and that of cooperative HD-PSR-PA can be up to 30.8% (when the number of failed disks is three).

6 RELATED WORK

Erasure coding in traditional RAID storage. There are plentiful literatures on improving erasure coding on traditional RAID storage [26], including improving the performance of coding operations and minimizing the disk I/O for recovery and degraded reads. For optimizing the coding operations, RDP codes [8] and EVENODD codes [6] are two of the most widely used RAID-6 codes [10], which both use the exclusive-or operations to replace the expensive multiplication of Galois Fields. STAR codes [15] and generalized EVENODD codes [7] provide higher fault tolerance than RDP codes [8] and EVENODD codes [7]. For minimizing the disk I/O for recovery and degraded reads, Xiang et al. [35] design algorithms that can reduce up to 25% of symbols to be read. However, this algorithm lacks generality because it is designed for RDP codes [8] and EVENODD codes [6]. Rotated codes proposed by Khan et al. [16] establishes a shortest path model, which can generate recovery schemes that require minimal amount of read data for any given erasure code. Luo et al. [20] improve Khan's algorithm by allowing the most heavily loaded disks take the least load. Fu et al. [11] propose a search-based algorithm which can find the near-optimal scheme with the minimized retrieved symbols on the heaviest loaded disks.

However, an HDSS has many more disks than the traditional RAID storage server [26], and thus enables multiple stripes to parallel the disk failure recovery, yet incurring a new memory competition challenge which is unique to the HDSS. Existing research has not proposed an effective solution to exploit parallelism among multiple stripes.

Repair in erasure-coded storage. A rich body of literatures focus on improving the repair performance in erasure-coded storage. Some of them construct new repair-efficient codes. Regenerating codes [9, 24, 28, 30] mitigate the repair bandwidth by allowing more surviving nodes to provide computing power. Locally repairable codes [14, 17, 25, 32] reduce the repair I/O by adding local parity chunks. Hu et al. [13] propose combined locality that exploit parity locality and topology locality to address the repair problem for stripes with a very large size. Others design new repair strategies for practical erasure codes. PPR [21] divides the repair task into some smaller computing tasks and complete these tasks in some surviving nodes in parallel. PPR [21] reduces the repair time complexity from $O(k)$ to $O(logk)$. RP [18] fully utilizes the bandwidth resources by pipelining the repair of a chunk in units of slices, so that the time complexity approaches to $O(1)$. PPT [5] construct a tree-like pipelined path in which the slowest link will be replaced by another two links with the same receiver, which has a better repair performance than RP [18]. SMFRepair [36] utilizes the idle nodes outside the stripe to bypass the slowest link to improve the repair performance without leading to network congestion and competition. ECWide [13] improves the single-node repair by always selecting the requestor and helpers with the least load such that it is less likely that the multiple concurrent single-chunk repairs

make some nodes heavily loaded. RepairBoost [19] accelerates the single-node repair via a directed acyclic graph model to balance the repair traffic among concurrent multiple single-chunk repairs.

However, all of these studies are mainly designed for distributed storage, whose bottleneck is basically the limited network resources, while in an HDSS, data is transmitted over the bus, whose bandwidth is rarely the bottleneck of the system. Therefore, our method HD-PSR is the first to re-think how to apply repair parallelism from distributed storage to the HDSS.

7 CONCLUSIONS

We propose HD-PSR for a high-density storage server, an erasure-coded repair mechanism that exploiting parallelisms to accelerate disk failure recovery. We elaborate HD-PSR with three algorithms HD-PSR-AP, HD-PSR-AS, and HD-PSR-PA, which realize efficient single-disk repairs. We further propose a cooperative repair strategy to improve multi-disk recovery performance. We prototype HD-PSR and the Amazon EC2 experiments demonstrate the repair efficiency of HD-PSR in erasure-coded high-density storage.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (No. 61872414), Key Laboratory of Information Storage System Ministry of Education of China. The corresponding author is yuchonghu@hust.edu.cn.

REFERENCES

[1] 2014. The cost savings of high-density data center environment. https://datacenterpost.com/cost-savings-high-density-data-center-environments/.
[2] 2021. Openstack Swift Object Storage Service. http://swift.openstack.org.
[3] 2021. Reed-Solomon Erasure Coding in Go. https://pkg.go.dev/github.com/klauspost/reedsolomon.
[4] 2022. Dell high-density storage. https://www.dell.com/en-hk/shop/enterprise-products/powervault-md1220/spd/powervault-md1220.
[5] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. 2019. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network. In *Proc. of ICPP*. 1–10.
[6] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. on Computers* 44, 2 (1995), 192–202.
[7] Mario Blaum, Jehoshua Bruck, and Alexander Vardy. 1996. MDS array codes with independent parity symbols. *IEEE Trans. on Information Theory* 42, 2 (1996), 529–542.
[8] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. 2004. Row-diagonal parity for double disk failure correction. In *Proc. of USENIX FAST*. San Francisco, CA, 1–14.
[9] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. 2010. Network coding for distributed storage systems. *IEEE Trans. on Information Theory* 56, 9 (2010), 4539–4551.
[10] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. 2009. DiskReduce: RAID for data-intensive scalable computing. In *Proc. of PDSW*. 6–10.
[11] Yingxun Fu, Jiwu Shu, and Xianghong Luo. 2014. A stack-based single disk failure recovery scheme for erasure coded storage systems. In *Proc. of IEEE SRDS*. IEEE, 136–145.
[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proc. of ACM SOSP*. 29–43.
[13] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. 2021. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proc. of USENIX FAST*. 233–248.
[14] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proc. of USENIX ATC*. 15–26.
[15] Cheng Huang and Lihao Xu. 2008. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Trans. on Computers* 57, 7 (2008), 889–901.
[16] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads.. In *Proc. of USENIX FAST*, Vol. 20. 2208461–2208481.
[17] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. 2020. On fault tolerance, locality, and optimality in locally repairable codes. *ACM Trans. on Storage* 16, 2 (2020), 1–32.
[18] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. 2017. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*. 567–579.
[19] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick PC Lee, and Jiwu Shu. 2021. Boosting Full-Node Repair in Erasure-Coded Storage. In *Proc. of USENIX ATC*. 641–655.
[20] Xianghong Luo and Jiwu Shu. 2013. Load-balanced recovery schemes for single-disk failure in storage systems with any erasure code. In *Proc. of ICPP*. IEEE, 552–561.
[21] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-parallel-repair (ppr) a distributed technique for repairing erasure coded storage. In *Proc. of EuroSys*. 1–16.
[22] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*. 383–398.
[23] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The quantcast file system. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1092–1101.
[24] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandic. 2016. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*. 81–94.
[25] Dimitris S Papailiopoulos and Alexandros G Dimakis. 2014. Locally repairable codes. *IEEE Trans. on Information Theory* 60, 10 (2014), 5843–5855.
[26] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM SIGMOD*. 109–116.
[27] James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, et al. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage.. In *Fast*, Vol. 9. 253–265.
[28] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *Proc. of USENIX FAST*. 81–94.
[29] Korlakai Vinayak Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*.
[30] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. 2011. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Trans. on Information Theory* 57, 8 (2011), 5227–5239.
[31] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
[32] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. Xoring elephants: Novel erasure codes for big data. In *Proc. of ACM VLDB Endowment*.
[33] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.
[34] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*. 307–320.
[35] Liping Xiang, Yinlong Xu, John CS Lui, and Qian Chang. 2010. Optimal recovery of single disk failure in RDP code storage systems. *Proc. of ACM SIGMETRICS* 38, 1 (2010), 119–130.
[36] Hai Zhou, Dan Feng, and Yuchong Hu. 2021. Multi-level Forwarding and Scheduling Repair Technique in Heterogeneous Network for Erasure-coded Clusters. In *Proc. of ICPP*. 1–11.