Scale-RS: An Efficient Scaling Scheme for RS-Coded Storage Clusters

Jianzhong Huang, Xianhai Liang, Xiao Qin, *Senior Member, IEEE*, Ping Xie, and Changsheng Xie, *Member, IEEE*

Abstract—It is indispensable to scale erasure-coded storage clusters to meet requirements of increased storage capacity and I/O performance. In this study, we propose an efficient scaling scheme for Reed-Solomon-coded storage clusters called Scale-RS, which has three salient features. First, Scale-RS achieves uniform data distribution by equally placing data blocks among old and new chunks using a transposed data layout. Second, Scale-RS minimizes data movement incurred in the procedures of data redistribution and parity update. Scale-RS not only reaches the lower bound of data migration traffic by transferring necessary data blocks from old data chunks to new chunks, but it also reduces update traffic via generating parity difference blocks from data blocks stored in an individual data chunk. Third, Scale-RS improves the I/O performance of scaled storage clusters in terms of read parallelism and write throughput. We implement Scale-RS along with two alternative scaling schemes in a Reed-Solomon-coded storage cluster, on which real-world I/O traces are replayed. Experimental results demonstrate that Scale-RS achieves the highest read performance among the three scaling schemes after data redistribution. When it comes to scaling from six data chunks to nine, Scale-RS can outperform the other two scaling schemes in terms of aggregate write throughput by a factor of 2.85 and 3.05 under online filling and offline filling, respectively. We also show that user response time is slightly enlarged during data redistribution due to bandwidth competition between migration and user I/Os.

Index Terms—Erasure-coded storage cluster, cluster scaling, data redistribution, parity update

1 Introduction

It is important to scale storage systems to satisfy growing demands of storage capacity and I/O performance. In this study, we propose an efficient scaling scheme—*Scale-RS*—for Reed-Solomon-coded storage clusters. Scale-RS aims at boosting the I/O performance of scaled storage clusters in terms of read parallelism and write throughput.

1.1 Motivation

The following three factors motivate us to investigate scaling schemes for erasure-coded storage:

- high cost-effectiveness of erasure-code storage,
- dynamically changing application requirements, and
- urgent needs of scalability in storage capacity.

Motivation 1. Thanks to high fault tolerance, storage efficiency, as well as cost-effectiveness, erasure-coded storage has been widely used in cloud storage platforms [1], [2], [3], data centers [4], [5], [6], archive storage [7], [8], [9], and the like. Nowadays, popular erasure codes used in distributed storage include Reed-Solomon (RS) coding and parity array coding. RS coding [10], [11] supports higher fault-tolerance than XOR-based parity array coding (e.g., EVENODD [12], RDP [13], etc.) that tolerates less than three concurrent failures. For example, Windows Azure Storage (WAS) adopts a variant of RS coding to implement a four-fault-tolerant cluster system [3]; Facebook also implements RS coding in HDFS-RAID to tolerate four failures [4].

Motivation 2. A wide range of applications have various deployment requirements in terms of I/O performance, fault-tolerance, and storage efficiency. To meet a variety of application requirements, existing erasure-coded storage systems adopt different coding parameters (see k and r in Table 1). Ideally, coding parameters should be dynamically adjusted according to the changing access patterns of applications. For example, each map processes a single HDFS chunk in Hadoop; more chunks improve the map parallelism [14]. Increasing the k value improves read performance and storage efficiency, because (1) read performance is primarily affected by read parallelism and (2) storage efficiency k/(k + r) of (k + r, k) RS-coded storage increases with parameter k.

Motivation 3. It is necessary for a storage system to expand or shrink its storage volume by adding or removing storage devices. International Data Corporation (IDC) projects that the storage market will grow at a 53.4 percent compound annual growth rate (CAGR) between 2012 to 2016 [21]. The ever-growing amount of data requires highly scalable storage solutions [22]. Additionally, 'disk space utilization' is an important metric for storage systems [23]. It is needed to expand storage when disk space utilization reaches a predetermined threshold (e.g., 75 percent).

- *J. Huang, X. Liang, P. Xie, and C. Xie are the with Wuhan National Lab. for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China.*
  E-mail: {hjzh, cs_xie}@hust.edu.cn, {hustlxh, qhnuxp}@gmail.com.
- *X. Qin is with the Department of Computer Science and Software Engineering, Shelby Center for Engineering Technology, Samuel Ginn College of Engineering, Auburn University, AL 36849-5347.*
  E-mail: xqin@auburn.edu.

Manuscript received 7 Jan. 2014; revised 16 May 2014; accepted 16 May 2014. Date of publication 20 May 2014; date of current version 8 May 2015.
Recommended for acceptance by M.C. Chuah.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2326156

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.
1045-9219 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

TABLE 1
Comparisons of Erasure-Coded Distributed Storage

Storage System	$(k+r,k)$ Erasure Codes*
WAS [3]	a variant of Reed-Solomon coding $(k=12, r=4)$ [15]
GFS II [5], QFS [6]	Reed-Solomon encoding $(k=6, r=3)$
Facebook [4]	RS coding is used in HDFS-RAID $(k=10, r=4)$
HYDRAstor [16]	Cauchy-based Reed-Solomon Codes $(k=9, r=3)$
Tahoe-LAFS [9]	Vandermonde-based Reed-Solomon Codes [17]
Cleversafe [18], [19]	Cauchy-based Reed-Solomon Codes [20]

*k and r—number of data chunks and parity chunks, respectively.

Therefore, it is extremely vital to scale up storage clusters by adding new data nodes.

1.2 Data Layout of RS-Coded Storage

Fig. 1 illustrates how a data stream is physically stored using (k + r,k) RS encoding. A client collects data strips—each of which is usually 64 KB—into k 1 MB data buffers. When the buffers fill, it calculates an additional r parity blocks and sends all the $k+r$ blocks to $k+r$ different storage nodes. Each node writes a 1 MB block to disk, until the chunks on disk reach a maximum chunk size. Note that the default chunk size is 64 MB in GFS [5], HDFS [4] and QFS [6], and it is configurable (e.g., 128, 256, and 512 MB). At this point the client will have written k full data chunks and r parity chunks. If there is new data, the client will establish connections to $k+r$ new nodes and repeatedly perform the above process.

Both k data blocks and r parity blocks are exclusively stored in k data chunks $\{DC_1, DC_2, \ldots, DC_k\}$ and r parity chunks $\{PC_1, PC_2, \ldots, PC_r\}$, respectively. Both data and parity blocks in chunks are organized in a RAID-4-like data layout, thus forming a *$(k+r,k)$ RS-coded chunk group*. If a chunk is a whole node, then the erasure-coded storage cluster can be regarded as a centralized RAID like storage system. Each block can be partitioned into multiple strips, and the collection of k data strips and r associated parity strips is called a stripe [24]. For simplicity, we denote $k+r$ blocks as a stripe (see Fig. 2). Since any k of $k+r$ blocks in a stripe can recover the original k data blocks and each block is stored in a separate chunk, data are protected against the simultaneous failures of up to r chunks.

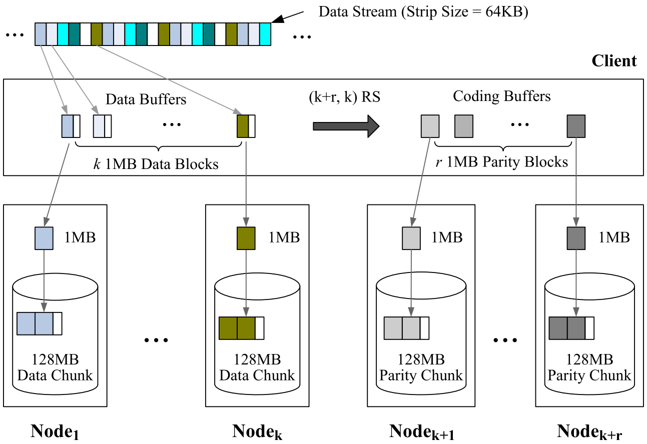


Fig. 1. Procedure of $(k+r,k)$ Reed-Solomon encoding.

1.3 Challenges and Strategies

Similar to *RAID scaling* [25], [26], [27], [28], [29], [30], we refer to node additions or removals in network-based erasure-coded storage clusters as '*cluster scaling*'. **In this study, 'cluster scaling$_{<k\rightarrow k+\Delta k>}$' means that Δk new nodes are placed into a cluster and Δk newly-created data chunks are stored on the new nodes**. We also denote a chunk group before scaling and after scaling by '*a source chunk group*' and '*a destination chunk group*', respectively. There are two types of cluster scaling: (1) scaling-up if $\Delta k > 0$ and (2) scaling-down if $\Delta k < 0$. We focus on the scaling-up case hereinafter, unless otherwise specified.

We are facing the following three challenges while designing a scaling scheme to expand erasure-coded chunk groups in storage clusters.

Challenge 1. How to maintain a uniform data distribution after scaling? If the uniformity of data distribution is violated, the initially balanced load will be served by a portion of data chunks. Load imbalance usually adversely degrades read performance.

Challenge 2. How to minimize data movement induced by data redistribution? To preserve the data distribution uniformity of destination chunk groups, it is inevitable to migrate data blocks among new and old chunks. Such data migrations cause updates of corresponding parity blocks, thereby yielding extra update traffic.

Challenge 3. How to improve the I/O performance of scaled storage clusters? A scaled storage cluster should exhibit high read parallelism, which in turn leads to improved read performance. Additionally, write throughput is an important requirement for data archival—one of in-production applications of erasure-code storage.

Because data and parity blocks stored in chunks are organized in a RAID-4-like layout, we should investigate the applicability of RAID scaling schemes for RS-coded chunk groups. Existing RAID scaling approaches focus on RAID-0 [25], [26], RAID-4 [31], RAID-5 [27], [28], [32], [33], [34], [35], and RAID-6 [29], [30], [36]. Different from RAID-0 scaling solutions that only address data migrations, cluster scaling must handle data migrations as well as parity updates. When RAID-4 scaling attempts to maintain a uniform data distribution across all data disks, a problem of large data

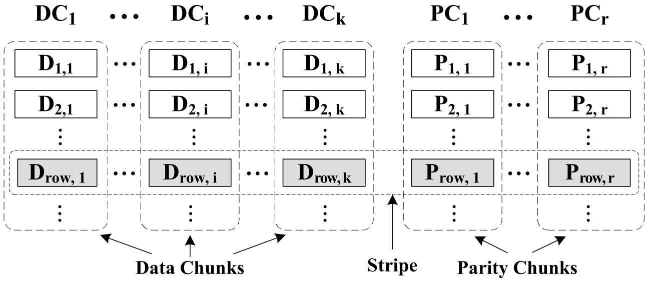
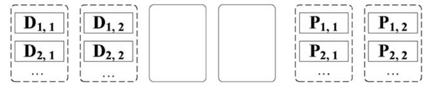
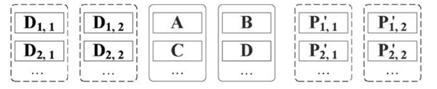
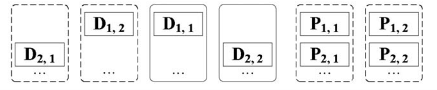
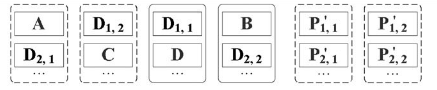
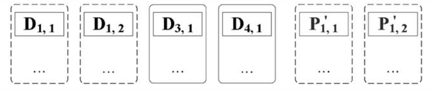
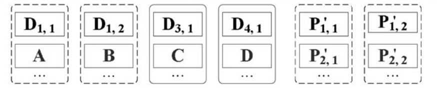
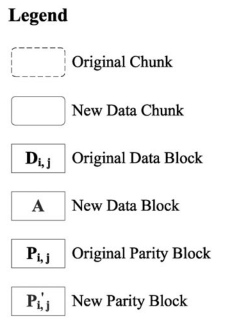


Fig. 2. Data layout of a (k + r,k) RS-coded chunk group, where k data blocks and r parity blocks are exclusively stored in k data chunks and r parity chunks.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Scaling Categories of Erasure-Coded Chunk Groups

Scaling Pattern	Stages	Diagram
No Data-Migration **(No-DM)**	**Redistribution:** No data migration happens, only zeroizing new data chunks on the new nodes.	 $D_{1,1}$ $D_{2,1}$ $D_{1,2}$ $D_{2,2}$ $P_{1,1}$ $P_{2,1}$ $P_{1,2}$ $P_{2,2}$
	Filling: New data is filled into the new data chunks, and all associated parity blocks are updated.	 $D_{1,1}$ $D_{2,1}$ $D_{1,2}$ $D_{2,2}$ A C B D $P'_{1,1}$ $P'_{2,1}$ $P'_{1,2}$ $P'_{2,2}$
Data-Migration w/o Parity-Update **(DM-Only)**	**Redistribution:** Data blocks are migrated along the same stripe, keeping the parity blocks untouched.	 $D_{2,1}$ $D_{1,2}$ $D_{1,1}$ $D_{2,2}$ $P_{1,1}$ $P_{2,1}$ $P_{1,2}$ $P_{2,2}$
	Filling: New data is filled into the scattered space on data chunks, thus the associated parity blocks are updated.	 A $D_{2,1}$ $D_{1,2}$ C $D_{1,1}$ D B $D_{2,2}$ $P'_{1,1}$ $P'_{2,1}$ $P'_{1,2}$ $P'_{2,2}$
Data-Migration w/ Parity-Update **(DM-PU)**	**Redistribution:** Old data blocks are migrated to new chunks, and the associated parity blocks are updated.	 $D_{1,1}$ $D_{1,2}$ $D_{3,1}$ $D_{4,1}$ $P'_{1,1}$ $P'_{1,2}$
	Filling: New data is written to data chunks in a full-stripe manner, then new parity blocks are written accordingly.	 $D_{1,1}$ A $D_{1,2}$ B $D_{3,1}$ C $D_{4,1}$ D $P'_{1,1}$ $P'_{2,1}$ $P'_{1,2}$ $P'_{2,2}$



migration occurs. Although an alternative RAID-4 scaling scheme can reduce the number of data blocks to be moved by migrating certain blocks to newly added disks [31], the RAID-4 scaling scheme suffers from a very expensive cost of parity updates during new data filling. Traditional RAID-5 scaling schemes are based on round-robin layout. If these RAID-5 scaling schemes are deployed in RS-coded storage clusters, then the clusters will inevitably encounter high I/O overhead incurred by data migrations and parity updates. The parity layout of each RAID-6 code is unique; RAID-6 scaling techniques designed for a specific RAID-6 code are inadequate for RS-coded storage clusters. In short, there is a wide application gap between RAID scaling schemes and cluster scaling.

In this study, we design an efficient cluster scaling scheme called Scale-RS, which exploits the structural properties of erasure codes to optimize both data migrations and parity updates. Scale-RS not only makes the total number of moved data blocks equal to the lower bound of data migration traffic, but it also minimizes the data movement induced by parity updates.

1.4 Roadmap

The remainder of this paper is organized as follows. Section 2 summarizes the preliminaries of this study. The design of Scale-RS is detailed in Section 3. Section 4 describes the experimental settings and results. Section 5 surveys the related work of storage scaling schemes. Section 6 discusses implementation issues. Finally, we conclude our work in Section 7.

2 Preliminaries

Recognizing that RS codes are widely deployed in in-production storage systems (see Table 1), we investigate scaling schemes for RS-coded storage clusters in this study.

2.1 Update Penalty in RS-Coded Storage Clusters

There is an update penalty issue in erasure-coded storage systems [37], [10], [38]—modifying any data block leads to updates of corresponding parity blocks. There exist two updating methods, namely, reconstruction write (a.k.a., RCW) [39] and read-modify-write (a.k.a., RMW) [40]. In the 'RMW' updating method, parity block $\mathrm{P_{row,j}}$ in parity chunk $\mathrm{PC_j}$ is updated to $\mathrm{P_{row,j}} + \Delta\mathrm{P_{row,j}}$ when data block $\mathrm{D_{row,i}}$ in data chunk $\mathrm{DC_i}$ is changed to $\mathrm{D'_{row,i}}$, where parity difference block $\Delta\mathrm{P_{row,j}}$ equals to $\alpha_{\mathrm{j,i}}(\mathrm{D'_{row,i}} - \mathrm{D_{row,i}})$, and coefficient $\alpha_{\mathrm{j,i}}$ denotes an element at the jth row and the ith row of a redundancy matrix.

In *cluster scaling*$_{<k \rightarrow k+\Delta k>}$, all *r* parity chunks should be updated when Δk new data chunks join a $(\mathrm{k}+\mathrm{r},\ \mathrm{k})$ RS-coded chunk group. If Round-Robin method [26], [34] is adopted, almost all data blocks will be migrated, and all parity blocks must be regenerated using the encoding procedure. Therefore, we should consider the following two issues when designing the Scale-RS scheme: (1) how to minimize the number of migrated data blocks, and (2) how to reduce the data movement induced by parity updates.

2.2 Categories of Cluster Scaling

One cluster scaling process can be logically divided into two stages: data redistribution and data filling. Data redistribution is a composite operation—apart from migrating data blocks, all associated parity blocks should be updated accordingly. Therefore, there exist the following three cluster-scaling patterns from a methodological point of view (see Table 2):

(1) *No* D*ata-*M*igration* (*No-DM*). No data migration is involved in the *No-DM*-based scaling scheme, which only places Δk new nodes into a storage cluster, resulting in zero migration overhead. In the data filling stage, new data is filled into new chunks and associated parity blocks are updated.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

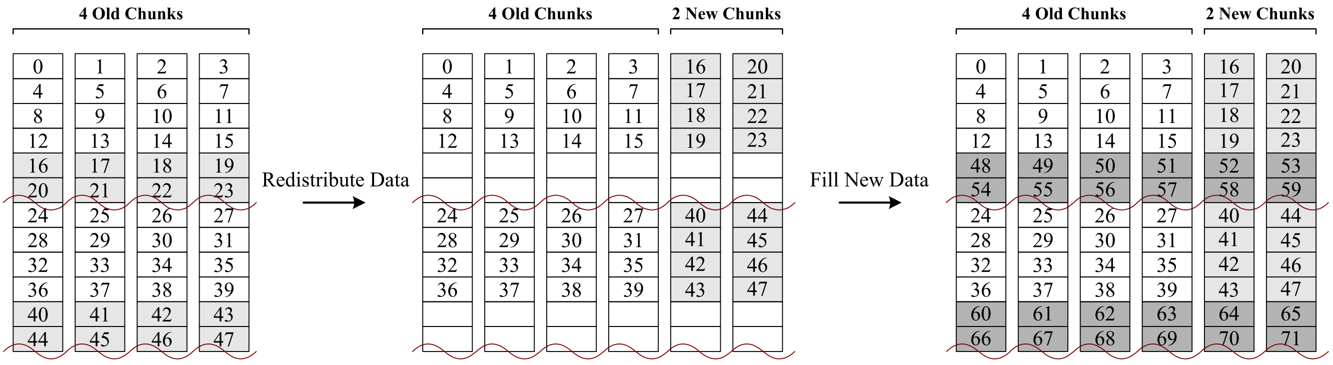


Fig. 3. Cluster scaling from four data chunks to six using Scale-RS. Each data block is a basic unit in both data redistribution and data filling stages.

(2) Data-Migration *Only* (*DM-Only*). The *DM-Only*-based scaling scheme moves data blocks within the same stripe from old data chunks to new data chunks (just like McPod [31]), thereby avoids modifying parity blocks after writing mapping metadata. When new data is filled into the scattered space among all data chunks, the associated parity blocks are updated.

(3) Data-*M*igration with *P*arity-*U*pdate (*DM-PU*). When data blocks are migrated from old data chunks to new data chunks, associated parity blocks are to be updated. In the data-filling stage, new data is written to all data chunks stripe by stripe (i.e., full-stripe write), and new parity blocks are directly written to the parity chunks.

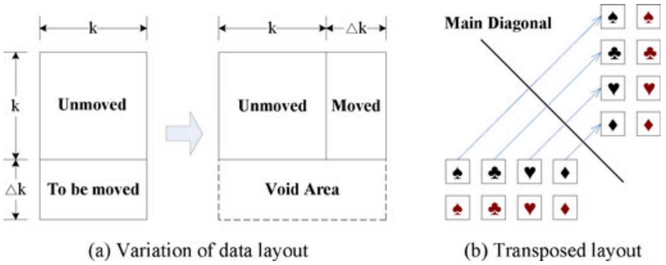
The *No-DM* pattern causes load imbalance due to non-uniform distribution of user I/Os among all data chunks (old and new). Additionally, compared to *DM-PU*, both *No-DM* and *DM-Only* patterns suffer from higher update penalty after the data redistribution stage. Therefore, Scale-RS takes the *DM-PU* pattern.

3 Scale-RS: An Efficient Scaling Scheme

3.1 Overview

In the *cluster scaling*$_{<\mathrm{k}\rightarrow\mathrm{k}+\Delta\mathrm{k}>}$ $\mathrm{k}+\Delta\mathrm{k}$ sequential stripes are grouped into one *scaling region*. As shown in Fig. 3, different scaling regions are separated with a wavy line. For each scaling region, the ways to data redistribution and data filling are identical.

Scale-RS moves only enough data blocks from old data chunks to new chunks, thereby avoiding data migration among old data chunks. As shown in Fig 4a, the data space in a scaling region can be divided into three zones: original and unmoved data, original and moved data, and new data. The zone of unmoved original data blocks is a square of side length k. New data blocks are filled into the void area according to the new striping rule of destination chunk groups. The void area is a rectangle of height $\Delta\mathrm{k}$ and width $\mathrm{k}+\Delta\mathrm{k}$.



(a) Variation of data layout (b) Transposed layout

Fig. 4. Data layout in scaling regions before scaling and after scaling.

There are several steps involved in parity updates, including retrieving moved data blocks, reading parity blocks, calculating new parity blocks, and writing resulting parity blocks. To reduce the data movement induced by parity updates, Scale-RS adopts a transposed data layout to redistribute the moved original data blocks (see Fig. 4b). According to the transposed data layout, Scale-RS can generate associated parity difference blocks from data blocks in an individual data chunk rather than that in multiple data chunks, thus optimizing the step of 'retrieving moved data blocks'.

3.2 The Data Migration Stage

The notation summarized in Table 3 facilitates the presentation of the algorithms in Scale-RS.

3.2.1 Addressing Function

Let array ***H*** record the history of scaling operations, and *H [t]* denotes the number of data chunks after the tth scaling. Function Addressing(t, H, x) calculates the address of block

TABLE 3
Symbols and Definitions

Symbols	Definition
k, r	Number of data chunks and parity chunks, respectively
k'	Number of data chunks after scaling
$\mathrm{DC_i}$	ith data chunk, $i \in \{1, 2, \ldots, k\}$
$\mathrm{PC_j}$	jth parity chunk, $j \in \{1, 2, \ldots, r\}$
$\mathrm{N(DC_i)}$	Node storing data chunk $\mathrm{DC_i}$
$\mathrm{N(PC_j)}$	Node storing parity chunk $\mathrm{PC_j}$
$\mathrm{D_{row,i}}$	Data block at rowth row in data chunk $\mathrm{DC_i}$
$\mathrm{P_{row,j}}$	Parity block at rowth row in parity chunk $\mathrm{PC_j}$
$\Delta\mathrm{P_{row,j}}$	Parity difference block of $\mathrm{P_{row,j}}$
C	Capacity of data chunk, e.g., number of data blocks
H	Scaling history. H[0] is the initial number of data chunks. H[t] denotes the number of data chunks after the tth scaling.
col, row	Chunk ordinal number and block ordinal number of a block
$\alpha_{\mathrm{j,i}}$	Element at jth column and ith row of Redundancy Matrix

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

x after the tth scaling. With the addressing function in place, Scale-RS is able to offer high scaling flexibility by supporting multiple scaling operations.

Function 1: Addressing(t, H, x)

```
Input:
      t: scaling times
      H: scaling history, H[0],H[1], ... ,H[t]
      x: logical block number
Output:
      col: the data chunk holding data block x
      row: physical block number
1  if $t == 0$ then                                   //Initial Layout
2    $k = H[0],\quad col = x \bmod k,\quad row = x/k$
3  end
4  $k = H[t-1],\quad k' = H[t]$
5  if $x \in [0,\ k \times C - 1]$ then               //block x is original
6      $[col_o, row_o]$ =Addressing$(t-1, H, x)$
7      $p = row_o \bmod k'$
8      if $0 \le p < k$ then                           //Not Moved
9          $col = col_o,\quad row = row_o$
10     else                                            //To be Moved
11         [col,row] = Moving$(col_o, row_o, k')$
12     end
13 else if $x \in [k \times C,\ k' \times C - 1]$ then //block x is new
14     [col,row] = Filling(x, k, k$'$)
15 end
16 return $col, row$
```

Data blocks are placed to *k* data chunks in a round-robin manner when a new $(\mathrm{k}+\mathrm{r},\ \mathrm{k})$RS-coded chunk group is created. The initial address of block *x* is calculated by one modular and one division (see line 2).

Let us investigate the tth scaling, where the number of data chunks is increased from *k* to *k*$'$ (see line 4). If block *x* is an original block, Scale-RS calculates its old address $(\mathrm{col_o},\ \mathrm{row_o})$ of the previous scaling (see line 6). If block *x* is a new block, it is filled to the location whose address is calculated by function Filling(x, k, $\mathrm{k'}$) (see line 14).

If block *x* is within the unmoved data zone, then Scale-RS keeps its chunk ordinal number and block ordinal number unchanged (see line 9); otherwise, Scale-RS invokes function Moving$(\mathrm{col_o},\ \mathrm{row_o},\ \mathrm{k'})$ to change its block address (see line 11).

3.2.2 Moving Function

Now we consider function Moving$(\mathrm{col_o}, \mathrm{row_o}, \mathrm{k'})$. According to the transposed data layout in Fig. 4b, the data block's location is transposed to its destination location. If a data block is to be moved, Scale-RS only swaps its chunk ordinal number and block ordinal number.

As to one scaling region of $\mathrm{k} + \Delta\mathrm{k}$ sequential stripes, Scale-RS moves a total of $\mathrm{k} \times \Delta\mathrm{k}$ data blocks from old data chunks to new data chunks. The theoretical minimum amount of migrated data is $(\mathrm{k'} \times \mathrm{k}) \times (\Delta\mathrm{k}/\mathrm{k'}) = \mathrm{k} \times \Delta\mathrm{k}$, where the lower bound of migration fraction is $\Delta\mathrm{k}/\mathrm{k'}$. That is, Scale-RS reaches the theoretical minimum data migration.

Function 2: Moving$(\mathrm{col_o}, \mathrm{row_o}, \mathrm{k'})$

```
Input:
      $col_o$: the data chunk holding a data block
      $row_o$: physical block number in $col_o{}^{th}$ data chunk
      $k'$: the number of data chunks after scaling
Output:
      col: new data chunk holding data block $(col_o, row_o)$
      row: physical block number
col = $row_o \bmod k'$
row = $(row_o/k') \times k' + col_o$
return $col, row$
```

3.2.3 Filling New Data Blocks

After the data redistribution is completed, new data blocks are written (i.e., filled) into an empty stripe of the scaled chunk group, according to a new striping rule of the destination chunk group.

Function Filling(x, k, $\mathrm{k'}$) calculates the address of new data block *x*. The ordinal number of the data chunk holding block *x* is calculated by a modular: col $= (\mathrm{x} - \mathrm{k} \times \mathrm{C}) \bmod k'$. The calculation of its physical block number (viz., block ordinal number) is relatively complicated, because the new data block should be filled into the lower $\mathrm{k'} - \mathrm{k}$ rows of the scaling region.

Function 3: Filling(x, k, $\mathrm{k'}$)

```
Input:
      x: logical block number
      $k'$: the number of data chunks after scaling
Output:
      col: new data chunk holding data block x
      row: physical block number
$y = x - k \times C,\quad \Delta k = k' - k$
col = $y \bmod k'$
row = $(\frac{y}{k \times k'} \times k') + k + (\frac{y}{k'} \bmod \Delta k)$
return $col, row$
```

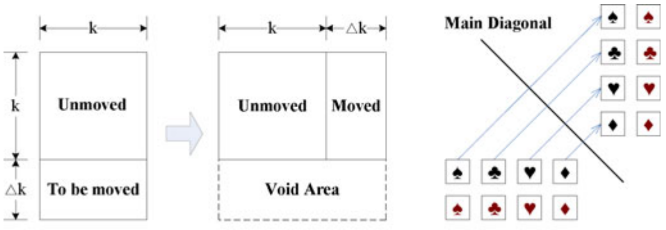
When new data blocks are filled into an entire stripe across $\mathrm{k'}$ data chunks, associated *r* parity blocks can be directly produced using the $(\mathrm{k'} + \mathrm{r},\ \mathrm{k'})$ RS encoding. Consequently, such a full-stripe write exhibits higher filling-time performance than partial-stripe writes in the cases of *No-DM* and *DM-Only*. Partial-stripe writes need to perform the 'reading-old-parity-block' step before calculating new parity blocks.

3.3 The Parity Updating Stage

3.3.1 Generating Parity Blocks

Recall that (see Section 2.1) if data block $\mathrm{D_{row,i}}$ in data chunk $\mathrm{DC_i}$ is modified to $\mathrm{D'_{row,i}}$, then parity block $\mathrm{P_{row,j}}$ in parity chunk $\mathrm{PC_j}$ will be updated by adding parity difference $\alpha_{j,i}(\mathrm{D'_{row,i}} - \mathrm{D_{row,i}})$. Similarly, when original data blocks are moved to new data chunks, the parity difference block equals to the linear combination of the moved data blocks with corresponding coefficients, since the new data chunks are initially zeroized. For example, if data blocks $\mathrm{D_{5,1}}$ and $\mathrm{D_{6,1}}$ are respectively moved to data chunks $\mathrm{DC_5}$ and $\mathrm{DC_6}$,

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.



(a) Variation of data layout (b) Transposed layout

Fig. 5. Scaling from four data chunks to six data chunks using Scale-RS, where old data blocks in lower $\Delta k = 2$ rows are migrated to $\Delta k = 2$ new data chunks (i.e., DC_5 and DC_6), and $r = 2$ associated parity difference blocks are delivered to parity chunks. '$D_{row,i}$ $[DC_i]$' represents that data block $D_{row,i}$ comes from data chunk DC_i, and '$\Sigma_{row=5}^{6}\alpha_{j,row}D_{row,i}$ $[DC_i]$' indicates that parity difference block '$\Sigma_{row=5}^{6}\alpha_{j,row}D_{row,i}$' is delivered from data chunk DC_i.

then the associated parity difference block for parity chunk PC_1 will be $\alpha_{1,5}D_{5,1} + \alpha_{1,6}D_{6,1}$.

Storage nodes offer sufficient computing capability in addition to I/O services [8], [24], [41]; thus, the calculation of parity difference can be accomplished by the nodes accommodating old data chunks, new data chunks, or parity chunks. Here, we denote nodes storing old data chunk DC_i, new data chunk $DC_{i'}$ and parity chunk PC_j as $N(DC_i)$, $N(DC_{i'})$, and $N(PC_j)$, respectively. There are three policies of calculating parity difference blocks:

- *Policy-1*—by node $N(DC_i)$, with $i \in \{1, 2, \ldots, k\}$. Node $N(DC_i)$ calculates *r* parity difference blocks, which are delivered to *r* parity chunks.
- *Policy-2*—by node $N(DC_{i'})$, with $i' \in \{k+1, \ldots, k+\Delta k\}$. After receiving Δk data blocks from an old data chunk, node $N(DC_{i'})$ computes *r* parity difference blocks and forwards them to *r* parity chunks;
- *Policy-3*—by node $N(PC_j)$, with $j \in \{1, 2, \ldots, r\}$. After receiving Δk data blocks from an old data chunk, node $N(PC_j)$ calculates its parity difference using the received data blocks.

A key design goal of Scale-RS is to minimize data movement induced by parity updates. Such a design goal spurs us to choose Policy-1 rather the other two policies to generate parity difference blocks. Analytically, Policy-1 causes the fewest number of transferred blocks among the three policies. Compared to Policy-1, Policy-2 has an extra data-forwarding step, and Policy-3 suffers from an enormous amount of network traffic. Fig. 5 depicts a diagram of $cluster\ scaling_{<4\to 6>}$, where node $N(DC_i)$ not only delivers $\Delta k = 2$ original data blocks $\{D_{5,i}, D_{6,i}\}$ to $\Delta k = 2$ new data chunks $\{DC_5, DC_6\}$, but also calculates $r+2$ parity difference blocks $\{\Sigma_{row=5}^{6}\alpha_{1,row}D_{row,i}, \Sigma_{row=5}^{6}\alpha_{2,row}D_{row,i}\}$ for parity chunks $\{PC_1, PC_2\}$, with $i \in \{1, 2, \ldots, k\}$. Each parity difference block is generated from data blocks in an individual data chunk, thereby exploiting data locality.

3.3.2 Procedure of Updating Parity Blocks

When migrating original data blocks or filling new data blocks, corresponding parity blocks should be updated. Procedure Updating(k, k', r) shows the parity updating within a scaling region. When original data blocks at rows $\{k+1, k+2, \ldots, k'\}$ are migrated to new data chunks, parity blocks at rows $\{1, 2, \ldots, k\}$ will be updated using the *RMW* method. When new data blocks are filled into void area, parity blocks at rows $\{k+1, k+2, \ldots, k'\}$ will be yielded by the encoding algorithm.

Procedure 1: Updating(k, k', r)

Input:
k: the number of data chunks before scaling
k': the number of data chunks after scaling
r: the number of parity chunks

```
switch the time point do
  case data blocks at rows $\{k+1, k+2, \ldots, k'\}$ have been migrated
      foreach $i \in \{1, 2, \ldots, k\}$ do
          foreach $j \in \{1, 2, \ldots, r\}$ do
              $P_{i,j} += \Sigma_{row=k+1}^{k'}\alpha_{j,row}D_{row,i}$
          end
      end
  endsw
  case rows $\{k+1, k+2, \ldots, k'\}$ have been filled with new blocks
      foreach $i \in \{k+1, k+2, \ldots, k'\}$ do
          foreach $j \in \{1, 2, \ldots, r\}$
              $P_{i,j} = \Sigma_{col=1}^{k'}\alpha_{j,col}D_{i,col}$
          end
      end
  endsw
endsw
```

The updating procedure properly performs regardless of the total number *C* of data blocks in a chunk. If *C* is a multiple value of $(k + \Delta k)$, all data blocks can be divided into *rgn* (for, $rgn = C/(k + \Delta k)$) scaling regions, each of which can be processed by the updating procedure; otherwise, there exist some empty data blocks in the moved area of the rgnth scaling region after the data redistribution stage, and the above updating procedure still works when the empty data blocks are initialized to 0.

After Δk data blocks are written to Δk new data chunks, the Δk blocks should also be transferred to each of r parity chunks to accomplish parity updates [42]. With the transposed layout, only one parity difference block (i.e., the linear combination of the Δk data blocks) is delivered to one parity chunk. Therefore, Scale-RS minimizes the data movement incurred by parity updates.

3.4 Optimization in Scale-RS

3.4.1 Write Aggregation

When *k* original data blocks in *k* old data chunks are sent to new data chunk $DC_{i'}$, node $N(DC_{i'})$ may write the received data blocks in an arbitrary order. Motivated by the fact that large block requests improve disk I/O performance [43], Scale-RS merges multiple blocks into a single request. We refer to this optimization technique as *write aggregation*. Fig. 6 shows that after four data blocks (e.g., blocks 16, 17, 18, and 19) are delivered from four old data chunks to new data chunk DC_5, node $N(DC_5)$ buffers the data blocks to a pre-allocated RAM space and then writes the buffered blocks in the form of a single write request.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

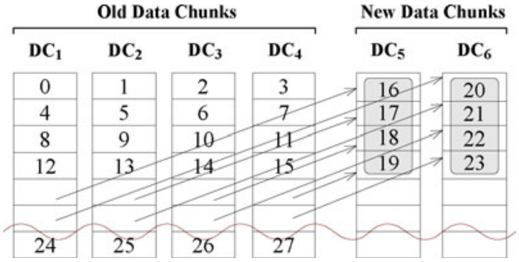


Fig. 6. Write aggregation in $clusterscaling_{<4\rightarrow 6>}$. Four sequential blocks are merged and written in the form of a single request.

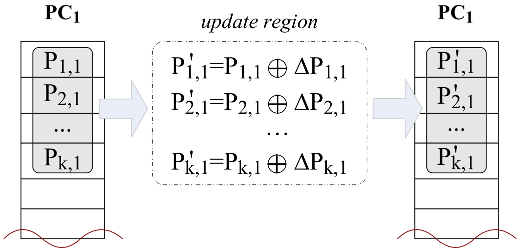


Fig. 7. Deferred update for parity chunk PC_1 under $cluster\ scaling_{<k\rightarrow k'>}$. *k* old parity blocks are read through a single read and *k* new parity blocks are written via a single write request.

Each chunk in HDFS or QFS is stored as a separate file to a native file system (e.g., ext3, ext4) on a datanode. It is feasible to use the space reservation feature of the underlying file system to enable blocks to be written contiguously (see Section 3.1 in [6]). In this case, *write aggregation* makes *k* data blocks be written in a sequential manner, thus improving the write throughput by mitigating disk seek time over multiple contiguous blocks.

3.4.2 Decoupling Parity Update from Data Migration

Parity update is a composite operation. For example, the RMW method relies on three steps to modify a parity block; the three updating steps are reading an old parity block, computing a new parity block, and writing a resulting parity block. It is conventional to couple both data migration and parity update into an atomic operation to assure the data consistency of each stripe. As a result, updating latency incurred by parity update is a significant contributor of the data-redistribution time.

Essentially, data migration and parity update can be accomplished in an asynchronous fashion, suggesting that Scale-RS may decouple parity update from data migration. In Scale-RS, node $\mathrm{N(DC_i)}$ storing old data chunk $\mathrm{DC_i}$ transfers $\Delta\mathrm{k}$ original data blocks and *r* parity difference blocks to new data chunks and parity chunks, respectively. Once the data blocks are successfully written to the new data chunks and the parity difference blocks are received by nodes storing parity chunks, the migration process terminates and continues a next migration. And nodes storing parity chunks $\{\mathrm{PC_1}, \mathrm{PC_2}, \ldots, \mathrm{PC_r}\}$ independently update parity blocks by merging the received parity difference blocks. The decoupling method can guarantee the stripe consistency, because parity difference blocks can also be calculated from $\Delta\mathrm{k}$ moved data blocks and then stored to new data chunks $\{\mathrm{DC_{k+1}}, \mathrm{DC_{k+2}}, \ldots, \mathrm{DC_{k+\Delta k}}\}$.

3.4.3 Deferred Update

After node $\mathrm{N(PC_j)}$ storing parity chunk $\mathrm{PC_j}$ has read a local old parity block $\mathrm{P_{row,j}}$ and received a parity difference block $\Delta\mathrm{P_{row,j}}$, a new parity block $\mathrm{P'_{row,j}}$ will be successfully generated. The operation of reading a local parity block or receiving a parity difference block over network may stall parity updates from the perspective of I/O path. To address this performance issue, Scale-RS adopts the *deferred update* technique. As shown in Fig. 7 node $\mathrm{N(PC_1)}$ not only buffers *k* received parity difference blocks $\{\Delta\mathrm{P_{1,1}}, \Delta\mathrm{P_{2,1}}, \ldots, \Delta\mathrm{P_{k,1}}\}$ to a pre-allocated RAM space (i.e., *update region*), but also reads *k* old parity blocks $\{\mathrm{P_{1,1}}, \mathrm{P_{2,1}}, \ldots, \mathrm{P_{k,1}}\}$ via a single read. If two blocks in a block pair (e.g., $\mathrm{P_{row,1}}$ and $\Delta\mathrm{P_{row,1}}$) have already been buffered, then the node will carry out an XOR calculation of $\mathrm{P_{row,1}} \oplus \Delta\mathrm{P_{row,1}}$. Furthermore, *k* new parity blocks $\{\mathrm{P'_{1,1}}, \mathrm{P'_{2,1}}, \ldots, \mathrm{P'_{k,1}}\}$ can be written to $\mathrm{PC_1}$ adopting the *write aggregation* method (see Section 3.4.1).

3.5 Features of Scale-RS

Scale-RS has four salient features as follows:

(1) *Uniformly distributing data after scaling*. Each data chunk, either old or new, has *k* data blocks after data redistribution; Scale-RS preserves the uniformity of data distribution even after multiple scaling operations.

(2) *Minimizing the total number of migrated data blocks*. The number of migrated data blocks from old data chunks to new data chunks $\mathrm{k} \times \Delta\mathrm{k}$, which is the theoretical minimum amount of migrated data for one scaling region of $\mathrm{k} + \Delta\mathrm{k}$ sequential stripes.

(3) *Optimizing parity updates in both cases of data redistribution and data filling*. Using the transposed data layout, Scale-RS minimizes updating traffic when generating parity difference blocks. In data filling, the full-stripe write eliminates the 'reading-old-parity-block' step, which is required in the partial updating methods.

(4) *Increasing the storage efficiency of RS-coded chunk groups*. For example, the storage efficiency of the source (6, 4)RS-coded chunk group is 66.67 percent; the storage efficiency of the destination chunk group is 75 percent when the chunk group is scaled from four data chunks to six.

4 Performance Evaluation

4.1 Experimental Setup

Our testbed is an RS-coded storage cluster that consists of 15 commodity storage nodes and one client node. All the nodes are connected through a Cisco WS-C2960S-24TS-S switch. Each storage node contains an Intel(R) E5800 @ 3.2 GHz CPU, 4 GB DDR3 memory, and Intel G41 Chipset Mainboard with on board integrated Gigabit Ethernet interface. All the disks attached in the storage nodes are West Digital's Enterprise WD1002FBYS SATA2 disks. The operating system running in the storage nodes is Ubuntu 10.04 X86 64 (Kernel 2.6.32); the operation system installed in the client node is Fedora 12 X86 64 (Kernel 2.6.32). The client node is a Dell R720 Server with two Xeon(R) E5-2609 @2.40 GHz (four cores) CPUs, 32 GB DDR3 memory, and the Intel C600 series Chipset Mainboard.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

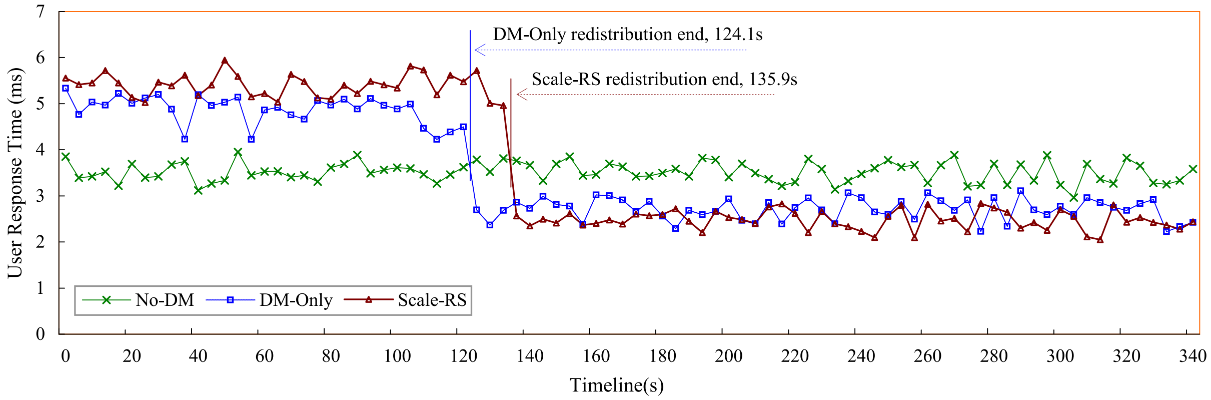


Fig. 8. Comparison of user response times among three scaling schemes (i.e., Scale-RS and two schemes based on No-DM and DM-Only patterns).

4.2 Prototype and Methodology

4.2.1 Prototype Design

To resemble real-world workload, we implement an application-level trace replayer on the storage cluster. The trace replayer is running on the client node, acting as multiple concurrent users whose average response times are measured. We adopt an open-loop model during the online scaling, where traces are replayed according to timestamps logged in trace files, i.e., I/O arrival rates are independent of I/O request completions [44]. The trace replayer issues I/O requests to appropriate data chunks according to address mappings.

It is relatively fair to make comparisons between scaling strategies incorporated in storage systems sharing the same configuration; similar comparisons can be found in the literature [31]. For example, a group of comparative experiments were performed on the same RAID system, where the *DM-Only*-based McPod scheme and the *DM-PU*-based MD-Reshape are deployed. In addition, *No-DM*, *DM-Only* and *DM-PU* are the three cluster-scaling patterns for RAID-4-like chunk groups (see Section 2.2), one of the evaluation goals is to evaluate scaling performance of the solutions based on the three scaling patterns in the realm of storage clusters. Therefore, we compare *DM-PU*-based Scale-RS and the two alternative scaling schemes based on *No-DM* and *DM-Only* patterns. To quantitatively evaluate I/O performances of our Scale-RS scheme and the two alternative scaling schemes, we implement the three scaling schemes in the RS-coded storage cluster. A scaling process reorganizes data blocks among old and new data chunks, and updates associated parity blocks. Aiming to minimize update traffic, Scale-RS makes the storage nodes storing old data chunks calculate parity difference blocks, which are delivered to the corresponding parity chunks. ZFec library [45] is employed for the classic Reed-Solomon coding.

4.2.2 Evaluation Methodology

An application scenario of erasure-coded storage systems is to archive data that are no longer modified [3]. The archival storage is usually characterized as Write-Only-Read-Many (WORM) I/Os. We evaluate the online scaling performance of Scale-RS using the Web-2 trace that represents read-intensive applications [46]. Additionally, it is demanding to sustain high write throughput for data archival. For instance, high write throughput is indispensable for archiving video data from surveillance cameras in a timely manner. Therefore, we focus on three performance metrics, namely, user response time during data redistribution, total redistribution time, and write throughput in data-filling stage.

To have an insight into the impact of background data redistribution on foreground user I/Os, we collect the response times of all user I/Os and calculate the average user response time per 1,000 user I/Os. All average user response times form an average user response time series.

Assume there are 100 chunks of size 128 MB stored on each storage node (e.g., 128 MB $\times 100 = 12{,}800$ MB), which is sufficiently large to cover the footprint of the evaluated workload. During the data filling stage, Δk × 12,800 MB is the total amount of data written to the void area in destination chunk groups. Evidence shows that a configuration of '$r = 3$' achieves a sufficiently large Mean Time To Data Loss or MTTDL for archival storage systems [8]. Therefore, we set parameter r to be 3 in our tests.

To make a fair comparison between Scale-RS and the two alternative scaling schemes, we also incorporate the deferred update method to improve parity updates incurred in the data filling stage within both the *No-DM*-based and *DM-Only*-based schemes. Furthermore, the write aggregation method is applied to the *DM-Only*-based scheme to optimize the data redistribution process.

4.3 Trace-Driven Evaluations

4.3.1 Data Redistribution

We compare the performances of the three scaling schemes running in two phases: during data redistribution and after data redistribution. Fig. 8 plots the average user response time series as the time increases from 0 to 340 with an increment of 20 seconds, with coding parameters k, Δk, and r are set to 6, 3, and 3, respectively.

We draw three observations from Fig. 8. First, the Scale-RS scheme spends more time than *DM-Only* to accomplish data redistribution, because Scale-RS needs to handle parity updates besides data migrations. Thanks to the two optimization techniques (i.e., 'decoupling parity update from data migration' and 'deferred update'), parity update overhead is mostly hidden by overlapping with data migration latency. As such, the redistribution time of Scale-RS is slightly more than that of the *DM-Only*-based one (i.e.,

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

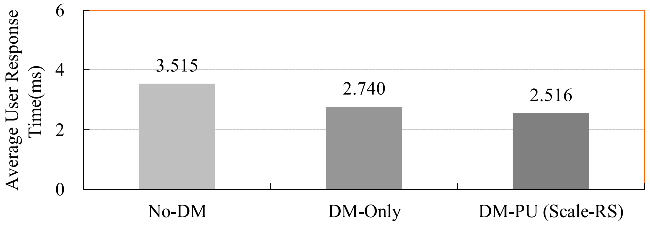


Fig. 9. Comparison of average user response times of the three scaling schemes in the case of 'after data redistribution'.

135.9 versus 124.1 s). Second, compared with *No-DM*, both Scale-RS and *DM-Only* increase the user response time during data redistribution, because data redistribution I/Os compete disk bandwidth with user I/Os. Third, the *No-DM*-based scheme has higher user response time than both Scale-RS and DM-Only after data redistribution. This is mainly because there are $k = 6$ data chunks serving user I/O requests in No-DM, whereas there are $\mathrm{k}' = 9$ data chunks in Scale-RS and DM-Only.

As mentioned in Section 1.3, it is challenging to achieve high I/O performance for scaled storage clusters. Although Scale-RS exhibits marginally longer response time than No-DM and DM-Only during the course of data redistribution, Scale-RS offers the highest read performance after data redistribution. The reason lies in the fact that Scale-RS not only has a much stronger spatial locality than *DM-Only*, but it also achieves a higher read parallelism than *No-DM*. Fig. 9 reveals that Scale-RS outperforms No-DM and DM-Only in terms of average user response time by a factor of 1.40 and 1.09, respectively.

4.3.2 Offline Data Filling

When it comes to *offline data filling*, all chunks (i.e., old data chunks, new data chunks, and parity chunks) are dedicated to serve data filling operations rather than user I/Os. Fig. 10 plots the aggregate write throughput of the three scaling schemes under offline filling. Again, parameters k, Δk, and r are set to 6, 3, and 3, respectively.

The results plotted in Fig. 10 confirm that Scale-RS has the best write performance among the three scaling schemes. The reason is two-fold. First, the data filling operations of *No-DM* and *DM-Only* cause all old parity blocks in parity chunks to be updated, thereby bringing a penalty to write throughput; whereas Scale-RS directly generates $r = 3$ new parity blocks and simply writes the new parity blocks to $r = 3$ parity chunks. Second, more stripes are involved in both *No-DM* and *DM-Only* than that in Scale-RS; that is, *No-DM* and *DM-Only* should update more parity blocks than Scale-RS. For example, assume 36 new data blocks is filled into a destination chunk group, then *No-DM* and *DM-Only* need to update parity blocks in $36/\Delta \mathrm{k} = 12$ stripes. If the RMW update method [40] is adopted, then the 'reading-old-parity-block' step is required before calculating new parity blocks; if the RCW update method [39] is applied, then the 'reading-old-data-block' step is triggered before calculating new parity blocks. However, there are only $36/k' = 36/9 = 4$ stripes involved in the Scale-RS case, and the filling procedure is equivalent to a full-stripe write for Scale-RS.

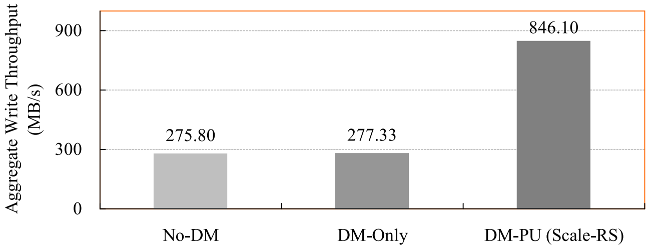


Fig. 10. Comparison of aggregate write throughput under offline data filling. Parameters k = 6, Δk = 3, and r = 3.

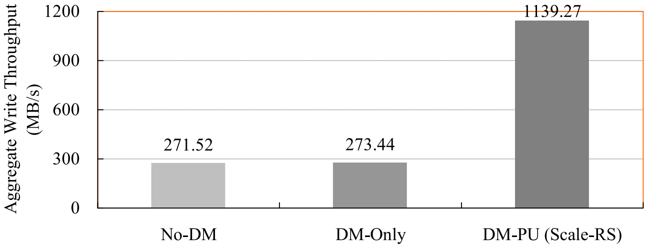


Fig. 11. Comparison of aggregate write throughput in the offline data filling case. Parameters k = 9, Δk = 3 and r = 3.

To examine the sensitivity of filling time to number k of data chunks, we conduct a group of tests in the offline data filling case, where parameters k, Δk, r are set to 9, 3, and 3, respectively. Fig. 11 shows the write throughput of the three scaling schemes.

Figs. 10 and 11 show that the aggregate write throughput of *No-DM* is close to that of *DM-Only* in both cases of $k = 6$ and $k = 9$, because parity updates in parity chunks become a data-filling performance bottleneck for these two schemes. On the contrary, Scale-RS exhibits higher write throughput when the k value is increased from 6 to 9. Specifically, Scale-RS outperforms the other two scaling schemes in terms of aggregate write throughput by a factor of 3.05 and 4.17 when the number k of data chunks is 6 and 9, respectively. Scale-RS performs better when the destination chunk group has more data chunks, because more data chunks help to offer higher write bandwidth and to decrease the number of stripes associated with parity updating (i.e., $36/k' + 36/12 = 3$ stripes).

4.3.3 Online Data Filling

In the case of *online data filling*, a destination chunk group simultaneously serves user read requests and data filling operations. Fig. 12 plots both aggregate write throughput and average user response times of the three scaling schemes under online data filling, where the Web-2 trace is replayed and parameters k, Δk and r are respectively set to 6, 3, and 3.

We draw two observations from Figs. 9 and 12a. First, the data-filling operation has almost no impact on user read performance of *No-DM*. No impact is expected, because user I/O requests and data filling operations are respectively served by old data chunks and new data chunks in *No-DM*. Second, in the Scale-RS and *DM-Only*-based scaling schemes, the average user response time increases under online filling compared to that after data redistribution. The increased user response times are attributed to bandwidth

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

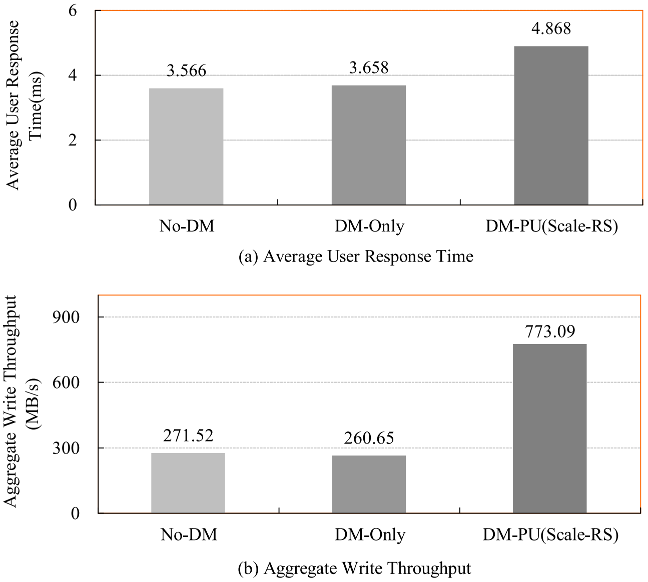


Fig. 12. Aggregate average user response times and write throughput of the three scaling schemes under online data filling. Parameters k = 6, Δk = 3 and r = 3.

competition between data filling operations and user I/Os under online filling.

Three observations can be drawn from Figs. 10 and 12b. First, the aggregate write performance of *No-DM* in the online and offline filling cases are almost identical. Second, the aggregate write throughput of *DM-Only* under online filling is close to that under offline filling, because the parity update in parity chunks is still the data-filling bottleneck for *DM-Only* under online filling. Last, Scale-RS's aggregate write throughput under online filling decreases by 8.63 percent compared to that under offline filling, because user I/Os compete with filling operations for disk bandwidth in the online filling case. In particular, Scale-RS can outperform the two alternative scaling schemes in terms of aggregate write throughput by a factor of 2.85 to 2.97 under online filling for $cluster\ scaling_{<6 \to 9>}$.

4.4 Summaries of Evaluation

We summarize the observations drawn from the above experimental results as follows:

- Both Scale-RS and *DM-Only* have similar data redistribution performance. Compared to *No-DM*, the user response times of Scale-RS and *DM-Only* increase due to disk bandwidth competition between data redistribution operations and user I/Os.
- After data redistribution, Scale-RS achieves the lowest average user response time among the three scaling schemes. That is, Scale-RS is conducive to sustaining high read performance for scaled storage clusters.
- Scale-RS has an outstanding performance under data filling. It outperforms the other two alternative scaling schemes in terms of write throughput by a factor of 2.85 and 3.05 under online filling and offline filling for $cluster\ scaling_{<6 \to 9>}$, respectively; and the speedup factor is 4.17 under offline filling as far as scaling from 9 data chunks to 12 is concerned.

5 Related Work

5.1 Storage Scaling Categories

Redundancy is a common approach to improve storage reliability, either by replicating data blocks or by storing additional information (e.g., parity blocks generated by erasure codes).

Replica-based distributed storage systems usually adopt randomized data distribution strategies for online placement and reorganization of replicated data. For example, the RUSH algorithm utilizes a mapping function to map replicated objects to a scalable collection of storage servers or disks [48]; the CRUSH algorithm employs a scalable pseudo-random data redistribution function to distribute and reorganize replicated data [49]. Random Slicing provides a simple yet efficient randomized data distribution strategy for replica-based storage systems [22]. Moreover, the rack-aware replica policy in HDFS exhibits random placement decision for non-local replicas, which can be on any rack and within any node of the rack [50].

Erasure coding schemes (e.g., array codes, Reed-Solomon codes) are implemented by striping data blocks across several storage devices (e.g., disks and nodes). There are two cases considered in the scaling of erasure-coded storage: (1) block-level XOR-based RAIDs (e.g., RAID-4, RAID-5, and RAID-6), where both data and parity blocks are placed across disks according to a pre-calculated pattern. Such a deterministic scaling in RAID usually alters its parity chain, thus causing data redistribution; (2) file-level erasure-coded storage clusters (e.g., HydraFS [51], DRFS [52]), where a namespace server (MDS) maintains the mapping of file blocks to the physical location. That is, file blocks are randomly distributed. When Δk new nodes are added to expand an existing storage cluster, the new nodes become target nodes of data replacement. A simple way to utilize the new nodes is to create a new RS-coded chunk group within the storage cluster.

5.2 Storage Scaling Schemes

Our Scale-RS attempts to scale existing chunk groups for RS-coded storage clusters. Because blocks in chunks are organized in a RAID-4-like data layout within a $(k+r, k)$RS-coded chunk group, it is necessary to investigate existing RAID scaling schemes. Table 4 summarizes several storage scaling schemes from the aspects of distribution uniformity, data migration, storage efficiency, parity update approach during data migration, and the like.

Existing scaling schemes designed for RAIDs include SLAS [26], SCADDAR [47], FastScale [25], McPod [31], ALV [32], MiPiL [33], MDM [28], GA [34], GSR [27], PBM [35], SDM [29], and so on. Data redistribution in scaling RS-coded chunk groups consists of data migration and parity update; the data migration can be completed using RAID-0 scaling approaches (e.g., Round-Robin, FastScale, etc.). Unfortunately, scaling approaches to single-parity RAID-5 and dual-parity RAID-6 are not adequate for in-production RS-coded storage clusters tolerating more than two failures, because these scaling schemes are tailored for the specific parity layout of RAID-5 and RAID-6.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Comparisons of the Storage Scaling Schemes

Scaling Scheme	Target Storage	Uniform Data Distribution	Minimal Data Migration	Highest Storage Efficiency	Parity Update During Data Migration
RR (e.g., SLAS [26])	RAID-0, RAID-4, RAID-5	√	×	√	–
Semi-RR (e.g., SCADDAR [47])	RAID-0, RAID-4, RAID-5	×	×	√	–
FastScale [25]	RAID-0, RAID-10, RAID-01	√	√	√	–
McPod [31]	RAID-4	√	√	√	without parity update
ALV [32]	RAID-5	√	√	√	write alignment technique
MiPiL [33]	RAID-5	√	√	√	piggyback parity update
MDM [28]	RAID-5	×	√	×	without parity update
GA [34]	RAID-5	√	×	√	not mentioned
GSR [27]	RAID-5	√	√	√	not mentioned
PBM [35]	RAID-5	×	√	√	no parity recalculation
SDM [29]	RAID-6	√	√	√	not mentioned
Scale-RS (Ours)	RS-Coded Chunk Groups	√	√	√	deferred update

Round-Robin (RR) algorithm preserves the uniformity of data distribution in any case of scaling (e.g., RAID-0, RAID-4, and RAID-5) [26], but RR causes almost 100 percent of data moved as well as 100 percent of parity updated. Compared to RR, Semi-RR algorithm significantly reduces data migration [47]; however, Semi-RR fails in achieving distribution uniformity after subsequent scaling operations.

FastScale [25] is a scaling solution for striping without parity (e.g., RAID-0, RAID-01, and RAID-10). FastScale moves only enough data blocks from old disks to new disks without migrating data among old disks. The downside of FastScale is that it leads to large update overhead due to partial-stripe writes during data filling.

McPod is a data redistribution approach to accelerating RAID-4 scaling [31]. McPod maintains a uniform data distribution across all data disks while optimizing data migration with a set of techniques (e.g., outsourcing all parity updates to a surrogate disk).

ALV is a data redistribution approach to RAID-5 scaling [32]. ALV deploys a reordering window to change the transfer order of data blocks to access multiple successive blocks via a single I/O. ALV is an extension of the RR algorithm; and as such, ALV suffers from large I/O overhead caused by data migrations and parity updates.

MiPiL is a RAID-5 scaling approach [33], which not only maintains a uniform distribution for regular data and parity data, but also minimizes data migration using piggyback parity updates and lazy metadata updates. MiPiL preserves simple data management since it designs a deterministic placement strategy appropriate for RAID-5.

MDM [28] is a RAID-5 scaling method with minimal data movement. Although MDM can eliminate parity updates, it is not feasible for $(k+r, k)$ RS-coded storage when r is greater than or equal to 2. This is because new data blocks are distributed down a diagonal in MDM, and the layout after scaling becomes much more complex than a typical RAID-5 one. Furthermore, MDM does not increase the data storage efficiency after scaling.

Similar to ALV that accelerates RAID-5 scaling by scheduling data blocks, Gradual Assimilation (GA) algorithm [34] achieves the assimilation of the new disks by a sequential reorganization of full stripes. The GA algorithm adopts round-robin order to redistribute data blocks; as the result, all parities need to be updated after data migration is completed.

Global Stripe-based Redistribution (GSR) is proposed to accelerate RAID-5 scaling [27]. GSR maintains the layout of most stripes while sacrificing a small portion of stripes. The main limitations of GSR are two-fold: First, it suffers from high overhead in updating both mapping metadata and parity blocks; Second, it brings a large performance penalty under the workload with a strong locality.

Parity-based Migration (PBM) is an expansion method for RAID-5 [35]. PBM minimizes data migration while evenly distributing parity blocks. Unfortunately, PBM makes the scaled RAID a non-standard RAID-5 distribution, which may lead to an imbalanced data redistribution.

Stripe-based Data Migration (SDM) is a stripe-level scaling scheme developed for RAID-6 [29]. It is common that a RAID-6 scaling scheme is designed for a special RAID-6 code, because the parity layout of each RAID-6 code is unique and the parity chain of one RAID-6 code is various. Typical parity chains include diagonal chain, anti-diagonal chain, horizontal chain, and vertical chain.

6 Further Discussions

This paper addresses the case of $\textit{cluster scaling}_{<k\rightarrow k+\Delta k>}$ where Δk is greater than zero. If no new data chunks are created and inserted to an existing chunk group when Δk new nodes join a cluster, then the redundancy parameters (i.e., k and r) of the chunk group remain unchanged; thus, the storage efficiency does not increase. In this case, new nodes are merely used to expand the cluster's capacity.

There is another way to expand an existing RS-coded chunk group—Δk old data chunks in another chunk group are inserted to the chunk group, and the associated parity chunks are updated accordingly. Essentially, such a scaling method is equivalent to the *No-DM*-based one.

Uniform data distributions are achieved at the cost of data migration. To meet the uniform-data-distribution

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

requirement [25], a scheme of *cluster scaling*$_{<k \rightarrow k+\Delta k>}$ has to move $\Delta k/(k+\Delta k)$ of old data blocks from old data chunks to new chunks. Therefore, there is a tradeoff between distribution uniformity and data-migration I/O traffics.

Scale-RS supports bidirectional scaling operations (i.e., scaling-up and scaling-down operations). When scaling from k data chunks to k' where k is larger than k', the transposed data layout still helps to minimize the data movement induced by parity updates. In particular, after receiving $k-k'$ data blocks from $k-k'$ data chunks to be evicted, the receiving node can generate r parity difference blocks from the received $k-k'$ data blocks.

Apart from sending Δk data blocks to Δk new data chunks, a storage node storing an old data chunk is responsible for calculating parity difference blocks. Nevertheless, the node does not become an I/O and CPU performance bottleneck during the data redistribution stage. The reason is two-fold: (1) such calculation operations are executed using in-memory Δk data blocks without incurring any disk reads, and (2) storage nodes have sufficient computing capability for RS encoding.

7 Conclusion and Future Work

This paper proposed a new scaling scheme called Scale-RS, which aims to expand RS-coded chunk groups for storage clusters. Scale-RS is conducive to optimizing both data migrations and parity updates. On one hand, Scale-RS moves only necessary data blocks from old data chunks to new data chunks without migrating data among old data chunks; on the other hand, with the transposed data layout in place, Scale-RS generates associated parity difference blocks from data blocks stored in an individual data chunk, thereby substantially reducing update traffic.

We implemented the prototypes for Scale-RS as well as two alternative scaling schemes in an erasure-coded storage cluster, on which real-world I/O traces were replayed. The extensive experiments demonstrate that Scale-RS slightly increases user response time during data redistribution due to bandwidth competition between migration and user I/Os; Scale-RS achieves the highest read performance among the three scaling schemes in the case of 'after data redistribution'. The experimental results also illustrate that Scale-RS outperforms the other two scaling schemes in terms of aggregate write throughput under data filling. For example, in the *cluster scaling*$_{<6 \rightarrow 9>}$ case, the speedup factors are 2.85 and 3.05 under online filling and offline filling, respectively.

It is shown that data migrations compete disk and network resources with user requests. As a future research direction, we plan to develop novel strategies to schedule migration I/Os among data chunks according to access locality in user I/O workload, thereby aiming to minimize interference between migration I/Os and user I/Os.

Acknowledgments

This work was supported in part by the National High Technology Research Program of China under Grant No. 2013AA013203, the National Basic Research Program of China under Grant No. 2011CB302303, and the Fundamental Research Funds for the Central Universities under No. 2014QN012. Xiao Qin's work was supported by the US National Science Foundation (NSF) under Grants CCF-0845257 (CAREER), CNS-0917137 (CSR), and CCF-0742187 (CPA). Changsheng Xie is the corresponding author of this paper.

References

[1] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal , M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 143–157.

[2] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 251–264.

[3] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Techno. Conf.*, 2012, pp. 15–26.

[4] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1013–1020.

[5] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 1–14.

[6] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[7] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch, "A decentralized algorithm for erasure-coded virtual disks," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2004, pp. 125–134.

[8] M. Storer, K. Greenan, E. Miller, and K. Voruganti, "Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–16.

[9] TAHO-LAFS. (2010). Tahoe: The least-authority filesystem. Open source code distribution [Online]. Available: http://tahoe-lafs.org/trac/tahoe-lafs

[10] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in raid-like systems," *Softw. Practice Exp.*, vol. 27, no. 9, pp. 995–1012, 1997.

[11] M. Manasse, C. Thekkath, and A. Silverberg, "A Reed-Solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage," *Proc. Inf.*, 2009, pp. 1–11.

[12] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An optimal scheme for tolerating double disk failures in raid architectures," in *Proc. 21st Annu. Int. Symp. Comput. Arch.*, 1994, pp. 245–254.

[13] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 1–14.

[14] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly, 2012.

[15] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Industrial Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.

[16] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra, "Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, pp. 225–238.

[17] L. Rizzo, " On the feasibility of software FEC," Dip. di Ingegneria dell'Informazione, Univ. di Pisa, Italy, DEIT, Tech. Rep. LR-970131, pp. 1–16, 1997.

[18] I. Cleversafe. (2008). Cleversafe dispersed storage," Open source code distribution [Online]. Available: http://www.cleversafe.org/downloads, 2008.

[19] J. K. Resch and J. S. Plank, "Aont-RS: Blending security and performance in dispersed storage systems," in *Proc. 9th USENIX Conf. File stroage Technol.*, 2011, pp. 191–202.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

[20] F. F. J. MacWilliams, and N. N. J. A. Sloane, *The Theory of Error-correcting Codes: Part 2*. Amsterdam, The Netherlands: Elsevier, 1977, vol. 16.

[21] D. Vesset, "Worldwide big data technology and services 2012-2016 forecast," IDC, Framingham, MA, USA, Tech. Rep. 238746, 2012.

[22] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes, "Reliable and randomized data distribution strategies for large scale storage systems," in *Proc. 18th Int. Conf. High Perform. Comput.*, 2011, pp. 1–10.

[23] G. Newgaard, "White paper: Hadoop on emc isilon scale-out nas," EMC Corp., Tech. Rep. H10528.1, 2012.

[24] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 253–265.

[25] W. Zheng and G. Zhang, "Fastscale: Accelerate raid scaling by minimizing data migration." in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 149–161.

[26] G. Zhang, J. Shu, W. Xue, and W. Zheng, "Slas: An efficient approach to scaling round-robin striped volumes," *ACM Trans. Storage*, vol. 3, no. 1, pp. 1–39, 2007.

[27] C. Wu and X. He, "Gsr: A global stripe-based redistribution approach to accelerate raid-5 scaling," in *Proc. 41st Int. Conf. Parallel Process.*, 2012, pp. 460–469.

[28] S. R. Hetzler, "Data storage array scaling method and system with minimal data movement," U.S. Patent 8,239,622, Aug. 2012.

[29] C. Wu, X. He, J. Han, H. Tan, and C. Xie, "Sdm: A stripe-based data migration scheme to improve the scalability of raid-6," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 284–292.

[30] C. Wu and X. He, "A flexible framework to enhance raid-6 scalability via exploiting the similarities among mds codes," in *Proc. 42nd Int. Conf. Parallel Process.*, 2013, pp. 542–551.

[31] G. Zhang, J. Wang, K. Li, J. Shu, and W. Zheng, "Redistribute data to regain load balance during raid-4 scaling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 99, no. PrePrints, p. 1, 2014.

[32] G. Zhang, W. Zheng, and J. Shu, "ALV: A new data redistribution approach to raid-5 scaling," *IEEE Trans. Comput.*, vol. 59, no. 3, pp. 345–357, Mar. 2010.

[33] G. Zhang, W. Zheng, and K. Li, "Rethinking raid-5 data layout for better scalability," *IEEE Trans. Comput.*, vol. 99, no. PrePrints, p. 1, 2013.

[34] J. L. Gonzalez and T. Cortes, "Increasing the capacity of raid5 by online gradual assimilation," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/O*, 2004, pp. 17–24.

[35] Y. Mao, J. Wan, Y. Zhu, and C. Xie, "A new parity-based migration method to expand raid-5," *IEEE Trans. Parallel Distrib Syst.*, vol. 99, no. PrePrints, p. 1, 2013.

[36] G. Zhang, K. Li, J. Wang, and W. Zheng, "Accelerate rdp raid-6 scaling by reducing disk i/os and xor operations," *IEEE Trans. Comput.*, vol. 99, no. PrePrints, p. 1, 2013.

[37] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. Patterson, "Failure correction techniques for large disk arrays," *ACM SIGARCH Comput. Archit. News*, vol. 17, no. 2, pp. 123–132, 1989.

[38] J. S. Plank, M. Blaum, and J. L. Hafner, "Sd codes: Erasure codes designed for how storage systems really fail," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 95–104.

[39] S. Savage and J. Wilkes, "Afraid: A frequently redundant array of independent disks," in *Proc. USENIX Annu. Tech. Conf.*, 1996, pp. 3–16.

[40] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Comput. Sur.*, vol. 26, no. 2, pp. 145–185, 1994.

[41] M. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2005, pp. 336–345.

[42] F. Zhang, J. Huang, and C. Xie, "Two efficient partial-updating schemes for erasure-coded storage clusters," in *Proc. IEEE 7th Int. Conf. Netw., Archit. Storage*, 2012, pp. 21–30.

[43] Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Comput. Surv.*, vol. 43, no. 3, pp. 23–52, 2011.

[44] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *Proc. 3rd conf. Netw. Syst. Des. Implementation*, 2006, pp. 239–252.

[45] Z. Wilcox-O'Hearn. (2012). Zfec 1.4.24. open source code distribution [Online]. Available: http://pypi.python.org/pypi/zfec

[46] M. Liberatore. (2007). Search engine i/o, umass trace repository [Online]. Available: http://traces.cs.umass.edu/

[47] A. Goel, C. Shahabi, S.-Y. D. Yao, and R. Zimmermann, "Scaddar: An efficient randomized technique to reorganize continuous media blocks," in *Proc. 18th Int. Conf. Data Eng.*, 2002, pp. 473–482.

[48] R. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proc. 18th Int. Parallel Distrib Process. Symp.*, 2004, pp. 96–105.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. SuperComput.*, 2006, pp. 122–133.

[50] D. Borthakur. (2009). The hadoop distributed file system: Architecture and design, Hadoop Apache Project [Online]. Available: http://hadoop.apache.org/common/docs/current/hdfs_design.html

[51] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 197–210.

[52] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "Hdfs raid," technical Talk, Yahoo! Developer Network, 2010.

Jianzhong Huang received the PhD degree in computer architecture in 2005, and completed the postdoctoral research in information engineering in 2007 from the Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently an associate professor in the Wuhan National Laboratory for Optoelectronics at HUST. His research interests include computer architecture and dependable storage systems. He received the National Science Foundation of China in Storage System Research Award in 2007.

Xianhai Liang received the BS degree in computer science and technology from the Wuhan University of Technology (WHUT), China, in 2012. He is currently working toward the MS degree at HUST. His research interests include networked storage systems and file system.

Xiao Qin (S'00-M'04-SM'09) received the BS and MS degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 1992 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska-Lincoln, in 2004. He is currently an associate professor with the Department of Computer Science and Software Engineering, Auburn University. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He received the US National Science Foundation (NSF) Computing Processes and Artifacts Award, the NSF Computer System Research Award in 2007, and the NSF CAREER Award in 2009. He is a senior member of the IEEE.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.

Ping Xie received the BS degree in physics and the MS degree in computer application from Qinghai Normal University, China, in 2002 and 2008, respectively. He is currently working toward the PhD degree in the Huazhong University of Science and Technology (HUST). His research interests include erasure codes and dependable storage systems.

Changsheng Xie received the BS and MS degrees in computer science both from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1982 and 1988, respectively. He is currently a professor in the Department of Computer Engineering at HUST. He is also the director of the Data Storage Systems Laboratory of HUST and the deputy director of the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, I/O system, and networked storage system. He is the vice chair of the expert committee of Storage Networking Industry Association (SNIA), China. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Auburn University. Downloaded on October 15,2024 at 13:10:38 UTC from IEEE Xplore. Restrictions apply.