Efficient Parity Update for Scaling RAID-like Storage Systems

Dongdong Sun[1], Yinlong Xu[1,2], Yongkun Li[1,3], Si Wu[1], Chengjin Tian[1]
1. School of Computer Science and Technology, University of Science and Technology of China
2. Collaborative Innovation Center of High Performance Computing, National University of Defense Technology
3. AnHui Province Key Laboratory of High Performance Computing, Hefei, China
Email: {sunddd, wusi, jiyingtk}@mail.ustc.edu.cn, {ylxu, ykli}@ustc.edu.cn

***Abstract*—It is inevitable to scale RAID systems with the increasing demand of storage capacity and I/O throughput. When scaling RAID systems, we will always need to update parity to maintain the reliability of the storage systems. There are two schemes, read-modify-write (RMW) and read-reconstruct-write (RCW), to update parity. However most existing scaling approaches simply use RMW to update parity. While in many scenarios for existing scaling approaches, RCW performs better in terms of the number of scaling I/Os.**

In this paper, we propose an algorithm, called *EPU*, to analyze which of RCW and RMW is better for a scaling scenario and select the more efficient one to save the scaling I/Os. We apply EPU to online scaling scenarios and further use two optimizations, I/O overlap and access aggregation, to enhance the online scaling performance. Using Scale-CRS, one of the existing scaling approaches, as an example, we show via numerical studies that Scale-CRS+EPU reduces the amount of scaling I/Os over the traditional Scale-CRS in many scaling cases. To justify the online efficiency of EPU, we implement both Scale-CRS+EPU and Scale-CRS in a simulator with Disksim as a working module. Through extensive experiments, we show that Scale-CRS+EPU reduces the scaling time and the online response time of user requests over Scale-CRS.

***Key words*—RAID scaling; Erasure code; Parity update; Performance evaluation**

I. Introduction

Redundant Array of Independent Disks (a.k.a, RAID) [1], which is composed of multiple inexpensive devices, is an efficient choice for storage systems to achieve high throughput, large capacity and high reliability. To ensure data availability, erasure codes are often introduced into RAIDs. According to the level of fault tolerance, RAIDs can be classified into RAID-0 without fault resilience, RAID-5 against one disk failure, RAID-6 against double disk failures and others which against three or more disk failures.

A Maximum Distance Separable (MDS) erasure code encodes k data blocks into a *stripe* of n $(k < n)$ coded ones such that any k out of them are sufficient to decode the original data blocks. Thus, MDS erasure codes achieve a certain level of fault resilience with theoretically lowest storage overhead. An erasure code is *systematic* when the original k data blocks are included in the n coded ones, while the other $n - k$ encoded blocks are named parity blocks. Due to the low storage overhead and easy access to the original data blocks, systematic MDS codes are widely used in RAIDs.

There are many implementations of RAID-6 codes, such as EVENODD [2], P-Code [3], X-code [4], RDP [5]. As storage systems continue to grow in size, multiple disk failures often occur [6]. Reed-solomon (RS) [7] codes and Cauchy Reed-solomon (CRS) [8] codes are good choices for large-scale storage systems as they are designed to tolerate a general number of disk failures. For example, both Google and Facebook adopt RS codes in their distributed file systems [9] [10] [11]. CRS codes are also widely used in commercial systems, such as NetApp [5], OceanStore [12], etc.

One challenge in RAID-like storage systems is that the volume of data to be stored continues to explode, and hence the foreground applications often require larger storage capacity and higher I/O bandwidth. A storage system addresses this challenge through adding new disks into the current configuration [13]. However, the process of scaling up the storage system is non-trivial. On the one hand, part of the data blocks in old disks should be migrated into new disks to regain a balanced data load after scaling. On the other hand, the parity blocks must be updated accordingly as the data layout within a stripe is changed. Both the data migration and the parity update processes induce some scaling I/O requests. Besides, the scaling processes are often conducted online to provide unstopped services to the foreground applications [14]. Therefore, a scaling process should schedule the user I/O requests and the scaling I/O requests appropriately to perform efficient online scaling.

There are many approaches to scale RAID-like storage systems, including FastScale [15], Gradual Assimilation (GA) [16], ALV [17], Semi-RR [18], MDM [19], GSR [20], MiPiL [21], SDM [22], RS6 [23], Scale-RS [24], Scale-CRS [25]. Among them, FastScale is proposed for RAID-0 scaling. GA [16], ALV [17], MDM [19], GSR [20], MiPiL [21] are aimed at RAID-5 systems, SDM [22] are designed for RAID-6 systems, RS6 [23] is dedicated for RAID-6 system with RDP code, Scale-RS [24] is designed for systems with RS codes, and Scale-CRS [25] is aimed at systems with CRS codes. These approaches are primarily designed to balance the load after scaling, minimize the volume of data blocks to be migrated and minimize the number of parity blocks to be updated. Nevertheless, none of them pay attention to optimizing parity update.

There are two ways, read-modify-write (RMW) and read-

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

reconstruct-write (RCW), to update the parity blocks. RMW reads an old parity block and several data blocks in the same stripe to modify the parity block, while RCW reads all data blocks in the same stripe to reconstruct a parity block. The existing scaling approaches update the parity blocks with RMW by default. However, we find that in many cases for the existing scaling approaches, RCW outperforms RMW in terms of scaling I/Os. Combining RCW and RMW, the parity update processes of existing scaling approaches can be further optimized. Furthermore, since RCW has to read more data blocks, it may have more overlaps between scaling I/O requests and user I/O requests. By caching the data blocks acquired by the scaling I/Os for an appropriate time, RCW may save more user I/Os.

In this paper, we propose an efficient parity update mechanism (*abbr. EPU*), which first calculates the numbers of I/Os with RMW and RCW to update a group of parity blocks, and then chooses the one with less I/Os. We also use the scaling I/Os to save the user I/Os so as to enhance the online scaling performance of EPU. Our EPU mechanism is applicable to various existing scaling approaches to save the scaling I/Os and improve the online scaling performance. In particular, we use Scale-CRS [25] as an example, and deploy EPU atop Scale-CRS (abbr. Scale-CRS+EPU). Our numerical studies show that Scale-CRS+EPU reduces the amount of scaling I/Os by 1.22% to 31.15% compared to Scale-CRS under different scaling scenarios. We implement both Scale-CRS+EPU and Scale-CRS in a simulator that uses Disksim as the storage components to verify the effectiveness of our EPU mechanism in online scaling environment. Experimental results demonstrate that Scale-CRS+EPU reduces the scaling time by as much as 17.07% and the user response time by as much as 7.39% over Scale-CRS.

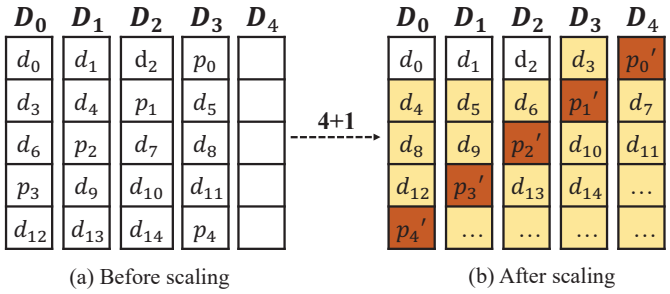
The rest of the paper is organized as follows. Section II reviews some existing scaling methods and illustrates the differences between RCW and RMW via a motivating example. Section III details the design of *EPU*. Section IV provides the numerical studies of EPU and Scale-CRS under plenty of CRS scaling scenarios. Section V presents the performance evaluation and discussions on the experimental results. Finally, Section VI concludes this paper.

II. Background and Motivation

In this section, we first review some background on the existing scaling methods, and then illustrate the differences between RMW and RCW via a motivating example.

A. Background

Traditional methods aim to preserve the round-robin data distribution after adding new disks (e.g., Gradual Assimilation(GA) [16], ALV [17]), and these methods are called *round-robin scaling*. For example, GA uses a predefined schedule to ensure that the scaling process is finished at a given time for scaling RAID-5 systems. ALV adjusts the migrating order of data blocks and aggregates small I/Os into large I/Os for RAID-5 scaling. Round-robin scaling maintains the desired



(a) Before scaling (b) After scaling

Fig. 1. An example of round-robin scaling for RAID-5 from 4 disks to 5 disks.

data layout after scaling. However, it is penalized with huge data migration overhead and parity update cost. To illustrate round-robin scaling, we show an example in Figure 1, which scales a RAID-5 system with 4 disks to one with 5 disks by using round-robin scaling. Here $D_0 \sim D_4$ denote disks and D_4 is the newly added one, and we emphasize that in this paper we only consider homogeneous disks where each disk is configured with the same I/O throughput. d_i denotes a data block and p_j denotes a parity block. As shown in Figure 1, all data blocks are moved except those in the first stripe (i.e., d_0, d_1, d_2). Worse still, all parity blocks should be updated accordingly.

To alleviate the huge migration overhead and update cost, lately proposed methods try to migrate the minimum amount of data blocks from old disks to new disks to achieve an uniform data distribution after scaling, and meanwhile optimize the parity update process. These methods include FastScale [15], MDM [19], GSR [20], MiPiL [21], RS6 [23], SDM [22], Scale-RS [24], Scale-CRS [25], which are called *minimum data migration scaling* in this paper. For instance, FastScale minimizes data migration for RAID-0 scaling, and achieves almost identical post-scaling performance as that of round-robin scaling. MDM minimizes data migration for RAID-5 scaling. It does not need to update the parity blocks, but the storage efficiency cannot be improved after scaling and the post-scaling data layout is disordered. GSR is also for RAID-5 scaling. It maintains most of the stripes and destructs a small portion of stripes to achieve minimum data migration. MiPiL is another scaling method for RAID-5, it maintains uniform data and parity distribution after scaling, and uses piggyback parity update and lazy metadata update to further optimize the scaling process.

For RDP-coded RAID-6 scaling, RS6 exploits piggyback parity update and further select the logical disk numbers of the new disks to reduce the amount of I/Os for parity update. For various RAID-6 codes (e.g., RDP, P-Code) scaling, SDM minimizes data migration and optimizes parity update based on the future parity layout.

Nowadays, RS codes and CRS codes are most widely deployed in large-scale distributed storage systems to support higher fault-tolerant ability. Scale-RS scales RS-coded storage systems with minimum volume of data migration and reduces

network traffic for parity update via generating intermediate results of several data blocks that stored in the same disks. Scale-CRS is an efficient approach for scaling CRS-coded storage systems. It minimizes data migration and optimizes parity update for scaling different instances of CRS codes. We illustrate Scale-CRS via an example in Figure 2.

As shown in Figure 2, data blocks d_2, d_3 are migrated from D_0 to D_2 and data blocks d_6, d_7 are migrated from D_1 to D_3 to achieve the minimum data migration. Scale-CRS determines a post-scaling encoding matrix and a data migration policy so as to update the parity blocks with optimized I/O overhead. Scale-CRS adopts RMW to update parity blocks. For example, $p_0^{'} = p_0 \oplus d_2 \oplus d_6$, $p_1^{'} = p_1 \oplus d_3 \oplus d_7$, $p_2^{'} = p_2 \oplus d_2 \oplus d_6$, $p_3^{'} = p_3 \oplus d_3 \oplus d_7$, $p_4^{'} = p_4 \oplus d_2 \oplus d_3 \oplus d_7$, $p_5^{'} = p_5 \oplus d_7$, $p_6 = p_6 \oplus d_2 \oplus d_6$, $p_7^{'} = p_7 \oplus d_2 \oplus d_3 \oplus d_6 \oplus d_7$.

B. Motivation

With RMW to update a parity block, we should read the parity block and several data blocks in the same stripe, and then modify it. While with RCW, we should read all the data blocks, and then reconstruct the parity block. The existing scaling approaches mainly focus on minimizing data migration and keeping even data distribution after scaling. They pay no attention to the scheme for parity update and update parity blocks with the default RMW scheme. However, we conducted simulations and found that for many scaling scenarios, RCW performs better with fewer scaling I/Os.

We still use Figure 2 to briefly introduce RCW and RMW, and motivate our work. In Figure 2, $p_0 = d_0 \oplus d_4$, $p_0^{'} = d_0 \oplus d_4 \oplus d_2 \oplus d_6$. If we use RMW to update p_0 to $p_0^{'}$, we should read d_2, d_6 and p_0 first, and then perform the operation $p_0^{'} = p_0 \oplus d_2 \oplus d_6$. Otherwise, if we use RCW, we should read d_0, d_2, d_4, d_6, and then perform the operation $p_0^{'} = d_0 \oplus d_4 \oplus d_2 \oplus d_6$. From above, we know that to update p_0 to $p_0^{'}$, RMW needs to read 3 data/parity blocks, while RCW needs to read 4 data blocks.

However, from another point of view, we can consider the update of $p_0 \sim p_7$ as one task. During the scaling process, we migrate d_2, d_3 from D_0 to D_2 and d_6, d_7 from D_1 to D_3. Therefore, we must read d_2, d_3, d_6, d_7 for data migration, and we can cache them for parity update. Now we rethink the processes of RCW and RMW for parity update.

If we use the default RMW scheme to update the parity blocks, data blocks d_2, d_3, d_6, d_7 are needed to update the parity blocks $p_0 \sim p_7$. Considering that d_2, d_3, d_6, d_7 can be cached in main memory when performing the data migration process, the scaling process requires to read $p_0 \sim p_7$ and write $p_0 \sim p_7$ to complete the parity modification process. Hence, the parity update I/Os include eight reads as well as eight writes.

Alternatively, if we use RCW, $d_0 \sim d_7$ are needed to re-calculate $p_0^{'} \sim p_7^{'}$. Similarly, because d_2, d_3, d_6, d_7 are cached in main memory, the scaling process must read d_0, d_1, d_4, d_5 and write $p_0^{'} \sim p_7^{'}$ to complete parity update. Thus, the parity update I/Os are reduced to four reads and eight writes.

III. Efficient Parity Update Mechanism

Our primary objective is to minimize I/Os for the update of parity blocks. We propose the *EPU* algorithm to select one from RCW and RMW with fewer I/Os (See subsection III-A2). To enhance the online scaling performance of a scaling process, we also propose two techniques to further optimize the scaling process, i.e., the overlap between scaling I/Os and user I/Os, and the access aggregation technique (See subsection III-B).

A. EPU *Algorithm*

1) Algorithm Preliminaries: We divide a RAID into units, each of which consists of one or several stripes. The scaling scheme in each unit is exactly the same, so we only focus on the scaling within one unit.

The scaling process is composed of data migration and parity update. The data migration process reads data blocks to be migrated from old disks, and writes them into new disks. The parity update process is in charge of renewing the parity blocks. We select one from RCW and RMW with fewer I/Os for parity update, while considering that the migrated data blocks are already in memory during the data migration process and they do not need to be read from disk again for parity update.

This paper is to optimize the parity update process for RAID scaling. Given a scaling approach, we decide to select RCW or RMW to update a parity block. With RMW, the parity update process consists of:

- Read parity blocks to be updated;
- Read the data blocks for updating the parity blocks (except for the migrated data blocks);
- Update the parity blocks;
- Write back the updated parity blocks.

While with RCW, the parity update process becomes:

- Read the original data blocks (except for the migrated data blocks);
- Reconstruct the parity blocks;
- Write back the updated parity blocks.

To calculate the numbers of I/Os for updating the parity blocks, we define the following sets to record the data/parity blocks to be read or the number of blocks :

- A is a set consisting of the parity blocks in a scaling unit.
- B is a set consisting of the migrated data blocks in a scaling unit.
- C is a set consisting of the data blocks which are needed for updating the parity blocks with RMW.
- D is a set consisting of the data blocks which are needed for updating the parity blocks with RCW.
- E is a set consisting of all data blocks which are encoded into a parity block p before scaling.
- F is a set consisting of all data blocks which are encoded into a parity block $p^{'}$ after scaling.
- m is a variable to record the number of all data/parity blocks (except for the migrated ones) for updating the parity blocks in a unit with RMW.

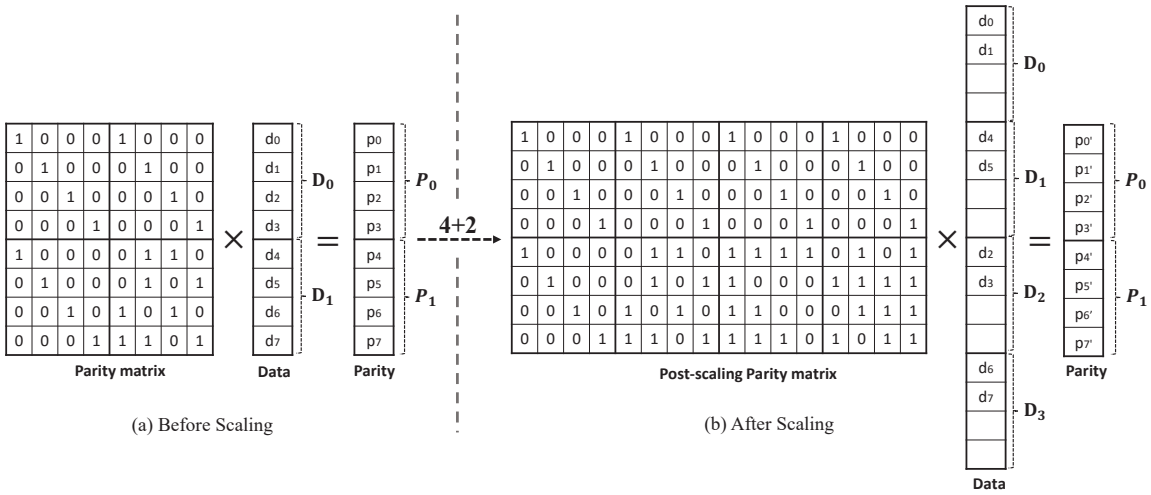


Fig. 2. An example of CRS scaling from 4 disks to 6 disks.

Algorithm 1 The Efficient Parity Update Scheme In Scaling Process (*EPU*)

1: **Initialization:**
$A \leftarrow \{the\ parity\ blocks\ of\ a\ scaling\ unit\}$, $B \leftarrow \{the\ moved\ data\ blocks\ of\ a\ scaling\ unit\}$, $C, D, E, F \leftarrow \emptyset$, $m \leftarrow |A|, n \leftarrow 0$;
2: **for** $P \in A$ **do**
3: $E \leftarrow \{the\ data\ blocks\ that\ encodes\ P\}$;
4: $F \leftarrow \{the\ data\ blocks\ that\ encodes\ P'\}$;
5: $G \leftarrow (E \cup F) - (E \cap F)$;
6: $C \leftarrow C \cup G$, $D \leftarrow D \cup F$;
7: **if** $G = \emptyset$ **then**
8: $m \leftarrow m - 1$
9: **end if**
10: **end for**
11: let $C \leftarrow C - B$;
12: let $D \leftarrow D - B$;
13: $m \leftarrow m + |C|$,$n \leftarrow |D|$;
14: **if** $m < n$ **then**
15: use RMW to update parity
16: **else**
17: use RCW to update parity
18: **end if**

- n is a variable to record the number of all data blocks (except for the migrated ones) for updating the parity blocks in a unit with RCW.

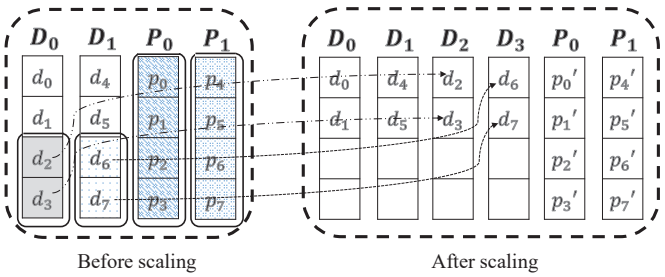
Now we are ready to present the parity update algorithm.

2) Algorithm Design: To balance the data distribution after the scaling process and minimize the data/parity blocks to be migrated, approaches for RAID scaling usually group some stripes into a unit and scale a unit to some new stripes after scaling. Algorithm 1 is operated on a scaling unit. As the data migration process is decided, we can get the set of data/parity blocks to be moved. For each parity block in a scaling unit, we can first get the data/parity blocks needed for updating it with RCW and RMW respectively. Then we can obtain the sets of data/parity blocks needed for updating the parity blocks in a unit with RCW or RMW respectively. The elements in the sets except those being migrated are blocks needed to be read from disks for updating the parity blocks. Comparing the cardinality of the two sets, we can conclude which parity update scheme, RCW or RMW, is better to update parity blocks. In the algorithm, Line 1 to Line 10 are dedicated to calculate the data/parity blocks for parity updating, Line 11 to Line 13 are dedicated to subtract the migrated data blocks from the total needed blocks.

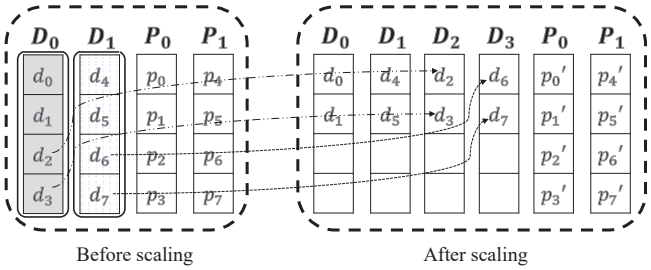
Now we use the example in Figure 2 to elaborate the algorithm execution flow. In this example, a scaling unit consists of 8 data blocks and 8 parity blocks. The migrated data blocks are d_2, d_3, d_6, d_7. Thus, the set of migrated data blocks $B = \{d_2, d_3, d_6, d_7\}$, and the set of parity blocks $A = \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. The parity block before scaling $p_0 = d_0 \oplus d_4$, and the parity block after scaling $p_0{}' = d_0 \oplus d_4 \oplus d_2 \oplus d_6$. Then we can get set $E = \{d_0, d_4\}$ and $F = \{d_0, d_4, d_2, d_6\}$. Next we calculate the data blocks need to be read for updating p_0 with RMW, $G = (E \cup F) - (E \cap F) = \{d_2, d_6\}$. Along with the other parity blocks (i.e., p_1, p_2, p_3) are processed the same as p_0, we can get $C = \{d_2, d_3, d_6, d_7\}$, $D = \{d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$. Finally, we subtract the elements in set B from set C and D. Now, we get $C = \emptyset$, $D = \{d_0, d_1, d_4, d_5\}$, $m = 8, n = 4$ (Note that for this example, m is the number of parity blocks which need to be read by the RMW scheme, n is the number of data blocks which need to be read by the RCW scheme). Thus, RCW is better than RMW in this example. EPU chooses the RCW scheme.

B. Online Optimization

1) The overlap between scaling I/Os and user I/Os: RAIDs are usually scaled online, i.e., they still serve user requests while being scaled. That is to say, user I/Os and scaling I/Os are processed interactively. We can further exploit user I/Os



(a) Aggregation I/Os of *RMW*



(b) Aggregation I/Os of *RCW*

Fig. 3. Aggregation I/Os

to speed up the scaling process. If a data block is accessed by a user request, it will be resident in memory. If this data block is also accessed by the scaling process, the system can read it from memory rather than from disk.

All blocks read by RCW to update parity are data blocks, which may overlap with the user I/Os. However, RMW needs to read parity blocks for parity update, while the parity blocks will not be accessed by user I/Os (*Note: we omit the degraded read accesses by user-level applications in this paper*). That is to say, the probability of the blocks needed by RCW overlap with the user I/Os is higher than that of RMW.

2) Access Aggregation: Since hard disk performs better with sequential access than random access, the response time of a disk depends not only on the volume of data/parity blocks being accessed but also their sequentiality. Thus, we also consider I/O aggregations of RCW and RMW in EPU. As shown in Figure 3(a), if we use RMW to update the parity blocks in the stripe, the system will issue four read requests, the first two requests read blocks d_2, d_3 and d_6, d_7 respectively, and the third request reads parity blocks p_0, p_1, p_2, p_3, and the fourth request reads parity blocks p_4, p_5, p_6, p_7. However, if we use RCW, as shown in Figure 3(b), the system issues only two read requests, the first request reads blocks d_0, d_1, d_2, d_3, and the second request reads blocks d_4, d_5, d_6, d_7.

Access aggregation converts several small requests into a large sequential request. As a result, the seek overhead is amortized by multiple requests. Thus, if the data/parity blocks that need to be read for parity update can be accessed through a single larger I/O, the system will have a higher throughput. Since data blocks needed by RCW are always successive, it may benefit the scaling process compared with RMW even if it needs a little more blocks.

IV. Numerical Study

In this section, we compare the numbers of read I/Os (*Note: without considering I/O overlap and access aggregation*) when we use Scale-CRS+EPU (abbr. EPU) and Scale-CRS to scale some common system settings of CRS codes. Table I and Table II show the number of scaling cases under which EPU can improve the performance. Figure 4 shows how many scaling I/Os EPU can reduce under certain system settings.

TABLE I
Number of advantageous scenarios of EPU when using Jerasure library to generate the post-scaling matrices.

m	2	3	4	5
Total (cases)	324	304	284	263
EPU (cases)	16	285	269	255
Equal (cases)	308	19	15	8

TABLE II
Number of advantageous scenarios of EPU when using the optimized post-scaling matrices.

m	2	3	4	5
Total (cases)	324	304	284	263
EPU (cases)	16	25	36	48
Equal (cases)	308	279	248	215

Scale-CRS should determine a post-scaling encoding matrix and a data migration policy to complete the scaling process. The post-scaling encoding matrix can be generated in two ways, one is from the Jerasure library, the other is from [25]. Jerasure is an open-source erasure code library developed by J.S. Plank et. al [26]. It is an efficient implementation of various erasure codes with optimized encoding performance. The method described in [25] is specially designed for generating post-scaling matrix to reduce the scaling I/Os. However, the matrix generated as in [25] may sacrifice the encoding performance compared with those generated by Jerasure library. We will evaluate the impact of the two matrix generation methods in the following.

Firstly we examine whether EPU reads fewer data/parity blocks for parity update. Suppose that we are to scale a (k, m, w) CRS code to a $(k + t, m, w)$ one, where k, m, t, w are the number of data disks before scaling, the number of parity disks, the number of data disks to be added, and the number of blocks on each disk in a stripe. We use the two ways to generate the post-scaling matrices and run the cases of $3 \leq w \leq 4$, $2 \leq m \leq 5$, $t \leq k$ and $2 \leq k \leq 17$ with EPU and Scale-CRS. For each case, we record the numbers of data/parity blocks to be read in a unit by EPU and Scale-CRS, respectively. The results are shown in Table I and Table II respectively, where "Equal" means EPU and Scale-CRS read the same number of data/parity blocks for parity update and "EPU" means EPU reads fewer data/parity blocks than Scale-CRS.

As we can see in Table I, when $m = 2$, there are 324 scaling cases in total, and EPU has better performance under 15 cases, while EPU has identical performance to Scale-CRS

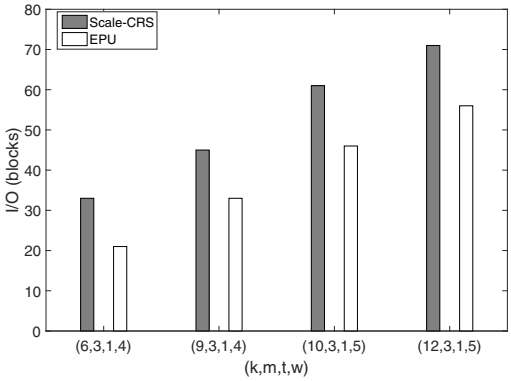


Fig. 4. I/O comparison between Scale-CRS and EPU

under the remaining 309 cases. When $m = 3$, there are 304 scaling cases in total, 285 of them are better to use EPU, and 19 cases are free to choose EPU or Scale-CRS. When $m = 4$, EPU outperforms Scale-CRS under 284 scaling cases and has the identical performance under the remaining 15 cases. For $m = 5$, EPU performs better under 255 cases and has the same performance with Scale-CRS under 8 cases. The results when using the method as described in [25] to generate the post-scaling matrix are shown in Table II. We can see that even if the optimized post-scaling matrix is used, there are still plenty of cases where EPU can reduce the amount of scaling I/Os of Scale-CRS.

We can get several conclusions from the above comparisons. One is that in many scaling cases, EPU can reduce the amount of scaling I/Os over Scale-CRS, while in other cases, EPU has the same performance to Scale-CRS. The reason is that Scale-CRS always chooses RMW, while EPU is more intelligent to choose RMW or RCW. The other is that the percentage that EPU performs better grows as m increases. The reason is that it is more likely to use RCW to update the parity when there are relatively more parity blocks.

In our numerical analysis of read I/Os for parity update, when using Jerasure library, for $3 \leq w \leq 6$, $2 \leq m \leq 5$, $t \leq k$ and $2 \leq k \leq 17$, EPU can reduce 3.13% to 31.15% read/write compared with Scale-CRS for CRS scaling. Even if optimized post-scaling encoding matrix is used, our numerical analysis shows that EPU can reduce 1.22% to 16.67% read/write over Scale-CRS in the advantageous cases. We will demonstrate in the following section that these savings of scaling I/Os will convert directly into the reduction of scaling completing time and user response time in online scaling.

In particular, we show the numerical study results of some common scaling cases in Figure 4. The quadruple in x-axis means (k,m,t,w). For example, when (k,m,t,w) is (6,3,1,4), EPU needs to read 21 data blocks to update the parity blocks in a stripe while Scale-CRS needs to read 33 data/parity blocks. Note that these parameters are commonly deployed in real storage systems. We will evaluate the scaling performance of EPU with these system settings in our experiments.

We also emphasize that EPU is applicable to various existing scaling approaches. To be more specific, for round-robin scaling approaches, our EPU algorithm always chooses RCW to update parity blocks, which typically requires less amount of scaling I/Os than the default RMW. For Scale-RS, when the number of unmoved data blocks in each stripe is less than the number of parity blocks in each stripe, EPU atop Scale-RS can reduce the scaling I/Os of Scale-RS. Otherwise, EPU atop Scale-RS performs the same as the original Scale-RS. For other approaches like GSR, MiPiL, RS6, our EPU algorithm can also enhance their performance in some scaling cases.

V. Experimental Evaluation

In the last section, our numerical analysis shows that EPU reduces the scaling I/Os in some cases. In this section, we examine the performance of Scale-CRS and Scale-CRS+EPU (abbr. EPU) in some cases with a simulation system.

A. The Configuration of Simulation System

We conduct detailed simulations using a simulator that uses Disksim as the slave module. Our simulator consists of a workload generator and a disk array as shown in Figure 5. The workload generator consists of a trace parser, which reads trace files and issues I/O requests at the appropriate time so that an exact workload is reproduced on the disk array.

The disk array is composed of an array controller and the storage components. The array controller is logically divided into three parts, a workload analyzer, an address mapper and a data redistributor. The workload analyzer monitors the user I/O rate, which will be used by the data redistributor. The address mapper, according to the data layout strategy, forwards incoming I/O requests to the corresponding disks. The data redistributor is in charge of reorganizing the data on the array and it adjusts the scaling I/O rate based on the user I/O rate and the maximum I/O throughput of the disks.

We implement the workload generator and the array controller in C++ on linux system, and use DiskSim [27] to simulate the bottom storage components. DiskSim is an efficient, accurate and highly-configurable disk system simulator developed to support research on various aspects of storage subsystem architecture. Disksim has been widely used in various research projects. The simulated disks are configured as Seagate Cheetah 15000 RPM, each with a capacity of 146.8GB. We deploy our simulator on an Inter i5-4460 Quad-Core 3.20GHz PC with 4GB DRAM.

B. Workloads

In our experiments, We use the following real-system disk I/O traces with different characteristics.

- WebSearch is obtained from the Storage Performance Council (SPC) [28], a vendor-neutral standards body. It was collected from a system running a web search engine. The read dominated WebSearch trace exhibits the strong locality in its access pattern.
- Financial is also from SPC [28]. It was collected from OLTP applications running at a large financial institution, and it is write dominated.

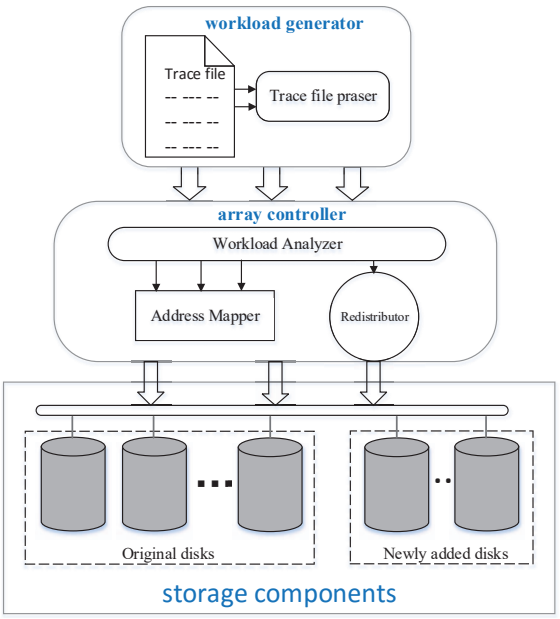


Fig. 5. Simulation system diagram.

C. The Design of Experiments

We conduct extensive online scaling experiments to evaluate the performance of Scale-CRS and EPU. In our experiments, the scaling process is operated stage by stage. In each stage, the system serves S user requests at the first chance. Then the system issues several scaling requests at the end of a stage. The time duration of each stage is denoted as a time slot. The number of user requests at each time slot is determined by the workload. In order to control the negative effect of the scaling process, we need to adjust the scaling I/O rate based on the user I/O rate and maximum I/O rate of the disks. The user I/O rate (scaling I/O rate) is calculated as the amount of blocks accessed by user requests (scaling requests) divided by the time span between the first user request and the last user request during a scaling stage. The maximum I/O rate of the disks is the upper bound of request transmission frequency. If the user request transmission frequency exceeds the upper bound, then we will not issue any scaling requests in this time slot. Based on the user I/O rate and maximum I/O rate, we can decide the amount of scaling requests issued at each time slot, so that we can control the response time of each user request being acceptable. In our experiments, we set the max I/O rate configurable, and we vary the max I/O rate so as to simulate different available bandwidths of the disks.

Note that EPU chooses RCW or RMW to update parity blocks more efficiently. However, EPU does not change the data layout after scaling, and so the post-scaling performance of EPU is exactly the same as that of Scale-CRS. Here we emphasize that EPU mainly focuses on reducing the overhead that is induced by parity update during the scaling process. As a result, it can accelerate the scaling process and reduce the user I/O response time during scaling.

D. Experiment Results

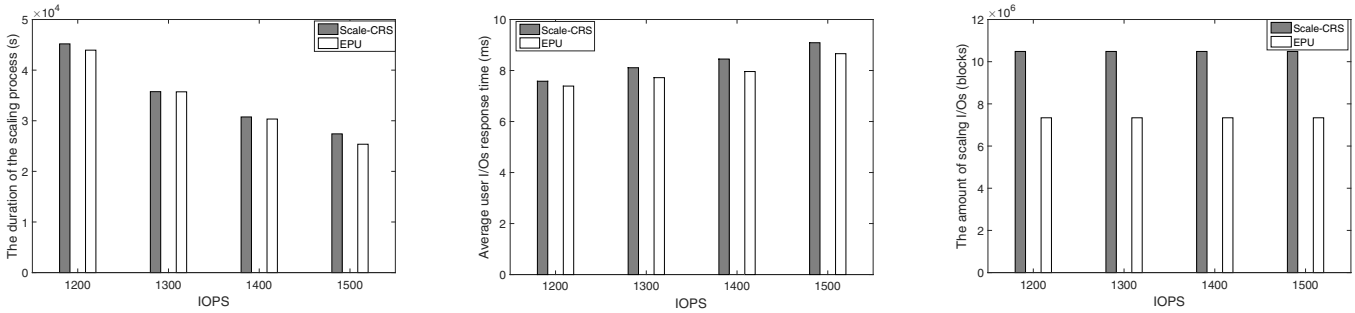
There are several factors, scaling I/O rate, disk capacity, foreground workload, and system scale, which may affect the online scaling performance. In the following, we will evaluate the performance of EPU with different settings.

Performance with different I/O rates. In this experiment, we aim to evaluate the influence of scaling I/O rate on the system performance (including the completion time of the scaling process, user I/O response time, the amount of scaling I/Os). We take the scaling scenario of adding one data disk (i.e., t=1) to a nine-disk CRS-coded RAID ($k = 6, m = 3, w = 4$) as an example. The disk size is fixed as 4GB and the block size is fixed as 4KB. We use the WebSearch2 as the foreground workload. We use the maximum I/O rates of the disks, 1200, 1300, 1400, 1500 blocks per second. The results are shown in Figure 6.

Figure 6(a) shows the time durations of each scaling operation, and we can see that for this scaling scenario, EPU performs better than Scale-CRS in terms of the time duration of the scaling process. The main reason of this improvement is that in this scaling scenario, EPU uses RCW to reduce the amount of scaling I/Os. We further record the amount of scaling I/Os to verify our analysis, and the result is shown in Figure 6(c). As we can see, EPU reduces the amount of scaling I/Os over Scale-CRS significantly, which confirms our analysis.

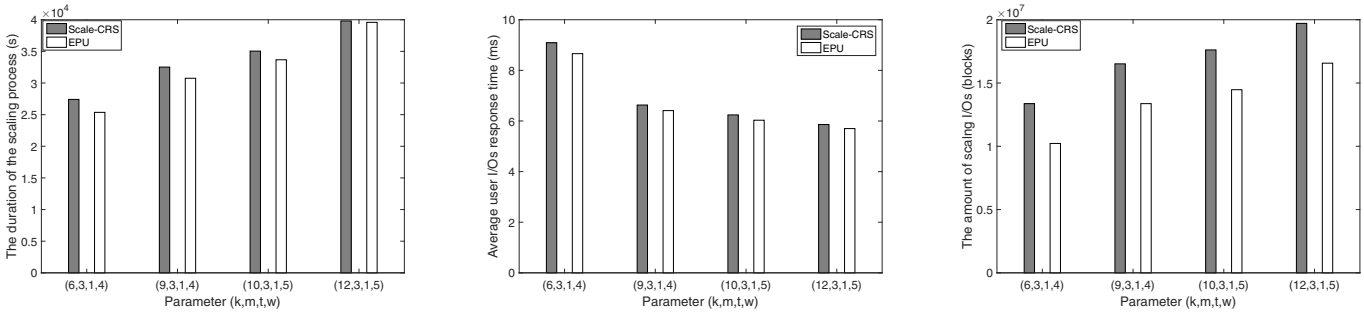
Since we control the I/O rate in the scaling process, almost the same number of blocks will be issued into the storage system in a time slot. Therefore, we expect that the average response times of user I/Os for EPU and Scale-CRS are almost the same. However, as we see in Figure 6(b), compared with Scale-CRS, EPU has a slight improvement in user I/O response time. The reason is that although EPU issues nearly the same number of blocks in each time slot as Scale-CRS does, EPU uses the I/O overlap and access aggregation techniques to save more I/Os. As the maximum I/O rate increases from 1200 blocks per second to 1500 blocks per second, the time duration of the scaling process with EPU decreases from 43946 seconds to 25362 seconds while that with Scale-CRS decreases from 45187 seconds to 27410 seconds. We can see that EPU outperforms Scale-CRS under different system loads. Further more, we can see that, as the maximum I/O rate increases, the average response time of user I/Os increases. The reason is that when the maximum I/O rate increases, there will be more scaling requests issued at each scaling time slot and the contention between scaling I/Os and user I/Os becomes more severe. As a result, the load of each disk becomes heavier and the user request response time becomes longer.

Performance with different system scales. In the above experiments, we vary the maximum I/O rate to evaluate the impact of scaling I/O rate on the performance of EPU. Now we fix the maximum I/O rate as 1500 blocks per second. We examine the scaling performance with some commonly used system scales of (k=6, m=3, t=1, w=4), (k=9, m=3, t=1, w=4), (k=10, m=3, t=1, w=5), (k=12, m=3, t=1, w=5)). In this



(a) Scaling Completion Time (b) Average user I/O response time (c) The amount of scaling I/Os

Fig. 6. The influence of IOPS to the scaling process



(a) Scaling Completion Time (b) Average user I/O response time (c) The amount of scaling I/Os

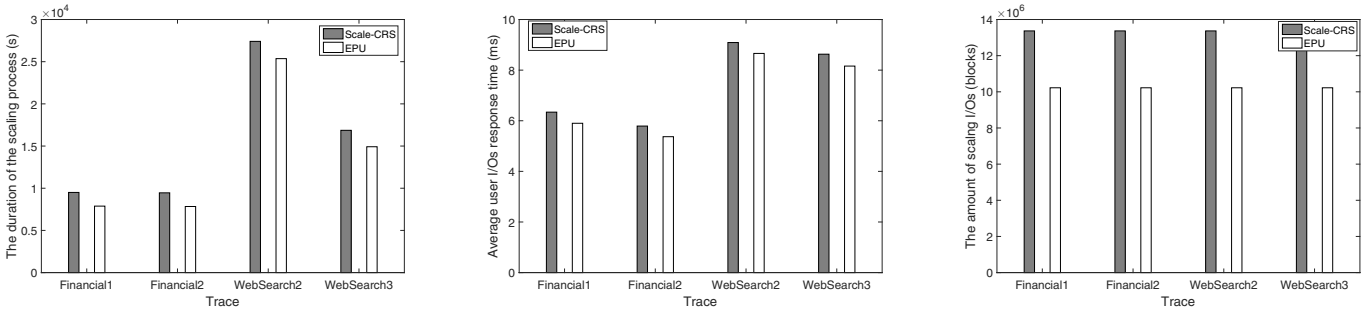
Fig. 7. The performance of EPU and Scale-CRS under different parameters

experiment, we still use the disk size of 4GB and WebSearch2 as the foreground workload. As we can see in Figure 7, the online scaling result is consistent with our numerical analysis in Section IV. For these system scales, EPU outperforms Scale-CRS in terms of the time duration of the scaling process, average response time of user I/Os during the scaling process, number of scaling I/Os. Figure 7(a) shows the time durations of each scaling cases, for (k=6, m=3, t=1, w=4), our EPU needs 25362 seconds to complete the scaling process while Scale-CRS needs 27410 seconds. In other words, EPU reduces the scaling time by 7.47% compared with Scale-CRS. For (k=9, m=3, t=1, w=4), (k=10, m=3, t=1, w=5) and (k=12, m=3, t=1, w=5), EPU reduces the scaling time by 5.44%, 3.91% and 0.56%, respectively. The reason of this improvement is that our EPU mechanism needs fewer blocks to update parity, as shown in Figure 7(c), EPU really reduces the number of scaling I/Os. Figure 7(b) shows the user I/O response time during the scaling process. For (k=6, m=3, t=1, w=4), (k=9, m=3, t=1, w=4), (k=10, m=3, t=1, w=5) and (k=12, m=3, t=1, w=5), EPU can reduce the user I/O response time by up to 4.78%, 3.36%, 3.33% and 2.83%, respectively.

Performance under different workloads. Now we aim to examine the performance of EPU with different workloads. We set the maximum I/O rate as 1500 blocks per second, and the disk size is fixed as 4GB, and use system scale (k=6, m=3, t=1, w=4). Then we conduct the scaling process with Financial1, Financial2, WebSearch2 and WebSearch3. Figure 8 presents the performance of EPU and Scale-CRS with different workloads. With the Financial1 workload, EPU needs 7881 seconds to finish the scaling process, while Scale-CRS needs 9503 seconds. That is to say, EPU reduces the scaling time by 17.07%. With Financial2, WebSearch2, and WebSearch3 workloads, EPU can reduce the scaling time by 17.15%, 7.47% and 11.54%, respectively. Figure 8(b) shows the average response time of user requests, we can find that EPU reduces the response time of user requests by 6.96% under Financial1 workload. For Financial2, WebSearch2 and WebSearch3 traces, EPU can reduces the response time of user request by 7.39%, 4.78% and 5.42%. We can see that EPU perform better with the Financial traces than with WebSearch traces. We find that our EPU perform better under heavy foreground workload.

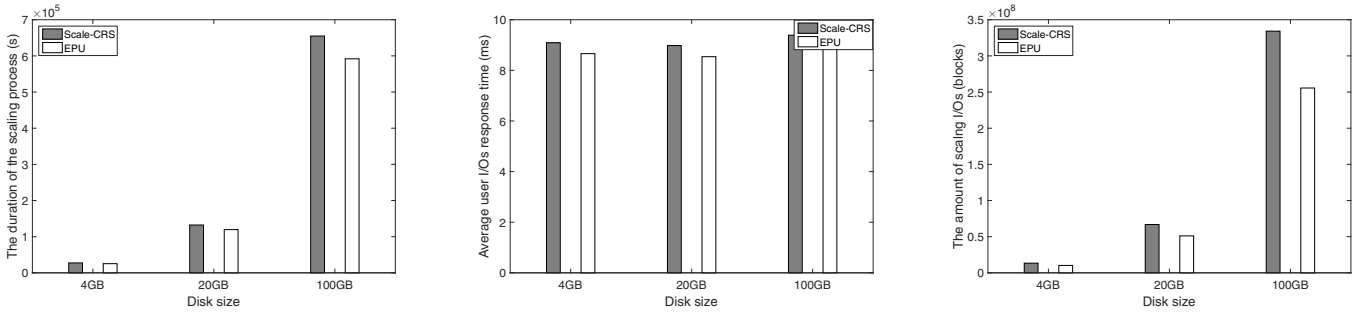
Performance with different disk capacities. In this experiment, we vary the disk capacities as 4GB, 20GB and 100GB. We also use the system scale of (k=6,m=3,t=1,w=4) as the example. The maximum I/O rate is set as 1500 blocks per second. We still use WebSearch2 as the foreground workload. Due to the disk capacity is increased to 100GB, if the online scaling process does not finish when the workload generator reaches the end of the trace file, the workload will be replayed again until the online scaling process finishes.

As the capacity of a disk increases from 4GB to 20GB and 100GB, the time duration of the scaling operation increases linearly. This is because that scaling larger disks needs more



(a) Scaling Completion Time (b) Average user I/O response time (c) The amount of scaling I/Os

Fig. 8. The performance of EPU and Scale-CRS under different workloads



(a) Scaling Completion Time (b) Average user I/O response time (c) The amount of scaling I/Os

Fig. 9. The influence of disk size to the scaling process

scaling I/Os, and the amount of scaling I/Os increases linearly with the increase of the capacity. Figure 9(c) verifies our analysis. We can see that there is a small fluctuation of the response time of user request as shown in Figure 9(b), this is because of the locality of user requests. When the user requests are near the processing scaling area, the user request will get faster response because of less disk seek. This inspires us to scale the area accessed by user requests currently at the first chance. Therefore, the user requests will get a shorter response time and there will be more overlaps between scaling I/Os and user I/Os. We pose the analysis of this factor as our future work.

VI. Conclusions

In this paper, we propose an online scaling scheme, EPU, for RAID system with erasure code. EPU selects RCW or RMW with less I/Os to update parity blocks. EPU aims to accelerate the scaling process by reducing the scaling I/Os induced by parity update. To further improve online scaling performance, EPU uses I/O overlap and access aggregation. EPU is applicable to various scaling approaches to save the scaling I/Os and improve the online scaling performance. Using Scale-CRS as an example, our numerical study shows that Scale-CRS+EPU can reduce the number of scaling I/Os in many scaling cases. We have also implemented a trace-driven simulator, and conducted detailed experiments using this simulator. The results show that EPU can reduce the scaling I/Os by 1.22% to 31.15%. It is also efficient in reducing the scaling time and improving the response time of user requests during the scaling process. Besides, EPU can be readily deployed in real storage systems.

Acknowledgment

This work was supported by National Nature Science Foundation of China under Grant No. 61379038 and 61303048, Anhui Provincial Natural Science Foundation under Grant No. 1508085SQF214 and Huawei Innovation Research Program under Grant No. HIRPO20140301.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the International Conference on Management of Data (SIGMOD '88)*. ACM, Jun 1988, pp. 109–116.

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

[3] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-Code: A New RAID-6 Code with Optimal Properties," in *Proceedings of the 23rd International Conference on Supercomputing (ICS)*. ACM, 2009, pp. 360–369.

[4] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, Jan 1999.

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

[6] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn *et al.*, "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage." in *FAST*, vol. 9, 2009, pp. 253–265.

[7] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[8] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," ICSI Technical Report No. TR-95-048, August 1995.

[9] A. Oriani, I. C. Garcia, and R. Schmidt, "The Search for a Highly-Available Hadoop Distributed Filesystem," Citeseer, Tech. Rep., 2010.

[10] A. Fikes, "Storage architecture and challenges," *Talk at the Google Faculty Summit*, 2010.

[11] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems." in *OSDI*, 2010, pp. 61–74.

[12] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Computing*, vol. 5, no. 5, pp. 40–49, Sep 2001.

[13] S. Ghandeharizadeh and D. Kim, "On-line reorganization of data in scalable continuous media servers," in *Database and Expert Systems Applications*. Springer, 1996, pp. 751–768.

[14] D. A. Patterson *et al.*, "A Simple Way to Estimate the Cost of Downtime." in *LISA*, vol. 2, 2002, pp. 185–188.

[15] W. Zheng and G. Zhang, "FastScale: Accelerate RAID Scaling by Minimizing Data Migration." in *FAST*, 2011, pp. 149–161.

[16] J. L. Gonzalez and T. Cortes, "Increasing the capacity of RAID5 by online gradual assimilation," in *SNAPI '04:Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os, Antibes Juan-les-Pins*, Sep 2004, pp. 17–24.

[17] G. Zhang, W. Zheng, and J. Shu, "ALV: A New Data Redistribution Approach to RAID-5 Scaling," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 345–357, March 2010.

[18] A. Goel, C. Shahabi, S. Y. D. Yao, and R. Zimmermann, "SCADDAR: An efficient randomized technique to reorganize continuous media blocks," in *2002 18th International Conference on Data Engineering*, 2002, pp. 473–482.

[19] S. R. Hetzler, "Data storage array scaling method and system with minimal data movement," US Patent 20080276057, 2008.

[20] C. Wu and X. He, "GSR: A Global Stripe-Based Redistribution Approach to Accelerate RAID-5 Scaling," in *2012 41st International Conference on Parallel Processing (ICPP)*, Sep 2012, pp. 460–469.

[21] G. Zhang, W. Zheng, and K. Li, "Rethinking raid-5 data layout for better scalability," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2816–2828, 2014.

[22] 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 *2012 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep 2012, pp. 284–292.

[23] G. Zhang, K. Li, J. Wang, and W. Zheng, "Accelerate RDP RAID-6 Scaling by Reducing Disk I/Os and XOR Operations," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 32–44, Jan 2015.

[24] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie, "Scale-RS: An Efficient Scaling Scheme for RS-Coded Storage Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1704–1717, June 2015.

[25] S. Wu, Y. Xu, Y. Li, and Z. Yang, "I/O-Efficient Scaling Schemes for Distributed Storage Systems with CRS Codes," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[26] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2," Technical Report CS-08-627, University of Tennessee, Tech. Rep., 2008.

[27] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," *Parallel Data Laboratory*, p. 26, 2008.

[28] O. Application, "I/O. UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage. 2007.