Accelerate RDP RAID-6 Scaling by Reducing Disk I/Os and XOR Operations

Guangyan Zhang, Keqin Li, Jingzhe Wang, and Weimin Zheng

Abstract—Disk additions to an RAID-6 storage system can increase the I/O parallelism and expand the storage capacity simultaneously. To regain load balance among all disks including old and new, RAID-6 scaling requires moving certain data blocks onto newly added disks. Existing approaches to RAID-6 scaling, restricted by preserving a round-robin data distribution, require migrating all the data, which results in an expensive cost for RAID-6 scaling. In this paper, we propose RS6—a new approach to accelerating RDP RAID-6 scaling by reducing disk I/Os and XOR operations. First, RS6 minimizes the number of data blocks to be moved while maintaining a uniform data distribution across all data disks. Second, RS6 piggybacks parity updates during data migration to reduce the cost of maintaining consistent parities. Third, RS6 selects parameters of data migration so as to reduce disk I/Os for parity updates. Our mathematical analysis indicates that RS6 provides uniform data distribution, minimal data migration, and fast data addressing. We also conducted extensive simulation experiments to quantitatively characterize the properties of RS6. The results show that, compared with existing "moving-everything" Round-Robin approaches, RS6 reduces the number of blocks to be moved by 60.0%–88.9%, and saves the migration time by 40.27%–69.88%.

Index Terms—Data migration, load balance, migration parameter, parity update, RAID-6 scaling

1 Introduction

1.1 Motivation

RAID-6 [1]–[3] storage systems provide large I/O bandwidth via parallel I/O operations and tolerate two disk failures by maintaining dual parity. With higher possibility of multiple disk failures [4], [5], RAID-6 has received more attention than ever. RAID-based architectures are also used in clusters and large-scale storage systems [6], [7]. A typical instance is the NetApp WAFL file system [8] that works with RAID-6. As user data grow rapidly, applications often demand increasingly larger storage capacity. One solution is adding more disks to a RAID-6 array. It is also desirable that the bandwidth of a RAID-6 array increases with the member disks [9], [10]. Such disk addition is termed as *RAID-6 scaling*.

In order to regain load balance after RAID-6 scaling, data need to be redistributed evenly among all disks including old and new. In today's server environments, the cost of downtime is extremely high [11]. Therefore, RAID-6 scaling requires an efficient approach to redistributing the data online with the following requirements. (1) Data redistribution should be completed in a short time. (2) The impact of data redistribution on application performance should be quite low. (3) Data reliability should be guaranteed during the scaling process.

- *G. Zhang, J. Wang, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: gyzh@tsinghua.edu.cn, wjz416@gmail.com, zwm-dcs@tsinghua.edu.cn.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, New York 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 18 May 2013; revised 05 Oct. 2013; accepted 10 Oct. 2013. Date of publication 20 Oct. 2013; date of current version 12 Dec. 2014. Recommended for acceptance by Y. Pan.
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/TC.2013.210

There are multiple coding methods proposed for RAID-6 arrays. According to the layout of data and parity, RAID-6 codes can be categorized into horizontal codes [3], [13]–[15] and vertical codes [12], [16]–[20]. Existing scaling approaches are proposed for general case in RAID-0 or RAID-5 [21]–[27]. They cannot adapt to various coding methods in RAID-6, and therefore are not suitable for RAID-6 scaling. An efficient approach to RAID-6 scaling should be designed based on the characteristics of each coding method respectively. Currently, the RDP code represents the best performing RAID-6 codes for storage systems [28]. This paper focuses on the problem of RDP-based RAID-6 scaling.

Typical RAID scaling approaches [22], [24]–[26] preserve a round-robin data distribution after adding disks. Although Round-Robin is a simple approach to implement on RAID-6, it results in high overhead. First, all data blocks are migrated based on the round-robin order in the scaling process. Second, all parities need to be recalculated and modified during RAID scaling. The expensive cost for RAID-6 scaling means that either data redistribution will be completed in a long time, or the impact of data redistribution on application performance will be significant. There are some optimizations of data migration [22], [26] proposed for RAID scaling, e.g., I/O aggregation and rate control. They can be used to improve the performance of RAID-6 scaling to certain extent, but still suffer from large data migration and heavy parity updates.

1.2 Initial Idea and Technical Challenges

Our initial idea to improve the efficiency of RDP RAID-6 scaling is minimizing data migration in the process of RAID scaling. Zheng and Zhang [27] proposed the FastScale approach to accelerating RAID-0 scaling by minimizing data migration. FastScale provides a good starting point for efficient scaling of RAID-6 arrays. However, optimizing data

redistribution for RAID-6 scaling will be more difficult, due to the need of maintaining consistent, dual parities.

One of the reasons is that the dual parity layouts of RAID-6 codes are complex. Another reason is that the stripes are dramatically changed after scaling. Using FastScale in RAID-6 scaling directly will bring high overhead of parity migration, modification, and computation. Moreover, the parity disk in a RAID-6 array tends to become a performance bottleneck during RAID scaling. It is a technical challenge to design a new data redistribution scheme for RDP RAID-6 scaling, which provides uniform data distribution, minimal data migration, and efficient parity updates.

1.3 Our Contributions

In this paper, we propose RS6—a new data redistribution approach to accelerating RDP RAID-6 scaling. Different from the round-robin scheme, no data are moved among original disks. RS6 moves data blocks from old disks to new disks just enough to preserve the uniformity of data distribution. With the restriction of maintaining a uniform data distribution, RS6 reaches the lower bound of the migration fraction. Furthermore, the RS6 migrating scheme keeps as many blocks within the same row or diagonal parity set as possible. Consequently, there are more opportunities to use the read-modify-write alternative for efficient parity updates.

RS6 also optimizes online parity updates with two techniques. First, it piggybacks parity updates during data migration to reduce the numbers of additional disk I/Os and XOR computations. Second, with different migration parameters, the RS6 migrating scheme performs obviously different numbers of data reads or XOR operations dedicated for parity updates (see Section 6.2). RS6 selects the best migration parameters to minimize the cost of parity updates while requiring minimal data migration.

RS6 has several unique features as follows.

- RS6 maintains a uniform data distribution across old data disks and new data disks after RAID-6 scaling.
- RS6 minimizes the amount of data to be migrated during a scaling operation.
- RS6 minimizes the cost of parity updates while providing the above two features.
- RS6 computes the location of a block easily without any lookup operation.
- An array scaled using the RS6 approach can be reconstructed after the loss of any two of its disks.

We conducted extensive simulation experiments to quantitatively characterize the properties of RS6. The results show that, compared with existing "moving-everything" approaches, RS6 reduces the number of blocks to be moved by 60.0%–88.9%, and saves the migration time by 40.27%–69.88%.

1.4 Paper Organization

The rest of the paper is organized as follows. We review related work in Section 2. In Section 3, we present how RS6 accelerates RAID-6 scaling with three techniques. In Section 4, we develop our addressing algorithm. We examine the properties of RS6 in Section 5, and compare the performance of RS6 with that of existing solutions in Section 6. Finally, we conclude this paper in Section 7.

2 Related Work

Efforts concentrating on RAID scaling approaches are divided into two categories, i.e., optimizing the process of data migration and reducing the number of data blocks to be moved.

2.1 Optimizing Data Migration for RAID Scaling

The conventional approaches to RAID scaling preserve the round-robin order after adding disks. All data blocks are migrated in the scaling process. Brown [24] designed a reshape toolkit in the Linux kernel (MD-Reshape), which writes mapping metadata with a fixed-size data window. User requests to the window have to queue up until all data blocks within the window are moved. Therefore, the window size cannot be too large. Metadata updates are quite frequent. Gonzalez and Cortes [22] proposed a gradual assimilation approach (GA) to control the speed of RAID-5 scaling.

The MDM method [23] eliminates the parity modification cost of RAID-5 scaling by exchanging some data blocks between original disks and new disks. However, it does not guarantee an even data and parity distribution. Also, it does not increase (just maintains) the data storage efficiency after adding more disks.

A patent [29] presents a method to eliminate the need to rewrite the original data blocks and parity blocks on original disks. However, the obvious uneven distribution of parity blocks will bring a penalty to write performance.

Franklin et al. [30] proposed to use spare disks to provide immediate access to new space. During data redistribution, new data are mapped to spare disks. Upon completion of the redistribution, new data are copied to the set of data disk drives. Similar to WorkOut [31], this kind of method requires spare disks to be available.

Zhang et al. [25], [26] discovered that there is always a reordering window during data redistribution for round-robin RAID scaling. By leveraging this insight, they proposed the ALV approach to improving the efficiency of RAID-5 scaling. However, ALV still suffers from large data migration.

2.2 Reducing Data Migration for RAID Scaling

With the development of object-based storage, randomized RAID [21], [32]–[34] reduce data migration while delivering a uniform load distribution. The cut-and-paste placement strategy [34] uses randomized allocation strategy to place data across disks. Seo and Zimmermann [35] proposed an approach to finding a sequence of disk additions and removals for the disk replacement problem. The goal is to minimize the data migration cost. The SCADDAR algorithm [21] uses a pseudo-random function to minimize the amount of data to be moved. RUSH [36], [37] and CRUSH [38] are two algorithms for online placement and reorganization of replicated data. The Random Slicing strategy [39] keeps a small table with information about previous insert and remove operations to reduce the required amount of randomness. These randomized strategies are designed for object-based storage systems. They only provide mapping from logical addresses to a set of storage devices, while the data placement on a storage device is resolved by additional software running on the device itself.

The HP AutoRAID [40] allows an online capacity expansion without requiring data migration. Newly created RAID-5

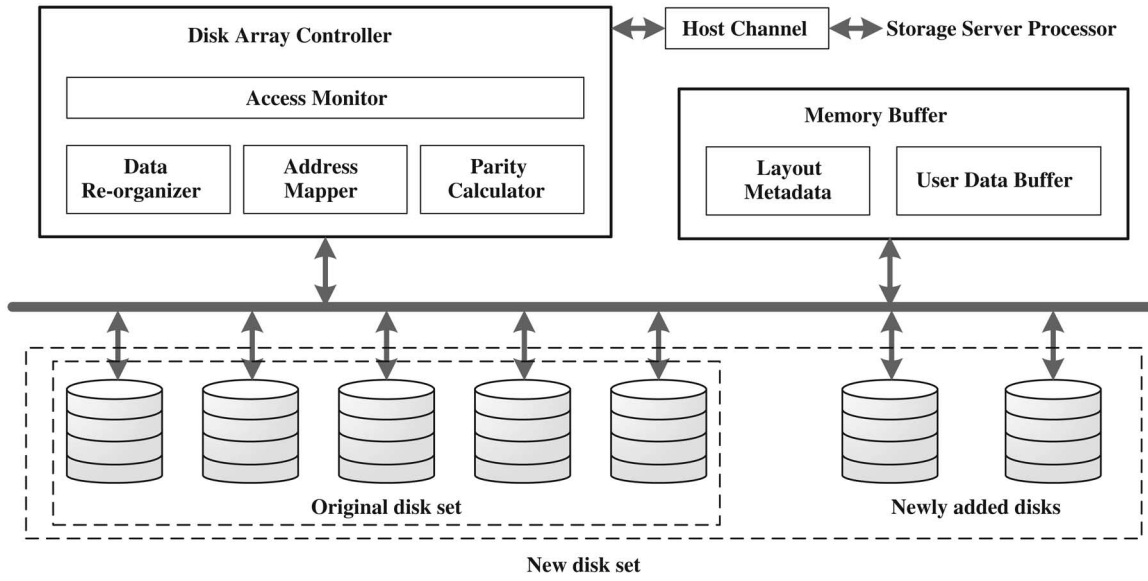

Fig. 1. The disk array architecture. The new disk set has larger I/O parallelism than the original disk set.

volumes use all the disks in the system, but previously created RAID-5 volumes continue to use only the original disks.

To reduce data migration, the semi-RR algorithm [21] modifies the round-robin scheme, and requires a block movement only if the target disk number is one of new disks. Semi-RR reduces data migration significantly. Unfortunately, it does not guarantee uniform distribution of data blocks after subsequent scaling operations. This will deteriorate the initial equally distributed load.

The GSR approach [41] to RAID-5 scaling divides data on the original array into two consecutive sections. It moves the second section of data onto the new disks, while keeping the first section of data unmoved. In this way, GSR minimizes data migration and parity modification. The main limitation of GSR is the performance of RAID systems after scaling. Especially, there is a large performance penalty when the workload exhibits a strong locality in its access pattern.

The SDM scheme [42] increases the size of the row parity set, while keeping the size of the diagonal parity set unchanged. It provides uniform data distribution and minimal data migration. Unfortunately, SDM can be used in RAID-6 scaling for one time, rather than for multiple times. Furthermore, a RAID-6 array scaled using SDM is unable to be reconstructed after the loss of any two disks of the array.

Zheng and Zhang [27] proposed the FastScale approach to RAID-0 scaling. FastScale minimizes data migration while maintaining a uniform data distribution. FastScale provides a good starting point for RAID-6 scaling. However, RAID-6 scaling is more challenging, as discussed in Section 1.2.

3 How RS6 Performs RAID Scaling

RS6 accelerates RAID-6 scaling with three techniques, i.e., minimizing data migration, piggyback parity updates, and selecting migration parameters. The first technique improves the migration efficiency of regular data. The three techniques reduce the overhead of parity updates during RAID scaling together. To better understand how RS6 works, we first give an overview of the data migration process.

3.1 System Architecture and Procedure Overview

Fig. 1 demonstrates the architecture of a typical disk array. A disk array is made up of a disk array controller, some hard disks, and a memory buffer. The memory buffer caches data and layout metadata for higher performance. The array controller consists of four modules: an address mapper, a parity calculator, a data re-organizer, and an access monitor. The address mapper, according to the layout metadata, forwards incoming I/O requests to the corresponding disks. The parity calculator performs XOR calculations when writing data, recovering data, or migrating data. The data re-organizer moves the data on the array. The access monitor keeps track of the popularity of data on the hard disk array. A disk array is connected to a storage server over a host channel.

Before scaling, only the disks in the original disk set serve I/O requests from the storage server. When the storage capacity is almost exhausted or the I/O bandwidth is unable to meet the performance requirement, new disks will be added to the RAID-6 array. First of all, the data re-organizer selects the best combination of two migration parameters to minimize the cost of parity updates. One of the two parameters determines two disks between which new disks are inserted. The other parameter determines which data will be moved. See Section 3.4 for more details.

Then, data redistribution is performed by the following four steps. First, the data re-organizer reads data from the original disks into user data buffer. Second, the parity calculator generates a new parity block according to the data in the data buffer. Third, the data re-organizer writes data and the new parity into new locations simultaneously, and writes zero to original locations of moved data. Fourth, the data re-organizer modifies the layout metadata in the memory buffer. The layout metadata will be synchronized onto disks when necessary.

Once an I/O request from a storage server arrives, the address mapper forwards this request to the corresponding disk according to the layout metadata. In this way, RAID-6 scaling can be performed without stopping foreground applications.

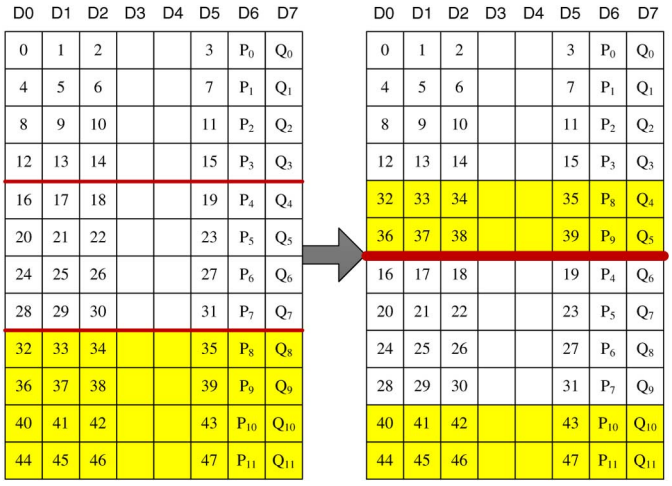

Fig. 2. RS6's normalizing operation during RAID-6 scaling from 6 disks to 8. Here, $N_{i-1} = 4$, $N_i = 6$, so we have $L = 12$, which is the least common multiple of 4 and 6. One region consists of 12 rows.

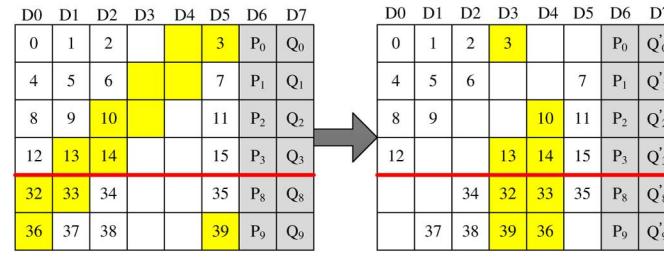

Fig. 3. RS6's data migration during RAID-6 scaling from 6 disks to 8.

As more and more data blocks are moved to new disks, the newly added disks are gradually available to serve user requests. When data redistribution is completed, a balanced load distribution among all the disks is obtained.

3.2 Minimizing Data Migration

During RAID-6 scaling, RS6 moves a fraction of existing data blocks from original disks to new disks. The goal is that data migration is minimal while data distribution across all the disks is uniform. RAID-6 scaling is over when data migration is finished. After that, newly added capacity is available, and new data can be filled into the RAID-6 gradually. This subsection focuses on an overview of how RS6 minimizes data migration for RAID-6 scaling. For more details on data migration and data filling, see the description in Sections 4.2 and 4.3.

To understand how the RS6 approach works, and how it minimizes data migration while maintaining uniform data distribution across all the disks, we take the ith RAID scaling operation from $N_{i-1}+2$ disks to N_i+2 as an example. We suppose each disk consists of s data blocks. Before this scaling operation, there are $N_{i-1} \times s$ data blocks stored on N_{i-1} data disks. The blocks on the two parity disks will keep unmoved.

Suppose that newly added disks are inserted after Disk 2 according to the result of selecting the best migration parameters. We let L denote the least common multiple of N_{i-1} and N_i. Each L consecutive locations in a data disk are grouped into a *segment*. For the N_i data disks, N_i segments with the same physical address are grouped into one *region*. Locations on all disks with the same block number form a *row* or a *row stripe*. For different regions, the ways for data migration and data filling are completely identical. Therefore, we will focus on one region.

In one region, as shown in Fig. 2, there are L/N_{i-1} small stripe sets before scaling, while there are L/N_i large stripe sets after scaling. Before data migration, RS6 introduces a normalizing operation to shuffle rows, which helps to reduce the cost of parity updates. The first L/N_i small stripe sets keep unmoved. The remaining rows are divided into L/N_i shares evenly, which are moved behind small stripe sets in turn. Take Fig. 2 as an example. Rows 8 and 9 are put behind the first small stripe set. Rows 10 and 11 are put behind the second small stripe set. Thus, the first large stripe set includes Rows 0, 1, 2, 3, 8, and 9. It should be noted that the normalization of the logical view does not result in any data migration.

After being normalized, different large stripe sets have the same way for data migration and data filling. Therefore, we will focus on one large stripe set. In a large stripe set, as shown in Fig. 3, all data blocks within a parallelogram will be moved. Such a parallelogram is termed a *moving parallelogram*. The base of a moving parallelogram is $N_i - N_{i-1}$, and the height is N_i. To restrict the moving parallelogram within the N_i data disks, the disk number of each data block within the moving parallelogram is modulared by N_i. Fig. 3 depicts the moving trace of each migrating block. For one moving data block, only its disk number is changed while its physical block number is unchanged. The source locations of moving data are the intersection of the moving parallelogram and original disks. The target locations of moving data are on new disks and outside the moving parallelogram.

After data migration, in a large stripe set, only the moving parallelogram is empty. The intersection of the moving parallelogram and each disk has $N_i - N_{i-1}$ data blocks. Therefore, each data disk, either old or new, has N_{i-1} data blocks. That is to say, RS6 regains a uniform data distribution. In a large stripe set, the total number of data blocks to be moved is the area of the intersection of the moving parallelogram and original disks, i.e., $(N_i - N_{i-1}) \times N_{i-1}$. This reaches the minimal number of moved blocks in each large stripe set, i.e., $(N_{i-1} \times N_i) \times (N_i - N_{i-1})/N_i = N_{i-1} \times (N_i - N_{i-1})$. We can claim that the RAID scaling using RS6 minimizes data migration while maintaining uniform data distribution across all data disks.

After data migration, RS6 maintains consistent row and diagonal parities for data in a large stripe set. Hence, an array scaled using RS6 is able to tolerate two simultaneous disk failures. After RAID-6 scaling, new data can be filled gradually. $N_i - N_{i-1}$ new data blocks are placed into each row consecutively. These new blocks are distributed in a round-robin order. More details about data filling are given in Section 4.3.

3.3 Piggyback and Aggregated Parity Updates

RDP RAID-6 arrays can protect against double disk failures by maintaining dual parity information. RS6 moves a block by cutting it and pasting it into another location within the same row stripe. This migrating operation does not change the total content of a row stripe, and therefore does not require updating a row parity.

RS6 changes the total contents of a diagonal stripe, and therefore requires updating a diagonal parity. The moving scheme of RS6 reduces the cost of diagonal parity updates significantly. Take Fig. 3 as an example. After RAID scaling, we have $Q'_1 = P_2 \oplus B_{15} \oplus B_{33} \oplus B_{39} \oplus B_1 \oplus B_4$. Here, B_i denotes data block i, which is noted by i in the figure. Before RAID scaling, we have $Q_1 = P_2 \oplus B_{15} \oplus B_1 \oplus B_4$. So, we have $Q'_1 = Q_1 \oplus B_{33} \oplus B_{39}$. In this manner, to calculate Q'_1 that is the XOR sum of six blocks, only two XOR operations are required.

RS6 also piggybacks parity updates during data migration to reduce the number of additional disk I/Os. Let us continue the above example. Since B_{33} and B_{39} need to be moved, they will be in the memory buffer and available for computing parity. As a result, to maintain a consistent diagonal parity, Q_1, only a parity read and a parity write are added. In addition, all these three reads from Q_1, B_{33}, and B_{39}, are on three disks, and therefore can be done in parallel. Similarly, the three writes to Q'_1, B_{33}, and B_{39}, can also be done in parallel. This parallelism will further reduce the cost of parity updates.

Since disk I/O performs much better with large sequential access, RS6 writes multiple successive diagonal parity blocks via a single I/O if possible. For example, as shown in Fig. 3, only two I/Os, instead of six I/Os, are required to write all diagonal parity blocks in a large stripe set. The first I/O is responsible for writing diagonal parity blocks Q'_0, Q'_1, Q'_2, and Q'_3. The second I/O is responsible for writing diagonal parity blocks Q'_8, and Q'_9. It is worthwhile to mention that if they are only logically in sequential order, the benefits of writing all of them in one operation is less than if the blocks are actually physically adjacent.

3.4 Selecting Migration Parameters

There are two factors in the RS6 moving scheme that affect the I/O cost of diagonal parity updates. One is to decide two disks between which new disks are inserted. The other is to decide which data are to be moved. Before one RAID-6 scaling operation, RS6 selects the best migration parameters to minimize the number of disk I/Os for updating diagonal parities.

For different large stripe sets, the ways for updating diagonal parities are completely identical. Therefore, we will focus on one large stripe set. Without loss of generality, we assume that this scaling operation is from m data disks to $m+n$ data disks. We let idx denote the location of the first new disk in a large stripe set. Let $bias$ denote the distance between the upper-left vertex of the moving parallelogram and the upper-right vertex of all the data blocks in the large stripe set. Take Fig. 3 as an example, we have $idx = 3$ and $bias = 1$.

As shown in Fig. 4, RS6 traverses all possible values of idx and $bias$. For each combination of them, it calculates the number of data reads for updating diagonal parities using the *getCost* function. Then, RS6 selects the best combination of idx and $bias$ with the minimal cost.

For given idx and $bias$, the *getCost* function is used to calculate the number of data reads for updating diagonal parities in a large parity set. There are usually two alternative methods to compute the new diagonal parity, namely, reconstruction-write and read-modify-write. The main difference between the two methods lies in the data blocks that must be pre-read for the computation of the new parity blocks [43].

As shown in Fig. 5, S_i denotes the set of those blocks protected by diagonal parity Q_i before scaling (line 2). S'_i denotes the set of those blocks protected by diagonal parity Q'_i after scaling (line 5). M denotes the set of data blocks moved in this scaling operation (line 3). With piggyback parity updates,

Algorithm: FindBestParam($m, n, idx, bias$).

Input: The input parameters are m and n, where
m: the number of old disks;
n: the number of new disks.
Output: The output parameters are idx and $bias$, where
idx: the position for inserting new disks;
$bias$: the offset of the first chosen diagonal.

```
minCost ← ∞;                                   (1)
for (i ∈ [0, m])                               (2)
    for (b ∈ [0, m + n − 1])                   (3)
        c ← getCost(m, n, i, b);               (4)
        if (c < minCost) then                  (5)
            minCost ← c;                       (6)
            idx ← i;                           (7)
            bias ← b.                          (8)
```

Fig. 4. The FindBestParam algorithm used in RS6.

Algorithm: getCost($m, n, idx, bias$).

Input: The input parameters are m, n, idx, and $bias$, where
m: the number of old disks;
n: the number of new disks;
idx: the position for inserting new disks;
$bias$: the offset of the first chosen diagonal.
Output: The number of blocks to be read for calculating diagonal parities.

```
for (i ∈ [0, m − 1])                                              (1)
    S_i ← the set of blocks protected by diagonal parity Q_i
before scaling;                                                   (2)
M ← the set of data blocks moved in this scaling operation;       (3)
for (i ∈ [0, m + n − 1])                                          (4)
    S'_i ← the set of blocks protected by diagonal parity Q_i
after scaling;                                                    (5)
for (i ∈ [0, m − 1])                                              (6)
    A ← S_i − M;                                                  (7)
    B ← S'_i − M;                                                 (8)
    T ← A ∩ B;                                                    (9)
    if (1 + |A| < 2|T|) then    //read-modify-write              (10)
        C_i ← {Q_i} ∪ (A − T) ∪ (B − T);                          (11)
    else    //reconstruction-write                              (12)
        C_i ← B;                                                  (13)
for (i ∈ [m, m + n − 1])                                          (14)
    C_i ← S'_i − M ;                                              (15)
return | ∪_{i=0}^{m+n−1} C_i|.                                   (16)
```

Fig. 5. The getCost function used in the FindBestParam algorithm.

Algorithm: Addressing(t, H, s, x, d, b).

Input: The input parameters are t, H, s, and x, where
t: the number of scaling times;
H: the scaling history ($H[0], H[1], ..., H[t]$);
s: the number of data blocks in one disk;
x: a logical block number.
Output: The output parameters are d and b, where
d: the disk holding block x;
b: the physical block number on disk d.

$b \leftarrow$ x2b(t, H, s, x); (1)
Normalizing(t, H, b, T); (2)
$d \leftarrow$ Mapping(t, H, s, x, T). (3)

Fig. 6. The Addressing algorithm used in RS6.

the reconstruction-write method reads blocks in the set B. T denotes the intersection of A and B. The read-modify-write method reads parity block Q_i and blocks in the set $(A - T) \cup (B - T)$. Therefore, the reconstruction-write method reads $|B|$ blocks, while the read-modify-write method reads $1 + |A| + |B| - 2|T|$ blocks.

RS6 chooses one of the two methods for the first m diagonal stripe to minimize the number of pre-read operations. If we have $1 + |A| + |B| - 2|T| < |B|$ (i.e., $1 + |A| < 2|T|$), the read-modify-write method is used (line 11). Otherwise, the reconstruction-write method is used (line 13). For the last n diagonal stripe, only the reconstruction-write method is used (line 15). Finally, the number of disk I/Os for updating all diagonal parities in a large stripe set is returned.

Because the best parameters will not change dynamically, the *FindBestParam* function is executed only once at the beginning of a scaling operation. The result is saved and can be used directly in the process of RAID scaling. Therefore, RS6 selects the best migration parameters without bringing any additional overhead into RAID scaling.

4 How RS6 Performs Data Addressing

In this section, we describe how RS6 maps a logical address to the corresponding physical address in a RAID-6 system.

4.1 The Addressing Algorithm

Fig. 6 shows the *Addressing* algorithm to minimize data migration required by RAID scaling. A two-dimensional array H is used to record the history of RAID scaling. $H[0][0]$ is the initial number of data disks in the RAID. After the ith scaling operation, the RAID consists of $H[i][0]$ data disks. $H[i][1]$ and $H[i][2]$ are the corresponding migration parameters, idx and $bias$, for the ith scaling operation.

First, the *Addressing* algorithm uses the *x2b* function (shown in Fig. 7) to get the physical block number for logical block x. This physical block number will not change after any number of scaling operations, because RS6 only moves data blocks within the same row stripe. Second, it calculates t logical row numbers according to the physical block number and saves them in an array, T, with the *Normalizing* function (shown in Fig. 8). $T[i]$ is the row number in a logical view

Algorithm: x2b(t, H, s, x).

Input: The input parameters are t, H, s, and x, where
t: the number of scaling times;
H: the scaling history ($H[0], H[1], ..., H[t]$);
s: the number of data blocks in one disk;
x: a logical block number.
Output: The physical block number of block x.

if ($x < H[0][0] \times s$) **then** (1)
  return $x/H[0][0]$; (2)
for($i = 1$; $i \leq t$; $i++$) (3)
  if ($x < H[i][0] \times s$) **then** (4)
    break; (5)
$m \leftarrow H[i-1][0]$; (6)
$n \leftarrow H[i][0] - m$; (7)
return $(x - m \times s)/n$. (8)

Fig. 7. The x2b function used in the Addressing algorithm.

during the ith scaling. Finally, it uses the *Mapping* function to calculate the disk number for a logical block number, x.

The *x2b* function. After the ith scaling operation, $(H[i][0] - H[i-1][0]) \times s$ locations are available for new data blocks. First, the *x2b* function determines the number of scaling times after which the data block, x, is added. Then, it gets the physical block number easily.

The *Normalizing* function. Before data migration, RS6 uses the *Normalizing* function to construct a logical view by shuffling rows, which helps to reduce the cost of diagonal parity updates. It should be noted that the normalization of the logical view does not result in any data migration. When a

Algorithm: Normalizing(t, H, b, T).

Input: The input parameters are t, H, and b, where
t: the number of scaling times;
H: the scaling history;
b: the physical block number;
Output: The output parameter is T, where
T: the transforming history of the physical block.

if ($t = 0$) **then** (1)
  $T[t] \leftarrow b$; (2)
else (3)
  $m \leftarrow H[t-1][0]$; //the number of old disks (4)
  $n \leftarrow H[t][0] - m$; //the number of new disks (5)
  Normalizing($t - 1, H, b, T$) ; (6)
  $L \leftarrow$ lcm($m, m + n$); (7)
  $g \leftarrow T[t-1]/L$; (8)
  $b_1 \leftarrow T[t-1] \bmod L$; (9)
  $g_0 \leftarrow b_1/m$; (10)
  if ($g_0 < L/(m+n)$) **then** (11)
    $T[t] \leftarrow g \times L + (m+n) \times g_0 + b_1 \bmod m$; (12)
  else (13)
    $b_2 \leftarrow b_1 - m \times (L/(m+n))$; (14)
    $g_1 \leftarrow b_2/n$; (15)
    $T[t] \leftarrow g \times L + (m+n) \times g_1 + m + b_2 \bmod n$. (16)

Fig. 8. The normalizing procedure used in the Addressing algorithm.

RAID is constructed from scratch (i.e., $t = 0$), the logical row number is just the physical block number (lines 1–2). Let us examine the tth ($t > 0$) scaling, where n disks are added into a RAID made up of m disks (lines 4–5).

RS6 calculates its logical row number, $T[t-1]$, for the $(t-1)$th scaling (line 6). We let L denote the least common multiple of m and $m+n$ (line 7). Each L consecutive rows are grouped into a *region*. For different regions, the ways for row shuffling are completely identical. There are $L/(m+n)$ large stripe sets after scaling. There are L/m small stripe sets before scaling. The first $L/(m+n)$ small stripe sets keep unmoved. The left rows are divided into $L/(m+n)$ shares evenly, which are moved behind small stripe sets in turn.

In the logical view for the tth scaling, physical row b is logical row b_1 (line 9) in region g (line 8). It is in small stripe set g_0 before this scaling (line 10). If g_0 is one of the first $L/(m+n)$ small stripe sets (line 11), there are g regions and g_0 large stripe sets in front of logical row b_1 (line 12). Otherwise, row b_1 is in share g_1 (line 15). There are g regions, g_1 large stripe sets, and one small stripe set in front of logical row b_1 (line 16).

The *Mapping* function. Finally, let us see the *Mapping* function. When a RAID is constructed from scratch (i.e., $t = 0$), it is actually a round-robin RDP RAID-6 array. The disk number of block x can be calculated via one modular operation (line 3).

Let us examine the tth ($t > 0$) scaling, where n disks are added into a RAID made up of m disks (lines 4–5).

Case 1. If block x is an original block (line 8), RS6 calculates its original disk number, d_0, before the tth scaling (line 9). If we have $d_0 \geq idx$, RS6 adds d_0 by n due to the addition of n new disks (line 11).

- If data block x needs to be moved (line 19), RS6 changes the disk ordinal number via the *Moving* function (line 20).
- If data block x does not need to be moved (line 21), RS6 keeps the disk number unchanged (line 22).

Case 2. If block x is a new block (line 23), RS6 determines which disk places it on via the *Placing* function (line 24).

Algorithm: Mapping(t, H, s, x, T).

Input: The input parameters are t, H, s, x, and T, where
t: the number of scaling times;
H: the scaling history $(H[0], H[1], ..., H[t])$;
s: the number of data blocks in one disk;
x: a logical block number;
T: the transforming history of the physical block.

Output: The disk holding block x.

```
if (t = 0) then                                                    (1)
    m ← H[0][0]; //the number of initial disks                     (2)
    return x mod m;                                                (3)
m ← H[t − 1][0]; //the number of old disks                         (4)
n ← H[t][0] − m; //the number of new disks                         (5)
idx ← H[t][1]; //index                                             (6)
bias ← H[t][2]; //bias                                             (7)
if (x ∈ [0, m × s − 1]) then //an old data block                   (8)
    d0 ← Mapping(t − 1, H, s, x, T);                               (9)
    if (d0 ≥ idx) then                                             (10)
        d0 ← d0 + n;                                               (11)
    b1 ← T[t] mod (m + n);                                         (12)
    vl ← (m + n − 1 − bias − b1) mod (m + n);                      (13)
    vr ← (vl + n − 1) mod (m + n);                                 (14)
    if (vl < vr) then                                              (15)
        Sm ← [vl, vr];                                             (16)
    else                                                           (17)
        Sm ← [0, vr] ∪ [vl, m + n − 1];                            (18)
    if (d0 ∈ Sm) then //to be moved                                (19)
        return Moving(m, n, idx, bias, d0, b1);                    (20)
    else //not to be moved                                        (21)
        return d0;                                                 (22)
else //a new data block                                            (23)
    return Placing(m, n, s, bias, x, t, T);                        (24)
end if.                                                            (25)
```

Fig. 9. The Mapping function used in the Addressing algorithm.

4.2 Moving Old Data

The code of lines 12–19 in Fig. 9 is used to decide whether data block x will be moved during a RAID scaling. As shown in Fig. 3, there is a moving parallelogram in each large stripe set. The base of the parallelogram is n, and the height is $m+n$. If and only if a data block is within the moving parallelogram, it will be moved. In the logical view after being normalized, the row number of block x in a large stripe set is b_1 (line 12). One parallelogram mapped to row b_1 is a line segment, whose two end points are v_l and v_r (lines 13–14). If d_0 is within the line segment, block x is within the moving parallelogram, and therefore it will be moved (line 19).

Once a data block is determined to be moved, RS6 changes its disk number with the *Moving* function given in Fig. 10. As shown in Fig. 11, the intersection of a moving parallelogram and original disks is divided into four parts: the upper triangle, the lower triangle, the upper parallelogram, and the lower parallelogram. How a data block moves depends on which part it lies in. No matter which is bigger between m and n, the upper triangle and the lower triangle keep their shapes unchanged. The upper triangle will be moved to the left by n disks (lines 9, 21), while the lower triangle will be moved to the right by n disks (lines 11, 23). However, the two parallelograms are sensitive to the relationship between m and n. They are twisted from parallelograms to rectangles when $m \geq n$ (line 12), and from rectangles to parallelograms when $m < n$ (line 24). RS6 keeps the relative locations of all data blocks in the same row. After data migration, the whole moving parallelogram becomes empty, while any location in the other part has a data block.

4.3 Placing New Data

When block x is at a location newly added after the last scaling, it is addressed via the *Placing* function given in Fig. 12. If block x is a new block, it is the yth new block (line 1). In the logical view after being normalized, the row number of block x is b_1 (line 2). In row stripe b_1, the first new block is on disk v_l (line 3). Let us go backward to line 13 in Fig. 9. v_l is one of two end points of a line segment that a moving parallelogram is mapped to row b_1. Block x is the rth new data block in stripe b_1 (line 4). Therefore, the disk number of block x is $(v_l + r) \bmod (m+n)$ (line 5). The order of placing new blocks is shown in Fig. 13.

Algorithm: Moving($m, n, idx, bias, d_0, b$).

Input: The input parameters are m, n, idx, $bias$, d_0, and b, where

m: the number of old disks;
n: the number of new disks;
idx: the position for inserting new disks;
$bias$: the offset of the first chosen diagonal;
d_0: the disk number before a block is moved;
b: the logical value of physical block number.

Output: The disk number after a block is moved.

```
if (m ≥ n) then                                                    (1)
    y ← (m + n − 1 − idx − bias) mod (m + n);                      (2)
    y0 ← (y − (n − 1)) mod (m + n);                                (3)
    if (y0 + 2 × (n − 1) ≥ m + n) then                             (4)
        y ← m;                                                     (5)
        δ ← y0 + 2 × (n − 1) − (m + n − 1);                        (6)
        b ← (b − δ) mod (m + n);                                   (7)
    if (b ∈ [y − (n − 1), y]) then                                 (8)
        return (d0 − n) mod (m + n);                               (9)
    if (b ∈ (y, y + (n − 1)]) then                                 (10)
        return (d0 + n) mod (m + n);                               (11)
    return (d0 + b − y) mod (m + n);                               (12)
else                                                               (13)
    y ← (m + n − 1 − idx − bias) mod (m + n);                      (14)
    y0 ← (y − (m − 1)) mod (m + n);                                (15)
    if (y0 + 2 × (m − 1) ≥ m + n) then                             (16)
        y ← n;                                                     (17)
        δ ← y0 + 2 × (m − 1) − (m + n − 1);                        (18)
        b ← (b − δ) mod (m + n);                                   (19)
    if (b ∈ [y − (m − 1), y]) then                                 (20)
        return (d0 − n) mod (m + n);                               (21)
    if (b ∈ (y, y + (m − 1)]) then                                 (22)
        return (d0 + n) mod (m + n);                               (23)
    return (d0 + y − b) mod (m + n);                               (24)
end if.                                                            (25)
```

Fig. 10. The Moving function used in the Mapping function.

4.4 Remarks about the Addressing Algorithm

The addressing algorithm of RS6 is very simple and elegant. For instance, the *Addressing* algorithm requires fewer than 50 lines of C code. This reduces the likelihood that a bug will cause a data block to be mapped to a wrong location.

When a disk array boots, the RAID topology needs to be obtained from disks. The RS6 algorithm depends on how many disks are added during each scaling operation. If a RAID-6 array is scaled for t times, RS6 needs to store $t+1$ integers ($H[i][0]$, $0 \le i \le t$) on stable storage. $H[i][1]$ and $H[i][2]$ ($0 \le i \le t$) can be calculated when the array is loaded.

5 Theoretical Proofs of Properties

For a RAID-6 scaling operation, it is desirable to ensure an even load distribution on all data disks and minimal block movement. Furthermore, since the location of a block may be changed during a scaling operation, another objective is to quickly compute the current location of a block. In this section, we formally prove that RS6 satisfies all the three requirements.

Theorem 5.1. *RS6 maintains a uniform data distribution after each RAID scaling.*

Proof Sketch. Assume that there are N_{i-1} old data disks and $N_i - N_{i-1}$ new data disks during a RAID scaling. In the logical view after being normalized, the whole data space is divided into multiple regions with the same size. For different regions, the ways for data migration and data filling are completely identical. One region is divided into multiple large stripe sets with the size of $N_i \times N_i$ locations. After being normalized, different large stripe sets have the same way for data migration and data filling. Therefore, it suffices to show that RS6 maintains a uniform data distribution in each large stripe set after this RAID scaling.

Before this RAID scaling, there are N_i data blocks on each of the N_{i-1} old data disks. As shown in Fig. 11, the intersection of the moving parallelogram and each disk has $N_i - N_{i-1}$ data blocks. Therefore, each old data disk has $N_i - (N_i - N_{i-1}) = N_{i-1}$ data blocks after this scaling.

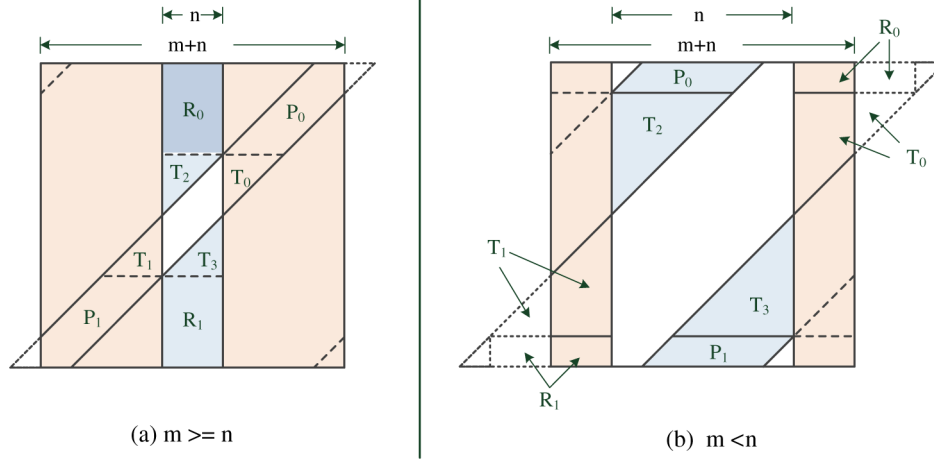

Fig. 11. The variation of data layout involved in migration. T_0 is moved onto the location of T_2. T_1 is moved onto the location of T_3. When $m \ge n$, P_0 is twisted to a rectangle and moved onto the location of R_0; P_1 is twisted to a rectangle and moved onto the location of R_1. When $m < n$, R_0 is twisted to a parallelogram and moved onto the location of P_0; R_1 is twisted to a parallelogram and moved onto the location of P_1.

Algorithm: Placing$(m, n, s, bias, x, t, T)$.

Input: The input parameters are m, n, s, $bias$, x, t and T, where

m: the number of old disks;
n: the number of new disks;
s: the number of data blocks in one disk;
$bias$: the offset of the first chosen diagonal;
x: a logical block number;
t: the number of scaling times;
T: the transforming history of the physical block.

Output: The disk holding block x.

$$y \leftarrow x - m \times s; \quad (1)$$
$$b_1 \leftarrow T[t]; \quad (2)$$
$$v_l \leftarrow (m + n - 1 - bias - b_1) \bmod (m + n); \quad (3)$$
$$r \leftarrow y \bmod n; \quad (4)$$
$$d \leftarrow (v_l + r) \bmod (m + n); \quad (5)$$
return d. (6)

Fig. 12. The Placing function used in the Mapping function.

According to the RS6 moving scheme, no data blocks are moved onto the moving parallelogram. So, each new disk holds at most $N_i - (N_i - N_{i-1}) = N_{i-1}$ data blocks after this scaling. All new disks can hold at most $N_{i-1} \times (N_i - N_{i-1})$ data blocks. Since each old disk contributes $N_i - N_{i-1}$ blocks to the new disks, $N_{i-1} \times (N_i - N_{i-1})$ data blocks are put onto new disks. Consequently, any location on new disks outsides the moving parallelogram holds a data block. Therefore, each new data disk has $N_i - (N_i - N_{i-1}) = N_{i-1}$ data blocks after this scaling. Each disk, either old or new, has N_{i-1} data blocks. That is to say, RS6 regains a uniform data distribution. □

Theorem 5.2. *RS6 performs the minimum number of data migration during each RAID scaling.*

Proof Sketch. Assume that there are N_{i-1} old data disks and $N_i - N_{i-1}$ new data disks during a RAID scaling. Again, it suffices to show that RS6 performs the minimum number of data migration in each large stripe set during this RAID scaling.

To maintain a uniform data distribution, the minimum number of blocks to be moved is $(N_{i-1} \times s) \times (N_i - N_{i-1})/N_i$, where each old data disk has s data blocks. For one large stripe set, each old disk has N_i data blocks. Therefore, the minimum number of blocks to be moved for one large stripe set is $(N_{i-1} \times N_i) \times (N_i - N_{i-1})/N_i = N_{i-1} \times (N_i - N_{i-1})$.

As shown in Fig. 11, the intersection of the moving parallelogram and each disk has $N_i - N_{i-1}$ data blocks. In other words, each old disk contributes $N_i - N_{i-1}$ blocks to the new disks. In total, $N_{i-1} \times (N_i - N_{i-1})$ data blocks are moved onto new disks, which is exactly the minimum number of blocks to be moved. □

Theorem 5.3. *The Addressing algorithm of RS6 has time complexity $O(t)$ after t RAID scalings.*

Proof Sketch. Let $T(t)$ denote the time complexity of the Addressing algorithm. Let $T_1(t)$, $T_2(t)$, and $T_3(t)$ denote the time complexities of the x2b, Normalizing, and Mapping functions, respectively. Then, we have $T(t) = T_1(t) + T_2(t) + T_3(t)$.

For the x2b function, we have $T_1(t) \leq c_1 + c_2 t$, for all t, where c_1 and c_2 are some two constants. Since Normalizing is a recursive algorithm, we can represent $T_2(t)$ by using a recurrence relation. First, it is clear that $T_2(0) = c_3$ for some constant c_3. Next, we have $T_2(t) = T_2(t-1) + c_4$, for all $t \geq 1$, where c_4 is some constant. Solving the above recurrence relation, we get $T_2(t) = c_3 + c_4 t$.

Since Mapping is a recursive algorithm, we can represent $T_3(t)$ by using a recurrence relation. First, it is clear that $T_3(0) = c_5$ for some constant c_5. Next, we notice that both the Moving function and the Placing function take constant time. Thus, we have $T_3(t) \leq T_3(t-1) + c_6$, for all $t \geq 1$, where c_6 is some constant. Solving the above recurrence relation, we get $T_3(t) \leq c_5 + c_6 t$.

In summary, we have $T(t) = T_1(t) + T_2(t) + T_3(t) \leq (c_1 + c_3 + c_5) + (c_2 + c_4 + c_6)t = O(t)$. □

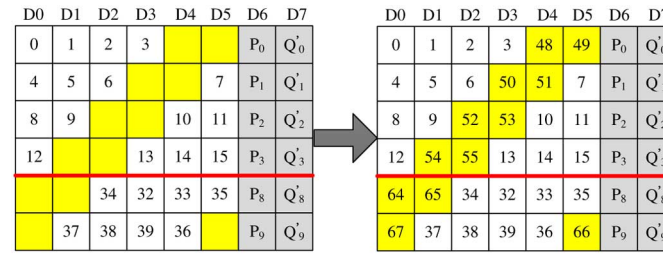

Fig. 13. The layout of new data blocks.

6 Experimental Evaluation

This section mainly presents results of a comprehensive experimental evaluation comparing RS6 with existing solutions. We first demonstrate the importance of RS6's selecting the best migration parameters. Then, we examine the properties of RS6—uniform data distribution, minimal data migration, and fast data addressing. Finally, we analyze their performance in the redistribution time, and explain the reason behind their difference.

6.1 Evaluation Methodology

To examine the importance of RS6's selecting the best migration parameters, we traverse all combinations of idx and $bias$ for a RAID-6 scaling operation, and calculate the corresponding numbers of data reads and XOR operations dedicated for parity updates. The larger the differences among the numbers of data reads (or XOR operations) with different parameters are, the more important the selection of migration parameters is.

To quantitatively characterize the properties of RS6, we compare RS6 with the round-robin algorithm [22], [24], [25] and the Semi-RR algorithm [21] via simulation experiments. ALV [26], MDM [23] and FastScale [27] cannot be used in RAID-6, so they are not compared. An RDP array is defined by a controlling parameter p, which must be a prime number greater than 2 [3]. From a 6-disk RDP RAID-6 array, we add two, four, two, four, two disks in turn using the three algorithms respectively. Each disk has a capacity of 128 GB, and

TABLE 1
The Numbers of Data Reads for Parity Updates with Different Parameters

bias \ idx	0	1	2	3	4
0	19	17	13	12	13
1	15	12	9	8	10
2	14	12	10	10	14
3	15	14	13	14	17
4	17	17	17	16	15
5	21	21	18	15	15

The maximum is 21 while the minimum is only 8.

the size of a data block is 64 KB. In other words, each disk holds 2×1024^2 blocks.

Finally, to compare their performance in the redistribution time, we calculate total numbers of disk I/Os and XOR operations during the above five scaling operations, and further get the corresponding redistribution time during an off-line scaling.

6.2 Best Migration Parameters

In this section, we examine the importance of RS6's selecting the best migration parameters. For this purpose, we inspect the RAID-6 scaling operation from four data disks to six data disks. We traverse all combinations of idx and $bias$, and calculate the corresponding numbers of data reads dedicated for parity updates. The numbers of data reads for parity updates with different parameters are compared in Table 1. We can see that the numbers of data reads in the 30 cases are significantly different. The maximum is 21, while the minimum is only 8.

We also traverse all combinations of idx and $bias$, and calculate the corresponding numbers of XOR operations dedicated for parity updates. The numbers of XOR operations for parity updates with different parameters are listed in Table 2. Similarly, the numbers of XOR operations in the 30 cases are significantly different. The maximum is 23, while the minimum is only 7.

We can safely claim that the numbers of data reads or XOR operations dedicated for parity updates change obviously with different parameters. RS6 selects the best migration parameters according to the number of disk I/Os for updating diagonal parities. For the above example, RS6 will select $(idx = 3, bias = 1)$ as the best migration parameters since 8 is the minimal number of data reads.

The number of XOR operations is not considered as a selecting metric for two reasons. First, there is a linear relationship between the number of data reads and the number of XOR operations, if the impact of piggyback parity updates is not taken into account. Second, the time for performing a XOR operation on two data blocks is usually a few microseconds, which is negligible compared to milliseconds of disk I/O time.

TABLE 2
The Numbers of Data XOR Operations for Parity Updates with Different Parameters

bias \ idx	0	1	2	3	4
0	23	22	18	14	15
1	18	14	7	7	11
2	16	9	9	11	14
3	15	14	14	16	17
4	18	19	19	17	15
5	23	23	23	20	17

The maximum is 23 while the minimum is only 7.

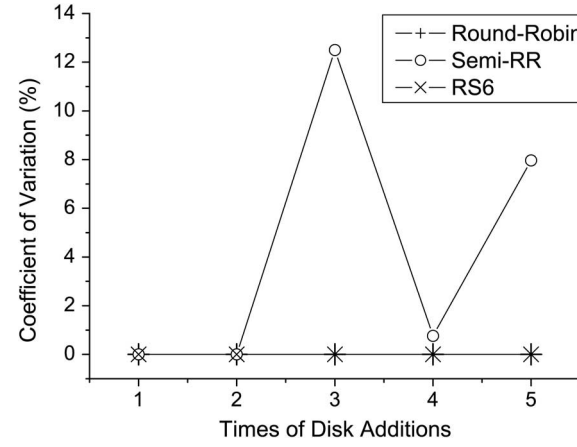

Fig. 14. Comparison of uniformity of data distribution.

6.3 Uniform Data Distribution

We use the coefficient of variation of the numbers of blocks on different disks as a metric to evaluate the uniformity of data distribution across all the data disks. The coefficient of variation expresses the standard deviation as a percentage of the average. The smaller the coefficient of variation is, the more uniform the data distribution is.

Fig. 14 plots the coefficient of variation versus the number of scaling operations. For the round-robin and RS6 algorithms, both the coefficients of variation remain 0 percent as the times of disk additions increases. Conversely, the semi-RR algorithm causes excessive oscillation in the coefficient of variation. The maximum is even 12.5 percent. This indicates that RS6 maintains a uniform data distribution after each RAID scaling, while Semi-RR fails to do so.

6.4 Minimal Data Migration

Fig. 15 plots the migration fraction (i.e., the fraction of data blocks to be migrated) versus the number of scaling operations. Using the round-robin algorithm, the migration fraction is constantly 100%. This will bring a very large migration cost.

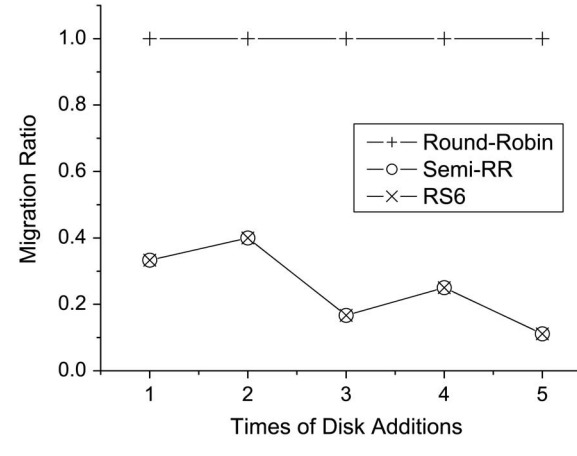

Fig. 15. Comparison of data migration ratio.

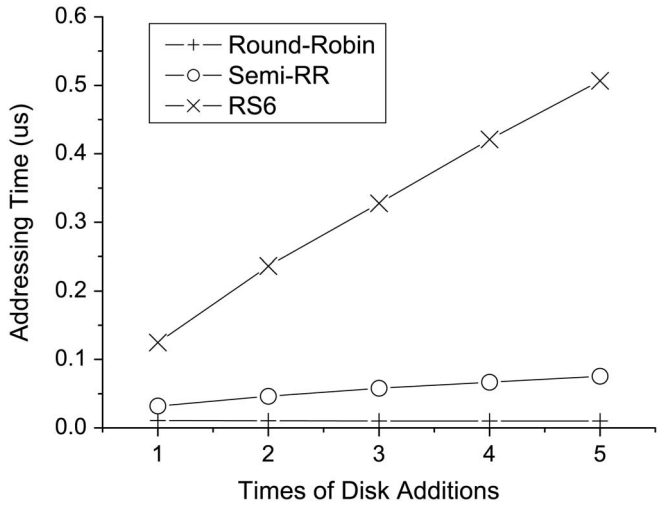

Fig. 16. Comparison of addressing time.

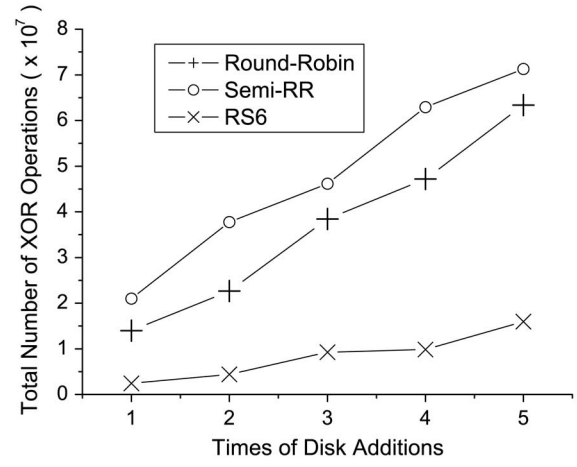

Fig. 18. Comparison of the number of XOR operations.

The migration fractions using the semi-RR algorithm and using RS6 are identical. They are significantly smaller than the migration fraction of using the round-robin algorithm. Assume that there are N_{i-1} old disks and $N_i - N_{i-1}$ new disks during a RAID scaling. To regain a uniform data distribution, the minimal number of blocks to be moved is $(N_{i-1} \times s) \times (N_i - N_{i-1})/N_i$, where each old data disk has s data blocks. Our numerical analysis indicates that the migration fractions using semi-RR and using RS6 reach this lower bound. That is to say, RS6 performs the minimum number of data migration during each RAID scaling. Compared with the round-robin algorithm, RS6 reduces the number of blocks to be moved by 60.0%–88.9%.

6.5 Fast Data Addressing

To quantitatively characterize the calculation overheads, we run different algorithms to calculate physical addresses for all data blocks on a scaled RAID. The whole addressing process is timed and then the average addressing time for each block is calculated. The testbed used in the experiment is an Intel Core i5-2300 2.80 GHz machine with 4 GB of memory. A Ubuntu 12.04 x64 is installed.

Fig. 16 plots the addressing time versus the number of scaling operations. The round-robin algorithm has a low calculation overhead of 0.01 μs or so. The calculation overheads using the semi-RR and RS6 algorithms both take on an upward trend. This is because the addressing algorithms of RS6 and Semi-RR have time complexity $O(t)$ after t RAID scalings. Among the three algorithms, RS6 has the largest overhead. Fortunately, the largest addressing time using RS6 is 0.51 μs, which is negligible compared to milliseconds of disk I/O time.

6.6 Total Scaling Time

To obtain total times of RAID-6 scaling operations using different approaches, we calculate total numbers of I/O operations and XOR operations. The total numbers of I/O operations during scaling using the three approaches are shown in Fig. 17. Compared with the round-robin approach, RS6 reduces the number of disk I/Os by 40.24%–69.88%.

We also calculate the total number of XOR operations under various cases as shown in Fig. 18. The round-robin and Semi-RR approaches have similar computation cost. RS6 decreases 74.82%–82.50% computation cost compared with other approaches.

The total scaling times in different cases are demonstrated in Fig. 19. Compared with other approaches, RS6 performs consistently better during each scaling, and decreases the scaling time by 40.27%–69.88%.

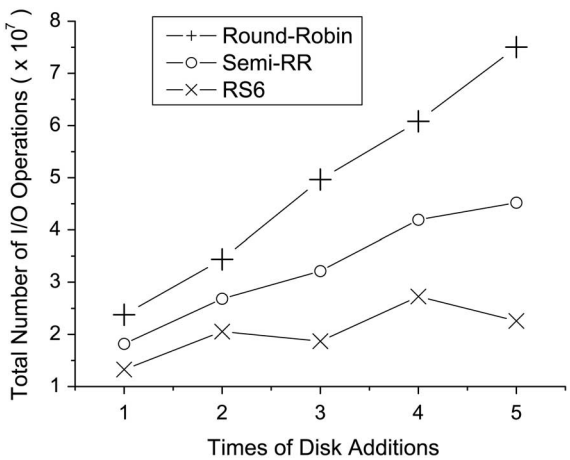

Fig. 17. Comparison of the number of disk I/Os.

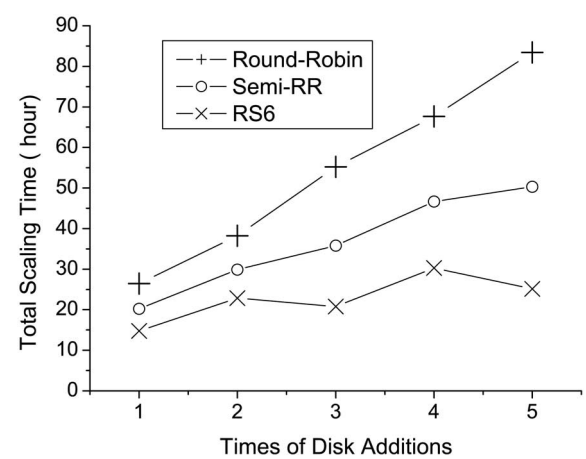

Fig. 19. Comparison of total scaling time.

7 Conclusions and Future Work

This paper presents RS6—a new approach to accelerating RDP RAID-6 scaling by reducing disk I/Os and XOR operations. First, RS6 minimizes the number of data blocks to be moved while maintaining a uniform data distribution across all data disks. Second, RS6 piggybacks parity updates during data migration to reduce the cost of maintaining consistent parities. Third, RS6 selects parameters of data migration so as to reduce disk I/Os for parity updates.

Some conclusions can be drawn from our simulation experiments and numerical analysis. First, the numbers of data reads or XOR operations dedicated for parity updates change obviously with different parameters. RS6 is able to choose the best migration parameters with the minimal number of disk I/Os for updating parities. Second, RS6 minimizes data migration, while maintaining a uniform data distribution. Third, although the time complexity of the RS6 addressing algorithm is $O(t)$ with t scaling operations, the addressing time that is less than 1 μs is acceptable. Finally, compared with existing "moving-everything" Round-Robin approach, RS6 can reduce the number of blocks to be moved by 60.0%–88.9%, and saves the migration time by 40.27%–69.88%.

RDP is a horizontal code, and RS6 does not handle RAID-6 scaling with vertical codes. We believe that RS6 provides a good starting point for efficient scaling of RAID-6 arrays based on vertical codes. In the future, we will focus on the problem of RAID-6 scaling with vertical codes.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 60903183, Grant 61170008, and Grant 61272055, in part by the National Grand Fundamental Research 973 Program of China under Grant 2014CB340402, and in part by the National High Technology Research and Development Program of China under Grant 2013AA01A210.

References

[1] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. Special Interest Group Manage. Data Conf. (SIGMOD)*, 1988, pp. 109–116.

[2] P. Chen et al., "RAID: High-performance, reliable secondary storage," *ACM Comput. Surveys*, vol. 26, no. 2, pp. 145–185, Jun. 1994.

[3] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. 3rd USENIX Conf. File Storage Technol. (FAST'04)*, Feb. 2004, pp. 1–14.

[4] E. Pinheiro, W. Weber, and L. Barroso, "Failure trends in a large disk drive population," in *Proc. 5th USENIX Conf. File Storage Technol. (FAST'07)*, Feb. 2007, pp. 17–29.

[5] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol. (FAST'07)*, Feb 2007, pp. 1–16.

[6] K. Hwang, H. Jin, and R. Ho, "RAID-x: A new distributed disk array for I/O-centric cluster computing," in *Proc. 9th Int. Symp. High-Perform. Distrib. Comput. (HPDC'00)*, Aug. 2000, pp. 279–286.

[7] Y. Saito et al., "FAB: Building distributed enterprise disk arrays from commodity components," in *Proc. Int. Conf. Archit. Support Program. Languages Oper. Syst. (ASPLOS'04)*, Oct. 2004, pp. 48–58.

[8] J. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. Smith, and E. Zayas, "Flexvol: flexible, efficient file volume virtualization in WAFL," in *Proc. USENIX Annu. Tech. Conf.*, Berkeley, CA, USA, 2008, pp. 129–142.

[9] X. Yu et al., "Trading capacity for performance in a disk array," in *Proc. 4th Conf. Symp. Oper. Syst. Design Implementation (OSDI'00)*, Oct. 2000, pp. 243–258.

[10] S. Ghandeharizadeh and D. Kim, "On-line reorganization of data in scalable continuous media servers," in *Proc. 7th Int. Conf. Database Expert Syst. Appl.*, Sep. 1996, pp. 755–768.

[11] D. A. Patterson, "A simple way to estimate the cost of down-time," in *Proc. 16th Large Installation Syst. Admin. Conf. (LISA'02)*, Oct. 2002, pp. 185–188.

[12] C. Jin et al., "P-code: A new RAID-6 code with optimal properties," in *Proc. 23rd Int. Conf. Supercomput. (ICS'09)*, 2009, pp. 360–369.

[13] M. Blaum et al., "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.

[14] M. Blaum and R. Roth, "On lowest density MDS codes," *IEEE Trans. Inf. Theory*, vol. 45, no. 1, pp. 46–59, Jan. 1999.

[15] J. Plank, "The RAID-6 liberation codes," in *Proc. File Storage Technol. (FAST'08)*, 2008, pp. 97–110.

[16] Y. Cassuto and J. Bruck, "Cyclic lowest density MDS array codes," *IEEE Trans. Inf. Theory*, vol. 55, no. 4, pp. 1721–1729, Apr. 2009.

[17] C. Wu et al., "H-code: A hybrid MDS array code to optimize partial stripe writes in RAID-6," in *Proc IEEE Int. Parallel Distrib. Process. Symp. (IPDPS'11)*, 2011, pp. 782–793.

[18] C. Wu et al., "HDP code: A horizontal-diagonal parity code to optimize I/O load balancing in RAID-6," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw. (DSN'11)*, 2011, pp. 209–220.

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

[20] L. Xu et al., "Low-density MDS codes and factors of complete graphs," *IEEE Trans. Inf. Theory*, vol. 45, no. 6, pp. 1817–1826, Sep. 1999.

[21] A. Goel, C. Shahabi, S.-Y. Yao, and R. Zimmermann, "SCADDAR: An efficient randomized technique to reorganize continuous media blocks," in *Proc. 18th Int. Conf. Data Eng. (ICDE'02)*, 2002, pp. 473–482.

[22] J. Gonzalez and T. Cortes, "Increasing the capacity of RAID5 by online gradual assimilation," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/Os*, Sep. 2004, pp. 17–24.

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

[24] N. Brown. (2010, Feb.) *Online RAID-5 Resizing. Drivers/md/raid5.c in the Source Code of Linux Kernel 2.6.32.9* [Online] Available: http://www.kernel.org/.

[25] 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, Mar. 2007, article 3.

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

[27] W. Zheng and G. Zhang, "FastScale: Accelerate RAID scaling by minimizing data migration," in *Proc. 9th USENIX Conf. File Storage Technol. (FAST'11)*, Feb. 2011, pp. 149–161.

[28] J. S. Plank, J. Luo, C. D. 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. (FAST'09)*, 2009, pp. 253–265.

[29] C. B. Legg, "Method of increasing the storage capacity of a level five RAID disk array by adding, in a single step, a new parity block and N–1 new data blocks which respectively reside in a new columns, where N is at least two document type and number," US Patent 6000010, Dec. 1999.

[30] C. R. Franklin and J. T. Wong, "Expansion of RAID subsystems using spare space with immediate access to new space," US Patent 10/033,997, 2006.

[31] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao, "WorkOut: I/O workload outsourcing for boosting the RAID reconstruction performance," in *Proc. 7th USENIX Conf. File Storage Technol. (FAST'09)*, Feb. 2009, pp. 239–252.

[32] J. Alemany and J. S. Thathachar, "Random striping news on demand servers," Univ. of Washington, Seattle, WA, USA, Tech. Rep. TR-97-02-02, 1997.

[33] J. R. Santos, R. R. Muntz, and B. A. Ribeiro-Neto, "Comparing random data allocation and data striping in multimedia servers," in *Proc. Meas. Modeling Comput. Syst.*, 2000, pp. 44–55.

[34] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Efficient, distributed data placement strategies for storage area networks (extended abstract)," in *Proc. ACM Symp. Parallel Algorithms Archit.*, 2000, pp. 119–128.

[35] B. Seo and R. Zimmermann, "Efficient disk replacement and data migration algorithms for large disk subsystems," *ACM Trans. Storage J.*, vol. 1, no. 3, pp. 316–345, Aug. 2005.
[36] R. J. Honicky and E. L. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Proc. 17th Int. Parallel Distrib. Process. Symp. (IPDPS'03)*, Apr. 2003, pp. 1–10.
[37] R. J. 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. (IPDPS'04)*, Apr. 2004. pp. 1–10.
[38] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. Int. Conf. Supercomput. (SC)*, Nov. 2006. pp. 1–12.
[39] 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. (HiPC'11)*, Dec. 2011.
[40] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP autoRAID hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, Feb. 1996.
[41] C. Wu and X. He, "GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling," in *Proc. 41st Int. Conf. Parallel Process. (ICPP'12)*, 2012, pp. 460–469.
[42] 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. (CLUSTER'12)*, 2012, pp. 284–292.
[43] C. Jin, D. Feng, H. Jiang, L. Tian, J. Liu, and X. Ge, "TRIP: Temporal redundancy integrated performance booster for parity-based RAID storage systems," in *Proc. IEEE 16th Int. Conf. Parallel Distrib. Syst. (ICPADS'10)*, 2010, pp. 205–212.

Guangyan Zhang received the bachelor's and master's degrees in computer science from Jilin University, China, in 2000 and 2003, respectively, and the doctor's degree in computer science and technology from Tsinghua University, Beijing, China, in 2008. He is now an associate professor with the Department of Computer Science and Technology, Tsinghua University. His current research interests include big data computing, network storage, and distributed systems.

Keqin Li received the BS degree in computer science from Tsinghua University, China, in 1985, and PhD degree in computer science from the University of Houston, Texas, in 1990. He is a SUNY distinguished professor of computer science in the State University of New York, New Paltz. His research interests include design and analysis of algorithms, parallel and distributed computing, and computer networking.

Jingzhe Wang is now an undergraduate student in the Department of Computer Science and Technology at Tsinghua University, China. His current research interest include network storage.

Weimin Zheng received the master's degree from Tsinghua University, China, in 1982. He is a professor with the Department of Computer Science and Technology, Tsinghua University. His research covers distributed computing, compiler techniques, and network storage.