IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 30, 2018, accepted January 21, 2019, date of publication January 31, 2019, date of current version March 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2896263

A Fast Approach to Scale Up Disk Arrays With Parity Declustered Data Layout by Minimizing Data Migration

ZHIPENG LI[1], YINLONG XU[1], YONGKUN LI[1], CHENGJIN TIAN[1], AND JOHN C. S. LUI[2], (Fellow, IEEE)

[1]School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China
[2]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong

Corresponding author: Zhipeng Li (lizhip@mail.ustc.edu.cn)

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1003204, and in part by the National Nature Science Foundation of China under Grant 61832011 and Grant 61772486.

ABSTRACT Parity declustering is widely deployed in erasure-coded storage systems so as to provide fast recovery and high data availability. However, to perform scaling on such redundant array of inexpensive disks (RAIDs), it is necessary to preserve parity declustered data layout so as to preserve the properties after scaling. Unfortunately, existing scaling algorithms fail to achieve this goal so they cannot be applied for scaling RAIDs with parity declustering. To address this challenge, we develop an efficient online scaling scheme called parity declustering scaling (PDS), which employs an auxiliary balanced incomplete block design to define the data migration so as to preserve parity declustered data layout. Furthermore, PDS can also be applied to scale RAIDs for improving reliability and/or storage efficiency as options by allocating more parity blocks and/or data blocks in stripes. We provide theoretical proofs to formally show that PDS preserves parity declustered data layout, and achieves uniform distributions of data and parity blocks after scaling while requiring only the minimal data migration. We implement PDS in Linux kernel 3.14.72 and evaluate its performance with real-world traces. The results show that PDS can reduce 82.37 percent of scaling time and 18.25 percent of user response time during scaling on average, compared with ''moving-everything'' round-robin approach adapted to achieve parity declustered data layout after scaling.

INDEX TERMS Scaling, parity declustering, capacity expansion, reliability, data migration.

I. INTRODUCTION

Nowadays almost all professions in life, from enterprises to research academies, are continuously producing large amount of digital data, and as a result, the volume of digital data grows explosively. To meet the demand of large storage capacity, high I/O bandwidth, and data reliability, *Redundant Array of Inexpensive Disks (RAID)* [18] and distributed storage systems, both of which aggregate a set of individual storage devices, are two kinds of common solutions. Since these storage systems are typically built out of a large number of individual components that can be unreliable, component failures have become commonplace in modern storage systems [19].

Replication and *erasure code* are two common approaches to protect data against failures. In particular, modern storage systems usually adopt erasure codes to provide reliability with significantly lower storage overhead. However, in erasure-coded storage systems, the data reconstruction process for disk failures not only consumes a lot of bandwidth, but also opens the *window of vulnerability* for data loss [28]. As the storage capacity of modern disk is growing at a much faster rate than the disk I/O speed, the disk recovery process for modern disks takes a much longer time. This implies the lengthening of time in the window of vulnerability and it increases the chance of data loss.

To speed up the failure recovery process and provide highly-available arrays, parity declustering was first proposed by Muntz and Lui [16] as a data layout technique, and it was further realized by Holland and Gibson [13] based on *Balanced Incomplete Block Design (BIBD)* in practical systems. Due to the benefits of fast recovery and providing highly-available arrays, parity declustering has been implemented in the software RAID device driver RAIDFrame [6], it is also

The associate editor coordinating the review of this manuscript and approving it for publication was Sing Kiong Nguang.

deployed in the Panasas file system [25] and modern erasure-coded storage systems [2].

However, due to the increasing demand of storage capabilities, applications often require larger storage capacities and higher performance. This is normally achieved by adding new disks to the existing RAID system [7], [29]. In order to regain load balance, data need to be redistributed evenly among all disks. Since the RAID system contains more disks, concurrent disk failures are more likely to happen, and so the reliability of the disk array may need to be improved during data redistribution. Moreover, in today's server environments where applications access data constantly, the downtime cost can be extremely high [17]. To provide uninterrupted service, data redistribution needs to be performed online. Such disk addition to RAID system is known as *RAID scaling*.

Several challenges arise in scaling RAIDs when parity declustering is deployed. For scaling RAIDs with parity declustering, we not only need to satisfy the traditional scaling requirements, but also need to keep parity declustered data layout after scaling so as to preserve its nice properties. However, it is not an easy task due to the requirement of keeping parity declustered data layout after scaling. We describe more detailed discussions about the technical challenges of scaling RAIDs with parity declustering in Section III-B.

Even though there are many online scaling approaches, such as ALV [33], GSR [26], PBM [15], and MiPiL [32] for RAID-5 data layout; SDM [27], RS6 [30], and Xscale [31] for RAID-6 data layout; Round-Robin (RR) [5], [9], Semi-RR [8], and H-Scale [24] for different RAID levels, none of them can be deployed for scaling RAIDs with parity declustering except for RR only, which also needs some changes in its design. Furthermore, RR requires to migrate 100 percent of data blocks during scaling. This ''moving-everything'' approach will cause very expensive migration cost, thus degrades the scaling performance greatly. This motivates us to develop a new scheme for scaling RAIDs with parity declustering.

In this paper, we propose a novel scaling approach, *Parity Declustering Scaling (PDS)*, to efficiently scale RAIDs with parity declustering. PDS not only preserves parity declustered data layout after scaling, but also evenly redistributes data/parity blocks across all disks and minimizes the data migration. To the best of our knowledge, this is the first work to scale RAIDs with parity declustering, with the aim of preserving parity declustered data layout with minimal data migration.

The contributions of this work are as follows.

- We develop PDS with an auxiliary BIBD to define data migration. In particular, PDS has the following benefits. 1) The volume of migrated blocks is minimized. 2) The reliability and/or storage efficiency can be improved as options. 3) All data/parity blocks are evenly distributed across disks after scaling. 4) The data layout after scaling can also satisfy the requirements of parity declustering, i.e., it can also be defined by a BIBD.
- We provide theoretical proofs to formally show that PDS achieves all the above mentioned benefits. In particular, we theoretically prove the capability of our scheme for scaling RAID from one BIBD-defined data layout to another BIBD-defined data layout. That is, with our scaling scheme, the scaled RAID is also parity declustered. Thus, our scaling scheme supports successive scaling multiple times, while still preserves parity declustered data layout.
- We implemented PDS in the MD (Multiple Devices) driver in Linux kernel 3.14.72, and conducted experiments with real-world traces to show its performance. Results show that PDS can reduce 82.37 percent of scaling time and 18.25 percent of user response time during scaling on average, while keeping almost identical performance after scaling, compared with RR adapted to achieve parity declustered data layout after scaling.

The remainder of this paper proceeds as follows. Section II provides the background of parity declustering. Section III reviews related work on existing RAID-5 scaling approaches and then discusses the technical challenges of scaling for parity declustered data layout. We describe our motivations and present the main idea of PDS via an example in Section IV. We propose PDS in Section V, present its addressing algorithm in Section VI, prove its properties in Section VII, and discuss some further optimizations for PDS in Section VIII. We show experimental results in Section IX. Section X concludes the paper.

II. BACKGROUND

We first provide a brief overview of two commonly used erasure codes, RAID-5 and Reed-Solomon (RS) codes [20]. Next, we introduce basic concepts and properties of Balanced Incomplete Block Design (BIBD), which is the foundation of realizing parity declustered data layout. After that, we state the lemma of BIBD, which will be used in the proof of our Theorem 2. Finally, we take RAID-5 as an example to show how parity declustering uses the mathematical properties of BIBD to speed up disk failure recovery and provide high data availability.

A. RAID-5 AND RS CODES

Given n equal-size data blocks, say $B_0, B_1, \cdots, B_{n-1}$, RAID-5 encodes them into one parity block P by simply XOR-summing them as $P = B_0 \oplus B_1 \oplus \cdots \oplus B_{n-1}$. All of these $n+1$ blocks $B_0, B_1, \cdots, B_{n-1}$ and P form a *stripe* of size $n+1$ and are distributed evenly across $n+1$ disks. Any one of the $n+1$ blocks can be reconstructed with the other n blocks in the same stripe. Therefore, RAID-5 code can tolerate one disk failure. Fig. 1 shows a four-disk array with a left-symmetric[1] RAID-5 data layout, where the data and parity blocks in each stripe are rotationally stored across

[1] There is a variety of strategies in RAID-5 that evenly distributes the data blocks and parity blocks, typically four types of data and parity distribution are preferred, left-symmetric, left-asymmetric, right-symmetric and right-asymmetric [34].

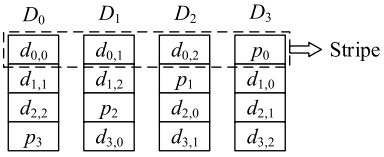


FIGURE 1. Layout of the left-symmetric RAID-5.

disks for load balance. If $d_{0,0}$ is lost for example, it can be reconstructed as $d_{0,0} = p_0 \oplus d_{0,1} \oplus d_{0,2}$.

Reed-Solomon (RS) family of codes are proposed to provide higher reliability. An RS code is associated with two parameters: k and m. A (k, m)-RS code encodes k data blocks into m parity blocks in a manner that guarantees the recoverability of all the k data blocks from any k out of these $k + m$ blocks. This collection of these $k + m$ data/parity blocks is called a stripe. Therefore, by distributing a stripe evenly across $k + m$ disks, a (k, m)-RS code can tolerate any m concurrent disk failures. RAID-5 and RS codes achieve the so-called *Maximum Distance Separable (MDS)* property, as they deliver optimal fault tolerance for the space dedicated to coding.

B. BIBD

BIBD [10] is a type of design in combinatorial design theory that concerns how to arrange elements of a finite set into subsets for certain "balance" properties. BIBD has many applications, such as experimental design, software testing, network topology, cryptography, and data layout. In particular, BIBD is used to formulate the data layout of parity declustering [13].

Definition 1: Given five positive integers b, v, k, r, λ and a base set $S = \{s_0, s_1, \cdots, s_{v-1}\}$ of v objects, a (b, v, k, r, λ)-BIBD on S is an arrangement of its objects into b tuples[2] *$\mathbb{B} = (T_0, T_1, \cdots, T_{b-1})$, which satisfies*

1) $|T_i| = k$ *for* $0 \le i \le b - 1$,
2) *each object s_j is exactly in r tuples,*
3) *each pair of objects s_{j_0}, s_{j_1} appear exactly in λ tuples for* $0 \le j_0 \ne j_1 \le v - 1$.

The five parameters b, v, k, r, λ of a BIBD satisfy the following two equations [10],

$$bk = vr, \tag{1}$$

$$\lambda(v - 1) = r(k - 1). \tag{2}$$

A BIBD can also be represented by means of an incidence matrix [10], the definition of an incidence matrix is as follows.

Definition 2: The incidence matrix of a (b, v, k, r, λ)-BIBD is a 0–1 *matrix $M = (m_{i,j})_{b\times v}$ defined as*

$$m_{i,j} = \begin{cases} 1 & s_j \in T_i, \\ 0 & s_j \notin T_i. \end{cases}$$

Furthermore, a BIBD satisfies the following lemma [23].

[2]The term *tuple* is called *block* in combinatorial design theory, but it is easily confused with the commonly held definition of a block as a contiguous chunk of data in storage. So we use tuple by following the work in [13].

Lemma 1: Suppose that $M = (m_{i,j})_{b\times v}$ is the incidence matrix of a (b, v, k, r, λ)-BIBD, and let $0 \le j_0, j_1 \le v - 1$, $j_0 \ne j_1$. Then the following properties hold.

1) $|\{i \mid m_{i,j_0} = 1, m_{i,j_1} = 1\}| = \lambda$,
2) $|\{i \mid m_{i,j_0} = 1, m_{i,j_1} = 0\}| = r - \lambda$,
3) $|\{i \mid m_{i,j_0} = 0, m_{i,j_1} = 1\}| = r - \lambda$,
4) $|\{i \mid m_{i,j_0} = 0, m_{i,j_1} = 0\}| = b - 2r + \lambda$.

Fig. 2a shows an example of a (4, 4, 3, 3, 2)-BIBD of base set $S = \{0, 1, 2, 3\}$ and 2b shows its incidence matrix. There are $v = 4$ objects and $b = 4$ tuples. Each tuple contains $k = 3$ objects and each object appears in $r = 3$ tuples. Each pair of objects appear in exactly $\lambda = 2$ tuples. Furthermore, we present an example to illustrate Lemma 1, let $(m_{i,j})_{4\times4}$ be the matrix shown in Fig. 2b and $j_0 = 0, j_1 = 1$, we have that

1) $|\{i \mid m_{i,0} = 1, m_{i,1} = 1\}| = |\{0, 1\}| = \lambda = 2$,
2) $|\{i \mid m_{i,0} = 1, m_{i,1} = 0\}| = |\{2\}| = r - \lambda = 1$,
3) $|\{i \mid m_{i,0} = 0, m_{i,1} = 1\}| = |\{3\}| = r - \lambda = 1$,
4) $|\{i \mid m_{i,0} = 0, m_{i,1} = 0\}| = b - 2r + \lambda = 0$.

C. PARITY DECLUSTERING

Parity declustering was first proposed by Muntz and Lui [16], and it was further realized based on BIBD in [13]. Fig. 2d is an example of parity declustered data layout constructed from the BIBD shown in Fig. 2a. In this example, four stripes of size three shown in Fig. 2c are distributed to four disks. Refer to Figs. 2d and 2b, in parity declustered data layout, stripes and disks correspond to tuples and objects in a BIBD respectively. For example, tuple T_0 defines the layout of stripe S_0, which consists of the three blocks on disks D_0, D_1, and D_2, respectively. As parity declustering assigns a block to the lowest available offset on the identified disk, Fig. 2e illustrates physical layout of the disk array. The storage space of a disk array is divided into many regions and parity declustering is performed within each region. Fig. 2e shows just one region.

Now assume that one disk in Fig. 2e (say disk D_2) fails. This disk contains three blocks which are parts of stripes S_0, S_2, and S_3 respectively. The other six surviving blocks belonging to these three stripes are stored on the other three surviving disks with two blocks on each disk. The shaded areas in Fig. 2e show that we only need to read exactly two blocks from each surviving disk to reconstruct the failed disk, and that is only two-third of the volume in each surviving disk. However, if a disk of RAID-5 shown in Fig. 1 fails, we should read the whole volume in each surviving disk to reconstruct the failed disk. Parity declustering requires less additional load on surviving disks for data reconstruction so that surviving disks can serve more user I/O requests, therefore resulting in higher user throughput during recovery and shorter recovery time.

More generally, in each region of a disk array with parity declustering constructed from a (b, v, k, r, λ)-BIBD, b stripes of size k are distributed across v disks with r blocks on each disk, and each pair of disks store exactly λ pairs of blocks belonging to the same stripe. Therefore, if a disk fails, we need to read exactly λ blocks from each surviving disk

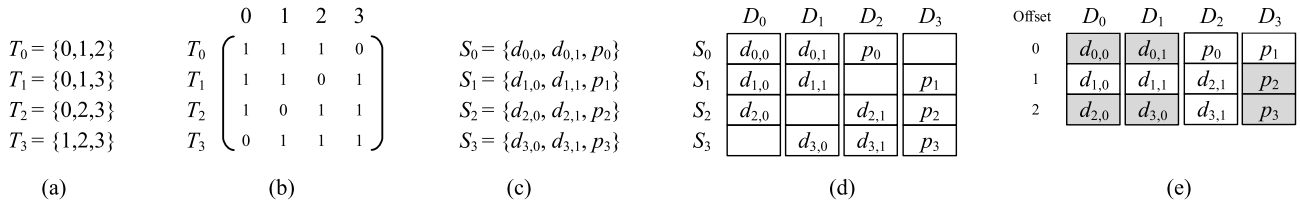


FIGURE 2. An example of parity declustering.

storing r blocks in each region for reconstruction. Furthermore, the fraction of volume to be read from each surviving disk is $\lambda/r = (k-1)/(v-1)$ according to (2).

D. FULL BLOCK DESIGN TABLE

Parity blocks should be evenly distributed across the array. Every data block update causes the update of the parity blocks in the same stripe, and so an uneven parity distribution would lead to imbalanced utilization, since the disks with more parity blocks would experience more load. However, it is apparent from Fig. 2e that parity blocks are not evenly distributed across the array within a region. But we can balance the distribution of parity blocks within a group of k regions, where k is a parameter in the (b, v, k, r, λ)-BIBD. We first arrange the objects in all tuples of a BIBD as a sub-matrix (e.g., the first four rows in the right side of Fig. 3), which correspond to the data layout of a region. Then we group k sub-matrices into a *full block design table* (e.g., the right side of Fig. 3), to define the data layout of a group of k regions, which is called a *perfect-parity-declustering* (abbr. as *PPD*) area in this paper. In the i-th region of a PPD area, parity blocks are stored in the disks corresponding to the objects in the $(k-i)$-th column of the i-th sub-matrix of the full block design table, as the shaded blocks in the left side of Fig. 3. Because the full block design table duplicates the original (b, v, k, r, λ)-BIBD k times, it corresponds to a $(kb, v, k, kr, k\lambda)$-BIBD.

Furthermore, if b is a multiple of v, parity blocks can be balanced perfectly within a region [21]. For example,

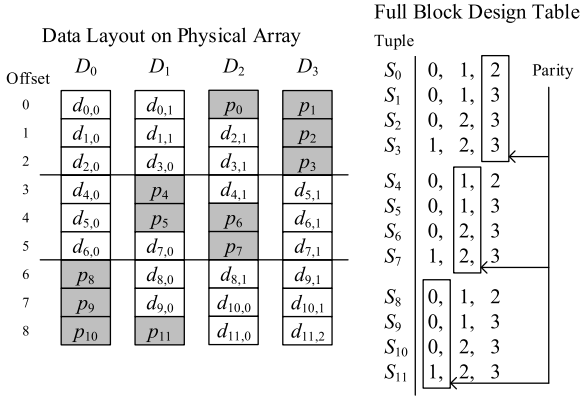


FIGURE 3. Full block design table.

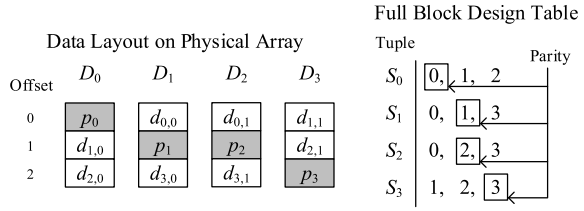


FIGURE 4. Optimized full block design table.

$b = v = 4$ in the BIBD in Fig. 2a, parity blocks can be distributed evenly across the array within a region as shown in Fig. 4. To present our scaling algorithm more concisely and save the pages, we will take Fig. 4 instead of Fig. 3 as an example of the data layout in a PPD area in the following. We also define a column of a PPD area as a *PPD-column*, which corresponds to all blocks of this PPD area on a disk, as shown in Fig. 12a. We will migrate blocks in unit of PPD-columns during scaling.

III. RELATED WORK AND TECHNICAL CHALLENGES

In this section, we first review existing RAID-5 scaling approaches, then we discuss the technical challenges of scaling for parity declustered data layout.

A. RELATED WORK ON RAID-5 SCALING

Existing approaches to scale up a RAID-5 disk array include Round-Robin (RR), ALV, GSR, MiPiL, H-Scale, CRAID, etc. In this section, we use a left-asymmetric RAID-5 [34] to illustrate the scaling processes of these approaches.

- **Round-Robin (RR)** is a traditional scaling approach that migrates almost all data to preserve the round-robin data distribution after adding disks. All parity blocks need to be recalculated after data migration, and the scaling process of RR is shown in Fig. 5. Large data migration results in expensive cost of a RAID-5 scaling. GA [9] controls the scaling speed to relieve online performance degradation during scaling to avoid downtime, and the GA's data migration process is similar to the RR's. Based on RR, Linux provides a reshape toolkit named MD-Reshape [5] in the MD (Multiple Devices) driver shipped with Linux kernel to support online capacity expansion.
- **ALV** [33] improves the efficiency of the RR scaling process by changing the transfer order of data blocks in

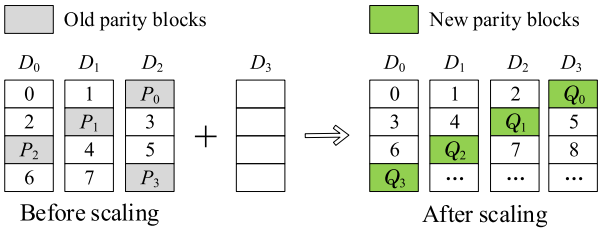


FIGURE 5. RAID-5 scaling from 3 disks to 4 using the Round-Robin approach.

order to aggregate migration I/Os for contiguous blocks. For example, in Fig. 6 (state 3), reads for blocks 6 and 8 are aggregated into one read I/O, hence reducing the total I/Os. ALV is an extension of the RR algorithm, and as such it still suffers from large data migration.

- **GSR** [26] is proposed to accelerate RAID-5 scaling and it achieves the minimal data migration. As illustrated in Fig. 7, stripes are classified into three categories: retained stripes ($S_0 \sim S_5$), remapped stripes ($S_6 \sim S_8$), and destructed stripes ($S_9 \sim S_{11}$). Data blocks in retained stripes are retained in the same disk; data blocks in remapped stripes are retained in the same disk by remapping to a new stripe; data blocks in destructed stripes are migrated to the new disk D_3. More generally, GSR divides data on the original array into two consecutive sections, and then moves the second section of data to the new disks, while keeping the first section of data unmoved. Its main limitation is the performance after scaling, accesses to the first section of data are served only by original disks, and accesses to the second section of data are served only by new disks, which brings a large performance penalty under the workload with a strong locality.
- **MiPiL** [32] is a RAID-5 scaling approach, which minimizes data migration while maintaining uniform data and parity distributions. Before data migration, MiPiL introduces a normalizing operation to shuffle columns and rows as illustrated in Fig. 8, then it moves the minimum number of data blocks from old disks to the new disk(s) for regaining a uniform data distribution. Furthermore, MiPiL optimizes online migration process by minimizing the number of mapping metadata writes.
- **H-Scale** [24] is a general scaling approach for different RAID levels, and it achieves fast RAID scaling via hybrid stripe layouts. H-Scale can perform RAID-5 scaling with minimal data migration. As illustrated in Fig. 9, blocks 0, P_1, and 5 are migrated to the new disk D_3, and their original positions are served for storing new data blocks of a new stripe. H-Scale stores the parity block of the new stripe, P'_0, on the new disk D_3, and as such no existing blocks are overwritten, therefore providing the reliability for the scaling process. However, it cannot regain a uniform data/parity distribution after scaling.
- **CRAID** [36] is a RAID architecture that uses a dedicated caching partition to capture frequently accessed data and redistributes data in this partition to incremental devices. As illustrated in Fig. 10, CRAID redistributes only hot data when new disks are added, therefore reducing the migration even further. CRAID requires less data migration for RAID scaling, while it has to identify those frequently accessed data all the time and perform additional I/O operations and parity computations for dirty blocks. Therefore, CRAID performs extra statistics of data accesses, and in turn suffers from additional spatial and temporal overheads [31]. It should be noted that RAID scaling involves occasional events, while statistics of data accesses are performed all the time by CRAID.

B. TECHNICAL CHALLENGES OF SCALING FOR PARITY DECLUSTERING

When implementing an erasure code into a RAID system, each disk is divided into many blocks. The erasure code is independently performed in each stripe, which consists of multiple blocks with exactly one block on a disk. The *stripe size* is defined as the number of blocks in a stripe. In traditional RAID systems, the stripe size is equal to the total number of disks in the system, that is, the blocks of each stripe are distributed across disks, with exactly one block on each disk. However, parity declustering defines a mapping that allows stripes with stripe size G to be distributed over C disks (C is larger than G). Muntz and Lui define the ratio $(G-1)/(C-1)$ as α [16], and it is further called the *declustering ratio* in [13]. This parameter indicates the fraction of

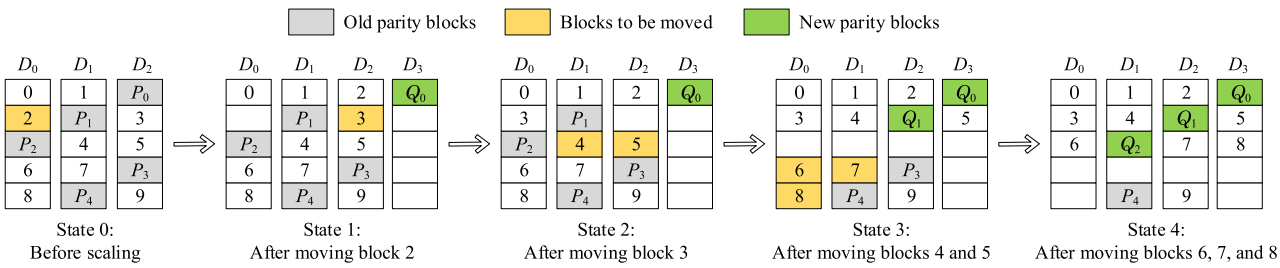


FIGURE 6. RAID-5 scaling from 3 disks to 4 using the ALV approach.

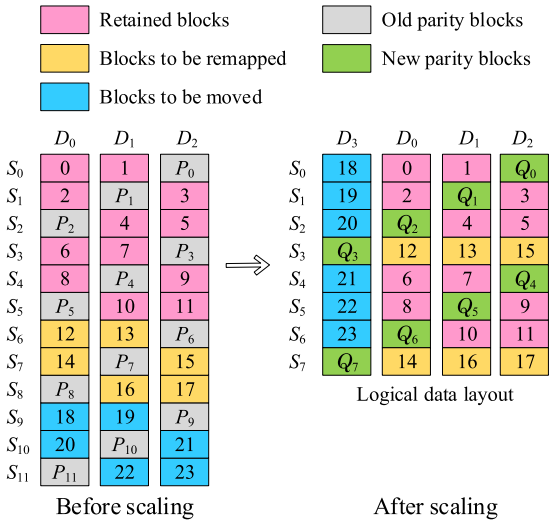


FIGURE 7. RAID-5 scaling from 3 disks to 4 using the GSR approach.

each surviving disk that must be read for reconstruction, and reconstruction time decreases as α drops [12], [13], [35].

Since any additional failure during the reconstruction process may result in data loss especially for the RAID levels with single fault-tolerant layouts, the reconstruction time is often referred to as "window of vulnerability" [28] that should be as small as possible. Parity declustering decreases reconstruction time by both reducing the per-disk load for reconstruction and utilizing all surviving disks in the array to participate in the reconstruction. When adding disks to a parity declustered array, reconstruction time is expected to be reduced since the declustering ratio can be decreased with the increasing number of disks in the array. Therefore, for scaling a parity declustered array, it is desirable to maintain parity declustered data layout within the whole array so as to preserve its nice properties and reduce the reconstruction time even further to minimize the window of vulnerability.

Even though CRAID [36] can expand volume for a parity declustered RAID array, it cannot regain parity declustered data layout after the expansion. Because CRAID is essentially a collection of independent RAID arrays that have been added to expand the storage capacity, only part of disks in the CRAID array can be used to participate in the reconstruction and reconstruction time cannot be further reduced with more additional disks in the CRAID array. For the other existing RAID scaling approaches, such as ALV [33], GSR [26], PBM [15], and MiPiL [32] for RAID-5 data layout; SDM [27], RS6 [30], and Xscale [31] for RAID-6 data layout; Round-Robin (RR) [5], [9], Semi-RR [8], and H-Scale [24] for different RAID levels, they all have a constraint that the stripe size of the erasure code deployed in the RAID system should be equal to the number of disks in the array. Thus, existing scaling approaches fail to scale RAIDs with parity declustered data layout which is defined by BIBD.

Our main idea to improve the efficiency of scaling RAIDs with parity declustered data layout is to minimize the data migrations in the scaling process. However, this is not an easy task due to the requirement of keeping parity declustered data layout after scaling. That is, the data layout in the scaled system must also satisfy the requirements defined by BIBD so as to preserve its nice properties. Besides, we have to also maintain the parity consistency so as to guarantee the reliability. Thus, the first challenge is to design an efficient data redistribution scheme which must achieve the parity declustered data layout, minimal data migration, and uniform data and parity distributions simultaneously.

Furthermore, as the reliability and/or storage efficiency can be improved as options in the scaling process, another technical challenge is to design a data allocation scheme for new data blocks which will fill in the scaled RAID arrays after data migration, and the data allocation scheme must also maintain the property of parity declustered data layout, uniform data and parity distributions, and provide higher reliability and/or better storage efficiency.

IV. MOTIVATIONS AND AN EXAMPLE

In this section, we first present our motivations through the design goals of scaling for parity declustered data layout,

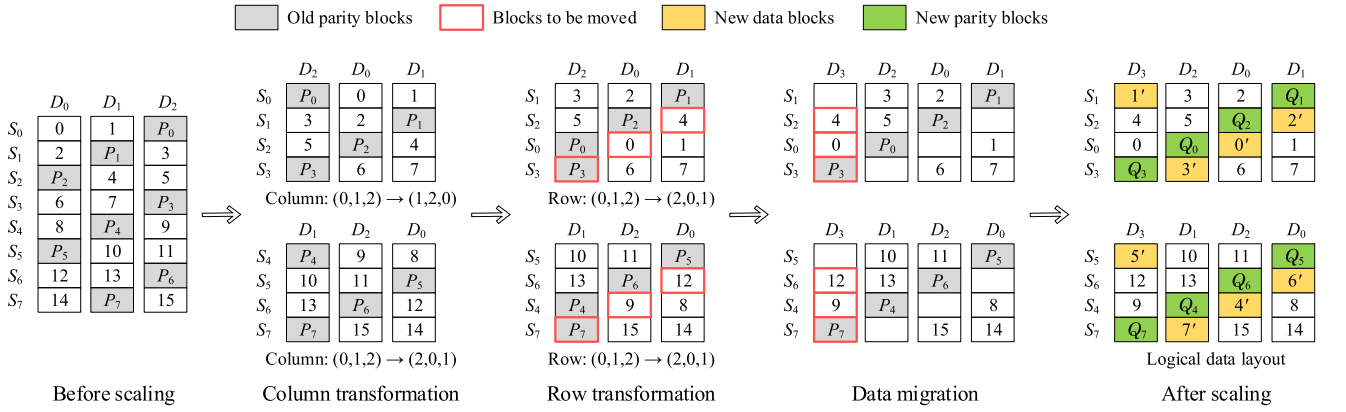


FIGURE 8. RAID-5 scaling from 3 disks to 4 using the MiPiL approach.

TABLE 1. Major notations used in this paper.

Notation	Description
D_i	The i-th disk in the RAID
S_i	The i-th stripe
$d_{i,j}$	The j-th data block in stripe S_i
p_i	The first parity block in stripe S_i
$p_{i,j}$	The j-th parity block in stripe S_i
$\mathbb{B}_p$	The primary BIBD which defines the data layout before scaling
$(b_p, v_p, k_p, r_p, \lambda_p)$	The five parameters of $\mathbb{B}_p$
$C_{i,j}$	The PPD-column in the i-th PPD area, which is stored in disk D_j before scaling
$\mathbb{B}_a$	The auxiliary BIBD for data migration
$(b_a, v_a, k_a, r_a, \lambda_a)$	The five parameters of $\mathbb{B}_a$
T_i^a	The i-th tuple of $\mathbb{B}_a$
$T_i^{\complement}$	Set $\{j \mid j \in \{0, 1, \cdots, v_a - 1\}\ and\ j \notin T_i^a\}$
x_{inc}	Number of new data blocks added to an original stripe
y_{inc}	Number of new parity blocks added to an original stripe
$d'_{i,j}$	The j-th new allocated data block for increasing stripe size in stripe S_i $(0 \le j \le x_{inc} - 1)$
$p'_{i,j}$	The j-th new allocated parity block for increasing stripe size in stripe S_i $(0 \le j \le y_{inc} - 1)$
$\mathbb{B}_{inc}$	The auxiliary BIBD for increasing stripe size
$(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc})$	The five parameters of $\mathbb{B}_{inc}$
T_i^{inc}	The i-th tuple of $\mathbb{B}_{inc}$
TBL_{inc}	The full block design table constructed from $\mathbb{B}_{inc}$ for increasing stripe size
$G_{i,j}^{d_k}$ & $G_{i,j}^{p_k}$	The IBA-groups where subscript i means the i-th scaled PPD area in a PIE and j means the j-th disk containing free storage units in a row, d_k and p_k mean the k-th new data block and the k-th new parity block to be added to an original stripe, respectively
$\mathbb{B}_{new}$	The auxiliary BIBD for placing new data after scaling
$(b_{new}, v_{new}, k_{new}, r_{new}, \lambda_{new})$	The five parameters of $\mathbb{B}_{new}$
TBL_{new}	The full block design table constructed from $\mathbb{B}_{new}$ for placing new data
$\mathbb{B}_o$	The objective BIBD after data migration
$(b_o, v_o, k_o, r_o, \lambda_o)$	The five parameters of $\mathbb{B}_o$
$\mathbb{B}_o^*$	The objective BIBD after the stripe size being increased
$(b_o^*, v_o^*, k_o^*, r_o^*, \lambda_o^*)$	The five parameters of $\mathbb{B}_o^*$

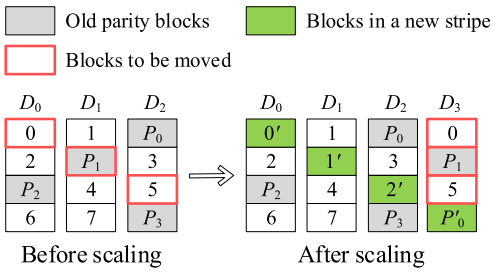


FIGURE 9. RAID-5 scaling from 3 disks to 4 using the H-Scale approach.

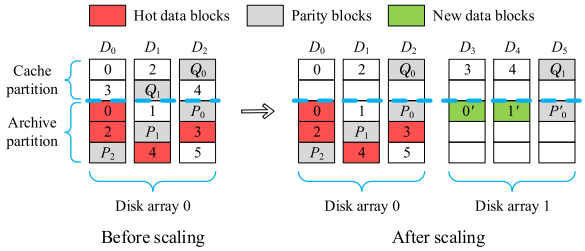


FIGURE 10. Addition of 3 disks to a 3-disk CRAID array.

then we show our main idea of our scaling algorithm PDS via an example. To facilitate our discussion, we summarize the major notations used for PDS in this paper in Table 1.

A. DESIGN GOALS OF SCALING FOR PARITY DECLUSTERING

Existing approaches to scale up RAID systems include ALV [33], GSR [26], PBM [15], and MiPiL [32] for RAID-5 data layout; SDM [27], RS6 [30], and Xscale [31] for RAID-6 data layout; Round-Robin (RR) [5], [9], Semi-RR [8], and H-Scale [24] for different RAID levels, but none of them can be applied for scaling RAIDs with parity declustering except for RR which also needs some changes in its design. This motivates us to design an efficient scaling algorithm for RAIDs with parity declustered data layout. In particular, our scaling algorithm aims for the following five objectives.

- **Objective 1** (*Parity Declustered Data Layout*): After scaling, the data layout of the RAID should still be parity declustered, i.e., the data layout can be defined by a BIBD.
- **Objective 2** (*Higher Reliability and/or Better Storage Efficiency*): When a RAID deploys an (x, y)-RS code with parity declustering, it can tolerate any y concurrent disk failures and the storage efficiency is $x/(x + y)$. As parity declustering decouples stripe size from the number of disks in the array, we expect that the reliability and/or storage efficiency can be improved as options with the help of the scaling process.
- **Objective 3** (*Uniform Data and Parity Distributions*): After scaling, each disk contains the same amount of

$T_0 = \{0,1,2,3\}$ $T_3 = \{0,3,5,6\}$ $T_5 = \{1,3,4,6\}$
$T_1 = \{0,1,4,5\}$ $T_4 = \{1,2,5,6\}$ $T_6 = \{2,3,4,5\}$
$T_2 = \{0,2,4,6\}$

FIGURE 11. A (7, 7, 4, 4, 2)-BIBD.

data blocks and the same amount of parity blocks so as to maintain load balance.

- **Objective 4** (*Minimal Data Migration*): Assume that n new disks are added to an array of m old disks storing S data/parity blocks in total. To achieve a uniform data distribution, the minimal number of blocks that have to be migrated from old disks to new disks for scaling is $S \times n/(n+m)$.
- **Objective 5** (*Fast Data Addressing*): After scaling, the location of a block can be computed by an algorithm with low time and space complexities.

B. AN EXAMPLE OF SCALING

To understand how the PDS algorithm works and how it satisfies the objectives as stated in Section IV-A, we take RAID scaling from four disks to seven as an example. As shown in Fig. 4, the data layout of an array with four disks is defined by a *primary* $(b_p, v_p, k_p, r_p, \lambda_p) = (4, 4, 3, 3, 2)$-BIBD $\mathbb{B}_p$ shown in Fig. 2a. We aim to scale up this RAID from four disks to seven. We select an *auxiliary* $(b_a, v_a, k_a, r_a, \lambda_a) = (7, 7, 4, 4, 2)$-BIBD $\mathbb{B}_a$ in Fig. 11 to define the data migration.

We divide all PPD areas into multiple groups, with each group consisting of b_a PPD areas and being named as a *perfect migrating entirety* (abbr. as *PME*). PDS performs scaling in unit of PMEs and migrates blocks in unit of PPD-columns. We diagram the data migration process in Fig. 12, where Fig. 12a is the physical data layout of a PPD area before data migration, Fig. 12b is a PME where each row corresponds to a PPD area in Fig. 12a, Fig. 12c is the auxiliary BIBD $\mathbb{B}_a$ for data migration, and Fig. 12d shows the data layout of the PME after data migration. We scale up the RAID in unit of $b_a = 7$ PPD areas, which form a PME. The data migration of the i-th PPD area is defined by the i-th tuple in $\mathbb{B}_a$. In Fig. 12b, one row corresponds to a PPD area, and $C_{i,j}$ denotes all blocks in the j-th PPD-column of the i-th PPD area, which are stored in disk D_j.

Now we migrate some PPD-columns from the four old disks to the three new disks for scaling according to $\mathbb{B}_a$ in Fig. 12c. Denote $\mathbb{B}_a = (T_0^a, T_1^a, \cdots, T_6^a)$. We redistribute the data layout of the i-th PPD area (i.e., the i-th row in Fig. 12b) according to the i-th tuple T_i^a. We know that before redistribution, the i-th PPD area consists of four PPD-columns, $C_{i,0}, C_{i,1}, C_{i,2}, C_{i,3}$, which are stored in disks D_0, D_1, D_2, D_3 respectively. Suppose that $T_i^a = \{i_0, i_1, i_2, i_3\}$, we should redistribute $C_{i,0}, C_{i,1}, C_{i,2}, C_{i,3}$ to disks $D_{i_0}, D_{i_1}, D_{i_2}, D_{i_3}$. For example, $T_2^a = \{0, 2, 4, 6\}$, we should redistribute $C_{2,0}$, $C_{2,1}, C_{2,2}, C_{2,3}$ to disks D_0, D_2, D_4, D_6. Since $C_{2,0}$ and $C_{2,2}$ are stored in disks D_0, D_2 respectively, to minimize the blocks being migrated, we only migrate $C_{2,1}, C_{2,3}$ from disks D_1, D_3 to disks D_4, D_6 respectively, as shown in the third row of Fig. 12d.

After data migration, we may exploit the free storage units in the following ways, such as

- We use the same erasure code as which before the scaling, i.e., we only increase the storage capacity and keep the erasure code unchanged.
- We add some data blocks into the stripes to increase the storage capacity and meanwhile increase the storage efficiency. For example, if we increase the stripe size from three to four in the above example by adding one more data block into each stripe, the storage efficiency increases from 2/3 to 3/4.
- We add some parity blocks into the stripes to increase the reliability. For example, if we increase the stripe size from three to four in the above example by adding one more parity block into each stripe, the disk array will tolerate two concurrent disk failures. However, adding one more parity block in each stripe makes the storage efficiency decrease from 2/3 to 2/4.
- We add some data blocks and some parity blocks simultaneously to increase storage efficiency and reliability.

Now we continue the example in Fig. 12 to explain how we add one data block and one parity block to a stripe in the scaled RAID while still keeping balanced distribution of data blocks and parity blocks, which is diagramed in Fig. 14. In

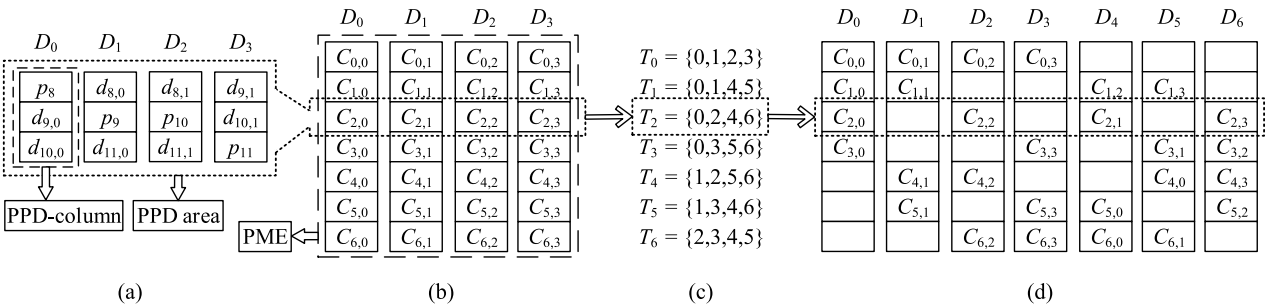


FIGURE 12. Data migration during scaling from 4 disks to 7. (a) Physical layout of a PPD area. (b) Layout of a PME before data migration. (c) The auxiliary BIBD. (d) Layout of a PME after data migration.

$$T_0 = \{0, 1\}$$
$$T_1 = \{0, 2\}$$
$$T_2 = \{1, 2\}$$

FIGURE 13. A (3, 3, 2, 2, 1)-BIBD.

Fig. 12d, there are three disks containing free storage units in each row. We use three objects 0, 1, 2 to represent the first, the second and the third disks containing free storage units in a row respectively, and use a tuple of two objects to define the allocation of two data/parity blocks to the three disks containing free storage units in a row. For example a tuple $T = \{0, 1\}$ means that we allocate two blocks to the first and the second disks containing free storage units in a row.

We diagram the allocation of new data/parity blocks in Fig. 14, where there are 3 scaled PMEs in Fig. 14a and each one has the same logical layout as the scaled PME in Fig. 12d. All free storage units are shadowed and we show the logical layout of a scaled PPD area with the new allocated data/parity blocks in Fig. 14b. To keep the balanced distribution of data blocks and parity blocks after adding one data block and one parity block to each stripe, we first select an other auxiliary $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc}) = (3, 3, 2, 2, 1)$-BIBD $\mathbb{B}_{inc}$ shown in Fig. 13. Then we use the full block design table constructed from $\mathbb{B}_{inc}$, which is shown in Fig. 14c, to define the allocation of new data/parity blocks.

In Fig. 14c, the i-th tuple defines the allocation of new data/parity blocks in the i-th PMEs. For example $T_0 = \{0, 1\}$ and 1 is labeled with P_0. So for each stripe in a row of the first PME, we allocate a data block to the first disk containing free storage units in the row and a parity block to the second disk containing free storage units in the row. Similarly T_1 and T_2 define the allocation of new data/parity blocks in the second PME and the third PME respectively. The logical layout of the scaled array with the new allocated data/parity blocks is shown in Fig. 14a. From Fig. 14a, we can find the distributions of data blocks and parity blocks are still balanced after adding new data/parity blocks due to the full block design table in Fig. 14c.

V. PDS APPROACH

A. BASIC IDEAS OF PDS

1) DATA MIGRATION PROCESS

Suppose we are to scale an array of v_p disks, which is deployed with parity declustering, to an array of v_o disks. Suppose the data layout of the system before scaling is defined by a $(b_p, v_p, k_p, r_p, \lambda_p)$-BIBD, $\mathbb{B}_p$, called *primary BIBD* in this paper. To achieve parity declustering in the

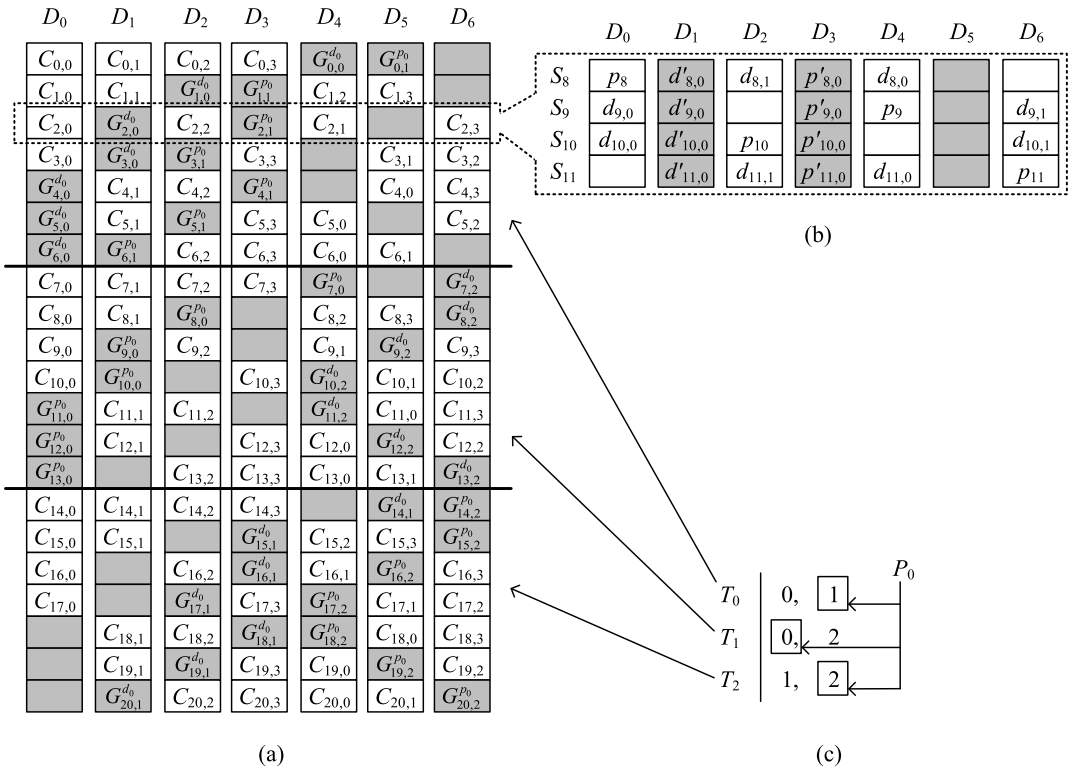


FIGURE 14. The allocation of a new parity block and a new data block into each stripe after data migration for improving reliability and storage efficiency. (a) Logical layout after the stripe size being increased. (b) Logical layout of a PPD area after the stripe size being increased. (c) Full block design table for increasing stripe size.

scaled system, we should guarantee that, in the scaled system, the data layout of the original blocks which belong to the old system can be defined by some $(b_o, v_o, k_o, r_o, \lambda_o)$-BIBD, $\mathbb{B}_o$, called *objective BIBD after data migration* in this paper.

To migrate some blocks from old disks to new disks for scaling, we use an auxiliary $(b_a, v_a, k_a, r_a, \lambda_a)$-BIBD, $\mathbb{B}_a$, called *auxiliary BIBD for data migration*, to define data migration of PDS. We set $v_a = v_o$ being the number of disks in the scaled system and $k_a = v_p$ being the number of disks in the system before scaling. Moreover, k_a is also the number of objects in a tuple of $\mathbb{B}_a$. We set $k_a = v_p$ to redistribute blocks from v_p old disks to v_o old/new disks according to the tuples of $\mathbb{B}_a$.

During the scaling process, we divide all PPD areas into multiple groups, with each group consisting of b_a PPD areas and being named as a *perfect migrating entirety* (abbr. as *PME*). PDS performs the data migration in unit of PMEs. The data migration of the i-th PPD area in a PME is defined by the i-th tuple of $\mathbb{B}_a$, where each object in a tuple corresponds to a PPD-column in a PPD area.

2) IMPROVING RELIABILITY AND/OR STORAGE EFFICIENCY

After data migration, we can improve reliability and/or storage efficiency as options. We allocate new data/parity blocks in the original stripes to improve the reliability and the storage efficiency, then the stripe size of the system is increased. To achieve parity declustering in the scaled system with stripe size increased, we should guarantee that, in the scaled system, the data layout of the original blocks which belong to the old system and the new blocks allocated for increasing stripe size can be defined by some $(b_o^*, v_o^*, k_o^*, r_o^*, \lambda_o^*)$-BIBD, $\mathbb{B}_o^*$, called *objective BIBD after the stripe size being increased* in this paper.

To allocate new data/parity blocks in the original stripes for improving the reliability and the storage efficiency, we use another auxiliary $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc})$-BIBD, $\mathbb{B}_{inc}$, called *auxiliary BIBD for increasing stripe size*, to define the allocation of new blocks. Because there are b_a PPD areas in a PME and there are b_p stripes in a PPD area, so there are totally $b_a \times b_p$ stripes in a PME. If we are to add x_{inc} new data blocks and y_{inc} new parity blocks to each of the original stripes, we need $(x_{inc} + y_{inc}) \times b_a \times b_p$ free storage units in a PME. During the scaling process, we add $v_a - v_p$ disks to the disk array. Because there are $r_p \times b_a$ storage units in each disk of a PME, the scaling process adds $(v_a - v_p) \times r_p \times b_a$ free storage units to each PME. To accommodate the new blocks added to free storage units, we have

$$\begin{aligned} x_{inc} + y_{inc} &\le (v_a - v_p) \times r_p \times b_a/(b_a \times b_p) \\ &= (v_a - v_p) \times r_p/b_p \quad \text{(eliminating } b_a\text{)} \\ &= (v_a - v_p) \times k_p/v_p. \quad \text{(by (1))} \end{aligned}$$

We set $v_{inc} = v_a - v_p$ being the number of new added disks in the scaling process and $k_{inc} = x_{inc} + y_{inc}$ being the number of new blocks added to each original stripe. Furthermore, when we want to increase reliability (i.e., $y_{inc} \neq 0$), we set b_{inc} being a multiple of v_{inc} so as to distribute parity evenly among distinct objects of the full block design table constructed from $\mathbb{B}_{inc}$.

Suppose that there are x_p data blocks and y_p parity blocks in each original stripe before scaling. So the stripe size before scaling is $k_p = x_p + y_p$. After allocating x_{inc} new data blocks and y_{inc} parity blocks into each of the original stripes, the stripe size of the scaled array becomes $k_o^* = k_p + x_{inc} + y_{inc}$ and the scaled array can tolerate $y_p + y_{inc}$ concurrent disk failures with storage efficiency of $(x_p + x_{inc})/k_o^*$. Moreover, as the number of disks in the scaled array is not changed in the process of improving reliability and/or storage efficiency, we have $v_o^* = v_o$.

To increase stripe size, we divide all PMEs into multiple groups, with each group consisting of b_{inc} PMEs and being named as a *perfect increasing stripe size entirety* (abbr. as *PIE*). PDS performs the process of improving reliability and/or storage efficiency with stripe size increased in units of PIEs. The allocation of new data/parity blocks into the original stripes of the i-th scaled PME is defined by the i-th tuple of $\mathbb{B}_{inc}$. Each object in a tuple of $\mathbb{B}_{inc}$ corresponds to the allocation of a new data/parity block into each original stripe of a scaled PME.

B. SCALING PROCESS OF PDS

Given an array of v_p disks, which is deployed with parity declustered data layout defined by a $(b_p, v_p, k_p, r_p, \lambda_p)$-BIBD, $\mathbb{B}_p$, we are to scale it up to v_o disks and then add x_{inc} ($x_{inc} \ge 0$) data blocks and y_{inc} ($y_{inc} \ge 0$) parity blocks into each original stripe such that the data layout of the scaled system can still be defined by a BIBD. We summarize the following six steps to conduct PDS scaling.

Step 1 (Auxiliary BIBD Selection for Data Migration): PDS needs an auxiliary $(b_a, v_a, k_a, r_a, \lambda_a)$-BIBD, $\mathbb{B}_a$, to define data migration. We should select $v_a = v_o$ being the number of disks in the system after scaling and $k_a = v_p$ being the number of disks in the system before scaling. Furthermore, we select the minimum possible value for b_a to minimize the size of the full block design table of the scaled system. Take Fig. 12c as an example.

Holland [12] gave a BIBD database, in which there are (b, v, k, r, λ)-BIBDs for all k when $v < 20$, and almost all k when $20 \le v \le 43$. Hanani [11] also presented some techniques to design (b, v, k, r, λ)-BIBDs for any k when $v \le 43$. Therefore, we can obtain an auxiliary BIBD for practical sizes of disk arrays.

Step 2 (Unit Identification for Data Migration): Given an auxiliary $(b_a, v_a, k_a, r_a, \lambda_a)$-BIBD $\mathbb{B}_a$ with $v_a = v_o$, $k_a = v_p$, and minimal b_a, we group b_a PPD areas into a PME and denote the j-th PPD-column of the i-th PPD area as $C_{i,j}$, which comprises all blocks of the i-th PPD area on disk D_j. Take Fig. 12b as an example. Then we design the data migration algorithm within a PME.

Step 3 (Data Migration): Suppose that $\mathbb{B}_a = \{T_0^a, T_1^a, \cdots, T_{b_a-1}^a\}$. The data migration in the i-th PPD area is defined

by T_i^a. We migrate data/parity blocks in unit of PPD-columns. For PPD-column $C_{i,j}$,

- if $j \in T_i^a$, we do not migrate $C_{i,j}$;
- otherwise, we migrate $C_{i,j}$ to disk D_{j_l}, where j_l is defined as follows. We first define two difference sets between base set $S_p = \{0, 1, \cdots, v_p - 1\}$ and T_i^a as $S_{from} = S_p - T_i^a$ and $S_{to} = T_i^a - S_p$. Suppose that j is the l-th smallest number in S_{from}, then j_l is the l-th smallest number in S_{to}.

For example, in Fig. 12, $S_p = \{0, 1, 2, 3\}$ and $T_2^a = \{0, 2, 4, 6\}$, which means before scaling, $C_{2,0}, C_{2,1}, C_{2,2}, C_{2,3}$ in the second PPD area are stored in disks D_0, D_1, D_2, D_3 respectively, and we are to redistribute them to disks D_0, D_2, D_4, D_6 in the scaled system. To minimize the blocks being migrated during the scaling process, we migrate $C_{2,1}$, $C_{2,3}$ from D_1, D_3 to D_4, D_6 respectively. From S_p and T_2^a, we can get $S_{from} = \{1, 3\}$ and $S_{to} = \{4, 6\}$, which means that the blocks to be migrated are stored in disks D_1, D_3, and they are to be migrated to disks D_4, D_6.

Step 4 (Auxiliary Full Block Design Table Construction for Increasing Stripe Size): If we are not to add any new data or parity blocks (i.e., $x_{inc} = 0$ and $y_{inc} = 0$) into the original stripes, PDS just skips this step. Otherwise, suppose that we are to add x_{inc} data blocks and y_{inc} parity blocks to each of the original stripes with $x_{inc} + y_{inc} > 0$ and $x_{inc} + y_{inc} \leq (v_a - v_p)k_p/v_p$.

We need to design an auxiliary full block design table TBL_{inc} for PDS to define the allocation of new blocks in the original stripes. Let base set $S' = \{0, 1, \ldots, v_a - v_p - 1\}$, where $v_a - v_p$ is the number of new disks added to the system, and it is also the number of disks containing free storage units in a row after we have migrated the data/parity blocks from old disks to new disks, such as $v_a - v_p = 3$ in Fig. 12d. We use $j \in S'$ to represent the j-th disk containing free storage units in the ascending order of the disk subscripts in a row for $0 \leq j \leq v_a - v_p - 1$.

Given $v' = v_a - v_p$ and $k' = x_{inc} + y_{inc}$, we first select a $(b', v', k', r', \lambda')$-BIBD $\mathbb{B}'$ from the BIBD database with base set S'. Then we construct a $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc})$-BIBD, $\mathbb{B}_{inc}$, from $\mathbb{B}'$ as follows and finally construct TBL_{inc} from $\mathbb{B}_{inc}$.

- If $y_{inc} = 0$, we choose $\mathbb{B}_{inc} = \mathbb{B}'$;
- otherwise, let $l = v'/\gcd(b', v')$, we choose $\mathbb{B}_{inc}$ to be l duplications of $\mathbb{B}'$ so as to ensure b_{inc} is a multiple of v_{inc}. Then $\mathbb{B}_{inc}$ is a $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc}) = (lb', v', k', lr', l\lambda')$-BIBD.

If $y_{inc} = 0$, since there is no parity assignment in TLB_{inc}, we use $\mathbb{B}_{inc}$ as TLB_{inc} directly.

Otherwise (i.e., $y_{inc} > 0$), we construct a TBL_{inc} from $\mathbb{B}_{inc} = (lb', v', k', lr', l\lambda')$. There are $v_{inc} = v' = v_a - v_p$ elements in TBL_{inc} and there are b_{inc} rows in TBL_{inc}. We select $l = v'/\gcd(b', v')$ such that lb' is a multiple of v_{inc}. In case of b_{inc} being a multiple of v_{inc}, Schwable and Sutherland [22] proposed an algorithm to label y_{inc} distinct objects with each of these y_{inc} objects holding a distinct label P_i $(0 \leq i \leq y_{inc} - 1)$, such that elements which correspond to objects labelled with P_i $(0 \leq i \leq y_{inc} - 1)$ are balanced perfectly among distinct objects of TBL_{inc}. Take Fig. 14c as an example.

Step 5 (New Block Allocation for Increasing Stripe Size): If we do not add any new data/parity blocks (i.e., $x_{inc} = 0$ and $y_{inc} = 0$) into the original stripes, PDS just skips this step. Otherwise, suppose that we need to add x_{inc} new data blocks $d'_{i,0}, d'_{i,1}, \cdots, d'_{i,x_{inc}-1}$ and y_{inc} new parity blocks $p'_{i,0}, p'_{i,1}, \cdots, p'_{i,y_{inc}-1}$ to the original stripe S_i.

Denote $\mathbb{B}_{inc} = \{T_0^{inc}, T_1^{inc}, \cdots, T_{b_{inc}-1}^{inc}\}$, PDS allocates new blocks in unit of PIEs. We allocate the new data/parity blocks into the original stripes in the x-th scaled PME of a PIE according to T_x^{inc}. For the x-th scaled PME, we define *increasing stripe size block allocation group* (abbr. as *IBA-group*) $G_{i,j}^{d_k}$ and $G_{i,j}^{p_k}$ as follows, where subscript i means the i-th PPD area in a PIE and j means the j-th disk containing free storage units in a row, and d_k and p_k mean the k-th new data block and the k-th new parity block to be added to an original stripe, respectively.

- We first denote the set of the unlabelled objects in T_x^{inc} as S_{data}. Suppose that j is the k-th smallest number in S_{data}, then we denote the b_p new allocated data blocks $d'_{b_p \times i+l,k}$ $(0 \leq l \leq b_p - 1)$ as $G_{i,j}^{d_k}$, where $b_a \times x \leq i \leq b_a(x+1) - 1$.
- Suppose that j is the object labelled with P_k in T_x^{inc}, then we denote the b_p new allocated parity blocks $p'_{b_p \times i+l,k}$ $(0 \leq l \leq b_p - 1)$ as $G_{i,j}^{p_k}$, where $b_a \times x \leq i \leq b_a(x+1) - 1$.

The IBA-groups $G_{i,j}^{d_k}$ and $G_{i,j}^{p_k}$ are added into b_p original stripes which belong to the $(i \bmod b_a)$-th scaled PPD area of the $\lfloor i/b_a \rfloor$-th scaled PME. We allocate $G_{i,j}^{d_k}$ and $G_{i,j}^{p_k}$ in disk D_l, where l is defined as follows. The data migration for the scaling process is defined by the auxiliary BIBD $\mathbb{B}_a = \{T_0^a, T_1^a, \cdots, T_{b_a-1}^a\}$. We use T_y^a to migrate data/parity blocks in the y-th PPD area of a PME. So we know from T_y^a that, in the y-th PPD area, the storage units in which disks are free. We first define the complement of tuple T_y^a in base set $S_a = \{0, 1, \cdots, v_a - 1\}$ as $T_y^{\complement} = \{z \mid z \in S_a \text{ and } z \notin T_y^a\}$, where each element in $T_y^{\complement}$ is the subscript of a disk in which the storage units of the y-th PPD area is free. Then l is the j-th smallest number in $T_{i \bmod b_a}^{\complement}$. Take Fig. 14a as an example.

Step 6 (Remapping): In the scaled system, there are three types of storage units. The first type corresponds to the storage units storing data and parity blocks in the RAID before scaling, such as all storage units occupied by $C_{i,j}$ in Fig. 12d. The second type corresponds to the storage units storing the new allocated data and/or parity blocks for increasing stripe size, such as all storage units occupied by IBA-groups $G_{i,j}^{d_0}$ and $G_{i,j}^{p_0}$ in Fig. 14a. Note that if we do not increase the stripe size of the scaled system, the second type of storage units does not exist. The third type corresponds to the storage units which are freed in old disks or are not used in new disks, such as all free storage units in Fig. 12d when we do not increase stripe size or in Fig. 14a when we increase stripe size. The addressing algorithm for the three types of storage units will be presented in the next section (i.e., Section VI) in detail.

Step 7 (Parity Update): Erasure codes guarantee data reliability by maintaining the parity information in a stripe. When we do not increase stripe size of the scaled system, because we do not change content of any data/parity block in an original stripe during the scaling process, the data migration does not induce update of parity blocks in the first type of storage units. However, when we increase the stripe size of the scaled system, we should update the parity blocks to keep the consistency due to new blocks added into the original stripe, To maintain parity consistency of the original stripes with the increased size, we need to read all data blocks to compute the parity blocks of the stripes by using the $(x_p+x_{inc}, y_p + y_{inc})$-RS code, then write these parity blocks into their corresponding storage units.

We can save disk I/O load for updating parity blocks of the stripes in the first and the second types of storage units and cleaning up the freed storage units of the third type in the old disks in the following two ways respectively.

- Since the new allocated data blocks (when $x_{inc} \neq 0$) and the migrated data blocks have been read into memory in the data migration, we can read these data blocks from memory instead of reading these blocks from disks to compute parity blocks to save I/O load.
- The freed storage units of the third type in old disks are still containing content of migrated blocks after data migration. If we clean up these storage units by writing 0 to maintain parity consistency, it will induce heavy disk I/O load. In order to save I/O load, we can use a piggyback parity update scheme, which will be introduced in Section VIII-A in detail.

VI. THE ADDRESSING ALGORITHM

After scaling, there are three types of storage units in the system. The volume of the first type equals to the size of all old disks, and the volume of the second type equals to the size of all new allocated blocks for increasing stripe size. Note that if we do not increase stripe size in the scaled system, the volume of the second type is 0. Furthermore, the volume of the second type and the third type in total equals to the size of all new added disks.

In the scaled system, for a block x with logical series number $n_{ls}(x)$,

- if $n_{ls}(x)$ is less than the size of all old disks, x is an original block which exists before scaling and it is stored in the first type of storage units;
- if $n_{ls}(x)$ is between the size of all old disks and the size of all blocks either in the original stripes that exist before scaling or in the new allocated blocks for increasing stripe size, x is a new allocated block for increasing stripe size and it is stored in the second type of storage units;
- otherwise, x is a new block and it is stored in the third type of storage units.

Blocks stored in three different types of storage units are located with the addressing algorithm in three different ways.

A. ADDRESSING THE FIRST TYPE OF STORAGE UNITS

If x is stored in a storage unit of the first type, PDS calculates its physical address (d, f) in the old system, which means x is stored in disk D_d with offset f before scaling. Since each PME has b_a PPD areas and each PPD-column contains r_p blocks, we know that x is stored in the $l = (\lfloor f/r_p \rfloor \bmod b_a)$-th PPD area of the $\lfloor f/(r_p \times b_a) \rfloor$-th PME. Therefore x is stored in $C_{l,d}$ in the $\lfloor f/(r_p \times b_a) \rfloor$-th PME before scaling.

Tuple T_l^a of the auxiliary BIBD $\mathbb{B}_a$ and disk number d decide into which disk x is migrated. If $d \in T_l^a$, PDS does not migrate $C_{l,d}$ during scaling, x is still stored in disk D_d with offset f after scaling. Otherwise, PDS migrates $C_{l,d}$ to disk $D_{d'}$, where d' is calculated according to the migration algorithm as follows. First, calculating $S_{from} = \{0, 1, \cdots, v_p - 1\} - T_l^a$ and $S_{to} = T_l^a - \{0, 1, \cdots, v_p - 1\}$. Second, suppose that d is the i-th smallest number in S_{from}, then d' is the i-th smallest number in S_{to}. Note that PDS does not change the offset of x during scaling, so x is stored in disk $D_{d'}$ with the same offset f after scaling.

B. ADDRESSING THE SECOND TYPE OF STORAGE UNITS

Suppose that x is stored in a storage unit of the second type. Let the size of all old disks be s_{old}, i.e., the old system can store s_{old} blocks. Suppose that when the size of all original stripes increases from k_p to $k_p + x_{inc} + y_{inc}$, all original stripes with increased size contain s_{inc} storage units, i.e., the scaled system can store s_{inc} blocks. Then we have that $s_{old} \leq n_{ls}(x) < s_{inc}$ and $n'_{ls}(x) = (n_{ls}(x) - s_{old})$ is the logical series number of x in the second type of storage units. Note that if we do not increase stripe size in the scaled system, the second type of storage units do not exist and then $s_{old} = s_{inc}$.

We know that the stripe size is increased by $k_{inc} = x_{inc} + y_{inc}$, i.e., there are k_{inc} new allocated blocks added to each original stripe. So block x is added to the $s_{or}(x) = \lfloor n'_{ls}(x)/k_{inc} \rfloor$-th original stripe as the $b_{na}(x) = (n'_{ls}(x) \bmod k_{inc})$-th new block added into the stripe for increasing its size. As there are b_a PPD areas in a PME and each PPD area contains b_p original stripes, so there are $N_{stripe} = b_a \times b_p$ original stripes in a PME. We use a tuple in TBL_{inc} to allocate new blocks for all the N_{stripe} original stripes in a PME.

When block x is coming into the system and we are to allocate x, in TBL_{inc}, the number of tuples that has been used to allocate new blocks for all the original stripes in a PME is $N_{tuple} = \lfloor s_{or}(x)/N_{stripe} \rfloor = \lfloor \lfloor n'_{ls}(x)/k_{inc} \rfloor / N_{stripe} \rfloor = \lfloor n'_{ls}(x)/(k_{inc} \times N_{stripe}) \rfloor$. Furthermore, in the current tuple which is now used to allocated block x, there are $b_{na}(x)$ objects has been used. As each tuple in TBL_{inc} contains k_{inc} objects, then when we allocate block x, in TBL_{inc}, there are $N_{object} = N_{tuple} \times k_{inc} + b_{na}(x)$ objects that has been used. Then we use series number

$$\begin{aligned} n^*_{ls}(x) &= N_{object} \\ &= N_{tuple} \times k_{inc} + b_{na}(x) \\ &= \lfloor n'_{ls}(x)/(N_{stripe} \times k_{inc}) \rfloor \times k_{inc} \\ &\quad + (n'_{ls}(x) \bmod k_{inc}) \end{aligned}$$

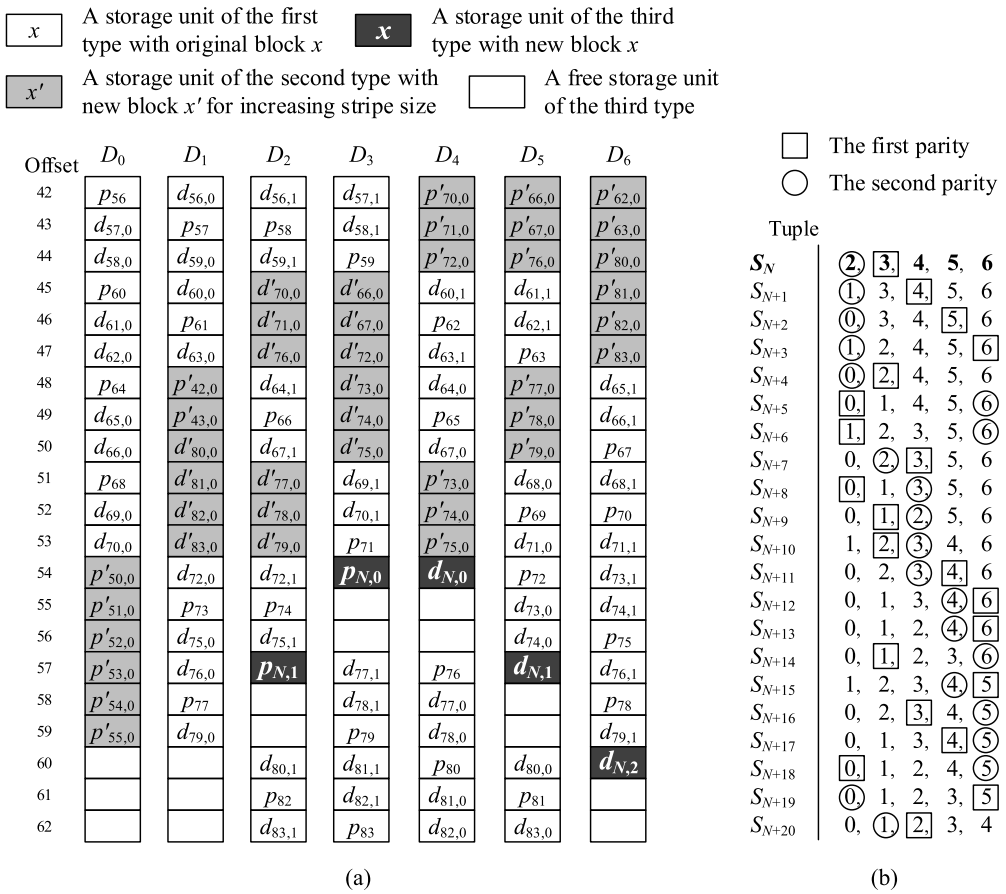


FIGURE 15. The placement of new data blocks $d_{N,0}$, $d_{N,1}$, $d_{N,2}$ and their parity blocks $p_{N,0}$, $p_{N,1}$ in the storage units of the third type. (a) Physical layout of the third PME with new data blocks. (b) Full block design table for new data.

to calculate logical address (d, f) according to the parity declustered data layout constructed from TBL_{inc}. Note that we use a tuple of TBL_{inc} to allocated N_{stripe} stripes instead of allocating just one stripe, i.e., we have that if $N_{stripe} = 1$, then $n^*_{ls}(x) = \lfloor n'_{ls}(x)/k_{inc} \rfloor \times k_{inc} + (n'_{ls}(x) \bmod k_{inc}) = n'_{ls}(x)$.

Since a PPD area contains b_p original stripes and k_{inc} new blocks are added to each original stripe, there are $b_p \times k_{inc}$ new blocks added into a PPD area. So block x is allocated into a stripe in the $i = \lfloor n'_{ls}(x)/(b_p \times k_{inc}) \rfloor$-th PPD area. We know that the i-th PPD area is migrated according to the (i mod b_a)-th tuple of $\mathbb{B}_a$ (i.e., $T^a_{i \bmod b_a}$). Thus, we allocate block x in disk $D_{d'}$, where d' is the d-th smallest number in $T^{\mathbf{C}}_{i \bmod b_a}$.

Since a PIE contains b_{inc} PMEs and there are b_a PPD areas in a PME, there are $b_{inc} \times b_a$ PPD areas in a PIE. So block x is allocated into a stripe in the $j = \lfloor i/(b_{inc} \times b_a) \rfloor$-th PIE. We allocate block x in the free storage unit with the lowest available offset on disk $D_{d'}$ in the j-th PIE. Since each PIE has the same data layout, the lowest available offset can be easily computed. Let f' be the the lowest available offset on disk $D_{d'}$ in the j-th PIE, As each disk contains $N_{total} = r_p \times b_a \times b_{inc}$ storage units in a PIE, then block x is stored in disk $D_{d'}$ with offset f'', where $f'' = j \times N_{total} + f'$.

C. ADDRESSING THE THIRD TYPE OF STORAGE UNITS

Suppose that x is stored in a storage unit of the third type. Then $n_{ls}(x) \geq s_{inc}$ and $n''_{ls}(x) = (n_{ls}(x) - s_{inc})$ is the logical series number of x in the third type of storage units. To make the data layout in the storage units of the third type be parity declustering, we should select a $(b'', v'', k'', r'', \lambda'')$-BIBD $\mathbb{B}''$ to define the data layout. In the scaled system, we should deploy the same erasure code for both of the stripes in the storage units of the third type and the stripes in the storage units of the first type and the second type. Therefore we set $k'' = k_p + x_{inc} + y_{inc}$ to ensure the same stripe size in the scaled system. We should set $v'' = v_o$ due to the number of disks in the scaled system. Then we find a $\mathbb{B}''$ in the BIBD database described in step 1 (i.e., Auxiliary BIBD Selection for Data Migration) of the scaling process of PDS in Section V-B. With $\mathbb{B}''$, we can build a new full block design table (denoted as TBL_{new}) to define the data layout in the storage units of the third type. Suppose TBL_{new} corresponds to a $(b_{new}, v_{new}, k_{new}, r_{new}, \lambda_{new})$-BIBD $\mathbb{B}_{new}$, we have $v_{new} = v_o$ and $k_{new} = k_p + x_{inc} + y_{inc}$.

For x's series number $n''_{ls}(x)$, we first calculate its logical address (d, f) according to the parity declustered data layout constructed from TBL_{new}, which means x should be stored in

disk D_d with offset f according to TBL_{new}. But note that here, the offset f is limited in the storage units of the third type, not the total storage space of the scaled RAID. So we should further map the offset f in the storage units of the third type to the physical address of x, i.e., the physical offset f' in the scaled system. Then we store block x into the corresponding storage unit in the scaled system. For example, let N be the number of stripes before scaling, Fig. 15 shows the placement of new data blocks $d_{N,0}$, $d_{N,1}$, $d_{N,2}$ and their parity blocks $p_{N,0}$, $p_{N,1}$ in the storage units of the third type.

From the proof of Theorem 3 in the next section, we can know that each disk contains $N_{third} = (b_a - r_a)(r_p \times b_{inc} - b_p \times r_{inc})$ storage units of the third type in a PIE in the scaled system. Therefore we store block x in the $l = \lfloor f/N_{third} \rfloor$-th PIE on D_d. In particular, we store x in the f''-th storage unit in the l-th PIE on D_d, which is the ($f \mod N_{third}$)-th storage unit of the third type in the l-th PIE on D_d. As there are $N_{total} = r_p \times b_a \times b_{inc}$ storage units in each PIE of a disk, we have $f' = N_{total} \times l + f''$.

If we do not increase the stripe size in the scaling process, we can just set $b_{inc} = 1$ such that a PIE just consists of $b_{inc} = 1$ PME and set $r_{inc} = 0$ in the calculation of f'.

This addressing algorithm is simply and can be easily implemented. A block address can be calculated quickly with low space overhead. Furthermore, several microseconds of the addressing calculation time is negligible compared to milliseconds of disk I/O time.

VII. THEORETICAL PROOFS OF PDS

We presented the addressing algorithm of PDS, which shows *fast data addressing* (i.e., Objective 5 as stated in Section IV-A) in the scaled system in the last section. In this section, we formally prove that PDS achieves the other objectives, i.e., Objectives 1, 3, 4 as stated in Section IV-A. In the following, we assume that the primary BIBD, the auxiliary BIBD for data migration, and the auxiliary BIBD for increasing stripe size are $(b_p, v_p, k_p, r_p, \lambda_p)$-BIBD $\mathbb{B}_p$, $(b_a, v_a, k_a, r_a, \lambda_a)$-BIBD $\mathbb{B}_a$, and $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc})$-BIBD $\mathbb{B}_{inc}$ respectively. We first prove that the data layout in the scaled system after scaling can be defined by some final $(b_f, v_f, k_f, r_f, \lambda_f)$-BIBD $\mathbb{B}_f$ with $v_f = v_a$ and $k_f = k_p + k_{inc}$ (i.e., Objective 1), then prove that data blocks, and also parity blocks, are evenly distributed among disks (i.e., Objective 3), and finally prove the number of migrated blocks is minimal (i.e., Objective 4).

A. DATA LAYOUT WITH PARITY DECLUSTERING

In this section, we show that PDS achieves Objective 1 as stated in Section IV-A, i.e., the data layout after scaling is parity declustering. We know that there are three types of storage units in the scaled system. We will first show that the data layout in the storage units of the first type can be defined by a BIBD, then show that the data layout in the storage units of the first type and the second type can be defined by another BIBD, and finally show that the data layout in the scaled system (i.e., all storage units of the three types) can be defined by some final BIBD.

Theorem 1: The data layout in the storage units of first type is still parity declustered after data migration.

Proof: We first come to the first type of storage units. In each PME, we define b_o tuples $T^o_0, T^o_1, \cdots, T^o_{b_o-1}$ with base set $S = \{0, 1, \cdots, v_o - 1\}$ as follows. $T^o_i = \{j \mid$ *Stripe* S_i *has a block stored in disk* $D_j\}$.

Now we prove $\mathbb{B}_o = \{T^o_0, T^o_1, \cdots, T^o_{b_o-1}\}$ is a $(b_o, v_o, k_o, r_o, \lambda_o) = (b_p \times b_a, v_a, k_p, r_p \times r_a, \lambda_a \times \lambda_p)$-BIBD. Note that in the following of this proof, the statement is limited in a PME.

- b_o: After data migration, there are $b_p \times b_a$ stripes, which means there are $b_o = b_p \times b_a$ tuples in $\mathbb{B}_o$.
- v_o: There are v_a disks in the array, which means there are $v_o = v_a$ objects in base set S.
- k_o: Since stripe size is unchanged during data migration, each tuple containing $k_o = k_p$ objects in S.
- r_o: After data migration, there are $r_p \times r_a$ blocks on each disk, which means each object appears in $r_o = r_p \times r_a$ tuples of $\mathbb{B}_o$.
- λ_o: From $\mathbb{B}_a$, we know that for each pair of disks, there are exactly λ_a pairs of PPD-columns in the same PPD areas. From $\mathbb{B}_p$, we further know that each pair of PPD-columns in the same PPD area has λ_p pairs of related blocks.[3] So each pair of disks has $\lambda_a \times \lambda_p$ pairs of related blocks. Thus, each pair of objects appears in exactly $\lambda_o = \lambda_a \times \lambda_p$ tuples in $\mathbb{B}_o$.

According to Definition 1, $\mathbb{B}_o = \{T^o_0, T^o_1, \cdots, T^o_{b_o-1}\}$ is a $(b_p \times b_a, v_a, k_p, r_p \times r_a, \lambda_a \times \lambda_p)$-BIBD. So the data layout in the storage units of the first type in a PME is defined by an objective $(b_o, v_o, k_o, r_o, \lambda_o)$-BIBD $\mathbb{B}_o$. ■

Theorem 2: The data layout in the storage units of the first and the second types is parity declustered after the stripe size being increased.

The complete proof of this theorem is given in Appendix A. In the proof, for each PIE, we define b^*_o tuples $T^*_0, T^*_1, \cdots, T^*_{b^*_o-1}$ with base set $S = \{0, 1, \cdots, v^*_o - 1\}$ as $T^*_i = \{j \mid$ *Stripe* S_i *has a block stored in disk* $D_j\}$, then we prove that $\mathbb{B}^*_o = \{T^*_0, T^*_1, \cdots, T^*_{b^*_o-1}\}$ is a $(b^*_o, v^*_o, k^*_o, r^*_o, \lambda^*_o) = (b_p \times b_a \times b_{inc}, v_a, k_p + k_{inc}, r_a \times b_{inc} \times r_p + (b_a - r_a) r_{inc} \times b_p, \lambda_a \times b_{inc} \times \lambda_p + 2(r_a - \lambda_a) r_{inc} \times r_p + (b_a - 2r_a + \lambda_a)\lambda_{inc} \times b_p)$-BIBD, which implies that the data layout in the storage units of the first and the second types in a PIE is defined by an objective $(b^*_o, v^*_o, k^*_o, r^*_o, \lambda^*_o)$-BIBD $\mathbb{B}^*_o$.

Note that if we do not increase stripe size (i.e., $k_{inc} = 0$), we set $(b_{inc}, v_{inc}, k_{inc}, r_{inc}, \lambda_{inc}) = (1, v_a - v_p, 0, 0, 0)$ so as to make $(b^*_o, v^*_o, k^*_o, r^*_o, \lambda^*_o) = (b_o, v_o, k_o, r_o, \lambda_o)$. Furthermore, a PIE is just $b_{inc} = 1$ PME in this case.

Theorem 3: PDS keeps parity declustered data layout in the scaled system.

Proof: Now we come to the total data layout in the scaled system. The data layout in the storage units of

[3] A pair of related blocks means that the two blocks belong to the same stripe, such that one block is read for the recovery of another.

the third type is defined by TBL_{new} corresponding to a $(b_{new}, v_{new}, k_{new}, r_{new}, \lambda_{new})$-BIBD $\mathbb{B}_{new}$, where $v_{new} = v_o$ and $k_{new} = k_p + x_{inc} + y_{inc}$. In the storage units of the third type corresponding to all entries of a TBL_{new}, there are r_{new} blocks stored in each disk. Furthermore, in a PIE, there are r_o^* storage units of the first and the second types in each disk, while there are total $N_{total} = b_{inc} \times b_a \times r_p$ storage units in each disk. So in a PIE, there are $N_{third} = N_{total} - r_o^* = (b_a - r_a)(r_p \times b_{inc} - b_p \times r_{inc})$ storage units of the third type in each disk, which is usually not equal to r_{new}. So we should combine the storage units of the third type in $N_{pie} = r_{new}/\gcd(N_{third}, r_{new})$ PIEs as a group, and divide a group into $N_{new} = N_{third}/\gcd(N_{third}, r_{new})$ parts, then the data layout in each part can be defined by TBL_{new}.

Therefore, the data layout in N_{pie} PIEs is defined by N_{pie} copies of $\mathbb{B}_o^*$ and N_{new} copies of $\mathbb{B}_{new}$. As $v_{new} = v_o^*$ and $k_{new} = k_o^*$, the concatenation of the N_{pie} copies of $\mathbb{B}_o^*$ and N_{new} copies of $\mathbb{B}_{new}$ is still a BIBD $\mathbb{B}_f$ [23]. So the data layout in N_{pie} PIEs can be defined by $\mathbb{B}_f$. Thus, the data layout in the scaled system is still parity declustering. ■

In the example shown in Section IV-B, we can prove that $\mathbb{B}_o$ is a $(28, 7, 3, 12, 4)$-BIBD and $\mathbb{B}_o^*$ is a $(84, 7, 5, 60, 40)$-BIBD. From Fig. 15, we can see that $\mathbb{B}_{new}$ is a $(21, 7, 5, 15, 10)$-BIBD and each disk contains $N_{third} = 3$ storage units of the third type in a PIE. So the data layout in five PIEs is defined by five copies of $\mathbb{B}_o^*$ and one $\mathbb{B}_{new}$. Because their concatenation is a BIBD, the data layout after scaling is still parity declustered.

B. UNIFORM DATA AND PARITY DISTRIBUTIONS

In this section, we show that PDS achieves Objective 3 as stated in Section IV-A, i.e., data blocks, and also parity blocks, are evenly distributed among all disks in the scaled system.

Theorem 4: PDS maintains uniform data and parity distributions after scaling.

The complete proof of this theorem is given in Appendix B, where we prove that, for the storage units of each type, data blocks and parity blocks are distributed evenly among all disks separately.

In the example shown in Section IV-B, from Fig. 12d we can see that, after data migration, each disk, either old or new, has $r_a = 4$ PPD-columns in a PME. Each PPD-column contains $r_p = 3$ blocks including one parity block. So each disk contains $r_p \times r_a = 12$ blocks in the storage units of the first type in a PME, including four parity blocks. So data blocks and parity blocks in the storage units of the first type are evenly distributed among disks after scaling.

From Fig. 14a, we can see that, after the stripe size being increased, each disk, either old or new, has three IBA-groups with data blocks and three IBA-groups with parity blocks in a PIE. As each IBA-group contains four blocks, data blocks and parity blocks in the storage units of the second type are evenly distributed among disks after scaling.

From Fig. 15, we can see that $\mathbb{B}_{new}$ is a $(21, 7, 5, 15, 10)$-BIBD and by using all entries of TBL_{new} to place blocks, there will be $r_{new} = 15$ blocks including six parity blocks stored in each disk. So data blocks and parity blocks in the storage units of the third type are evenly distributed among all disks after scaling.

C. MINIMAL DATA MIGRATION

In this section, we show that PDS achieves Objective 4 as stated in Section IV-A, i.e., the number of migrated blocks is minimal.

Theorem 5: PDS performs the minimal data migration during scaling.

Proof: There are v_p disks in the old system and v_a disks in the scaled system as $v_a - v_p$ new disks are added in the system. In each PME of the old system, there are $r_p \times b_a$ blocks in each disk and totally $S = r_p \times b_a \times v_p$ blocks in the array. Therefore to reach even distribution of data/parity blocks, the minimum number of blocks to be migrated from old disks to new disks is $S \times (v_a - v_p)/v_a = (v_a - v_p)r_p \times b_a \times v_p/v_a$. As the auxiliary BIBD $\mathbb{B}_a$ satisfies $k_a = v_p$, the minimum number of migrated blocks is $(v_a - v_p)r_p \times b_a \times k_a/v_a = (v_a - v_p)r_p \times r_a$, according to (1).

In a PME, each disk in the scaled system, either new or old, contains $r_p \times r_a$ migrated blocks. PDS totally moves $r_p \times r_a \times (v_a - v_p)$ blocks into new disks, and does not migrate blocks among old disks. Therefore the number of migrated blocks is $r_p \times r_a \times (v_a - v_p)$, which reaches the minimum. ■

In the example shown in Section IV-B, from Fig. 12b we can see that, there are totally $r_p \times b_a \times v_p = 84$ blocks in a PME, and three disks are added to an array with four disks. Thus, the minimal number of migrated blocks is $84 \times 3/(3 + 4) = 36$. From Fig. 12d we can see that PDS just migrates 36 blocks, which reaches the minimum.

VIII. FURTHER DISCUSSIONS

A. PIGGYBACK PARITY UPDATES

Data reliability is ensured by erasure codes by maintaining the parity information in a stripe. When migrating a block, PDS copies the block to its new storage unit while still keeping its content in the freed storage unit, without need of erasing the old block by writing 0 to the freed storage unit. This does not maintain parity consistence of stripes in the third type of storage units, and therefore requires parity updates. PDS uses a piggyback scheme to minimize disk I/Os for updating parity blocks of the stripes in the third type of storage units. Before performing the scaling process, new disks are cleaned up by writing 0. This "cleaning up" operation does not take up the scaling time. Since migrated blocks will be read into memory during scaling, PDS encodes blocks of the same stripe in the freed storage units of the third type when they reside in memory, then writes the updated parity blocks into disks, which requires less disk I/Os.

Fig. 16 shows the advantage of the piggyback parity update scheme. After data migration, suppose that a stripe consists of three storage units of the third type U_0, U_1 in old disks and U_2 in a new disk storing A, B, and 0 respectively, storage unit

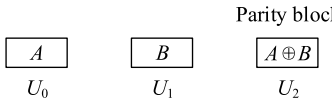


FIGURE 16. Piggyback parity update to maintain parity consistency.

U_2 belongs to the parity block. One way to maintain parity consistence of the stripe is to make all of U_0, U_1 and U_2 store 0, which needs to write 0 to U_0 and U_1 with two disk I/Os. However, A and B will be read into memory during scaling because they need to be migrated from U_0 and U_1, we can encode them as $A \oplus B$ when they are in memory, then write the parity block to U_2, which requires only one disk I/O.

B. SUPPORT TO RS CODES

By now we propose PDS with RAID-5, which can just tolerate a single disk failure, as an example. To achieve higher reliability than RAID-5, we can deploy RS codes with parity declustered data layout. For an (x, y)-RS code, a stripe contains x data blocks and y parity blocks. Moreover, an (x, y)-RS code can tolerate y concurrent disk failures. We can deploy an (x, y)-RS code to an array with m disks as follows. First, we select a (b, v, k, r, λ)-BIBD, $\mathbb{B}$, with $v = m$ and $k = x+y$. Second, we use the BIBD with duplicating $v/\gcd(b, v)$ times of $\mathbb{B}$ to construct the full block design table according to the "parity assignment graph" in the work [22] so as to distribute parity blocks evenly. Then the data layout with the RS code can be defined by the full block design table. PDS succeeds in scaling RS codes with parity declustering because it scales up an array with parity declustered data layout according to the full block design table.

C. SUCCESSIVE SCALING OPERATIONS

As time goes on, a scaled system may need scaling up again. Now we explain how to apply PDS to a scaled system. After the first scaling operation, the data layout in a number of PIEs can be defined by some BIBD (see the proof of Theorem 3), then we can redefine the PPD-column in the scaled system to perform the next scaling operation. Suppose that the system has already been performed $t-1$ scaling operations, now we are going to perform the t-th scaling operation.

- Suppose that TBL_i is the full block design table in the i-th scaled system for $0 \le i \le t$, and it corresponds to a $(b_i, v_i, k_i, r_i, \lambda_i)$-BIBD, which should satisfy that $k_i = k_{i-1} + k_i^{inc}$ for increasing the stripe size by k_i^{inc}.
- Suppose that $\mathbb{B}_i^a$ is the auxiliary BIBD that defines the data migration of the i-th scaling for $1 \le i \le t$, and it is a $(b_i^a, v_i^a, k_i^a, r_i^a, \lambda_i^a)$-BIBD, which should satisfy that $v_i^a = v_i$ and $k_i^a = v_{i-1}$.
- Suppose that $\mathbb{B}_i^{inc}$ is the auxiliary BIBD that defines the block allocation to increase stripe size in the i-th scaling process for $1 \le i \le t$, and $\mathbb{B}_i^{inc}$ is a $(b_i^{inc}, v_i^{inc}, k_i^{inc}, r_i^{inc}, \lambda_i^{inc})$-BIBD, which should satisfy that $v_i^{inc} = v_i - v_{i-1}$. Note that if we do not increase stripe size in the i-th scaling process (i.e., $k_i^{inc} = 0$), we just set $b_i^{inc} = 1$ and $r_i^{inc} = 0$ in the equations from (3) to (7).

During the i-th scaling process, $v_i - v_{i-1}$ new disks are added to the array with v_{i-1} old disks, the number of blocks in one PPD-column is len_i (defined by (3)), i.e., one PPD area contains len_i blocks on each disk when performing the i-th scaling process. Moreover, the number of stripes in one PPD area is N_i^{stripe} (defined by (4)). Because there are b_i^a PPD areas in a PME and there are N_i^{stripe} stripes in a PPD area, so there are totally $b_i^a \times N_i^{stripe}$ stripes in a PME. If we are to add k_i^{inc} new data/parity blocks to each of the original stripes, we need $k_i^{inc} \times b_i^a \times N_i^{stripe}$ free storage units in a PME. Because there are $len_i \times b_i^a$ storage units in each disk of a PME, the scaling process adds $(v_i - v_{i-1}) \times len_i \times b_i^a$ free storage units to each PME. To accommodate the new blocks added to free storage units, we have that

$$\begin{aligned} k_i^{inc} &\le (v_i - v_{i-1}) \times len_i \times b_i^a / (b_i^a \times N_i^{stripe}) \\ &= (v_i - v_{i-1}) \times len_i / N_i^{stripe}. \quad \text{(eliminating } b_i^a\text{)} \end{aligned}$$

After the i-th scaling, the number of storage units of the third type on each disk in one PIE is N_i^{third} (defined by (5)). Since the data layout of the third type of storage units is defined by TBL_i after the i-th scaling, by using all entries of TBL_i to place blocks N_i^{new} (defined by (6)) times, the new placed blocks will occupy the storage units of the third type in exactly N_i^{pie} (defined by (7)) PIEs. Therefore, the data layout of N_i^{pie} PIEs can be defined by a BIBD. So PDS succeeds in successive scaling operations with parity declustered data layout preserved. As a summary, we have

$$len_i = \begin{cases} r_0, & \text{if } i = 1; \\ len_{i-1} \cdot b_{i-1}^a \cdot b_{i-1}^{inc} \cdot N_{i-1}^{pie}, & \text{if } i > 1. \end{cases} \tag{3}$$

$$N_i^{stripe} = \begin{cases} b_0, & \text{if } i = 1; \\ N_{i-1}^{stripe} \cdot b_{i-1}^a \cdot b_{i-1}^{inc} \cdot N_{i-1}^{pie} + b_{i-1} \cdot N_{i-1}^{new}, & \text{if } i > 1. \end{cases} \tag{4}$$

$$N_i^{third} = (len_i \cdot b_i^{inc} - N_i^{stripe} \cdot r_i^{inc}) \cdot (b_i^a - r_i^a), \quad \text{for } i \ge 1. \tag{5}$$

$$N_i^{new} = \frac{N_i^{third}}{\gcd(N_i^{third}, r_i)}, \quad \text{for } i \ge 1. \tag{6}$$

$$N_i^{pie} = \frac{r_i}{\gcd(N_i^{third}, r_i)}, \quad \text{for } i \ge 1. \tag{7}$$

IX. PERFORMANCE EVALUATION

We evaluate PDS and compare with the existing "moving-everything" solution, i.e., the round-robin scheme, in different practical aspects, including scaling time, user response time during scaling, and user response time after scaling.

A. EVALUATION METHODOLOGY

We implement parity declustered data layout, PDS and round-robin scaling schemes in the MD (Multiple Devices) driver shipped with Linux kernel 3.14.72. The MD driver is a software RAID system that forms a common framework for all RAID systems, including RAID-5 and Linux RAID-6 [4].

In Linux RAID-6, a stripe contains two parity blocks such that the system could tolerate the failure of any two disks. The MD driver provides a reshape toolkit named MD-Reshape [5] to support online capacity expansion.

We conduct our evaluation on a storage server with an Intel Xeon E5-2609 2.40 GHz quad-core processor and 8 GB memory, running Ubuntu 14.04 with Linux kernel 3.14.72. Via a 6 GB/s SATA expansion card, 12 disks are connected to this server. Table 2 gives the key parameters of the disks. We set the block size as 256 KB throughout the evaluation. In all experiments, 10 GB capacity of each disk is used because using the whole volume of each disk will take quite a long time.

TABLE 2. Disk parameters.

Parameter	Value
Vender	SEAGATE
Disk Model	ST500DM002
Interface	SATA
Disk Capacity	500 GB
Rotation Speed	7200 RPMS
RAID Block Size	256 KB
Volume Capacity	10 GB

We evaluate our design by running trace-driven experiments over a real system. To replay I/O traces and collect block-level I/O information, we use the `blktrace` tool [1]. Our experiments use the following three real-system disk I/O traces with different characteristics. Table 3 gives a summary of the trace characteristics.

- Financial1 and Financial2 are from SPC (Storage Performance Council) [3]. They were collected from OLTP (On-Line Transaction Processing) applications running at two large financial institutions. Financial1 is write-dominated, while Financial2 is read-dominated.
- WebSearch2 is also from SPC. It was collected from a system running a web search engine. WebSearch2 is read-dominated and exhibits strong access locality.

B. PERFORMANCE DURING SCALING

We evaluate the scaling time and user I/O latency during scaling under different workloads. As data migration and user applications share and contend for the limited I/O resource of the storage system, migration I/Os and user I/Os interfere with each other. We can achieve different tradeoffs between the scaling time and user I/O response time by adjusting the parameters of `sync_speed_max` and `sync_speed_min` in MD-Reshape.

We conduct a scaling operation of adding n ($n = 1, 2, 3$) disks to an array of seven disks with stripe size five, including one parity per stripe. By default, with the round-robin approach, the parameters of `sync_speed_min` and `sync_speed_max` are set as 2 MB/s and 200 MB/s respectively. To fairly compare the user I/O latency of PDS and round-robin, for both scaling schemes, we should issue the same amount of data migration I/Os in each time slot.

TABLE 3. Trace characteristics.

Trace	Write Ratio	Total Requests	Requests/second
Financial1	0.768	5,334,987	122.0
Financial2	0.177	3,699,194	90.2
WebSearch2	2.16×10^{-4}	4,579,809	297.5

Because the volume of migrated data by PDS is less than that by round-robin, when adding $n = 1, 2, 3$ disks, we set `sync_speed_min` with PDS as 16, 9, 6.66 MB/s (i.e., $2\times(7+n)/n$ MB/s) respectively so as to ensure approximately the same volume of migrated data with round-robin in a time slot. The parameter of `sync_speed_max` with PDS is set as 200 MB/s, the same as round-robin.

Furthermore, we conduct a successive scaling operation by adding two disks each time without increasing stripe size to evaluate the performance during the second scaling process. For the second scaling process of PDS, we set `sync_speed_min` and `sync_speed_max` as 11 MB/s (i.e., $2 \times ((7 + 2) + 2)/2$ MB/s) and 200 MB/s respectively.

1) SCALING TIME

We compare the online scaling time between PDS and round-robin under different workloads, and we also measure the scaling time when the scaling process is performed offline, i.e., no traces are replayed during the scaling process. Fig. 17 shows the scaling time under the three workloads and in the offline case (marked as "offline"). We use (n, x, y) to represent the numbers of new added disks, new data blocks added to an original stripe and new parity blocks added to an original stripe, respectively. And the second scaling operation of adding two disks each time without increasing stripe size is marked as "2nd".

From Fig. 17, we find that the scaling time when running online workloads is much longer than that in the offline case. This is because the RAID also need to serve the user I/O requests in addition to the I/O migration requests, which burdens the storage system and so severely prolongs the scaling time. Note that only 10 GB capacity of each disk is used in the experiments, so the benefit of PDS must be

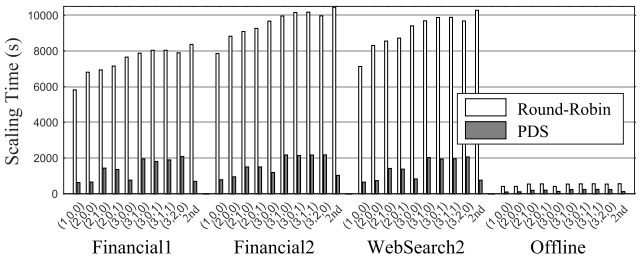


FIGURE 17. Scaling time comparison between PDS and Round-Robin under different workloads and in the offline case both for different (n, x, y) values, where n, x, y represent the numbers of new added disks, new data blocks added to an original stripe and new parity blocks added to an original stripe respectively, and for the second scaling operation of adding two disks each time (marked as "2nd").

larger in practical systems storing large amounts of data. Compared with round-robin, PDS significantly reduces the scaling time by 82.37 percent on average. In particular, for the WebSearch2 workload with $(n, x, y) = (2, 0, 0)$, the scaling time is reduced by 91.22 percent.

For round-robin, all blocks have to be migrated. However, PDS only migrates part of the blocks, for example, only $2/(2 + 7) = 22.22$ percent for adding two disks to an array of seven disks. The significant reduction of migrated data contributes to the most of the reduction of the scaling time. Furthermore, for round-robin, the storage units in old disks, which are invalid but still containing the content of the blocks that have already been migrated, need to be cleaned up so as to maintain parity consistence. This ''cleaning up'' process brings additional writes. While PDS updates parity blocks with piggyback scheme (discussed in Section VIII-A) that greatly reduces the additional writes to maintain parity consistence.

2) USER RESPONSE TIME

Now we compare the user I/O response time during the scaling process. Fig. 18 plots the average latency computed for every 100 seconds when adding two disks without increasing stripe size under the Financial2 workload. We denote the beginning time of running the two algorithms as t_b, and then denote the ending time of scaling with PDS and round-robin as t_{ep} and t_{er}, respectively. It also illustrates that round-robin takes a longer time for scaling than PDS. Note that we can trade off the scaling time for the user I/O latency during scaling, and we issue the same amount of data migration I/Os in each time slot for both scaling schemes so as to fairly compare the user I/O latency of PDS and round-robin.

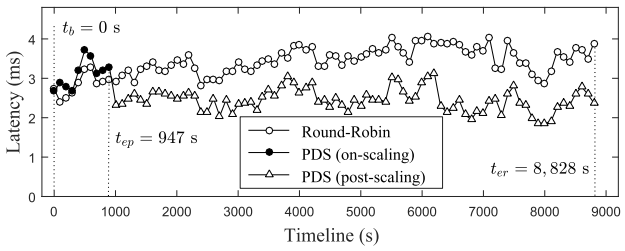


FIGURE 18. Performance comparison between PDS and Round-Robin when adding two disks without increasing stripe size under the Financial2 workload.

Fig. 19 shows the average latency of PDS and round-robin, the latency with round-robin is computed as the average latency of all requests between t_b and t_{er}, while for PDS, we measure the average latency between t_b and t_{ep}, which is denoted as ''PDS (on-scaling)'', as well as the average latency between t_b and t_{er}, which is denoted as ''PDS (on-scaling + post-scaling)''. The results demonstrate that PDS can significantly reduce the user I/O latency compared with round-robin. On average, the latency of PDS (on-scaling + post-scaling) is only 81.75 percent of which with round-robin. This is because PDS has a much shorter scaling time

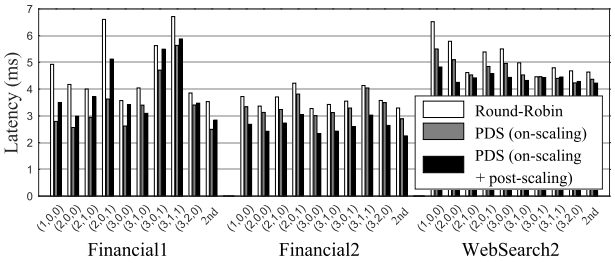


FIGURE 19. Average latency comparison between PDS and Round-Robin under different workloads for different (n, x, y) values and the second scaling operation of adding two disks each time during the scaling process.

than round-robin, and then it has a smaller influence on the user I/O response time. It should be noted that only 10 GB capacity of each disk is used in the experiments, so the benefit of PDS must be much larger in practical systems storing large amounts of data. Thus, scaling process may greatly degrade the storage system performance with ongoing applications, and efficient scaling schemes are very important to scale large-scale storage systems.

We also notice that the latency of PDS (on-scaling + post-scaling) is higher than that of PDS (on-scaling) under the Financial1 workload in Fig. 19, which implies that, for PDS under the Financial1 workload, the average latency after scaling is even higher than that during scaling. We analyze all the three trace files and find that the request rate of the Financial1 workload between t_{ep} and t_{er} happens to be much higher than those between t_b and t_{ep}.

C. PERFORMANCE AFTER SCALING

The data layout of a RAID scaled by PDS differs from ''standard'' parity declustered data layout preserved by the round-robin scaling scheme. Hence in this section we evaluate the storage performance, in terms of the average response time, after the RAID is scaled. Specifically we use the three workloads to measure the performance of the two RAIDs, scaled from the same RAID using PDS and round-robin. Each experiment lasts 30 minutes, records the latency of each user requests, and then computes the average latency.

First, we compare the performance of the two RAIDs, after one scaling operation using the two scaling approaches by adding n ($n = 1, 2, 3$) disks to a seven-disk array with stripe size five, including one parity per stripe. We use (n, x, y) to represent the numbers of new added disks, new data blocks added to an original stripe and new parity blocks added to an original stripe, respectively. Fig. 20 plots the average latencies for the two data layouts. It illustrates that, under the same situation, the average latencies of PDS and round-robin are almost identical. The reason why the two data layouts have approximately the same latencies is as follows. Since the stripe size after scaling is the same for both scaling methods under the same circumstances, and the number of blocks in one PPD-column is quite small, data parallelism is almost the same for the two data layouts in the scaled systems.

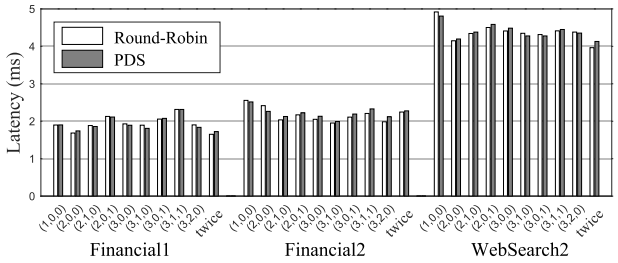


FIGURE 20. Average latency comparison between PDS and Round-Robin under different workloads after one scaling operation for different (n, x, y) values and two scaling operations of adding two disks each time (marked as "twice").

Second, we compare the storage performance after the RAID is scaled twice without increasing stripe size by adding two disks each time. Fig. 20 (marked as "twice") also shows the average latency for the two data layouts of the RAIDs that have been scaled twice. It again illustrates that their performances are almost identical under each workload.

X. CONCLUSION

This paper proposes PDS, a *parity declustering scaling* scheme that efficiently scales up RAIDs deployed with parity declustered data layout while requiring minimal data migration and allows to improve the reliability and/or storage efficiency of the RAID during scaling. PDS uses a BIBD-based data migration scheme to move blocks across disks, so as to guarantee parity declustered data layout with minimal data migration. It also allows to increase the reliability and/or storage efficiency by defining the allocation of new data/parity blocks, while still preserving parity declustered data layout. Theoretical proofs show that PDS can keep parity declustered data data layout while requiring the minimal data migration and improving the reliability and/or storage efficiency. Experimental results show that PDS can efficiently decrease the user response time during scaling and shorten the scaling time compared with RR adapted to achieve parity declustered data layout after scaling.

APPENDIX

A. PROOF OF THEOREM 2

Theorem 2: The data layout in the storage units of the first and the second types is parity declustered after the stripe size being increased.

Proof: For the data layout in the storage units of the first type and the second type, in each PIE, we define b_o^* tuples $T_0^*, T_1^*, \cdots, T_{b_o^*-1}^*$ with base set $S = \{0, 1, \cdots, v_o^*-1\}$ as follows. $T_i^* = \{j \mid \textit{Stripe } S_i \textit{ has a block stored in disk } D_j\}$.

Now we prove $\mathbb{B}_o^* = \{T_0^*, T_1^*, \cdots, T_{b_o^*-1}^*\}$ is a $(b_o^*, v_o^*, k_o^*, r_o^*, \lambda_o^*) = (b_p \times b_a \times b_{inc}, v_a, k_p + k_{inc}, r_a \times b_{inc} \times r_p + (b_a - r_a) r_{inc} \times b_p, \lambda_a \times b_{inc} \times \lambda_p + 2(r_a - \lambda_a) r_{inc} \times r_p + (b_a - 2r_a + \lambda_a)\lambda_{inc} \times b_p)$-BIBD. Note that in the following of this proof, the statement is limited in a PIE.

- b_o^*: Since each PIE consists of b_{inc} PMEs and each PME contains $b_p \times b_a$ stripes, so there are $b_p \times b_a \times b_{inc}$ stripes in a PIE after the stripe size being increased. So there are $b_o^* = b_p \times b_a \times b_{inc}$ tuples in $\mathbb{B}_o^*$.
- v_o^*: There are v_a disks in the array, which means there are $v_o^* = v_a$ objects in the base set, S.
- k_o^*: Since k_{inc} blocks are added into each stripe, each stripe contains $k_p + k_{inc}$ blocks after the stripe size being increased, which means each tuple containing $k_o^* = k_p + k_{inc}$ objects in S.
- r_o^*: r_o^* is the number of blocks stored in the storage units of the first and the second types in a PIE of each disk. We calculate r_o^* as follows.
  After the stripe size being increased, each disk contains r_a PPD-columns in a PME and there are b_{inc} PMEs in a PIE. Moreover, each PPD-column contains r_p blocks. So $N_{first} = r_a \times b_{inc} \times r_p$ blocks are stored in the storage units of the first type on each disk.
  Now we come to calculating the number of blocks stored in the storage units of the second type. Before scaling, there are b_a PPD areas in a PME. While after data migration, on each disk in a PME, the PPD-columns of just r_a PPD areas still stores the blocks in the system before scaling, i.e., the storage units in these PPD-columns are of the first type. The blocks in other $b_a - r_a$ PPD areas on the disk in a PME are migrated and the corresponding storage units are freed.
  Denote

$$N_{free} = b_a - r_a. \tag{8}$$

  Given disk D_l, suppose that the N_{free} PPD areas on D_l are the $h_0, h_1, \cdots, h_{N_{free}-1}$-th PPD areas in the PME, where $0 \le h_0 < h_1 < \cdots < h_{N_{free}-1} \le b_a - 1$. Suppose the disk number l is the w_i-th smallest number in $T_{h_i}^{\complement}$, where $i \in \{0, 1, \cdots, N_{free} - 1\}$. Since we set $v_{inc} = v_a - k_a$, we have $0 \le w_i \le v_{inc} - 1$. Let $M = (m_{i,j})_{b_{inc} \times v_{inc}}$ be the incidence matrix of $\mathbb{B}_{inc}$.
  When we allocate blocks in the storage units of the second type, according to the incidence matrix M, if $m_{i,w_j} = 1$, we allocate an IBA-group on disk D_l for increasing the stripes in the h_j-th PPD area $(0 \le j \le N_{free} - 1)$ of the i-th PME $(0 \le i \le b_{inc} - 1)$. Therefore, the number of IBA-groups allocated on disk D_l is

$$\begin{aligned} N_G &= \sum_{i=0}^{b_{inc}-1} \sum_{j=0}^{N_{free}-1} m_{i,w_j} \quad \text{(summing first on } j\text{)} \\ &= \sum_{j=0}^{N_{free}-1} \sum_{i=0}^{b_{inc}-1} m_{i,w_j} \quad \text{(summing first on } i\text{)} \\ &= \sum_{j=0}^{N_{free}-1} r_{inc} \quad \text{(by Defination 1)} \\ &= (b_a - r_a) r_{inc}. \quad \text{(by (8))} \end{aligned}$$

  Since $N_G = (b_a - r_a) r_{inc}$ is independent of the disk number l, the number of IBA-groups allocated on each disk is N_G. As each IBA-group contains b_p blocks, the number

of blocks stored in the storage units of the second type on each disk is $N_{second} = N_G \times b_p = (b_a - r_a)r_{inc} \times b_p$. Thus, there are $N_{first} + N_{second}$ blocks stored in the storage units of the first and the second types on each disk, which means each object appears in $r_o^* = N_{first} + N_{second} = r_a \times b_{inc} \times r_p + (b_a - r_a)r_{inc} \times b_p$ tuples of $\mathbb{B}_o^*$.

- λ_o^*: Now we prove that, there is a constant λ_o^* such that for each pair of disks D_{j_0} and D_{j_1} $(0 \le j_0 < j_1 \le v_a - 1)$, there are just λ_o^* stripes where both D_{j_0} and D_{j_1} store blocks in the storage units of the first and the second types.

  Given D_{j_0} and D_{j_1}, we define that a PPD-column or an IBA-group in D_{j_0} *is related to* another PPD-column or IBA-group in D_{j_1} when they contain blocks of the same stripes in the storage units of the first and the second types. Given D_{j_0} and D_{j_1} for $0 \le j_0 < j_1 \le v_a - 1$, we define some symbols in the following.
  - (C, C), a pair of related PPD-columns on disks D_{j_0} and D_{j_1}.
  - $(C, \varnothing)$, a PPD-column on D_{j_0} while there is no related PPD-column on D_{j_1}.
  - $(\varnothing, C)$, a PPD-column on D_{j_1} while there is no related PPD-column on D_{j_0}.
  - $(\varnothing, \varnothing)$, neither disk D_{j_0} nor disk D_{j_1} contains a PPD-column in a PPD area.
  - (C, G), a PPD-column on disk D_{j_0} and a related IBA-group on disk D_{j_1}.
  - (G, C), an IBA-group on disk D_{j_0} and a related PPD-column on disk D_{j_1}.
  - (G, G), a pair of related IBA-groups on disks D_{j_0} and D_{j_1}.
  - N_X, the number of Xs in a PME before allocating IBA-groups, where $X \in \{(C, C), (C, \varnothing), (\varnothing, C), (\varnothing, \varnothing)\}$.
  - N_X^*, the number of Xs in a PIE after allocating IBA-groups, where $X \in \{(C, C), (C, G), (G, C), (G, G)\}$.

  In the above definition, we use C to represent a PPD-column where the storage units are of the first types after the RAID scaling, while use $\varnothing$ to represent a PPD-column where the storage units are freed in the old disks or still free in the new disks after the data migration and before allocating the blocks of the second types for increasing stripe size. After we allocating blocks in a PPD-column of $\varnothing$, it will become a PPD-column of G. In the following, we will calculate the number of pairs (C, C), (C, G), (G, C) and (G, G) based on the auxiliary BIBD $\mathbb{B}_a$ and $\mathbb{B}_{inc}$ to get λ_o^*.

  As we migrate PPD-columns according to $\mathbb{B}_a$, we can apply the four properties of Lemma 1 and show that

$$N_{(C,C)} = \lambda_a, \tag{9}$$
$$N_{(C,\varnothing)} = r_a - \lambda_a, \tag{10}$$
$$N_{(\varnothing,C)} = r_a - \lambda_a, \tag{11}$$
$$N_{(\varnothing,\varnothing)} = b_a - 2r_a + \lambda_a. \tag{12}$$

  Since there are b_{inc} PMEs in a PIE, we have that $N_{(C,C)}^* = N_{(C,C)} \times b_{inc} = \lambda_a \times b_{inc}$.

  We know that the $N_{(C,\varnothing)}$ $(C, \varnothing)$'s are in $N_{(C,\varnothing)}$ PPD areas in a PME before allocating IBA-groups. Suppose that the $N_{(C,\varnothing)}$ PPD areas are the l_0, l_1, $\cdots$, $l_{N_{(C,\varnothing)}-1}$-th PPD areas in the PME, where $0 \le l_0 < l_1 < \cdots < l_{N_{(C,\varnothing)}-1} \le b_a - 1$. Suppose that the disk number j_1 is the x_i-th smallest number in $T_{l_i}^{\mathsf{C}}$, where $i \in \{0, 1, \cdots, N_{(C,\varnothing)} - 1\}$. Since we set $v_{inc} = v_a - k_a$, we have $0 \le x_i \le v_{inc} - 1$. Let $M = (m_{i,j})_{b_{inc} \times v_{inc}}$ be the incidence matrix of $\mathbb{B}_{inc}$. We know that there is a PPD-column as a $(C, \varnothing)$ in the l_j-th PPD area $(0 \le j \le N_{(C,\varnothing)} - 1)$ of the i-th PME $(0 \le i \le b_{inc} - 1)$ on disk D_{j_0} before allocating IBA-groups, we will allocate a related IBA-group on disk D_{j_1} when $m_{i,x_j} = 1$. Then a $(C, \varnothing)$ becomes a (C, G) in this case and we have that

$$\begin{aligned} N_{(C,G)}^* &= \sum_{i=0}^{b_{inc}-1} \sum_{j=0}^{N_{(C,\varnothing)}-1} m_{i,x_j} \quad \text{(summing first on } j\text{)} \\ &= \sum_{j=0}^{N_{(C,\varnothing)}-1} \sum_{i=0}^{b_{inc}-1} m_{i,x_j} \quad \text{(summing first on } i\text{)} \\ &= \sum_{j=0}^{N_{(C,\varnothing)}-1} r_{inc} \quad \text{(by Defination 1)} \\ &= (r_a - \lambda_a)r_{inc}. \quad \text{(by (10))} \end{aligned}$$

  Similarly, we can prove that $N_{(G,C)}^* = N_{(C,G)}^* = (r_a - \lambda_a)r_{inc}$.

  We know that there are $N_{(\varnothing,\varnothing)}$ $(\varnothing, \varnothing)$'s in a PME before allocating IBA-groups. Suppose that these $N_{(\varnothing,\varnothing)}$ PPD areas are the $f_0, f_1, \cdots, f_{N_{(\varnothing,\varnothing)}-1}$-th PPD areas in the PME, where $0 \le f_0 < f_1 < \cdots < f_{N_{(\varnothing,\varnothing)}-1} \le b_a - 1$. Suppose that the disk numbers j_0, j_1 are the y_i-th and the z_i-th smallest numbers in $T_{f_i}^{\mathsf{C}}$ $(0 \le i \le N_{(\varnothing,\varnothing)} - 1)$, respectively. Since $j_0 < j_1$ and we set $v_{inc} = v_a - k_a$, we have $0 \le y_i < z_i \le v_{inc} - 1$. We know that the f_j-th PPD area $(0 \le j \le N_{(\varnothing,\varnothing)} - 1)$ of the i-th PME $(0 \le i \le b_{inc} - 1)$ is a $(\varnothing, \varnothing)$ before allocating IBA-groups, we will allocate a pair of related IBA-groups on disks D_{j_0} and D_{j_1} when $m_{i,y_j} = 1$ and $m_{i,z_j} = 1$. Then a $(\varnothing, \varnothing)$ becomes a (G, G) in this case and we have that

$$\begin{aligned} N_{(G,G)}^* &= \sum_{i=0}^{b_{inc}-1} |\{j \mid m_{i,y_j} = 1, m_{i,z_j} = 1, \\ &\qquad 0 \le j \le N_{(\varnothing,\varnothing)} - 1\}| \quad \text{(summing on } i\text{)} \\ &= \sum_{j=0}^{N_{(\varnothing,\varnothing)}-1} |\{i \mid m_{i,y_j} = 1, m_{i,z_j} = 1, \\ &\qquad 0 \le i \le b_{inc} - 1\}| \quad \text{(summing on } j\text{)} \\ &= \sum_{j=0}^{N_{(\varnothing,\varnothing)}-1} \lambda_{inc} \quad \text{(by Defination 1)} \\ &= (b_a - 2r_a + \lambda_a)\lambda_{inc}. \quad \text{(by (12))} \end{aligned}$$

Since a (C, C) contains λ_p pairs of related blocks, a (C, G) or a (G, C) contains r_p pairs of related blocks and a (G, G) contains b_p pairs of related blocks, we have that a pair of disks D_{j_0} and D_{j_1} has $N_{related} = N^*_{(C,C)} \times \lambda_p + (N^*_{(C,G)} + N^*_{(G,C)})r_p + N^*_{(G,G)} \times b_p$ pairs of related blocks. Then each pair of disks has $N_{related}$ pairs of related blocks, which means that each pair of objects appears in exactly $\lambda^*_o = N_{related} = \lambda_a \times b_{inc} \times \lambda_p + 2(r_a - \lambda_a)r_{inc} \times r_p + (b_a - 2r_a + \lambda_a)\lambda_{inc} \times b_p$ tuples in $\mathbb{B}^*_o$.

According to Definition 1, $\mathbb{B}^*_o = \{T^*_0, T^*_1, \cdots, T^*_{b^*_o-1}\}$ is a $(b_p \times b_a \times b_{inc}, v_a, k_p + k_{inc}, r_a \times b_{inc} \times r_p + (b_a - r_a)r_{inc} \times b_p, \lambda_a \times b_{inc} \times \lambda_p + 2(r_a - \lambda_a)r_{inc} \times r_p + (b_a - 2r_a + \lambda_a)\lambda_{inc} \times b_p)$-BIBD. So the data layout in the storage units of the first and the second types in a PIE is defined by an objective $(b^*_o, v^*_o, k^*_o, r^*_o, \lambda^*_o)$-BIBD $\mathbb{B}^*_o$. ■

B. PROOF OF THEOREM 4

Theorem 4: PDS maintains uniform data and parity distributions after scaling.

Proof: In the following, for the storage units of each type, we prove that data blocks and parity blocks are distributed evenly among all disks separately.

- For the first type of storage units, from $\mathbb{B}_a$, we know that in each PME of the scaled system, there are r_a PPD-columns stored in each disk. Each PPD-column contains r_p blocks including the same number of parity blocks. So each disk contains $r_o = r_p \times r_a$ blocks including the same number of parity blocks in of storage units of the first type, i.e., the scaled system achieves even distribution of data blocks and parity blocks among distinct disks in the storage units of the first type in a PME.
- If we do not increase stripe size, the capacity of the storage units of the second type is 0. Otherwise, from the proof in Theorem 2, we know that each disk contains $N_{second} = (b_a - r_a)r_{inc} \times b_p$ blocks in the storage units of the second type in a PIE. When we do not increase the reliability (i.e., $y_{inc} = 0$), there is no parity blocks in the second type of storage units and each disk contains N_{second} data blocks in the storage units of the second type in a PIE.

  When we increase the reliability (i.e., $y_{inc} \neq 0$), according to step 4 (i.e., Auxiliary Full Block Design Table Construction for Increasing Stripe Size) of the scaling process of PDS in Section V-B, we know that there are b_{inc} tuples in $\mathbb{B}_{inc}$ and each tuple has an object labelled with P_x $(0 \leq x \leq y_{inc} - 1)$, so there are b_{inc} objects labelled with P_x in TBL_{inc}. Furthermore, these b_{inc} objects labelled with P_x in TBL_{inc} are balanced perfectly among the v_{inc} distinct objects of TBL_{inc}. Thus, we have that the number of object j $(0 \leq j \leq v_{inc} - 1)$ in TBL_{inc} labelled with P_x is b_{inc}/v_{inc}. Note that b_{inc} is a multiple of v_{inc} in this case. That is to say, for $\forall x \in \{0, 1, \cdots y_{inc} - 1\}$ and $\forall j \in \{0, 1, \cdots v_{inc} - 1\}$, we have that

$$|\{i \mid j \text{ in } T^{inc}_i \text{ is labelled with } P_x, 0 \leq i \leq b_{inc} - 1\}| = b_{inc}/v_{inc}. \quad (13)$$

  We calculate the number of parity blocks with parity type P_x stored in the storage units of the second type on each disk as following. From the proof of Theorem 2 (see the calculation for r^*_o), we know that for any disk, say disk D_l, where $0 \leq l \leq v_a - 1$, there are $N_{free} = b_a - r_a$ (i.e., (8)) PPD areas that do not contain a PPD-column on disk D_l in a PME before allocating IBA-groups. Moreover, in the proof of Theorem 2 (see the calculation for r^*_o), we have already assumed that 1) these N_{free} PPD areas are the h_0, h_1, $\cdots$, $h_{N_{free}-1}$-th PPD areas in the PME, where $0 \leq h_0 < h_1 < \cdots < h_{N_{free}-1} \leq b_a - 1$; 2) the disk number l is the w_i-th smallest number in $T^{\mathbb{C}}_{h_i}$, where $i \in \{0, 1, \cdots, N_{free} - 1\}$.

  We allocate IBA-group $G^{p_x}_{i \times b_a + h_j, w_j}$, which represents b_p parity blocks of type P_x, on disk D_l for the stripes in the h_j-th PPD area $(0 \leq j \leq N_{free} - 1)$ of the i-th PME $(0 \leq i \leq b_{inc} - 1)$ when object i in tuple $T^{inc}_{w_j}$ is labelled with parity type P_x $(0 \leq x \leq y_{inc} - 1)$. Therefore, the number of IBA-groups with parity blocks of parity type P_x allocated on disk D_l is

$$\begin{aligned} N^p_G &= \sum_{i=0}^{b_{inc}-1} |\{j \mid w_j \text{ in } T^{inc}_i \text{ is labelled with } P_x, \\ &\quad 0 \leq j \leq N_{free} - 1\}| \quad \text{(summing on } i\text{)} \\ &= \sum_{j=0}^{N_{free}-1} |\{i \mid w_j \text{ in } T^{inc}_i \text{ is labelled with } P_x, \\ &\quad 0 \leq i \leq b_{inc} - 1\}| \quad \text{(summing on } j\text{)} \\ &= \sum_{j=0}^{N_{free}-1} b_{inc}/v_{inc} \quad \text{(by (13))} \\ &= (b_a - r_a)b_{inc}/v_{inc}. \quad \text{(by (8))} \end{aligned}$$

  Since $N^p_G = (b_a - r_a)b_{inc}/v_{inc}$ is independent of disk number l and x in parity type P_x, the number of IBA-groups with parity blocks allocated on each disk is a constant, N^p_G. As each IBA-group with parity blocks contains b_p parity blocks, each disk contains $N^p_G \times b_p$ parity blocks with parity type P_x $(0 \leq x \leq y_{inc} - 1)$ in the storage units of the second type in a PIE. So data blocks and parity blocks in the storage units of the second type is uniformly distributed among all disks in a PIE.
- For the third type of storage units, as the data layout of the third type of storage units is defined by full block design table TBL_{new}, by using all entries of TBL_{new} to place blocks once, there will be r_{new} blocks including the same amount of parity blocks stored in each disk. Thus, we have that the scaled system achieves even distribution of data blocks and parity blocks among all disks in the storage units of the third type.

Above all, we have that data blocks and parity blocks are both distributed evenly among all disks in the storage units of each type. Thus, the scaled system achieves even distribution of data blocks and parity blocks among all disks. ■

ACKNOWLEDGMENT

This paper was presented in part at the 46th IEEE International Conference on Parallel Processing (ICPP), 2017 [14]. In the conference paper, we proposed an efficient scaling algorithm, *PDS (Parity Declustering Scaling)*, to scale up disk arrays with parity declustered data layout. However, the disk array remains the same reliability and storage efficiency after the scaling process. In this journal version, we extend our prior work in the following aspects.

- We propose a scheme within PDS to improve the reliability and/or storage efficiency during the scaling process as options.
- We provide theoretical proofs to formally show the benefits of the proposed scheme.
- We implement the scheme in Linux kernel 3.14.72 and conduct more experiments with real-world traces to evaluate its performance.

REFERENCES

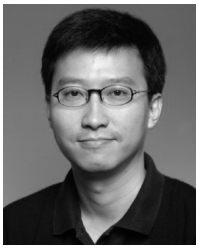
[1] Linux Man Page. *Blktrace*. Accessed: Apr. 2, 2018. [Online]. Available: https://linux.die.net/man/8/blktrace
[2] NetBSD Documentation. *NetBSD RAIDframe*. Accessed: Apr. 2, 2018. [Online]. Available: http://www.netbsd.org/docs/guide/en/chap-rf.html
[3] University of Massachuestts. OLTP Application I/O and Search Engine I/O. *UMass Trace Repository*. Accessed: Apr. 2, 2018. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage
[4] H. P. Anvin. (Jan. 2004). *The Mathematics of RAID-6*. Accessed: Apr. 2, 2018. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf
[5] N. Brown. *RAID-5 Resizing, Drivers/MD/Raid5.C in the Source Code of Linux Kernel 3.14.72*. Accessed: Apr. 2, 2018. [Online]. Available: https://www.kernel.org
[6] I. I. Courtright, V. William, G. Gibson, M. Holland, L. N. Reilly, and J. Zelenka, "RAIDframe: A rapid prototyping tool for RAID systems," School Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-97-142, 1997.
[7] S. Ghandeharizadeh and D. Kim, "On-line reorganization of data in scalable continuous media servers," in *Proc. Int. Conf. Database Expert Syst. Appl.*, 1996, pp. 751–768.
[8] A. Goel, C. Shahabi, S. D. Yao, and R. Zimmermann, "SCADDAR: An efficient randomized technique to reorganize continuous media blocks," in *Proc. 18th Int. Conf. Data Eng. (ICDE)*, 2002, pp. 473–482.
[9] J. L. Gonzalez and T. Cortes, "Increasing the capacity of RAID5 by online gradual assimilation," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/Os (SNAPI)*, 2004, pp. 12–24.
[10] M. Hall, *Combinatorial Theory*. Hoboken, NJ, USA: Wiley, 1986.
[11] H. Hanani, "Balanced incomplete block designs and related designs," *Discrete Math.*, vol. 11, no. 3, pp. 255–369, Jan. 1975.
[12] M. Holland, "On-line data reconstruction in redundant disk arrays," Ph.D. dissertation, School Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, 1994.
[13] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proc. 5th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 1992, pp. 23–35.
[14] Z. Li, Y. Xu, Y. Li, C. Tian, and Y. Bai, "PDS: An I/O-efficient scaling scheme for parity declustered data layout," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, 2017, pp. 402–411.
[15] Y. Mao, J. Wan, Y. Zhu, and C. Xie, "A new parity-based migration method to expand RAID-5," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 8, pp. 1945–1954, Aug. 2014.
[16] R. R. Muntz and J. C. S. Lui, "Performance analysis of disk arrays under failure," in *Proc. 16th Int. Conf. Very Large Data Bases (VLDB)*, 1990, pp. 162–173.
[17] D. A. Patterson, "A simple way to estimate the cost of downtime," in *Proc. 16th Large Installation Syst. Admin. Conf. (LISA)*, 2002, pp. 185–188.
[18] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 1988, pp. 109–116.
[19] E. Pinheiro, W. D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proc. 5th USENIX Conf. File Storage Technol. (FAST)*, 2007, pp. 17–28.
[20] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems," *J. Softw. Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, Sep. 1997.
[21] E. J. Schwabe and I. M. Sutherland, "Improved parity-declustered layouts for disk arrays," *J. Comput. Syst. Sci.*, vol. 53, no. 3, pp. 328–343, Dec. 1996.
[22] E. J. Schwabe and I. M. Sutherland, "Flexible usage of redundancy in disk arrays," *Theory Comput. Syst.*, vol. 32, no. 5, pp. 561–587, Oct. 1999.
[23] D. R. Stinson, *Combinatorial Designs: Constructions and Analysis*. New York, NY, USA: Springer-Verlag, 2004.
[24] J. Wan, P. Xu, X. He, J. Wang, J. Li, and C. Xie, "H-scale: A fast approach to scale disk arrays via hybrid stripe deployment," *ACM Trans. Storage*, vol. 12, no. 3, p. 16, Jun. 2016.
[25] B. Welch *et al.*, "Scalable performance of the Panasas parallel file system," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST)*, 2008, pp. 17–33.
[26] 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)*, 2012, pp. 460–469.
[27] 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)*, Sep. 2012, pp. 284–292.
[28] Q. Xin, E. L. Miller, T. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin, "Reliability mechanisms for very large storage systems," in *Proc. 20th IEEE/11th NASA Goddard Conf. Mass Storage Syst. Technol. (MSST)*, Apr. 2003, pp. 146–156.
[29] X. Yu *et al.*, "Trading capacity for performance in a disk array," in *Proc. 4th Conf. Symp. Oper. Syst. Design Implement. (OSDI)*, 2000, pp. 17–32.
[30] G. Zhang, K. Li, J. Wang, and W. Zheng, "Accelerate RDP RAID-6 scaling by reducing disk I/Os and XOR operations," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 32–44, Jan. 2015.
[31] G. Zhang, G. Wu, Y. Lu, J. Wu, and W. Zheng, "Xscale: Online X-code RAID-6 scaling using lightweight data reorganization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3687–3700, Dec. 2016.
[32] G. Zhang, W. Zheng, and K. Li, "Rethinking RAID-5 data layout for better scalability," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2816–2828, Nov. 2014.
[33] G. Zhang, W. Zheng, and J. Shu, "ALV: A new data redistribution approach to RAID-5 scaling," *IEEE Trans. Comput.*, vol. 59, no. 3, pp. 345–357, Mar. 2010.
[34] E. K. Lee and R. H. Katz, "The performance of parity placements in disk arrays," *IEEE Trans. Comput.*, vol. 42, no. 6, pp. 651–664, Jun. 1993.
[35] M. Holland, G. A. Gibson, and D. P. Siewiorek, "Architectures and algorithms for on-line failure recovery in redundant disk arrays," *Distrib. Parallel Databases*, vol. 2, no. 3, pp. 295–335, Jul. 1994.
[36] A. Miranda and T. Cortes, "CRAID: Online RAID upgrades using dynamic hot data reorganization," in *Proc. 12th USENIX Conf. File Storage Technol. (FAST)*, 2014, pp. 133–146.

ZHIPENG LI received the bachelor's degree in computer science from the University of Science and Technology of China, in 2013, where he is currently pursuing the Ph.D. degree with the School of Computer Science and Technology. His research interests include storage systems, solid-state devices, and erasure codes.

YINLONG XU received the B.S. degree in mathematics from Peking University, in 1983, and the M.S. and Ph.D. degrees in computer science from the University of Science and Technology of China, in 1989 and 2004, respectively, where he is currently a Professor with the School of Computer Science and Technology, and is leading a research group in doing some networking and high performance computing research. His research interests include network coding, storage systems, combinatorial optimization, design and analysis of parallel algorithms, and parallel programming tools. He received the Excellent Ph.D. Advisor Award of the Chinese Academy of Sciences, in 2006.

YONGKUN LI received the B.Eng. degree in computer science from the University of Science and Technology of China, in 2008, and the Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong, in 2012. He was a Postdoctoral Fellow with the Institute of Network Coding, The Chinese University of Hong Kong. He is currently an Associate Researcher with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include performance evaluation and architectural design of networking and storage systems.

CHENGJIN TIAN received the bachelor's degree from the School of Computer Science and Technology, University of Science and Technology of China, in 2016, where he is currently pursuing the master's degree with the School of Computer Science and Technology. His research interests include storage systems, including scaling and reliability issues.



JOHN C. S. LUI received the Ph.D. degree in computer science from UCLA. He is currently a Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His current research interests include communication networks, network/system security, network economics, network sciences, cloud computing, large scale distributed systems, and performance evaluation theory. He is an elected member of the IFIP WG 7.3, a fellow of ACM and IEEE, and Croucher Senior Research Fellow. He serves on the Editorial Board for the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the Journal of *Performance Evaluation*, and the *International Journal of Network Security*.