Parity-Switched Data Placement: Optimizing Partial Stripe Writes in XOR-Coded Storage Systems

Zhirong Shen, Jiwu Shu, *Member, IEEE*, and Yingxun Fu

Abstract—Erasure codes tolerate disk failures by pre-storing a low degree of data redundancy, and have been commonly adopted in current storage systems. However, the attached requirement on data consistency exaggerates partial stripe write operations and thus seriously downgrades system performance. Previous works to optimize partial stripe writes are relatively limited, and a general mechanism is still absent. In this paper, we propose a Parity-Switched Data Placement (PDP) to optimize partial stripe writes for any XOR-coded storage system. PDP first reduces the write operations by arranging continuous data elements to join a common parity element's generation. To achieve a deeper optimization, PDP further explores the generation orders of parity elements and makes any two continuous data elements associate with a common parity element. Intensive evaluations show that for tested erasure codes, PDP reduces up to 31.9 percent of write operations and further increases the write speed by up to 59.8 percent when compared with two state-of-the-art data placement methods.

Index Terms—Partial stripe writes, XOR-coded storage systems, data placement, parity generation

1 Introduction

Erasure coding has been commonly adopted in distributed storage systems to promise data reliability by pre-storing a low degree of data redundancy [1], [2], [3], [4]. One typical class of erasure codes is **M**aximum **D**istance **S**eparable (MDS) codes [5], [6], [7], [8], [9], [10], which provides the optimal storage efficiency. MDS codes are typically operated by two parameters k and m: an (k, m) MDS code takes k pieces of original data as input, and produces another m parity pieces, such that *any* k out of $k + m$ pieces are sufficient to reconstruct the original data. The $k + m$ dependent pieces collectively form a *"stripe"* and are distributed to $k + m$ disks, such that any m disk failures are tolerated. A family of MDS codes is XOR-based erasure code [7], [8], [9], [10], [11], [12], [13], [14], which only performs XOR operations for fast parity calculation and data reconstruction. XOR-based erasure codes have been popular solutions in current storage systems [15], [16], [17] (we call them "XOR-coded storage systems"). In XOR-coded storage systems, a data piece (resp. parity piece) will be partitioned into many "*data elements*" (resp. "*parity elements*") with equal-size.

One major issue that current XOR-coded storage systems suffer is how to efficiently handle the writes to the kept data. A write operation is usually processed by two write modes, i.e., *read-modify-write mode* for small writes (i.e., the updated size is less than half the stripe length) [8], [19] and *reconstruct-write mode* for large writes (i.e., the updated length is more than half the stripe length) [20], [21]. However, small writes are usually dominant operations in many real storage applications. We analyze several real workloads selected from MSR Cambridge Traces [18] in Fig. 1, which indicates that the small writes whose sizes are no larger than 8 KB take up more than 70 percent of all the write operations. Therefore, in this paper we mainly consider write operations in the read-modify-write mode.

The read-modify-write mode first retrieves the wanted data and then writes it back after modification. For such kind of operations, the attached consistency requirement will also trigger extra writes to the associated redundant information (i.e., parity elements). As a result, the size of data to be written is amplified and the system performance is downgraded. Meanwhile, write operations in XOR-coded storage systems can be classified into *full stripe writes* and *partial stripe writes* according to the different sizes of operated data in a stripe. Full stripe writes completely write all the data elements in a stripe and the optimal complexity of full stripe write can be achieved by MDS codes [8]. As a comparison, partial stripe writes only operate a subset of data elements in a stripe and they are still concerned in current storage system designs [6], [8], [9], [22], [23]. In this paper, we mainly consider *partial stripe writes to continuous data elements*, as other kinds of partial stripe writes (e.g., partial stripe writes to non-continuous data elements) can be treated as the combination of these basic operations.[1] Therefore, the optimization techniques for partial stripe writes to continuous data elements can also have a good effect on

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhirong.shen2601@gmail.com, {shujw, fu-yx10}@tsinghua.edu.cn.*

Manuscript received 11 July 2015; revised 11 Jan. 2016; accepted 25 Jan. 2016. Date of publication 4 Feb. 2016; date of current version 12 Oct. 2016. Recommended for acceptance by R. Vuduc. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2016.2525770

1. For example, if a write operation updates non-continuous data elements {`#1,#2,#4,#5`}, then it can be treated as the combination of the write operation to continuous data elements {`#1,#2`}, and the one to continuous data elements {`#4,#5`}. `#i` means the ith data element in a stripe.

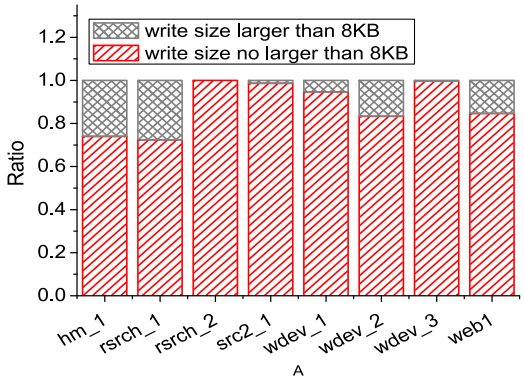


Fig. 1. Small write ratio of real workloads [18].

improving the performance of partial stripe writes to non-continuous data elements.

Some explicit RAID-6 codes (i.e., H-Code [8] and HV Code [9]) have considered the optimization of partial stripe writes under the premise that data elements are horizontally placed, yet two limitations remain. First, the existing works [8], [9] are specific RAID-6 codes. They are useless to optimize partial stripe writes for the storage systems that select other RAID-6 codes for data protection. For example, NetApp RAID-DP [16] (selects RDP Code [14] for RAID-6 protection [24]) and EMC Symmetrix DMX [15] (selects EVENODD Code [25] for RAID-6 protection). Second, the proposed RAID-6 codes for partial stripe write optimization only tolerate at most double disk failures. They cannot be deployed on the storage systems that require higher fault tolerance (e.g., the triple failure tolerance requirement in NetApp [26]).

Besides these two RAID-6 codes, RAID-Z [27] is an interesting scheme that is proposed to address the "write hole" problem[2] when combined with the "copy-on-write" transactional semantics in ZFS [28]. Additionally, RAID-Z also makes every block be its own stripe, such that every write in RAID-Z can be handled as a full stripe write and the stripe length in RAID-Z will be various. However, the data reconstruction in RAID-Z requires the understanding of RAID-Z geometry by scanning the file system metadata, which is hard to meet when the file system and RAID array are separate products [27]. Besides, RAID-Z can only resist no more than three disk failures [29], which is not suitable for the storage systems that have higher fault tolerance requirements [17]. Moreover, the realization of RAID-Z cannot be applied to current XOR-coded storage systems [15], [16], [17] that usually select fixed stripe length.

Due to these limitations, there is still no methodology to optimize partial stripe writes for any XOR-coded storage system, thus motivating our work.

In this paper, we propose a **P**arity-Switched **D**ata **P**lacement (PDP) scheme to optimize partial stripe writes for any XOR-coded storage system. As the triggered update to associated parity elements is the core reason that deteriorates a partial stripe write operation, PDP accordingly proposes several techniques to decrease the update operations to parity elements and thus improve the partial stripe writes performance. First, different from previous obstinate data placements, PDP respects the parity generation principle of any XOR-based code and arranges continuous data elements to generate parity elements. This design effectively reduces the updates to parity elements, especially for the write operation that operates the continuous data elements associated with a common parity element. Second, PDP also carefully designs the generation orders of parity elements and makes any two continuous data elements involved in a common parity element's generation. To our best knowledge, it is the *first* work to optimize partial stripe writes for any XOR-coded storage system.

2. It is the inconsistency risk between data information and parity information. It is caused by accident system crash when a write operation does not complete.

TABLE 1
Frequently Used Symbols and Descriptions

Symbols	Descriptions
p	prime number used to configure the stripe size
$\oplus$	XOR operation
`#i`	the ith data element in a stripe
$C_{i,j}$	cell in the ith row and the jth column
R_i	the ith redundant parity element
w	number of parity elements in a stripe
n	number of data elements in a stripe

Our contributions can be summarized as follows:

1) We propose a data placement design named Parity-Switched Data Placement to optimize partial stripe writes for any XOR-coded storage system by decreasing parity updates. To this end, PDP generates parity elements by using continuous data elements, explores generation orders of parity elements, and arranges any two continuous data elementsto share a common parity element.
2) We implement PDP in a real storage system equipped with several representative XOR-based erasure codes, and compare PDP with two start-of-the-art data placement methods. Results show that PDP decreases up to 31.9 percent of write operations and increases the write speed by up to 59.8 percent.

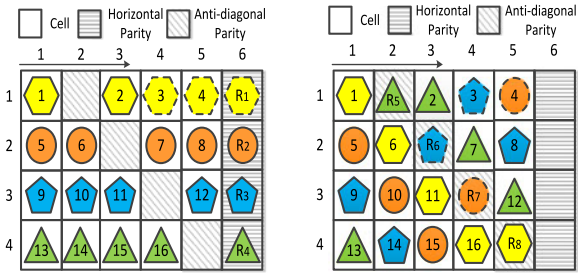
The rest of this paper continues as follows. Section 2 will introduce the research background. The motivations will be presented in Section 3. We then describe the detailed design of PDP in Section 4. Finally, we evaluate PDP in Section 5 and conclude our work in Section 6.

2 Background

2.1 Notations and Descriptions

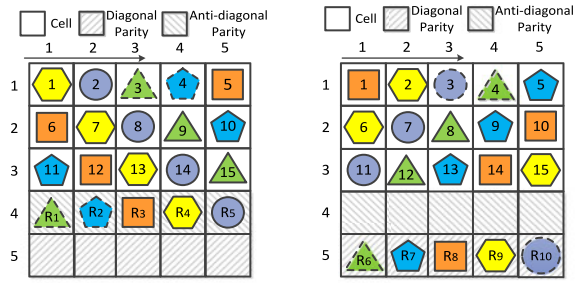
We first summarize the terms and notations that will be frequently referred throughout this paper. Meanwhile, we also list the symbols and their descriptions in Table 1.

Data element and parity element. Element is the basic information unit operated in XOR-coded storage systems. *Data elements* contain original data information, while *parity elements* keep redundant information of data elements. In figures of this paper, we use shapes to represent elements and use numbers to denote their logical orders. The data elements are *continuous* if their logical orders are neighboring. For instance, Fig. 2 shows the placement of data elements in a stripe, where `#i` means the ith data element in a stripe, and `#1` and `#2` are two continuous data elements.



(a) The Horizontal Parity. (b) The Anti-diagonal Parity.

Fig. 2. The layout of H-Code under the horizontal data placement over $p+1$ disks ($p=5$). A write operation to data elements #3 and #4 will need to update three parity elements. The dashed lines label the operated elements in this write operation.



(a) The Anti-diagonal Parity. (b) The Diagonal Parity.

Fig. 3. The layout of X-Code under the horizontal data placement over p disks ($p=5$). A write operation to data elements #3 and #4 will need to update four parity elements. The dashed lines label the operated elements in this write operation.

There are several methods to calculate parity elements, such as horizontal parity and diagonal parity. The horizontal parity element is calculated by performing XOR operations among the data elements in the same row. In this paper, we use R_i to denote the ith parity element in a stripe. For instance, in Fig. 2a, the horizontal parity element R_1 :=#1⊕#2⊕#3⊕#4. The diagonal (resp. anti-diagonal) parity connects the elements following the diagonal (resp. anti-diagonal) line. For example, the anti-diagonal parity element R_5 :=#13⊕ #2⊕ #7 ⊕ #12 as shown in Fig. 2b.

Cell. "Cell" denotes the storage area (e.g., sector) to store an element in this paper. We use $C_{i,j}$ to represent the cell whose position is at the ith row and the jth column in a stripe. For example, $C_{1,1}$ hosts the data element #1 in Fig. 2.

XOR-based erasure codes. XOR-based erasure codes calculate parity elements by just performing XOR operations. This kind of erasure codes owns better encoding/decoding/update efficiency, compared to the codes constructed over complicated operations in the finite field, such as Reed-Solomon Code [30] and SD Codes [31]. The representative XOR-based erasure codes include EVENODD Code [25], RDP Code [14], X-Code [10], P-Code [13], HDP Code [7], H-Code [8], HV Code [9], D-Code [32], STAR Code [33], and Cauchy Reed-Solomon Code [34].

Stripe. A maximal set of data elements and parity elements that have dependent relationship connected by an erasure code. Fig. 2 shows the layout of a stripe in H-Code.

Parity chain. A parity chain is composed of a collection of data elements and the parity element generated by them. In this paper, the elements in the same parity chain are marked in the same shape. For example, #1 involves in two parity chains in Fig. 2, i.e., {#1, #2, #3, #4, R_1} for the horizontal parity chain in Fig. 2a and {#1, #6, #11, #16, R_8} for the anti-diagonal parity chain in Fig. 2b. Therefore, when #1 is written, R_1 and R_8 should be correspondingly renewed for data consistency.

2.2 Existing Data Placement Methods

Data placement is usually established before data storage and will guide next data accesses to the locations where the requested elements reside. Currently, two data placement methods are commonly adopted in XOR-coded storage systems, i.e., horizontal data placement and vertical data placement.

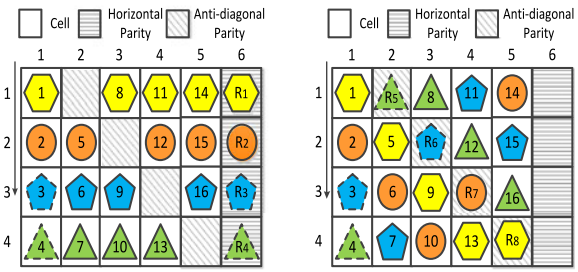
Horizontal data placement. Horizontal data placement "horizontally" lays continuous data elements across disks. For example, Figs. 2 and 3 illustrate the layout of H-Code [8] and X-Code [10] under the horizontal data placement.

The horizontal data placement brings two benefits. First, it takes full advantage of parallel technology [35]. For example, when requesting the data elements {#1, #2} in Fig. 2, the storage system can execute this operation in parallel by reading #1 from disk 1 and retrieving #2 from disk 3, thus the needed time will be shortened. Second, applying the horizontal data placement to a code with horizontal parity chains can effectively decrease the number of updated parity elements in partial stripe writes. For example, when #3 and #4 in Fig. 2 are written, then only 1 horizontal parity element (i.e., R_1 in Fig. 2a) and two anti-diagonal parity elements (i.e., R_6, R_7 in Fig. 2b) will be renewed, as these two data elements share a common horizontal parity element R_1. At last, only three parity elements are updated in this partial stripe write operation.

However, for the codes (e.g., X-Code [10], and P-Code [13]) that do not have horizontal parity chains, the horizontal data placement will lose the second advantage. This is because the continuous data elements laid on the same row may not have a common parity element in these codes. For example, Fig. 3 illustrates the layout of X-Code where data elements are horizontally placed. When the data elements #3 and #4 are written, the associated four parity elements (i.e., the anti-diagonal parity elements R_1 and R_2, and the diagonal parity elements R_6 and R_{10}) will be correspondingly renewed, requiring one more element update than the same write operation in H-Code (shown in Fig. 2).

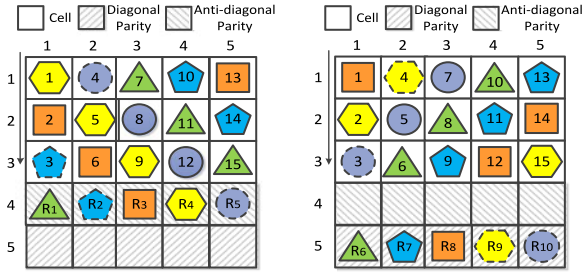
Vertical data placement. Vertical data placement puts continuous data elements along columns. As a comparison with Fig. 2, the layout of H-Code under the vertical data placement is illustrated in Fig. 4.

However, this placement method has two limitations. First, it restricts the parallel accesses of storage systems. For example, when reading data elements {#8, #9, #10} in Fig. 4, the disk 3 has to sequentially execute this operation. Second, the vertical data placement cannot provide a satisfied performance on partial stripe writes, as the continuous data elements stored in the same disk usually do not have any common parity element. For example, if data elements {#3, #4} in Fig. 4 are renewed, then the associated four parity elements (i.e., the horizontal parity elements R_3 and R_4, and the anti-diagonal parity elements R_5 and R_6) will be updated. Fig. 5 also presents the layout of X-Code under the vertical data placement. As the data elements #3 and #4 do not share



(a) The Horizontal Parity. (b) The Anti-diagonal Parity.

Fig. 4. The layout of H-Code under the vertical data placement over $p+1$ disks ($p=5$). A write operation to data elements #3 and #4 will need to update four parity elements. The dashed lines label the operated elements in this write operation.



(a) The Anti-diagonal Parity. (b) The Diagonal Parity.

Fig. 5. The layout of X-Code under the vertical data placement over p disks ($p=5$). A write operation to data elements #3 and #4 will need to update four associated parity elements. The dashed lines label the operated elements in this write operation.

any common parity element, a partial stripe write to them will also renew four parity elements (i.e., R_2, R_5, R_9, and R_{10}).

2.3 Existing Works to Optimize Partial Stripe Writes

To optimize partial stripe writes, several impressive works have been proposed, which can be classified into the following categories.

1) The optimization in RAID-5 [36]. Jin et al. [22] discuss different cases of partial stripe writes in RAID-5, and propose an adaptive control algorithm to reduce caused I/O operations.
2) The optimization in RAID-6. Two impressive works are H-Code [8] and HV Code [9], both of which are designed under the horizontal data placement. The layout of H-Code is shown in Fig. 2. H-Code owns a typical feature that any two continuous data elements share a common parity element. Therefore the writes to any two continuous data elements will renew three related parity elements. For example, both of data elements #3 and #4 join the generation of R_1. When performing a write to them, the horizontal parity element R_1 and the anti-diagonal parity elements R_6 and R_7 will be updated. This design principle is also followed by HV Code [9].
3) The optimization in RAID-Z [27]. RAID-Z avoids partial stripe writes by treating a block as a stripe, such that every write to a block can be handled as a full stripe write. The current realizations of RAID-Z includes RAIDZ-1 (like RAID-5), RAID-Z2 (tolerates double disk failures), and RAIDZ-3 (tolerates triple disk failures) [29].

2.4 Remaining Limitations of Existing Works

Though these impressive works [8], [9], [22], [27] can to some extent improve the performance of partial stripe writes, they still have significant limitations without being well addressed.

1) Both of H-Code and HV Code are two specific codes, they do not figure out how to optimize partial stripe writes in the storage systems equipped with other RAID-6 codes, such as NetApp RAID-DP [16] (selects RDP Code [14] for RAID-6 protection [24]) and EMC Symmetrix DMX [15] (selects EVENODD Code [25] for RAID-6 protection). Besides, they can only offer the protection of at most two disk failures. They are useless to the storage systems that have higher reliability requirements (e.g., the triple failure tolerance requirement in NetApp [26]).
2) As referred above, RAID-Z introduces variable stripe length, which is different from other XOR-based erasure codes [8], [9], [10], [14], [25]. This change makes the data reconstruction in RAID-Z more complex. It requires to scan the file system metadata to understand the RAID-Z geometry. This requirement is impossible to be fulfilled when the file system and RAID array are separate products [27]. Besides, current realizations of RAID-Z can only resist triple disk failures [29]. They cannot be deployed on the storage systems that have higher fault tolerance requirement [17]. Moreover, the realization of RAID-Z cannot be applied to current XOR-based erasure codes that have constant stripe length.

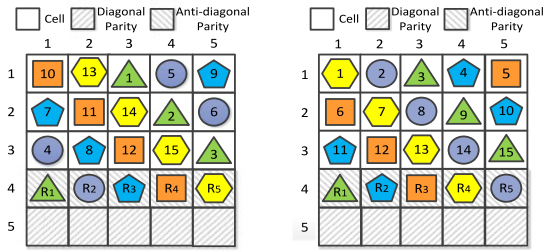
3 Motivation

Based on the above problems, we thus pose the following question: *Is there a methodology to optimize partial stripe writes for any XOR-coded storage system?*

As parity update is the key reason that deteriorates the performance of partial stripe writes, we then accordingly propose to optimize partial stripe writes by decreasing parity updates. Previous works [8], [9] achieve this by completely redesigning new parity generation principles under the horizontal data placement. Though this method is effective, it changes the parity generation principle and cannot be used for any XOR-based erasure code.

On the contrary, in this paper, we suggest respecting the protogenic parity generation principle of each XOR-based erasure code and associating parity elements with continuous data elements through data placement. *For any XOR-based erasure code, this method does not alter the parity generation, and thus does not change the fault tolerance of the original scheme*. In summary, the motivating arguments for our work are as follows.

Data placement principle. We first consider the optimization of partial stripe writes *in a common parity chain*. As referred above, we observe that the codes with horizontal parity chains can reduce the number of parity elements to be updated. This is because the horizontal parity in these

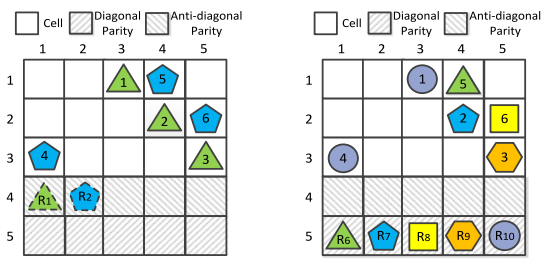


(a) Anti-diagonal Data Placement. (b) Horizontal Data Placement.

Fig. 6. Data placement principle. The generation of anti-diagonal parity elements in X-Code under the anti-diagonal data placement and the horizontal data placement, respectively.

codes makes continuous data elements in the same row share a common horizontal parity element, and then decreases the number of parity updates for a write operation to them. Inspired by this finding, for a code that does not have horizontal parity chains, such as X-Code [10] and P-Code [13], we can also accordingly place continuous data elements to generate parity elements. Therefore, when the data elements in a common parity chain are written, the number of parity elements to be updated will decrease. For example, we adopt "the anti-diagonal data placement" for X-Code by laying continuous data elements along the anti-diagonal line to produce the anti-diagonal parity elements in Fig. 6a. In this case, when writing continuous data elements {#1, #2, #3}, the number of anti-diagonal parity elements to be renewed will decrease to 1 (i.e., R_1). On the contrary, it needs to update three anti-diagonal parity elements (i.e., R_1, R_4, and R_5) when writing {#1, #2, #3} under the horizontal data placement (shown in Fig. 6b).

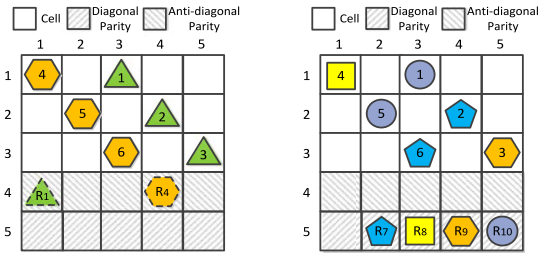
Parity generation order. Based on the data placement principle referred above, we also wish to optimize partial stripe writes *across parity chains*. For any such operation, we can always find a value i such that the data elements to be written are included in the i parity chains. For example, when writing {#2,#3,#4} in Fig. 6a, these operated data elements are covered by the data elements {#1,#2,...,#6} in the 2 parity chains (i.e., the parity chains that generate R_1 and R_2). Thus, the number of associated parity elements to be updated in this partial stripe write operation is no more than that of completely writing the data elements {#1, #2,...,#6} in this 2 parity chains.



(a) 2 anti-diagonal parity elements are updated when {#1,···,#6} are written.

(b) 5 diagonal parity elements are updated when {#1,···,#6} are written.

Fig. 7. Parity generation orders. An example when generating R_1 and R_2. In this case, when data elements {#1,...,#6} are written, then there are extra seven parity elements needed to update. The dashed lines indicate the parity elements generated by continuous data elements.



(a) 2 anti-diagonal parity elements are updated when {#1,···,#6} are written.

(b) 4 diagonal parity elements are updated when {#1,···,#6} are written.

Fig. 8. Parity generation orders. An example when generating R_1 and R_4. In this case, when data elements {#1, ..., #6} are written, then there are extra six parity elements need to be updated. The dashed lines indicate the parity elements generated by continuous data elements.

Derived from this observation, for the partial stripe write across parity chains, we propose to optimize it by lowering its upper-bound on the number of updated parity elements. Suppose there are w parity elements and n data elements in a stripe. Let the number of data elements in the first i parity chains be n_i ($n_i \leq n$). P_i is the number of updated parity elements when writing all the n_i data elements in the first i parity chains. Our objective can be formulated as:

$$\text{Minimize} \quad P_i. \tag{1}$$

Our second observation is that P_i will be influenced by the generation orders of parity elements. Two examples that have different parity generation orders in X-Code are shown in Figs. 7 and 8 respectively, both of which are under the anti-diagonal data placement.[3] Fig. 7 indicates that the generation of the first 2 parity elements $\{R_1, R_2\}$ (i.e., $i = 2$) will place data elements {#1, #2, ..., #6} (i.e., $n_2 = 6$). A write operation to these placed data elements will renew $P_2 = 7$ parity elements. As a comparison, Fig. 8 generates the first 2 parity elements in the order of $\{R_1, R_4\}$, and consequently updates 6 parity elements (i.e., $P_2 = 6$) for the same write operation. Thus the parity generation order should be carefully designed to optimize partial stripe writes across parity chains.

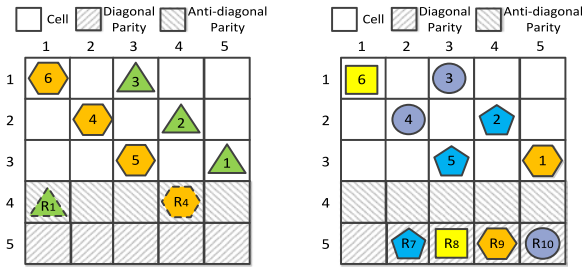
Position adjustment of data elements. The partial stripe writes can be further optimized by adjusting the positions of data elements, so that any two continuous data elements will always associate with a common parity element. This arrangement will further decrease the updates to parity elements, especially for the small write operations.

For instance, Fig. 8 gives the data placement before position adjustment, where two continuous data elements #3 and #4 do not in a common parity chain, thus a write operation to them will update 4 parity elements (i.e., R_1, R_4, R_8, and R_9). A new data placement after adjustment is shown in Fig. 9. The data elements #3 and #4 in this layout share the common diagonal parity element R_{10} (shown in Fig. 9b), and thus a write operation to them will renew only 3 parity elements (i.e., R_1, R_4, and R_{10}).

4 Parity-Switched Data Placement

From the motivation of Section 3, we propose a new data placement method called **P**arity-Switched **D**ata **P**lacement.

3. The encoding principle of X-Code can be reviewed in Fig. 3.



(a) The Anti-diagonal Parity (b) The Diagonal Parity

Fig. 9. Position adjustment of data elements. The layout of data elements after adjustment. The dashed lines indicate the parity elements generated by continuous data elements.

PDP includes two algorithms. The first algorithm explores the generation orders of parity elements, where continuous data elements will be placed by following the protogenic parity generation principle to calculate the parity elements. The second algorithm tries to make any two continuous data elements share a common parity element.

4.1 Determination of Parity Generation Orders

To obtain the optimal generation orders of parity elements that satisfy Equation (1), enumerating the orders has a high computational complexity. For example, for X-Code, its stripe has $2p$ parity elements;[4] thus, there will be $(2p)!$ permutations to seek the optimal parity generation orders. Therefore, we propose to explore the parity generation orders based on the greedy approach [37] as shown in Algorithm 1. Its *main idea* is to greedily select a parity element and generate it in the ith step ($1 \leq i \leq w$), such that the generation will cause the least parity elements to update when writing all the data elements in the first i parity chains.

Details of Algorithm 1. To place the data elements, we first assume that all the cells are initially blank and not placed with data elements. Suppose there are w parity elements and n data elements. Let $\mathcal{X}$ keep the parity elements that are finally selected to be generated by continuous data elements in this algorithm. $\mathcal{X}$ is first initialized as an empty set and gradually appended with one parity element after each greedy selection. Once a parity element is appended to $\mathcal{X}$, continuous data elements will then be placed on the corresponding cells to generate it. Suppose $|\mathcal{X}|$ denotes the number of parity elements in $\mathcal{X}$. The physical meaning of $\mathcal{X}$ is the optimal order of the first $|\mathcal{X}|$ data elements generated by continuous data elements, such that the minimal number of parity updates will be caused when writing all the data elements in the first $|\mathcal{X}|$ parity chains.

We then use $\mathcal{M}$ to include the updated parity elements when writing all the data elements in the first $|\mathcal{X}|$ parity chains, and suppose $|\mathcal{M}|$ represents the number of parity elements in $\mathcal{M}$. Then the objective of Algorithm 1 is to determine $\mathcal{X}$, such that $|\mathcal{M}|$ is minimized.

In the initialization, both $\mathcal{X}$ and $\mathcal{M}$ are first set as empty sets, and $\mathcal{U}$ includes all the w parity elements (step 1). In each iteration, the algorithm scans each candidate parity element $R_i \in \mathcal{U}$. If continuous data elements are placed on the corresponding cells to generate R_i, then $\mathcal{Y}_i$ denotes the parity elements to update when writing all the data elements in the parity chain of R_i (step 3). Therefore, if R_i is chosen to generate and appended to $\mathcal{X}$ in this iteration, then $\mathcal{M} \cup \mathcal{Y}_i$ includes the related parity elements to renew when completely writing all the data elements in the first $(|\mathcal{X}| + 1)$ parity chains. We record the number of parity elements in $\mathcal{M} \cup \mathcal{Y}_i$ (step 4). We then select the one R_j among the remaining candidates in $\mathcal{U}$, whose generation will result in the fewest parity updates when writing all the data elements in the first $(|\mathcal{X}| + 1)$ parity chains (step 5). After this selection, $\mathcal{X}, \mathcal{U}$, and $\mathcal{M}$ will be accordingly updated (step 6). To calculate R_j, the continuous data elements starting at the data element ranked first among the unplaced data elements, will be laid on the corresponding cells (step 7). The iteration will repeat until the generation of parity elements in $\mathcal{X}$ successfully places all the data elements (step 8). Finally, the remaining parity elements in $\mathcal{U}$ will be produced to complete the encoding process (step 9).

Algorithm 1. Parity Generation Orders

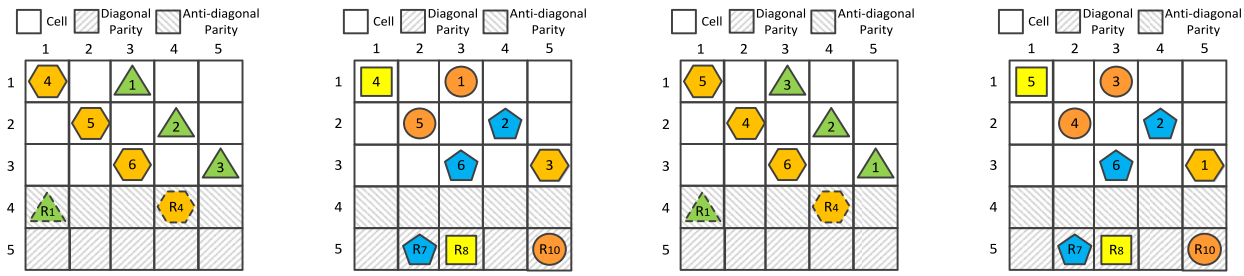
1 Set $\mathcal{X} = \emptyset, \mathcal{M} = \emptyset, \mathcal{U}$ includes w parity elements
2 **for** *each candidate parity element* $R_i \in \mathcal{U}$ **do**
3 Obtain $\mathcal{Y}_i$
4 Calculate $|\mathcal{M} \cup \mathcal{Y}_i|$
5 Find R_j, where $|\mathcal{M} \cup \mathcal{Y}_j| = \text{Min}\{|\mathcal{M} \cup \mathcal{Y}_i| | R_i \in \mathcal{U}\}$
6 $\mathcal{X} = \mathcal{X} \cup \{R_j\}, \mathcal{U} = \mathcal{U} - \{R_j\}, \mathcal{M} = \mathcal{M} \cup \mathcal{Y}_j$
7 Place next continuous data elements to generate R_j
8 Repeat step 2∼7 until all the data elements are placed
9 Generate the remaining parity elements in $\mathcal{U}$
10 **Return** $\mathcal{X}$

An example. We take X-Code with $p = 5$ as an example. In the initialization, both of $\mathcal{X}$ and $\mathcal{M}$ are set as empty sets, and $\mathcal{U} = \{R_1, R_2, \ldots, R_{10}\}$. We first consider R_1 (shown in Fig. 7a). Generating R_1 will place continuous data elements {#1, #2, #3} on the cells $\{C_{1,3}, C_{2,4}, C_{3,5}\}$. A write operation to them will renew four associated parity elements (i.e., $\mathcal{Y}_1 = \{R_1, R_7, R_9, R_{10}\}$[5]). Therefore, if we select R_1 as the first parity element to generate in $\mathcal{X}$, then $|\mathcal{M} \cup \mathcal{Y}_1| = 4$. It means that four parity elements will be updated when all the data elements in the first 1 parity chain are written. Following the same rule, we will calculate $|\mathcal{M} \cup \mathcal{Y}_i| = 4$ for $1 \leq i \leq 10$. Without loss of generality, we select R_1 as the first parity element to generate by placing {#1, #2, #3}, and append R_1 in $\mathcal{X}$. We then update $\mathcal{M} = \{R_1, R_7, R_9, R_{10}\}$ and $\mathcal{U} = \{R_2, R_3, \ldots, R_{10}\}$.

In the second run, we first consider R_2, whose generation will lay {#4, #5, #6} on the cells $\{C_{1,4}, C_{2,5}, C_{3,1}\}$ as shown in Fig. 7a. Writing all the data elements in the parity chain of R_2 will renew four related parity elements (i.e., $\mathcal{Y}_2 = \{R_2, R_6, R_8, R_{10}\}$). Therefore, if we select R_2 as the second parity element to generate, then writing all the data elements in the first 2 parity chains will update $|\mathcal{M} \cup \mathcal{Y}_2| = 7$ parity elements. Following this method, we then test other candidate parity elements in $\mathcal{U}$. For $R_4 \in \mathcal{U}$, its generation will place {#4, #5, #6} on $\{C_{1,1}, C_{2,2}, C_{3,3}\}$ as shown in Fig. 8. When writing all the data elements in the parity chain

4. p is a prime number, and is usually a parameter to configure the stripe size for RAID-6 codes.

5. R_1 is the anti-diagonal parity element associated with the three data elements (see Fig. 7a), while R_7, R_9, and R_{10} are diagonal parity elements associated with the three placed data elements (see Fig. 7b).



(a) Two generated parity elements R_1 and R_4.

(b) #3 and #4 share none common parity element.

(c) Adjust the positions of data elements.

(d) #3 and #4 have a common parity element R_{10}.

Fig. 10. An example of adjustment of data element positions. The shape with dashed line denotes the generated parity element in $\mathcal{X}$.

of R_4, the set of related parity elements is $\mathcal{Y}_4 = \{R_4, R_7, R_8, R_{10}\}$. Therefore, if we select R_4 as the second parity element to generate, then writing all the data elements in the first 2 parity chains should update $|\mathcal{M} \cup \mathcal{Y}_4| = 6$ ($< |\mathcal{M} \cup \mathcal{Y}_2| = 7$) parity elements (shown in Fig. 8). Finally, we update $\mathcal{X} = \{R_1, R_4\}$, $\mathcal{U} = \{R_2, R_3, R_5, R_6, \ldots, R_{10}\}$ and $\mathcal{M} = \{R_1, R_2, R_7, R_8, R_9, R_{10}\}$.

The iteration will repeat until all the data elements are placed. Finally, we can obtain $\mathcal{X} = \{R_1, R_4, R_2, R_3, R_5\}$ that are generated by continuous data elements, and then calculate the remaining parity elements in $\mathcal{U} = \{R_6, R_7, \ldots, R_{10}\}$ based on the placed data elements.

4.2 Positions Adjustment of Data Elements

Given the data layout derived from Algorithm 1, any two continuous data elements will either have a common parity element (e.g., #2 and #3 in Fig. 8a have the common parity element R_1) or associate with two parity elements in $\mathcal{X}$ whose generation orders are adjacent. For example, as shown in Fig. 8a, the continuous data elements #3 and #4 are in the parity chain of R_1 and R_4 whose generation orders are adjacent (refer to the example of Algorithm 1). For the second case, these two data elements may not have a common parity element and a write operation to them will cause many extra updates to parity elements. Targeting at this problem, Algorithm 2 proposes to further adjust the positions of data elements, such that any two continuous data elements will associate with a common parity element.

Algorithm 2. Adjustment of Data Element Positions

1 Set R_{cur} as the first parity element in $\mathcal{X}$
2 Find the next parity element $R_{nxt} \in \mathcal{X}$ after R_{cur}
3 **for** *each data element* `#i` *in the parity chain of* R_{cur} **do**
4 **for** *each data element* `#j` *in the parity chain of* R_{nxt} **do**
5 **if** *they are not fixed and in a common parity chain* **then**
6 Perform the position adjustments
7 Fix the two continuous data elements
8 Set $R_{cur} = R_{nxt}$
9 Repeat Step 2∼8 until R_{cur} is the last parity element in $\mathcal{X}$

Details of Algorithm 2. The algorithm starts from the first parity element in $\mathcal{X}$ and ends at the last one. Let R_{cur} be the currently generated parity element in $\mathcal{X}$, and $R_{nxt} \in \mathcal{X}$ be the next parity element to be produced after R_{cur} (step 1∼2). As the generation orders of R_{cur} and R_{nxt} are adjacent in $\mathcal{X}$ and data elements are placed continually in order when generating R_{cur} and R_{nxt}, the last data element in the parity chain of R_{cur} and the first data element in the parity chain of R_{nxt} are continuous. For example, R_1 and R_4 are two parity elements that are adjacently generated (shown in Fig. 8). The last data element in the parity chain of R_1 is #3 and the first data element in the parity chain of R_4 is #4. These two data elements are continuous. Therefore, the next step is to find the appropriate cells to place these two data elements, such that they will associate with a common parity element. Once these two continuous data elements are adjusted, they will be fixed and not be moved in next steps.

The algorithm then scans every pair of data elements (`#i`, `#j`), where `#i` and `#j` are in the parity chains led by R_{cur} and R_{nxt}, respectively (step 3∼4). If the data elements `#i` and `#j` are not fixed (i.e., they can be moved to other cells) and join a common parity element's generation (step 5), we then switch the position of `#i` with that of the last data element in the parity chain of R_{cur}, and swap the position of `#j` with that of the first data element in the parity chain of R_{nxt} (step 6). Thus the last data element in the parity chain of R_{cur} and the first data element in the parity chain of R_{nxt} will associate with a common parity element. Subsequently, these two continuous data elements will be fixed and not allowed to be moved in next steps (step 7). We then turn to the next parity element R_{nxt} (step 8). The loop will terminate when R_{cur} is the last parity element in $\mathcal{X}$ (step 9).

An example. We take X-Code with $p = 5$ as an example in Fig. 10. In Fig. 10a, as referred in Algorithm 1, R_1 and R_4 are the first two parity elements to be generated in $\mathcal{X}$. We first set $R_{cur} = R_1$ and $R_{nxt} = R_4$, and scan every pair of data elements in these two parity chains. Fig. 10b shows that data element #1 (i.e., `#i` in Algorithm 2) and #5 (i.e., `#j`) have a common parity element R_{10}. We then perform the movement by switching #1with the last data element in the parity chain of R_1 (i.e., #3) and swapping #5 with the first data element in the parity chain of R_4 (i.e., #4). Thus the two continuous data elements (i.e., #3 and #4) will still associate with a common parity element (i.e., R_{10}) as shown in Fig. 10d. After the adjustment, #3 and #4are not allowed to be moved in next position adjustments.

4.3 Enhancements

PDP also provides some potential enhancements, which are helpful to further pursue a better data placement layout.

1) We can also try more candidate parity elements that are considered to be included in $\mathcal{X}$. For example, we can try other candidate parity elements in Algorithm 1 when they have the same value of $|\mathcal{M} \cup \mathcal{Y}_i|$.

TABLE 2
An Analysis on Selected MSR Cambridge Traces

Traces	rsrch_1	rsrch_2	src2_1	wdev_2	wdev_3	web_3
Function	Research project	Research project	Source control	Test web server	Test web server	Web/SQL server
Num. of Write Operations	13,738	71,222	14,104	181,077	671	21,330
Average Write Size	12.2 KB	4.3 KB	13.4 KB	8.1 KB	4.4 KB	20.9 KB

2) Besides making any two data elements share a common parity element, we can also make the data elements which have the same parity element close in order. For example, we can further exchange the positions of #5 and #6 (i.e., move #5 to the cell $C_{3,3}$ and shift #6 to $C_{1,1}$) in Fig. 10d, and then #5 and #2 will have a common parity element R_7. As a consequence, less updates to parity elements will be caused when writing {#2, #3, #4, #5}.
3) We can also define the granularity when optimizing the partial stripe writes across parity chains. For example, we can set the granularity to be 2. In this example, the objective will be changed to reduce the total number of updated parity elements when writing two parity elements whose generation orders are adjacent.

4.4 Complexity Analysis

Computation complexity. Suppose there are w parity elements and n data elements. As Algorithm 1 requires to scan every parity element, its complexity is $O(w)$. For Algorithm 2, as the number of data elements in a parity chain is much less than n, the complexity of step 2$\sim$7 is no more than $O(n^2)$ and these steps will repeat at most w times. Therefore, the complexity of Algorithm 2 is $O(wn^2)$. Overall, both of these two algorithms preserve polynomial complexity.

Storage overhead. After the placement of data elements is determined, to record the localization of data elements in a stripe, PDP merely needs to keep an extra mapping table. Suppose a stripe has r rows and c columns, then the mapping table is composed of the tuples (#ELEMENT_ID, POSITION_ID), indicating that the data element #ELEMENT_ID is stored in the cell $C_{\text{POSITION\_ID}/c,\text{POSITION\_ID}\%c}$. Suppose there are n data elements in a stripe, then PDP only keeps n tuples and thus the storage complexity is $O(n)$.

5 Performance Evaluation

In this section, we mainly evaluate the efficiency improvement brought by PDP when applying it over the following representative XOR-based erasure codes.

- EVENODD Code [25] (over $p+2$ disks). EVENODD Code has horizontal parity chains and is a popular RAID-6 code in storage applications. We would like to know if PDP can further improve the performance of partial stripe writes for the codes with horizontal parity chains.
- X-Code [10] (over p disks). It is another impressive RAID-6 code constructed over diagonal parity chains and anti-diagonal parity chains. This evaluation will show the effect when applying PDP to the codes that do not have horizontal parity chains.
- HDP Code [7] (over $p-1$ disks). It is a recent RAID-6 code proposed for load balancing. It consists of horizontal parity chains and anti-diagonal parity chains.
- STAR Code [33] (over $p+3$ disks). It tolerates triple disk failures. We select it to show whether PDP can have effect on the codes with higher fault tolerance.

5.1 Evaluation Setup

In the tests, we choose $p=7$, which is used to configure the number of disks in a stripe. Under this value, the number of disks in the stripe of EVENODD Code, X-Code, HDP Code, and STAR Code is 9, 7, 6, and 10, respectively. This setting is similar with the configured parameters in many well-known erasure-coded storage systems [2], [38], [39]. For example, the number of disks in the stripe of GFS II [38] is 6. As the working principle of PDP is independent with the scale of storage systems, we believe that PDP will also sustain its effect when the storage systems expand. The test is run on a Linux server with an X5472 processor and 8 GB memory. The operating system is SUSE Linux Enterprise Server and the filesystem is EXT3. The deployed disk array consists of 15 Seagate/Savvio 10K.3 SAS disks, each of which has 300 GB storage capability and 10,000 rmp. The machine and the disk array are connected by a Fiber cable with the bandwidth of 800 MB/sec. These codes are realized based on Jerasure 1.2 [40], a widely-used library to realize erasure coding storage systems.

We select several real block-level traces with various access characteristics from the repository of MSR Cambridge Traces [18]. These traces are collected from 36 volumes consisting 179 disks on 13 servers for one week. We extract write patterns of each trace. Each write pattern describes the offset to trigger the write operation and the size of written data. We present their characteristics in Table 2.

We configure the disk array as a JBOD [41]. The element size is set as 4 KB, which is consistent with the block size in the disk. We then generate the data elements. For each code, the data elements are dispersed under the horizontal data placement, the vertical data placement, and our proposed PDP, respectively. Finally, we encode them by using every evaluated code.

For each trace, we strictly replay its write patterns to the data elements at the specified offsets and operate the same sizes of data. Every trace is repeated for five runs and we record the average results.

5.2 Storage Overhead to Maintain the Mapping Table

In this experiment, we vary the selection of p and record the storage space that a mapping table requires. The results are presented in Table 3. We can observe that the additional storage overhead caused by the mapping table is marginal. Take STAR Code as an example, the mapping table will

TABLE 3
Storage Overhead to Maintain A Mapping Table

Codes	EVENODD Code	X-Code	HDP Code	STAR Code
$p = 5$	0.06 KB	0.08 KB	0.03 KB	0.08 KB
$p = 7$	0.16 KB	0.14 KB	0.07 KB	0.16 KB
$p = 11$	0.43 KB	0.39 KB	0.31 KB	0.43 KB
$p = 13$	0.61 KB	0.56 KB	0.47 KB	0.61 KB

cause merely 0.61 KB when $p = 13$. Therefore, we believe that the storage overhead of PDP is acceptable.

5.3 Total Number of Write Operations

Suppose a trace includes N_p partial stripe write operations. For the ith ($1 \leq i \leq N_p$) partial stripe write operation, assume the number of write operations (i.e., the elements to be updated in a write pattern) is W_i, and then the total number of write operations is $W_{sum} = \sum_{i=1}^{N_p} W_i$. We replay the six traces and record the total number of elements to be updated in every trace. The final results are shown in Fig. 11.

We have two observations from Fig. 11. First, the vertical data placement usually introduces the most write operations. This is because continuous data elements in the same column usually do not have any common parity element, making the vertical data placement be less effective in partial stripe write optimization.

Second, for each code, PDP triggers the fewest write operations among the three data placements for all the evaluated traces. This result also demonstrates the generality of PDP to decrease the number of updated parity elements in partial stripe writes. Take the trace wdev_3 as an example, the reduction on the total number of write operations brought by PDP will be up to 19.2 percent (compared with the horizontal data placement) and 31.9 percent (compared with the vertical data placement).

5.4 Write Speed

Suppose the write speed under PDP is S_p and the write speed under the horizontal (resp. vertical) data placement is S_h (resp. S_v), and then the improvement brought by PDP is $I_{ph} = \frac{S_p}{S_h}$ compared with the horizontal data placement, and $I_{pv} = \frac{S_p}{S_v}$ compared with the vertical data placement. For each code, we replay the six traces and record the completion time. To clearly illustrate the gap among these three placement methods, we normalize the write speed under the horizontal data placement as 1. Besides, we also illustrate the error bars of the evaluation results as shown in Fig. 12.

We can derive two observations from Fig. 12. First, vertical data placement usually needs the most time to complete a write pattern in most cases. For example, when replaying the trace wdev_3 to the data encoded by EVENODD Code, the write speed of vertical data placement is 19.7 and 28.4 percent slower than those of horizontal data placement and PDP, respectively.

Second, PDP significantly accelerates the write operations compared with both the horizontal data placement and the vertical data placement. For example, when replaying the trace rsrch_2 to the data encoded by STAR Code, the write speed of PDP is 38.9 percent (resp. 59.8 percent) faster than that of the horizontal data placement (resp. the vertical data placement). The error bars also demonstrate that PDP will reach a faster write speed compared with the horizontal/vertical data placement for most cases.

5.5 Write Operations on the Most Loaded Disk

We also evaluate these three placement methods by measuring the number of write operations on the most loaded disk. Suppose the storage system consists of N_d disks and the ith disk serves Q_i write operations during the trace execution. This metric can be obtained by $Q_{max} = Max\{Q_i | 1 \leq i \leq N_d\}$. The results are shown in Fig. 13.

We can find that the vertical data placement usually causes the maximum number of write operations on the most loaded disk, especially for the traces rsrch_2 and wdev_3 whose average write sizes are around 4 KB. This is

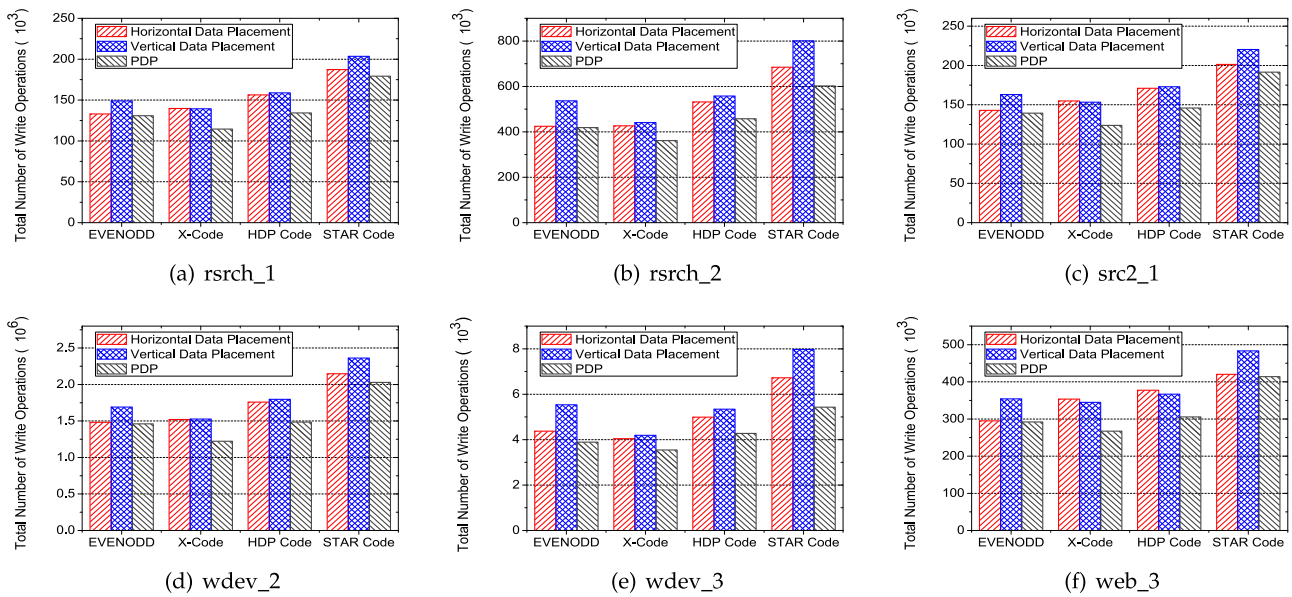


Fig. 11. The total number of write operations for evaluated traces. The smaller value indicates lighter load on the disks.

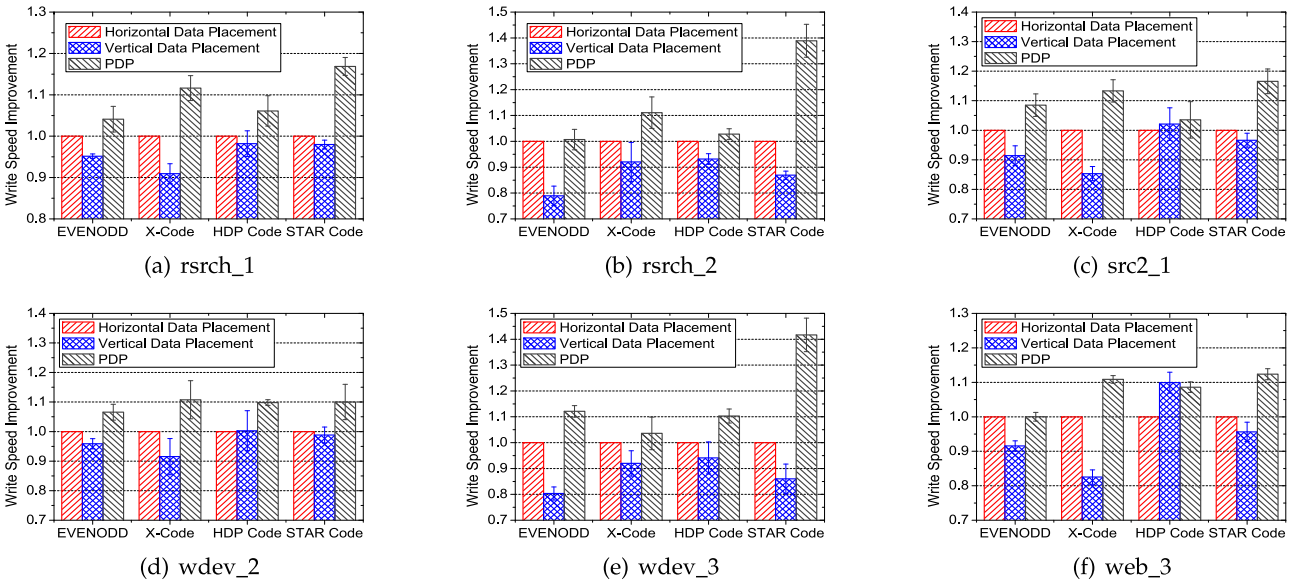


Fig. 12. The write speed improvement. The larger value indicates the faster write speed.

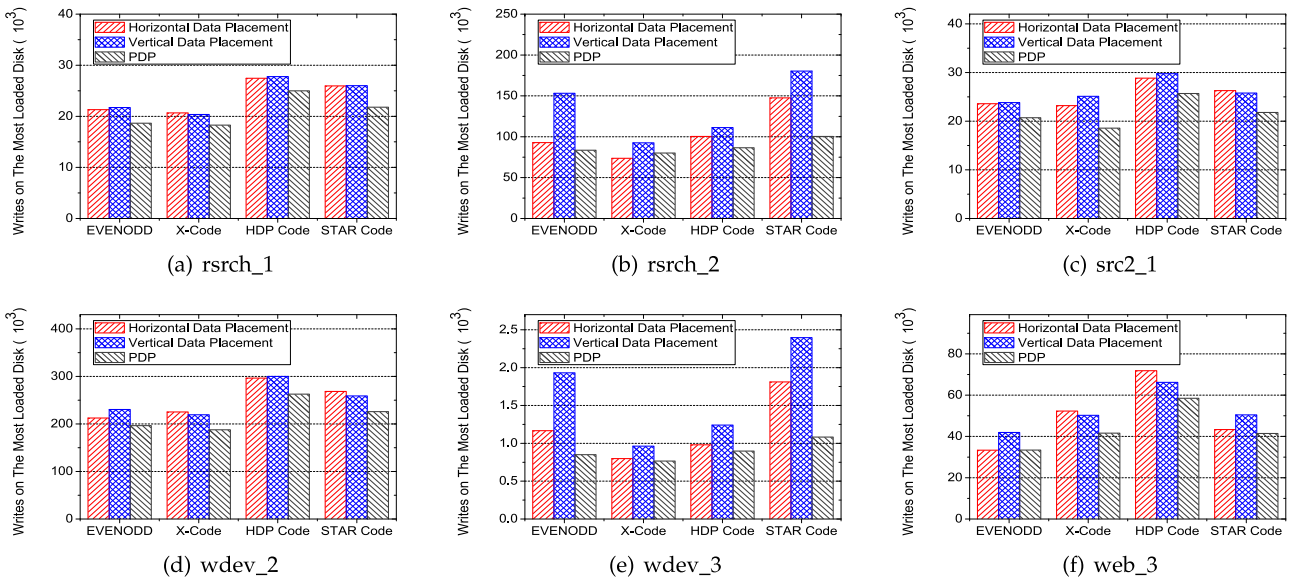


Fig. 13. The number of write operations on the most loaded disk. The smaller value indicates the less write operations on the most loaded disk.

TABLE 4
A Brief Comparison among Various Data Placement Methods

Data Placement	Parallel Access Support	Effect on Partial Stripe Write Optimization
Horizontal Data Placement	**Yes**	Works for the codes with horizontal parity chains
Vertical Data Placement	No	Easily introduce many parity updates for small writes
PDP	**Yes**	**Works for any XOR-based erasure code**

because in vertical data placement, the written data elements may usually lay in the same disk especially for the case with a small number of written data elements.

Moreover, PDP can effectively lighten the write burden on the most loaded disk. For example, in the trace wdev_3, compared with the vertical data placement (resp. the horizontal data placement), PDP decreases up to 56.0 percent of write operations (resp. 40.2 percent of write operations) on the most loaded disk.

5.6 Comparison

Table 4 summarizes the comparison among PDP, the horizontal data placement, and the vertical data placement. The comparison indicates that PDP is more general for partial

stripe write optimization and it also supports parallel access, which also make its read performance of PDP similar with that of the horizontal data placement.

6 Conclusion

In this paper, we propose a Parity-Switched Data Placement scheme to optimize partial stripe writes for any XOR-coded storage system. To decrease parity updates, PDP respects the parity generation principles of each code and selects parity elements to generate by using continuous data elements. In addition, PDP also explores further optimization by investigating parity generation orders and makes any two continuous data elements share a common parity element. Evaluations show that PDP reduces up to 31.9 percent of write operation and increases the write speed by up to 59.8 percent.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61232003, 61327902), and the state Key Laboratory of High-end Server and Storage Technology (Grant No. 2014HSSA02). Jiwu Shu is the corresponding author of this paper.

References

[1] M. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Proc. IEEE/IFIP DSN*, 2005.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, p. 2.

[3] Z. Shen, J. Shu, and Y. Fu, "Seek-efficient i/o optimization in single failure recovery for xor-coded storage systems," in *Proc. IEEE 34th Symp. Rel. Distrib. Syst.*, 2015, pp. 228–237.

[4] Y. Fu, J. Shu, Z. Shen, and G. Zhang, "Reconsidering single disk failure recovery for erasure coded storage systems: Optimizing load balancing in stack-level," *IEEE Trans. Parallel Distrib. Syst.*, to appear.

[5] Z. Shen, J. Shu, and Y. Fu, "Hv code: An all-around mds code for raid-6 storage systems," *IEEE Trans. Parallel Distrib. Syst.*, to appear.

[6] G.-L. Feng, R. H. Deng, F. Bao, and J.-C. Shen, "New efficient mds array codes for raid. part i. reed-solomon-like codes for tolerating three disk failures," *IEEE Trans. Comput.*, vol. 54, no. 9, pp. 1071–1080, Sep. 2005.

[7] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie, "Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw.*, 2011, pp. 209–220.

[8] C. Wu, S. Wan, X. He, Q. Cao, and C. Xie, "H-code: A hybrid mds array code to optimize partial stripe writes in raid-6," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 782–793.

[9] Z. Shen and J. Shu, "Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 550–561.

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

[11] J. J. Wylie and R. Swaminathan, "Determining fault tolerance of xor-based erasure codes efficiently," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw*, 2007, pp. 206–215.

[12] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu, "On the speedup of single-disk failure recovery in xor-coded storage systems: Theory and practice," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.

[13] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-code: A new raid-6 code with optimal properties," in *Proc. 23rd Int. Conf. Supercomput.*, 2009, pp. 360–369.

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

[15] D. Panchigar. (2009). Emc symmetrix dmxłraid 6 implementation [Online]. Available: http://storagenerve.com/2009/02/27/emc-symmetrix-dmx-raid-6-implementation/

[16] C. Lueth, "Raid-dp: Network appliance implementation of raid double parity for data protection," NetApp Inc., Tech. Rep. TR-3298, 2004.

[17] G. Zhang, G. Wu, S. Wang, J. Shu, W. Zheng, and K. Li, "Caco: An efficient cauchy coding approach for cloud storage systems," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 435–447, 2016.

[18] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, p. 10, 2008.

[19] R. A. DeMoss and K. B. DuLac, "Delayed initiation of read-modify-write parity operations in a raid level 5 disk array," U.S. Patent 5,388,108, Feb., 1995.

[20] A. Thomasian, "Reconstruct versus read-modify writes in raid," *Inf. Process. Lett.*, vol. 93, no. 4, pp. 163–168, 2005.

[21] W. Xiao, J. Ren, and Q. Yang, "A case for continuous data protection at block level in disk array storages," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 6, pp. 898–911, Jun. 2009.

[22] H. Jin, X. Zhou, D. Feng, and S. Zhang, "Improving partial stripe write performance in raid level 5," in *Proc. IEEE Int. Caracas Conf. Devices, Circuits Syst.*, 1998, pp. 396–400.

[23] E. D. Neufeld, "Method for improving partial stripe write performance in disk array subsystems," U.S. Patent 5,333,305, Jul. 26, 1994.

[24] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2010, pp. 119–130.

[25] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.

[26] A. Goel and P. Corbett, "Raid triple parity," *ACM SIGOPS Oper. Syst. Rev.*, vol. 46, no. 3, pp. 41–49, 2012.

[27] J. Bonwick. (2005, Nov.). Raid-z. [Online]. Available: http://blogs.sun.com/bonwick/entry/raid_z

[28] J. Bonwick and B. Moore. (2007). Zfs: The last word in file systems. [Online]. Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf

[29] C. Pugh, T. Carrol, and P. Henderson, "Ensuring high availability and recoverability of acquired data," in *Proc. IEEE/NPSS 24th Symp. Fusion Eng.*, 2011, pp. 1–5.

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

[31] J. S. Plank and M. Blaum, "Sector-disk (sd) erasure codes for mixed failure modes in raid systems," *ACM Trans. Storage*, vol. 10, no. 1, p. 4, 2014.

[32] Y. Fu and J. Shu, "D-code: An efficient raid-6 code to optimize i/o loads and read performance," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 603–612.

[33] C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 889–901, Jul. 2008.

[34] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," International Computer Science Institute, Berkeley, California, Tech. Rep. TR-95-048, 1995.

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

[36] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Newslett. ACM SIGMOD Rec.*, vol. 17, no. 3, pp. 109–116, 1988.

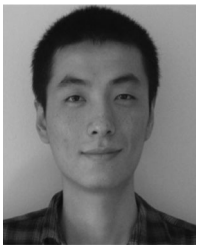
[37] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence.* Englewood Cliffs, NJ, USA: Prentice-Hall, 1995.

[38] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Oper. Syst. Des. Implementation*, 2010, pp. 61–74.

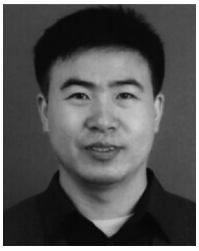
[39] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "Hdfs raid," *Tech talk. Yahoo Developer Netw.*, 2010.

[40] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08-627, vol. 23, 2008.

[41] L.-Y. Liu, T.-H. Lee, M. Schnapp, Y.-H. Wang, and C.-H. Pao, "Jbod subsystem and external emulation controller thereof," U.S. Patent App. 10/709,718, 2004.



Zhirong Shen receives the bachelor's degree from the University of Electronic Science and Technology of China. He is currently working toward the PhD degree from the Department of Computer Science and Technology at Tsinghua University. His current research interests include storage reliability and storage security.



Jiwu Shu received the PhD degree in computer science from Nanjing University in 1998, and finished the postdoctoral position research at Tsinghua University in 2000. Since then, he has been teaching at Tsinghua University. His current research interests include storage security and reliability, nonvolatile memory-based storage systems, and parallel and distributed computing. He is a member of the IEEE.

Yingxun Fu received the bachelor's degree from North China Electric Power University in 2007, the master's degree from the Beijing University of Posts and Telecommunications in 2010, and the doctor's degree from Tsinghua University in 2015. His current research interests include storage reliability and distributed systems.