# Correlation-Aware Stripe Organization for Efficient Writes in Erasure-Coded Storage: Algorithms and Evaluation

Zhirong Shen, *Member, IEEE*, Patrick P. C. Lee, *Senior Member, IEEE*, Jiwu Shu, *Fellow, IEEE*, and Wenzhong Guo, *Member, IEEE*

**Abstract**—Erasure coding has been extensively employed for data availability protection in production storage systems by maintaining a low degree of data redundancy. However, how to mitigate the parity update overhead of partial stripe writes in erasure-coded storage systems is still a critical concern. In this paper, we study this problem from two new perspectives: data correlation and stripe organization. We propose CASO, a *correlation-aware stripe organization* algorithm, which captures data correlation of a data access stream and uses the data correlation characteristics for stripe organization. It packs correlated data into a small number of stripes to reduce the incurred I/Os in partial stripe writes, and further organizes uncorrelated data into stripes to leverage the spatial locality in later access. We implement CASO over Reed-Solomon codes and Azure's Local Reconstruction Codes, and show via extensive trace-driven evaluation that CASO reduces up to 29.8 percent of parity updates and reduces the write time by up to 46.7 percent.

**Index Terms**—Correlation, stripe organization, partial stripe writes, erasure code

## 1 Introduction

Today's distributed storage systems continuously expand in scale to cope with the ever-increasing volume of data storage. In the meantime, failures also become more prevalent due to various reasons, such as disk crashes, sector errors, or server outages [3], [4], [5]. To achieve data availability, keeping additional redundancy in data storage is a commonly used approach to enable data recovery once failures occur. Two representatives of redundancy mechanisms are *replication* and *erasure coding*. Replication distributes identical replicas of each data copy across storage devices, yet it significantly incurs substantial storage overhead, especially in the face of massive amounts of data being handled nowadays. On the other hand, erasure coding introduces much less storage redundancy via encoding computations, while reaching the same degree of fault tolerance as replication [6]. At a high level, erasure coding performs encoding by taking a group of original pieces of information (called *data chunks*) as input and generating a small number of redundant pieces of information (called *parity chunks*), such that if any data or parity chunk fails, we can still use a subset of available chunks to recover the lost chunk. The collection of data and parity chunks that are encoded together forms a *stripe*, and a storage system stores multiple stripes of data and parity chunks for large-scale storage. Because of the high storage efficiency and reliability, erasure coding has been widely deployed in current production storage systems, such as Windows Azure Storage [2] and Facebook's Hadoop Distributed File System [7].

However, while providing fault tolerance with low redundancy, erasure coding introduces additional performance overhead as it needs to maintain the consistency of parity chunks to ensure the correctness of data reconstruction. One typical operation is *partial stripe writes* [8], in which a subset of data chunks of a stripe are updated. In this case, the parity chunks of the same stripe also need to be renewed accordingly for consistency. In storage workloads that are dominated by small writes [9], [10], partial stripe writes will trigger frequent accesses and updates to parity chunks, thereby amplifying I/O overhead and extending the time of write operation. Partial stripe writes also raise concerns for system reliability, as different kinds of failures (e.g., system crashes and network failures) may occur during parity renewals and finally result in the incorrectness of data recovery. Thus, making partial stripe writes efficient is critical for improving not only performance, but also reliability, in erasure-coded storage systems.

Our insight is that we can exploit *data correlation* [11] to improve the performance of partial stripe writes. Data chunks in a storage system are said to be *correlated* if they have similar semantic or access characteristics. In particular, correlated data chunks tend to be accessed within a short period of time with large probability [11]. By extracting data correlations from an accessed stream of data chunks, we can organize correlated data chunks (which are likely to be

- *Z. Shen and P.P.C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, 999077, Hong Kong. E-mail: zhirong.shen2601@gmail.com, pclee@cse.cuhk.edu.hk.*
- *J. Shu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: shujw@tsinghua.edu.cn.*
- *W. Guo is with the College of Mathematics and Computer Science, Fuzhou University, Fuzhou, China. E-mail: guowenzhong@fzu.edu.cn.*

*Manuscript received 11 Aug. 2018; revised 1 Nov. 2018; accepted 9 Dec. 2018. Date of publication 1 Jan. 2019; date of current version 12 June 2019.*
*(Corresponding author: Patrick P.C. Lee.)*
*Recommended for acceptance by W. Yu.*
*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.2018.2890635*

accessed simultaneously) into the same stripe, so as to reduce the number of parity chunks that need to be updated.

To this end, we propose CASO, a correlation-aware stripe organization algorithm. CASO carefully identifies correlated data chunks by examining the access characteristics of an access stream. It then accordingly classifies data chunks into either correlated or uncorrelated data chunks. For correlated data chunks, CASO constructs a correlation graph to evaluate their degrees of correlation and formulates the stripe organization as a graph partition problem. For uncorrelated data chunks, CASO arranges them into stripes by leveraging the spatial locality in future access.

CASO is applicable for general erasure codes, such as the classical Reed-Solomon (RS) codes [12], XOR-based codes [13], [14], [15], [16], [17], [18], and Azure's Local Reconstruction Codes (LRC) [2]. In addition, CASO complements previous approaches that optimize the performance of partial stripe writes at coding level [10], [17], [18] or system level [9], [19], and can be deployed on top of these approaches for further performance gains. To the best of our knowledge, CASO is the *first work* to exploit data correlation from real system workloads to facilitate stripe organization in erasure-coded storage, so as to mitigate the parity update overhead of partial stripe writes.

In summary, we make the following contributions.

- We carefully examine existing studies on optimizing partial stripe writes and identify the remaining open issues.
- We propose CASO to leverage data correlation in stripe organization for erasure-coded storage systems.
- We implement CASO over RS codes and Azure's LRC with different configurations and conduct extensive trace-driven testbed experiments. We show that CASO reduces up to 29.8 percent of parity updates and up to 46.7 percent of the average write time compared to the baseline stripe organization technique. Also, we show that CASO preserves the performance of degraded reads [20], which are critical recovery operations in erasure-coded storage. Furthermore, we show that CASO introduces only slight additional time overhead in stripe organization.

The source code of CASO is now available at *http://adslab.cse.cuhk.edu.hk/software/caso*.

The rest of this paper proceeds as follows. Section 2 presents the basics of erasure coding and reviews related work. Section 3 motivates our problem. Section 4 presents the detailed design of CASO. Section 5 evaluates CASO using trace-driven testbed experiments. Section 6 concludes the paper. In the digital supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2018.2890635, we also present the complexity analysis, the addressing issue of the trace replay in our experiments, and the future work.

## 2 Background and Related Work

### 2.1 Basics of Erasure Coding

We first elaborate the background details of erasure coding following our discussion in Section 1. An erasure code is typically constructed by two configurable parameters, namely $k$ and $m$. A $(k,m)$ erasure code transforms the original data into $k$ equal-size pieces of data information called *data chunks* and produces additional $m$ equal-size pieces of redundant information called *parity chunks*, such that these $k+m$ data and parity chunks collectively form a *stripe*. A storage system comprises multiple stripes, each of which is independently encoded and distributed across $k+m$ storage devices (e.g., nodes or disks). We say that a $(k,m)$ code is *Maximum Distance Separable (MDS)* if it ensures that *any* $k$ out of $k+m$ chunks of a stripe can sufficiently reconstruct the original $k$ data chunks, while incurring the minimum amount of storage redundancy among all possible erasure code constructions; that is, it can tolerate any loss of at most $m$ chunks with optimal storage efficiency.

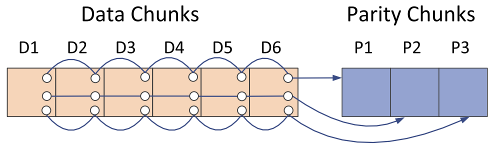



Fig. 1. Encoding of RS$(6,3)$ for a stripe, in which there are six data chunks and three parity chunks. If one of the data or parity chunks is lost, any six surviving chunks within the stripe can be used to reconstruct the lost chunk. Each line connects a parity chunk and its dependent data chunks in the encoding operation. For example, the first line connects the parity chunk $P_1$ with all six data chunks, meaning that $P_1$ is formed by the encoding of all the six data chunks.

Reed-Solomon codes [12] are one well-known family of MDS erasure codes that perform encoding operations based on Galois Field arithmetic [21]. RS codes support general parameters of $k$ and $m$, and have been widely deployed in production storage systems, such as Google [5] and Facebook [22]. In this paper, we denote the RS codes configured by the parameters $k$ and $m$ as RS$(k,m)$. Fig. 1 illustrates a stripe of RS$(6,3)$, in which there are six data chunks (i.e., $D_1 \sim D_6$) and three parity chunks (i.e., $P_1$, $P_2$, and $P_3$).

XOR-based codes are a special family of MDS codes that perform encoding using XOR operations only. Examples of XOR-based codes include RDP Code [13], X-Code [14], STAR Code [15], HDP Code [16], H-Code [17], and HV-Code [18]. XOR-based codes have higher computational efficiency than RS codes, but they often put restrictions on the parameters $k$ and $m$. For example, RDP Code and X-Code require $m=2$ and can only tolerate double chunk failures. XOR-based codes are usually used in local storage systems, such as EMC Symmetrix DMX [23] and NetApp RAID-DP [24].

Some recent studies (e.g., [2], [7]) focus on non-MDS codes that trade slight additional storage redundancy for repair efficiency. Local Reconstruction Codes [2] are one representative family of non-MDS codes deployed in Microsoft Azure. LRC keeps two types of parity chunks. In addition to the $m$ parity chunks (called the *global parity chunks* in LRC) derived from the $k$ data chunks, LRC further divides the $k$ data chunks of a stripe into $l$ local groups and maintains a parity chunk (called a *local parity chunk*) for each local group. By collectively keeping these two types of parity chunks, LRC is shown to significantly reduce I/Os in failure repair operations. In this paper, we denote the LRC configured by the parameters $k$, $l$, and $m$ as LRC$(k,l,m)$. Fig. 2 presents a stripe of LRC$(6,2,3)$, in which the six data chunks will be partitioned into two local groups (i.e., $\{D_1,D_2,D_3\}$ and

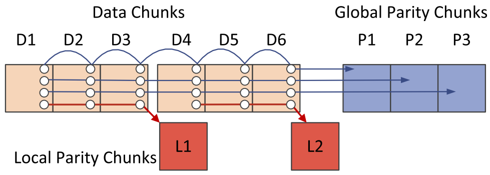



Fig. 2. Encoding of LRC$(6,2,3)$ for a stripe, in which there are six data chunks, two local parity chunks, and three global parity chunks. If one of the data chunks is corrupted, LRC can read the three surviving chunks within the same local group for data reconstruction.

$\{D_4, D_5, D_6\}$). Each local group generates a local parity chunk (i.e., $L_1$ and $L_2$ for the first and second local groups, respectively), and a stripe will maintain another three global parity chunks (i.e., $P_1$, $P_2$, and $P_3$). Suppose that the data chunk $D_1$ is corrupted. LRC$(6,2,3)$ can simply read the surviving three chunks (i.e., $D_2$, $D_3$, and $L_1$) from the first local group to repair $D_1$, thereby causing less repair traffic than the RS code in Fig. 1. Our work is applicable for RS codes, XOR-based codes, and Azure's LRC.

### 2.2 Partial Stripe Writes

Maintaining consistency between data and parity chunks is necessary during writes. Writes in erasure-coded storage systems can be classified into *full stripe writes* and *partial stripe writes* according to the write size. A full stripe write updates all data chunks of a stripe, so it generates all parity chunks from the new data chunks and overwrites the entire stripe in a single write operation. In contrast, a partial stripe write only updates a subset of data chunks of a stripe, and it must read existing chunks of a stripe from storage to compute the new parity chunks. Depending on the write size, partial stripe writes can be further classified into *read-modify-writes* for small writes and *reconstruct-writes* for large writes [25]. Since small writes dominate in real-world storage workloads [9], [10], we focus on read-modify-write mode, which performs the following steps when new data chunks are written: (i) reads both existing data chunks and existing parity chunks to be updated, (ii) computes the new parity chunks from the existing data chunks, new data chunks, and existing parity chunks via the linear algebra of erasure codes [9], and (iii) writes all the new data chunks and new parity chunks to storage. Clearly, the parity updates incur extra I/O overhead.

Extensive studies in the literature propose to mitigate parity update overhead. For example, H-Code [17] and HV Code [18] are new erasure code constructions that associate sequential data with the same parity information, so as to favor sequential access. Shen et al. [10] develop a new data placement that attempts to arrange sequential data with the same parity information for any given XOR-based code. Some approaches are based on parity logging [9], [19], which store parity deltas instead of updating parity chunks in place, so as to avoid reading existing parity chunks as in original read-modify-write mode.

### 2.3 Open Issues

When we examine existing studies on optimizing partial stripe writes, there remain two limitations.

*Negligence of Data Correlation.* Data correlation exists in real-world storage workloads [11]. Existing studies do not consider data correlation in erasure-coded storage systems, so they cannot fully mitigate parity update overhead. Specifically, if correlated data chunks are dispersed across many different stripes, then a write operation to those chunks will update all the parity chunks in multiple stripes. Note that some studies [10], [17], [18] favor sequential access, yet correlated data chunks may not necessarily be sequentially placed. Previous studies [11], [26], [27] exploit data correlation mainly to improve pre-fetching performance, but how to use this property to mitigate parity update overhead remains an open issue.

*Absence of an Optimization Technique for RS Codes and LRC.* Existing studies mainly focus on optimizing partial stripe writes for XOR-based codes [10], [17], [18]. Nevertheless, XOR-based codes often put specific restrictions on the coding parameters, while today's production storage systems often deploy RS codes or LRC for general fault tolerance (see Section 2.1). Thus, optimizing partial stripe writes for RS codes and LRC is still an imperative need.

## 3 Motivation

Many storage systems [28], [29] first keep new data in replication form and then encode the data after a period of time to maintain high storage efficiency. Since the popularity of the data being accessed tends to be stable in long term (e.g., hours or days) [30], we propose to capture the access correlations when the data is stored in replication form and improve the write efficiency when the data is later encoded by organizing the correlated data in the same stripe. Thus, for the applications with stationary access patterns, our proposed stripe organization remains effective in the long run. We pose the following question: *Given an access stream, how can we organize the data chunks into stripes based on data correlation, so as to optimize partial stripe writes?* In this section, we motivate our problem via trace analysis and an example.

### 3.1 Trace Analysis

We infer data correlation by a *black-box* approach, which finds correlated data chunks through analyzing a data access stream without requiring any modification to the underlying storage system [11]. We use two parameters to identify data correlation: *time distance* and *access threshold*. We say that *two data chunks are correlated if the number of times when they are accessed within a specific time distance reaches a given access threshold*.

To validate the significant impact of correlated data chunks in data accesses, we select several real-world block-level workloads from the MSR Cambridge Traces [31] (see Section 5 for details about the traces). Each trace includes a sequence of access requests, each of which describes the timestamp of a request (in terms of Windows filetime), the access type (i.e., read or write), the starting address of the request, and the size of the accessed data.

In this paper, we focus on improving the write efficiency. We assume that two data chunks are said to be correlated if *both of them are written by requests with the same timestamp value at least twice*. Note that the timestamp recorded in MSR Cambridge Traces is represented in units of ticks that correspond to 100-nanosecond intervals, yet the timestamp values in this analysis are rounded to the nearest 1,000. In

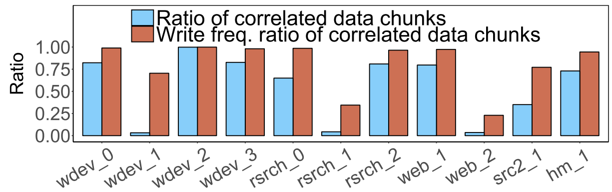



Fig. 3. Analysis on the real workloads about data correlation.

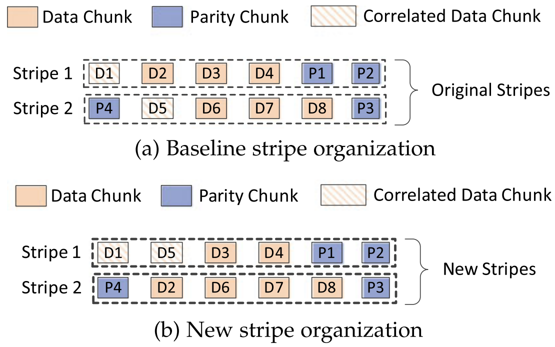



Fig. 4. Motivation: Two stripe organization methods.

other words, we set the time distance as 100 microseconds and the access threshold as two.

Let $n_c$ denote the number of correlated data chunks that we infer in a workload and let $f_c$ be the number of times written to these $n_c$ correlated data chunks over the entire workload. Suppose that $n_a$ denotes the number of all the distinct data chunks written in a workload, and $f_a$ represents the number of times written to these $n_a$ chunks in total. We consider *the ratio of the correlated data chunks* (denoted by $\frac{n_c}{n_a}$) and *the write frequency ratio of correlated data chunks* (denoted by $\frac{f_c}{f_a}$). We measure these two metrics in several selected workloads of the MSR Cambridge Traces, and the results are shown in Fig. 3. We make two observations.

- *The ratios of correlated data chunks vary across workloads.* For example, the ratio of correlated data chunks in wdev_2 is 99.9 percent and the ratio in wdev_1 is only 3.3 percent.
- *Correlated data chunks receive a number of writes.* For example, 70.5 percent of data writes are issued for correlated data chunks in wdev_1, while in wdev_2 the write frequency ratio of correlated data chunks reaches 99.9 percent.

In addition, previous work [32] reveals that most write requests usually access write-only data chunks. As correlated data chunks exhibit similar access characteristics, a write-only data chunk is expected to be more correlated to another data chunk that is also a write-only data chunk.

### 3.2 Motivating Example

Our trace analysis suggests that correlated data chunks receive a significant number of data accesses, and they tend to be accessed together. Thus, we propose to group correlated data chunks into the same stripes, so as to mitigate parity update overhead in partial stripe writes. We illustrate this idea via a motivating example. Fig. 4 shows two different stripe organization methods with RS$(4,2)$. Note that the placement of parity chunks is rotated across stripes to evenly distribute parity updates across the whole storage space, as commonly used in practical storage systems [33]. Thus, in Stripe 1, the last two chunks are parity chunks, while in Stripe 2, the parity chunks will be placed at the first and last column. Now, suppose that $D_1$ and $D_5$ are write-only data chunks and they are correlated. Fig. 4a shows a *baseline stripe organization* (BSO) methodology, which is considered for RS codes in the plugins of HDFS [34]. Specifically, BSO places sequential data chunks across $k+m$ storage devices in a round-robin fashion [35]. Suppose that the storage system caches the updates in the same time distance and flush them in batch. As $D_1$ and $D_5$ are correlated (i.e., be updated in a time distance), their updates are performed together. As shown in Fig. 4a, BSO places $D_1$ and $D_5$ in two different stripes. When $D_1$ and $D_5$ are updated, the associated four parity chunks $P_1$, $P_2$, $P_3$, and $P_4$ also need to be updated. On the other hand, by leveraging data correlation, the new stripe organization method (named CASO) can arrange $D_1$ and $D_5$ in the same stripe (shown in Fig. 4b). In this case, updating both chunks only needs to renew two associated parity chunks $P_1$ and $P_2$ once in the following ways: (i) read $P_1$ and $P_2$; (ii) update them based on the deltas of $D_1$ and $D_5$; and (iii) write back the new parity chunks $P_1'$ and $P_2'$.

In the encoding stage, both CASO and BSO need to retrieve $k$ data chunks of each stripe stripe and calculate the $m$ parity chunks for the stripe in RS$(k,m)$ (or $l+m$ local and global parity chunks for the stripe in LRC$(k,l,m)$). CASO does not change the number of stripes, and it introduces the same amount of I/O and computation cost as BSO in the encoding stage.

The address mapping information for stripe organization in CASO can be maintained in the RAID controller for RAID-based storage systems or in the master node that tracks the metadata of data storage for networked clusters (e.g., NameNode in HDFS [36]).

## 4 Correlation-Aware Stripe Organization

We now present CASO, a correlation-aware stripe organization algorithm. The main idea of CASO is to capture data correlations by first carefully analyzing a short period of an access stream and then separating the stripe organization for correlated and uncorrelated data chunks.

### 4.1 Stripe Organization for Correlated Data Chunks

Organizing correlated data chunks is a non-trivial task and is subject to two key problems: (i) how to identify data correlation and (ii) how to organize identified correlated data chunks into stripes. How to capture data correlation has been extensively studied, yet organizing the correlated data chunks into stripes is not equivalent to simply finding the longest frequent chunk sequence as in prior approaches such as C-Miner [11] and CP-Miner [37]. In stripe organization, we should select correlated data chunks that are predicted to receive the most write operations within a stripe, and the longest chunk sequence may not be the solution we expect.

#### 4.1.1 *Correlation Graph*

To evaluate the correlation among data chunks, CASO constructs an undirected graph $G(\mathcal{D},\mathcal{E},C)$ over correlated data chunks, which we call the *correlation graph*.

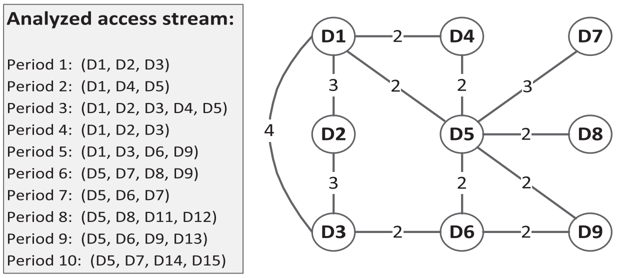



Fig. 5. An example of correlation graph constructed from an access stream.

In the correlation graph $G(\mathcal{D},\mathcal{E},C)$, suppose that $\mathcal{D}$ denotes the set of correlated data chunks that are identified, $n_c$ is the number of correlated data chunks, and $\mathcal{E}$ is a set of connections. If data chunks $D_i$ and $D_j$ are correlated (see Section 3.1 for the definition of correlation), then there exists a connection $E(D_i, D_j) \in \mathcal{E}$. $C$ is a correlation function that maps $\mathcal{E}$ to a set of non-negative numbers. For the connection $E(D_i, D_j) \in \mathcal{E}$, $C(D_i, D_j)$ is called the *correlation degree* between $D_i$ and $D_j$, which represents the number of times that both of $D_i$ and $D_j$ are requested within the same time distance in an access stream.

Fig. 5 presents an access stream which is partitioned into 10 non-overlapped periods according to a given time distance. If the access threshold is set as 2, then we can derive a set of correlated data chunks $\mathcal{D} = \{D_1, D_2, \ldots, D_9\}$ (i.e., $n_c = 9$) and accordingly construct a correlation graph. For example, as the number of periods when both of $D_1$ and $D_3$ are requested is four, we set $C(D_1, D_3) = 4$.

After establishing the correlation graph, the next step is to organize the correlated chunks into stripes. Suppose that there are $n_c$ correlated data chunks and the system selects RS$(k, m)$ for data encoding. Then the correlated data chunks will be organized into $\lambda = \lceil \frac{n_c}{k} \rceil$ stripes, namely $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_\lambda\}$. Note that the last stripe $\mathcal{S}_\lambda$ may include fewer than $k$ correlated data chunks, and it can be padded with dummy data chunks with all zeros.

Grouping the correlated data chunks will accordingly partition the correlation graph $G$ into $\lambda$ subgraphs termed $G_i(\mathcal{D}_i, \mathcal{E}_i, C)$ for $1 \le i \le \lambda$, where $\mathcal{D}_i$ $(1 \le i \le \lambda)$ denotes the set of data chunks in $G_i$. After graph partitioning, the correlated data chunks in a subgraph are organized into the same stripe. Suppose that $\mathcal{D}_i = \{D_{i_1}, D_{i_2}, \ldots, D_{i_k}\}$, and let $R(\cdot)$ be a function to calculate the sum of the correlation degrees of data chunks in a set. Then the sum of the correlation degrees of the data chunks in $\mathcal{D}_i$ is given by

$$R(\mathcal{D}_i) = \sum_{D_{i_x}, D_{i_y} \in \mathcal{D}_i, E(D_{i_x}, D_{i_y}) \in \mathcal{E}} C(D_{i_x}, D_{i_y}). \quad (1)$$

Let $\mathcal{O}$ be the set of all possible stripe organization methods. Then our objective is to *find an organization method that maximizes the sum of correlation degrees for the* $\lambda$ *correlation subgraphs*, so that the most writes are predicted to be issued to the data chunks within the same stripe. We formulate this objective function as follows:

$$\text{Max} \quad \sum_{i=1}^{\lambda} R(\mathcal{D}_i), \quad \text{for all possible methods in } \mathcal{O}. \quad (2)$$

For example, we configure $k = 3$ in erasure coding and group the nine correlated data chunks in Fig. 5 into three subgraphs as shown in Fig. 6. The data chunks grouped in the same subgraph will be organized into the same stripe. We can see that the sum of correlation degrees of the data chunks in these three subgraphs is $\sum_{i=1}^{3} R(\mathcal{D}_i) = 14$.

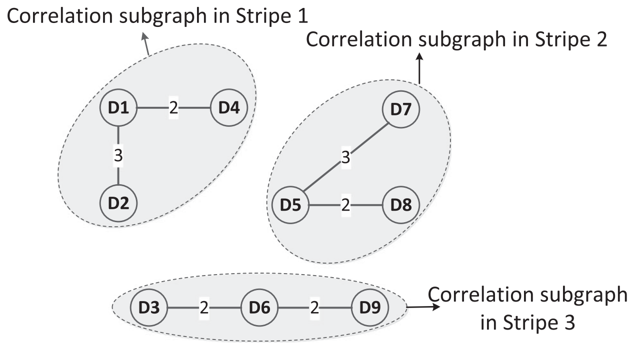



Fig. 6. An example of three correlation subgraphs (assuming that $k = 3$). There are three stripes, where $\mathcal{D}_1 = \{D_1, D_2, D_4\}$, $\mathcal{D}_2 = \{D_5, D_7, D_8\}$, and $\mathcal{D}_3 = \{D_3, D_6, D_9\}$. Then the correlation degrees of the data chunks in the three subgraphs are $R(\mathcal{D}_1) = 5$, $R(\mathcal{D}_2) = 5$, and $\mathcal{R}(\mathcal{D}_3) = 4$, respectively.

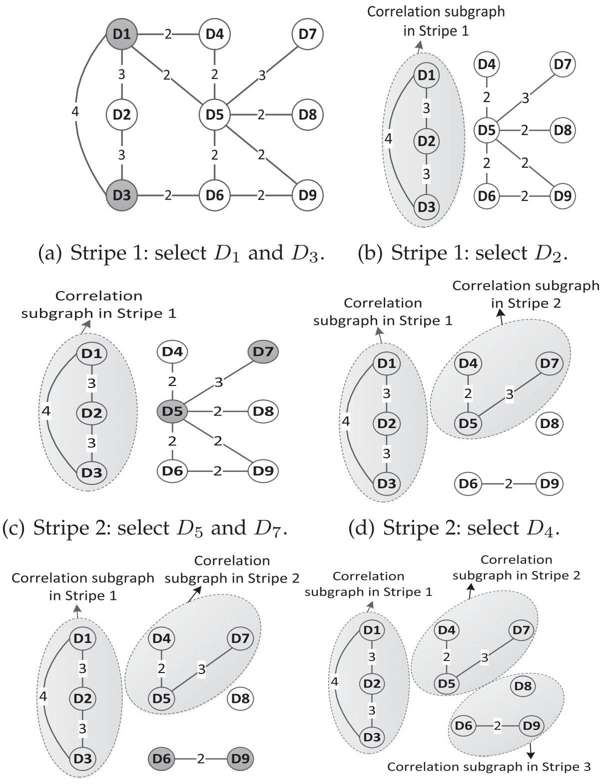
### 4.1.2 Correlation-Aware Stripe Organization Algorithm

Finding the organization method that maximizes the sum of correlation degrees through enumeration is extremely time consuming. It requires to iteratively choose $k$ correlated data chunks to construct a stripe from those that are unorganized yet. Suppose that there are $n_c$ correlated data chunks. Then the enumeration of all possible stripe organization methods will need $\binom{n_c}{k} \cdot \binom{n_c - k}{k} \cdots \binom{n_c - (\lambda-1)k}{k}$ tests,[1] where $\lambda = \lceil \frac{n_c}{k} \rceil$. To improve the search efficiency, we propose a greedy algorithm (see Algorithm 1) to organize the correlated data chunks. The *main idea* is that for each stripe, it first selects a pair of data chunks with the maximum correlation degree among those that are unorganized yet, and then iteratively chooses a data chunk that has the maximum sum of correlation degrees with those that have already been selected for the stripe.

**Algorithm 1.** Stripe Organization for Correlated Data Chunks

```
Input: A correlation graph G(D, E, C).
Output: The λ subgraphs.
Set D_i = ∅ for 1 ≤ i ≤ λ
for i = 1 to λ − 1 do
    Select D_{i1} and D_{i2} with the maximum correlation
    degree in G(D, E, C)
    Update D = D − {D_{i1}, D_{i2}}, D_i = {D_{i1}, D_{i2}}
    repeat
      for D_x ∈ D do
        Calculate θ_{x,i} = R(D_i ∪ {D_x}) − R(D_i)
      Find D_y, where θ_{y,i} = Max{θ_{x,i} | D_x ∈ D}
      Set D = D − {D_y}, D_i = D_i ∪ {D_y}
    until D_i includes k data chunks;
    Remove the connections between the data chunks in D_i
    and those in D over G(D, E, C)
Organize the remaining correlated data chunks into D_λ
```

1. $\binom{i}{j}$ denotes the number of combinations of selecting $j$ chunks from $i$ chunks, where $j \le i$.




(a) Stripe 1: select $D_1$ and $D_3$. (b) Stripe 1: select $D_2$.

(c) Stripe 2: select $D_5$ and $D_7$. (d) Stripe 2: select $D_4$.

(e) Stripe 3: select $D_6$ and $D_9$. (f) Stripe 3: select $D_8$.

Fig. 7. Example of organizing correlated data chunks in CASO.

In the initialization of Algorithm 1, $\mathcal{D}$ includes all the correlated data chunks. The set $\mathcal{D}_i$ ($1 \leq i \leq \lambda$), which is used to include the data chunks in the stripe $\mathcal{S}_i$, is set as empty (step 1). For the stripe $\mathcal{S}_i$ ($1 \leq i \leq \lambda - 1$), we first choose two data chunks that have the maximum correlation degree in $G(\mathcal{D}, \mathcal{E}, C)$ from those that have not been organized yet (step 3). These two data chunks will be excluded from $\mathcal{D}$ and added into $\mathcal{D}_i$ (step 4). After that, we scan every remaining data chunk $D_x$ in $\mathcal{D}$ and calculate its sum of correlation degrees with the data chunks in $\mathcal{D}_i$, which is denoted by $\theta_{x,i} = R(\mathcal{D}_i \cup \{D_x\}) - R(\mathcal{D}_i)$ (step 6$\sim$step 7). According to the definition of $R(\cdot)$ (see Equation (1)), we can deduce that

$$\theta_{x,i} = \sum_{D_j \in \mathcal{D}_i, E(D_x, D_j) \in \mathcal{E}} C(D_x, D_j).$$

We then choose the one $D_y$ that has the maximum sum of correlation degrees with the data chunks in $\mathcal{D}_i$, exclude it from $\mathcal{D}$, and append it to $\mathcal{D}_i$ (step 8$\sim$step 9). We repeat the selection of data chunks in $\mathcal{D}_i$ until $\mathcal{D}_i$ has included $k$ data chunks (step 10). Once these $k$ data chunks in $\mathcal{D}_i$ have been determined, the algorithm then removes the connections of the data chunks in $\mathcal{D}_i$ with those in $\mathcal{D}$, and turns to the organization of the next stripe (step 11). Finally, the storage system organizes the remaining correlated data chunks into $\mathcal{D}_\lambda$ (step 12).

**Example.** We show an example in Fig. 7 based on the correlation graph in Fig. 5. In this example, we set $k = 3$ and thus $\lambda = \lceil \frac{n_c}{k} \rceil = 3$. At the beginning, $\mathcal{D} = \{D_1, D_2, \ldots, D_9\}$ and $\mathcal{D}_i = \emptyset$ for $1 \leq i \leq 3$.

To determine the three data chunks in $\mathcal{D}_1$, we first select the two data chunks $D_1$ and $D_3$, which we find have the maximum correlation degree of $C(D_1, D_3) = 4$ in $G(\mathcal{D}, \mathcal{E}, C)$. Then we update $\mathcal{D} = \{D_2, D_4, D_5, \ldots, D_9\}$ and set $\mathcal{D}_1 = \{D_1, D_3\}$ (see Fig. 7a). The algorithm then scans the remaining data chunks in $\mathcal{D}$. We first consider $D_2$, which connects both $D_1$ and $D_3$ and has the sum of correlation degrees $C(D_2, D_1) + C(D_2, D_3) = 6$. We next turn to $D_4$ in $\mathcal{D}$, which only connects $D_1$ and has the correlation degree of $C(D_4, D_1) = 2$. We repeat the test for all the remaining data chunks in $\mathcal{D}$, and finally select $D_2$ that has the maximum sum of correlation degrees with the data chunks in $\mathcal{D}_1$. We update $\mathcal{D} = \{D_4, D_5, \ldots, D_9\}$ and $\mathcal{D}_1 = \{D_1, D_2, D_3\}$. Once the number of data chunks in $\mathcal{D}_1$ equals $k$ (i.e., 3 in this example), we delete the edges connecting the data chunks in $\mathcal{D}$ and those in $\mathcal{D}_1$ (i.e., $E(D_1, D_4)$, $E(D_1, D_5)$, and $E(D_3, D_6)$), as shown in Fig. 7b. Following this principle, we obtain $\mathcal{D}_2 = \{D_4, D_5, D_7\}$ (see Fig. 7d) and $\mathcal{D}_3 = \{D_6, D_8, D_9\}$ (see Fig. 7f). We can see that $\sum_{i=1}^{3} R(\mathcal{D}_i) = 17$.

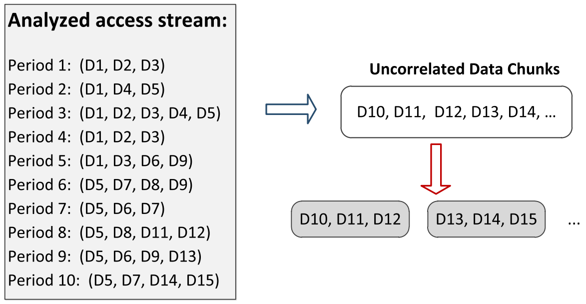



Fig. 8. An example of stripe organization for uncorrelated data chunks.

**Algorithm 2.** Stripe Organization for Uncorrelated Data Chunks

**for** *each uncorrelated data chunk* $D_i$ **do**
  Find the number of correlated data chunks $n_i$ whose chunk identities are smaller than $i$
  Organize it into the $(\lambda + \lceil \frac{i - n_i}{k} \rceil)$-th stripe
Store the $k$ data and $m$ parity chunks of each stripe on $k + m$ storage devices with only one chunk per device

### 4.2 Stripe Organization for Uncorrelated Data Chunks

We also consider the organization of uncorrelated data chunks. We have two observations.

1) Spatial locality can be utilized in stripe organization to reduce the parity updates in partial stripe writes. For example, if two sequential data chunks in the same stripe are written, then we only need to update their common parity chunks.
2) Uncorrelated data chunks still account for a large proportion of all the accessed data chunks in many workloads (e.g., `wdev_1`, `rsrch_1`, and `web_2` in Fig. 3).

Therefore, we propose to organize the uncorrelated data chunks in a round-robin fashion [35]. Algorithm 2 gives the main steps to organize uncorrelated data chunks.

**Example.** We set $k = 3$ in erasure coding. Fig. 8 shows an example based on the access stream in Fig. 5. From Fig. 5, the correlated data chunks are $\{D_1, D_2, \ldots, D_9\}$ and are organized into $\lambda = 3$ stripes. We then identify the uncorrelated ones. To organize $D_{10}$, it will be organized in the 4th stripe. Following this method, we can obtain the stripes that preserve a high degree of data sequentiality as shown in Fig. 8.

### 4.3 Extension for Local Reconstruction Codes

We will elaborate how to deploy CASO on top of Azure's LRC (see Section 2.1). As a non-MDS code, LRC keeps additional local parity chunks and hence suffers from more parity update overhead. How to reduce the updates to both global and local parity chunks is critical for improving the write performance of LRC.

**Algorithm 3.** Local Group Organization for Correlated Data Chunks

**Input:** (1) The $k$ data chunks $\mathcal{D}_i$ of the $i$-th stripe;
(2) The subgraph $G_i$ constructed by $\mathcal{D}_i$
**Output:** $l$ local groups of $\mathcal{D}_i$.

```
1  Set D_i = {D_{i_1}, D_{i_2}, ..., D_{i_k}}
2  Set L_j = ∅ for 1 ≤ j ≤ l
3  for 1 ≤ j ≤ l do
4      Select D_{j_1} and D_{j_2} from D_i with the maximum
       correlation degree in D_i
5      Update L_j = {D_{j_1}, D_{j_2}}, D_i = D_i − {D_{j_1}, D_{j_2}}
6      repeat
7          for D_{i_x} ∈ D_i do
8              Calculate θ_{i_x,j} = R(L_j ∪ {D_{i_x}}) − R(L_j)
9          Find D_{i_y}, where θ_{i_y,j} = Max{θ_{i_x,j} | D_{i_x} ∈ D_i}
10         Set L_j = L_j ∪ {D_{i_y}}, D_i = D_i − {D_{i_y}}
11     until L_j includes k/l data chunks;
12     Remove the connections between the data chunks in
       L_j and those in D_i over G_i
```

In view of this, we extend CASO to organize the local groups of LRC, with the objective of utilizing data correlations for reducing updates to both global and local parity chunks. For LRC$(k, l, m)$, the $k$ data chunks of a stripe will be organized into $l$ local groups, namely $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_l\}$ with $\frac{k}{l}$ data chunks per local group; for simplicity of our discussion, we now assume that $k$ is divisible by $l$. Algorithm 3 considers the local group organization of a stripe, and $\mathcal{D}_i$ denotes the set of $k$ data chunks in the $i$-th stripe. For each local group say $\mathcal{L}_j$ (where $1 \leq j \leq l$), CASO first selects two most correlated data chunks among the recorded data chunks in $\mathcal{D}_i$, which will be included in the local group $\mathcal{L}_j$ and evicted from $\mathcal{D}_i$ (steps 4$\sim$5). The algorithm then iteratively chooses the data chunk $D_{i_y} \in \mathcal{D}_i$ that owns the maximum correlation degree with those in $\mathcal{L}_j$, and accordingly updates $\mathcal{L}_j$ and $\mathcal{D}_i$ (steps 7$\sim$10). As such chunk $D_{i_y} \in \mathcal{D}_i$ always exists in each round of correlated chunk selection, we can repeatedly increase the number of data chunks in $\mathcal{L}_j$. When the number of data chunks in $\mathcal{L}_j$ reaches $\frac{k}{l}$, the algorithm then removes the connections of the data chunks selected in $\mathcal{L}_j$ with those in $\mathcal{D}_i$, and turns to the organization of next local group (step 12). The algorithm terminates when all the $k$ data chunks are successfully organized into $l$ local groups.

### 4.4 Complexity Analysis

We present the complexity analysis for the three algorithms in Section 1 of the supplementary file, available online.

TABLE 1
Characteristics of Selected Traces

| Trace | Write ratio | Num. of write requests | Write size |
|---|---|---|---|
| **High write ratios** | | | |
| `wdev_1` | 1.000 | 1,055 | 5.13 KB |
| `wdev_2` | 0.999 | 181,077 | 8.15 KB |
| `rsrch_1` | 0.997 | 13,738 | 12.17 KB |
| `wdev_3` | 0.984 | 671 | 4.35 KB |
| **Medium write ratios** | | | |
| `wdev_0` | 0.799 | 913,732 | 8.20 KB |
| `rsrch_2` | 0.343 | 71,223 | 4.25 KB |
| **Low write ratios** | | | |
| `hm_1` | 0.047 | 28,415 | 19.96 KB |
| `src2_1` | 0.021 | 14,104 | 13.37 KB |

## 5 Performance Evaluation

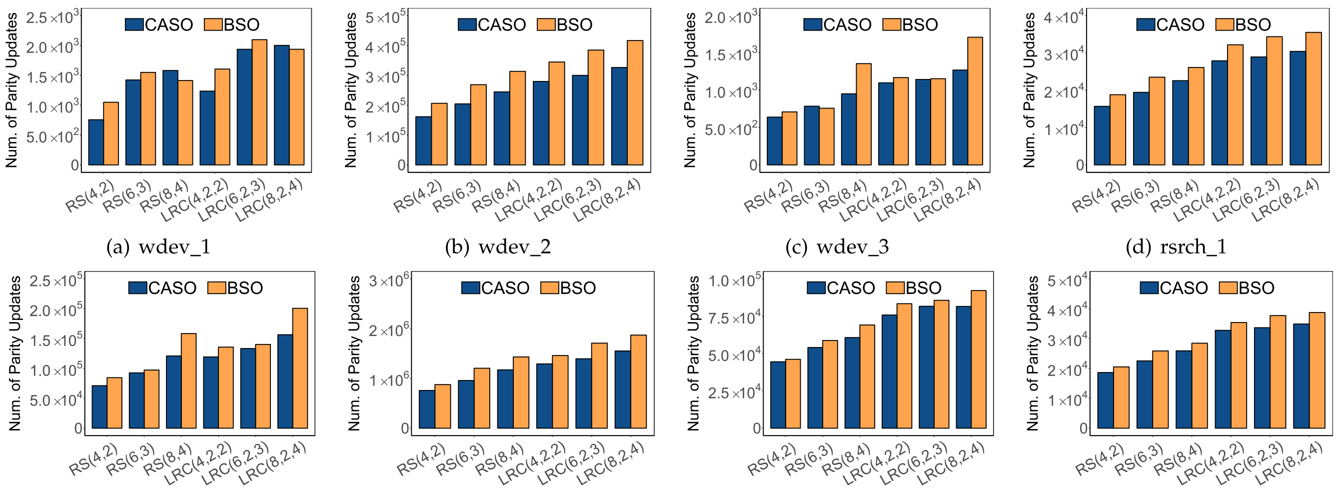
In this section, we carry out extensive testbed experiments to evaluate the performance of CASO.

*Selection of Codes and Traces.* We mainly consider three RS codes: RS$(4, 2)$, RS$(6, 3)$, and RS$(8, 4)$; note that RS$(6, 3)$ is also used in the Quancast File System [38] and HDFS Erasure Coding [39]. Based on the above three RS codes, we generate another three LRC variants by creating two local groups in a stripe of each RS code. The resulting three LRC variants are LRC$(4, 2, 2)$, LRC$(6, 2, 3)$, and LRC$(8, 2, 4)$, respectively.

Our evaluation is driven by real-world block-level traces from MSR Cambridge Traces [31], which describe various access characteristics of enterprise storage servers. The traces are collected from 36 volumes that span 179 disks of 13 servers for one week. Each trace records the starting position of the I/O request and the request size. As CASO is proposed for optimizing partial stripe writes, our goal is to systematically study the effect of CASO when being deployed in the applications with different degrees of write intensity.

To this end, we select eight traces for evaluation based on a new metric called *write ratio*. The write ratio of a volume is calculated by dividing the number of write requests to the number of all the access requests to that volume. To select the eight traces with significantly different write ratios, we first sort the 36 volumes according to their write ratios and classify them into three categories: (i) the top 12 volumes with high write ratios, (ii) the next 12 volumes with medium write ratios, and (iii) the last 12 volumes with low write ratios. We then select four volumes with high write ratios (i.e., `wdev_1`, `wdev_2`, `wdev_3`, and `rsrch_1`), two volumes with medium write ratios (i.e., `wdev_0` and `rsrch_2`), and another two volumes with low write ratios (i.e., `hm_1` and `src2_1`). These traces are collected from different applications. For example, the traces with prefixes `wdev`, `rsrch`, `src`, and `hm` are collected from web applications, research projects, source control, and hardware monitoring. We can thus use these traces to evaluate the effectiveness of CASO when it is deployed in different applications. Table 1 summarizes the characteristics of the selected traces, including their write ratios and average write sizes.

We run our experiments on a machine connected with a disk array, such that the machine simulates the functionalities




(a) wdev_1 (b) wdev_2 (c) wdev_3 (d) rsrch_1

(e) rsrch_2 (f) wdev_0 (g) hm_1 (h) src2_1

Fig. 9. Experiment 1 (Impact of different erasure codes on parity updates). The smaller value is better.

of a RAID controller. Take RS$(k, m)$ as an example. We manipulate the experiments on $k+m$ disks, where each disk exclusively stores one chunk of a stripe. Each disk stores the data sequentially with the increase of the stripe identity number.

When a block-level write request arrives, we first identify the chunk identities included in this request by dividing the access range to the chunk size. Thus, given a chunk identity, we can pinpoint the stripe identity and the disk identity of the chunk. For each stripe, we read the old data chunks, calculate the new parity chunks, and finally write the new data and parity chunks to the disks. The read and write operations in parity updates are realized by calling the POSIX asynchronous I/O (AIO) interfaces, with the disk identities and the logical offset (derived by multiplying the stripe identities with the chunk size) as parameters. We elaborate the addressing issue of the trace replay in Section 2 of the supplementary file, available online.

*Testbed.* Like previous work [18], [20], we use a node with a number of extended disks to study the performance of CASO. Our evaluation is run on a Linux server with an X5472 processor and 8GB 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 rpm. The machine and the disk array are connected by a Fiber cable with the bandwidth of 800 MB/sec. The selected erasure codes are realized based on Jerasure 1.2 [21].

*Methodology.* In the evaluation, the chunk size is set as 4 KB, which is consistent with the deployment of erasure codes in real storage systems [9], [40]. For each trace, we only select a small portion of write requests for correlation analysis. To describe the ratio of write requests of a trace that are analyzed in CASO, we first define the concept of *analysis ratio* as follows. CASO first classifies all the write requests into $w$ non-overlapped time windows with a constant time distance. In our test, we set the time distance as 1 millisecond. Suppose that CASO explores data correlation for the write requests in the first $w^*$ time windows (where $0 \leq w^* \leq w$). Then the analysis ratio can be calculated by $\frac{w^*}{w}$.

After correlation analysis, we first group the correlated data chunks that are identified into stripes based on Algorithm 1. For LRC, we further organize the correlated data chunks of a stripe into local groups based on Algorithm 3. For the remaining data chunks, we organize them based on their logical chunk addresses (see Algorithm 2). To fairly evaluate CASO, we replay the access requests (including the read and write requests) that are not used in the correlation analysis for each trace. We compare CASO with *baseline stripe organization* (BSO) in the evaluation.

For the experiments related to time performance, we repeat each experiment for five runs. We plot the average results and the error bars indicating the maximum and the minimum across the five runs.

*Experiment 1 (Impact of Different Erasure Codes on Parity Updates).* We first measure the number of parity updates incurred in partial stripe writes for different erasure codes. We set the analysis ratio as 0.5 and select the six erasure codes with different parameters: RS(4,2), RS(6,3), RS(8,4), LRC$(4,2,2)$, LRC$(6,2,3)$, and LRC$(8,2,4)$. The results are shown in Fig. 9. We make two observations.

First, CASO can reduce 13.1 percent of parity updates on average for different erasure codes under different real traces. In particular, when using RS$(8,4)$ in the trace `wdev_3`, CASO reduces 29.8 percent of parity updates compared to BSO. The reason is that CASO arranges the correlated data chunks together in a small number of stripes, such that the partial stripe writes to them are centralized to a few stripes and the number of parity chunks to be updated is reduced.

Second, CASO reduces the least parity updates for the traces with low write ratios. Specifically, CASO reduces 9.0 percent of parity updates on average for the traces with low write ratios (i.e., `hm_1` and `src2_1`), while the reduction for the traces with high and medium write ratios (i.e., `wdev_0`~`wdev_3`, `rsrch_1`~`rsrch_2`) is 14.5 percent. This observation indicates that with more write requests taken into account, CASO can capture more correlation and reduce the parity updates in next partial stripe writes.

*Experiment 2 (Impact of Different Analysis Ratios on Parity Updates).* To study the impact of analysis ratios on parity updates, we vary the analysis ratio from 0.1 to 0.7, and measure the number of resulting parity updates incurred in RS$(4,2)$ and LRC$(4,2,2)$ for CASO and BSO.

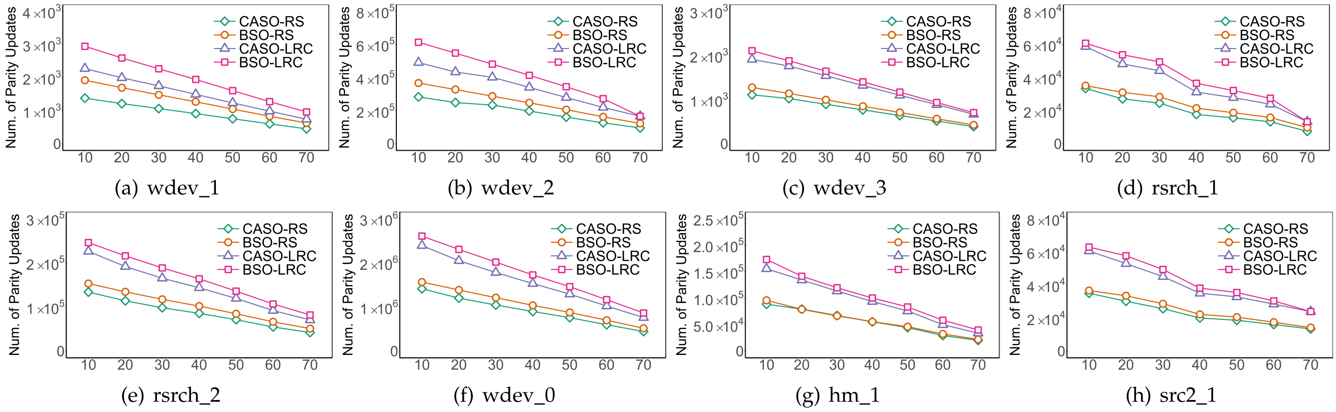



Fig. 10. Experiment 2 (Impact of different analysis ratios on parity updates for RS(4,2) and LRC(4,2,2)).

Fig. 10 illustrates the results. We observe that CASO generally reduces 13.5 percent of parity updates on average for different analysis ratios. Take the trace wdev_1 as an example. CASO cuts down about 22.8 percent of parity updates for LRC$(4,2,2)$ when the analysis ratio is 0.1, and this reduction increases to 23.0 percent when the analysis ratio reaches 0.7.

In addition, as described above, the evaluation measures the parity updates by using the remaining write requests that are not used in correlation analysis for validating the effectiveness of CASO. Therefore, fewer write requests can be replayed when the analysis ratio is larger, and hence the number of parity updates in both CASO and BSO drops when the analysis ratio increases.

*Experiment 3 (Average Write Time).* We further measure the average time for our testbed to complete a write request in different traces. We set the analysis ratio as 0.5 and run the tests for different RS codes and LRC. Fig. 11 illustrates the average time to complete a write request. A write request may update a single chunk or multiple chunks within or across stripes, depending on the starting address of the write request and the write size.

CASO reduces the write time by 14.6 percent on average for all the traces and erasure codes. In particular, when being applied to LRC$(8,2,4)$, CASO even reduces 46.7 percent of the write time for the trace wdev_2. The reason is that CASO significantly decreases the number of parity updates in partial stripe writes.

*Experiment 4 (Additional I/Os in Degraded Reads).* We also evaluate the performance of degraded reads in CASO. Degraded reads [2], [20], [41] usually appear when the storage system suffers from transient failure (i.e., the stored data chunks are temporarily unavailable). To serve degraded reads, the storage system will retrieve additional data and parity chunks to recover the lost chunk, and finally, the number of I/Os increases. To evaluate degraded reads when CASO and BSO are respectively deployed atop of an erasure code, we first construct the stripes of an erasure code by using CASO and BSO, respectively. We then erase the data on a disk, replay the read requests after the stripe organization is established for each trace, and record the average amount of data to be additionally read in one disk's failure. For each of RS codes and LRC, we repeat this procedure for every member disk's failure in a stripe.

To show the degree of I/O increase introduced by CASO in degraded reads, we define a new metric termed *increase ratio*, which can be calculated by the following equation.

$$\text{increase ratio} = \frac{\text{num. of additional chunks read in CASO}}{\text{num. of additional chunks read in BSO}}.$$

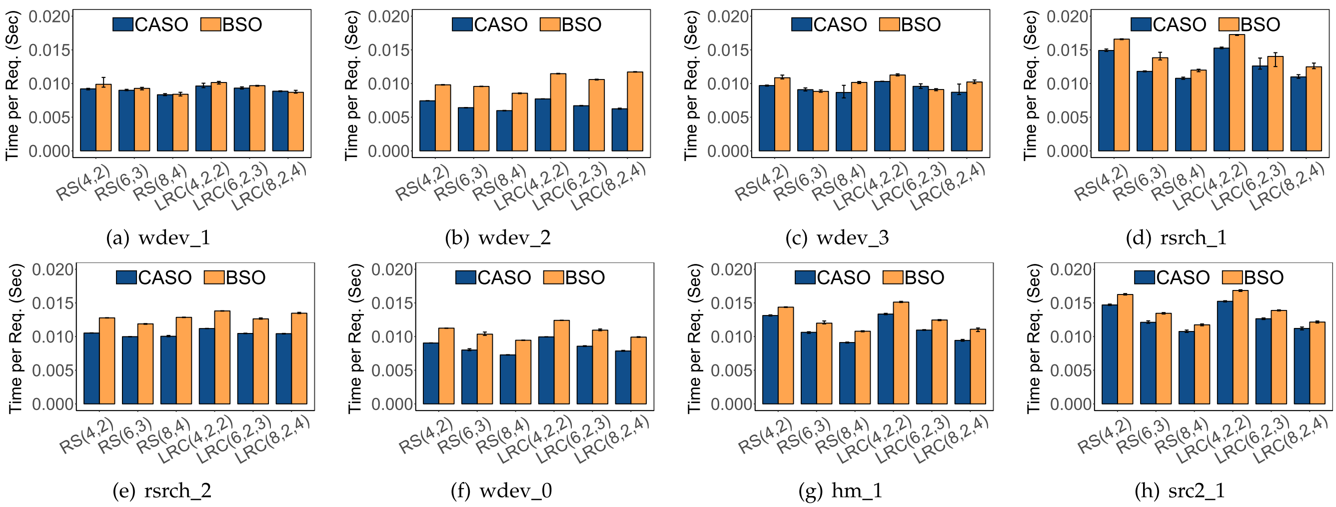



Fig. 11. Experiment 3 (Average time per write request). The smaller value is better.

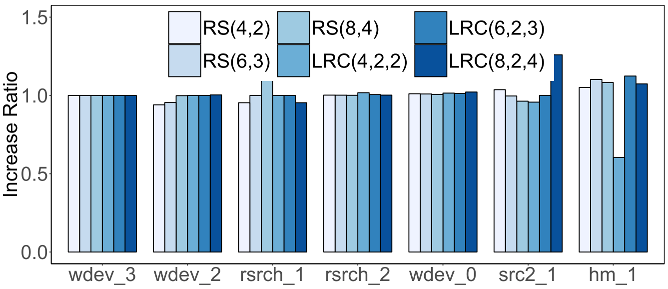



Fig. 12. Experiment 4 (Additional I/Os in degraded reads).

As the trace wdev_1 does not have any read request, it will not cause degraded read operations. Fig. 12 depicts the increase ratios of other seven traces. We can see that CASO only increases marginal I/O in degraded reads. More specifically, compared to BSO, CASO will read 1.1 percent (resp. 0.2 percent) of additional chunks on average in degraded reads for RS codes (resp. LRC). The reason is that CASO is designed to identify the correlation of data chunks in write requests and group those that have higher likelihood to be written within the same time window in the same stripe. This design may put the logically sequential data chunks into different stripes. As a consequence, the sequential data chunks requested in degraded reads will trigger the recovery across different stripes, thereby retrieving more additional data and parity chunks. However, we argue that compared to the write performance gains brought by CASO, this marginal cost is acceptable. In summary, it is more appropriate to deploy CASO in the applications with the access characteristics of intensive writes and infrequent reads.

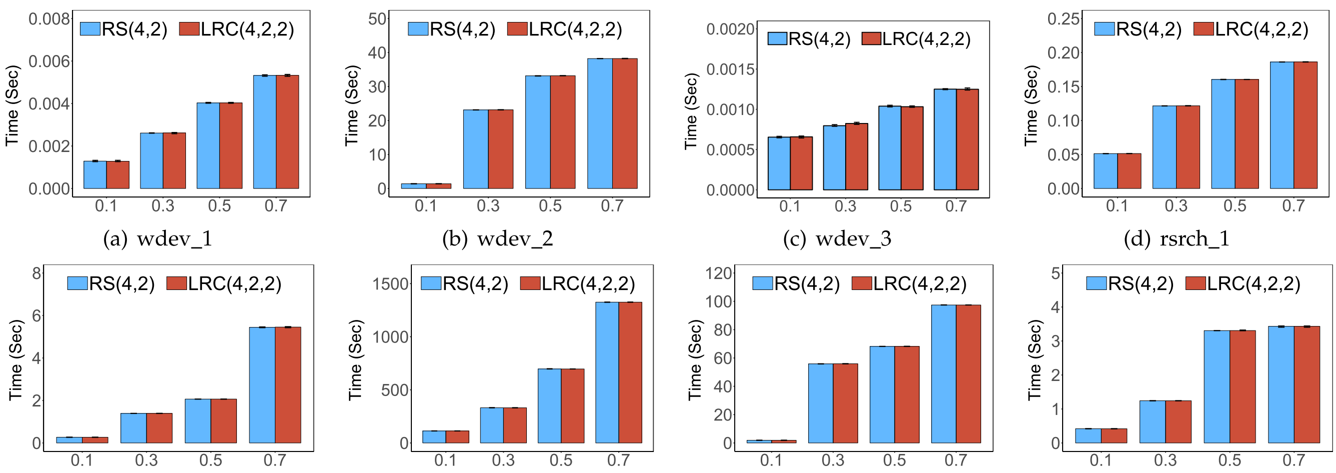
In addition, we can observe that the increase ratio has a significant fluctuation for some traces, such as src2_1 and hm_1. There is variety of causes of such fluctuation, such as the read patterns and the number of data chunks in a stripe (for RS codes) or in a local group (for LRC). Take the trace hm_1 as an instance. When replaying the read requests to the stripe of LRC$(4,2,2)$, we find that when CASO is applied, about 43 percent of the read requests will be issued to the correlated data chunks with the average read size of 2.3 chunks. In this case, as the average read size of these requests is larger than the number of disks in a local group of LRC$(4,2,2)$ (i.e., $\frac{k}{l}=2$), the requested data chunks will fall in the same local group with a large probability. Therefore, if a data chunk is temporarily unavailable, the system can first locate the local group where this unavailable data chunk belongs to and then reuse the available data chunks in the same local group that are requested for recovery.

*Experiment 5 (Stripe Organization Time).* In this experiment, we select RS$(4,2)$ and LRC$(4,2,2)$, vary the analysis ratio from 0.1 to 0.7, and measure the stripe organization time for different traces. The stripe organization time records the time to identify the correlated data chunks from given write requests, construct the correlation graph, partition the graph, and determine the stripe that each data chunk in a trace belongs to. Fig. 13 plots the average results. We can make four observations.

First, the stripe organization time generally increases with the analysis ratio. This is because CASO will find more correlated data chunks with a large probability when taking more write requests into correlation analysis. For example, when the analysis ratio is 0.1, it merely needs 0.27 seconds for CASO to organize the data chunks of the trace rsrch_2 for RS$(4,2)$. The stripe organization time will increase to 5.44 seconds when the analyze ratio is 0.7.

Second, the average stripe organization time of LRC is merely 0.1 percent more than that of RS codes. This finding indicates that the time for local group organization (see Algorithm 3) is marginal.

Third, the trace with more access requests does not definitely require more time in the stripe organization. For example, the trace wdev_2, though includes more access requests than hm_1 (see Table 1), calls for less time in the stripe organization. In the stripe organization, CASO identifies the access correlations and partitions the correlated graph. For the correlation identification, CASO needs to analyze the chunks that are written in the same time distance and capture the correlated data chunks. Thus, the number of time distances and the number of data chunks written in a time distance contribute to the correlation identification time. Also, for the graph partition, the identified




(a) wdev_1 (b) wdev_2 (c) wdev_3 (d) rsrch_1

(e) rsrch_2 (f) wdev_0 (g) hm_1 (h) src2_1

Fig. 13. Experiment 5 (Stripe organization time for RS(4,2) and LRC(4,2,2)).

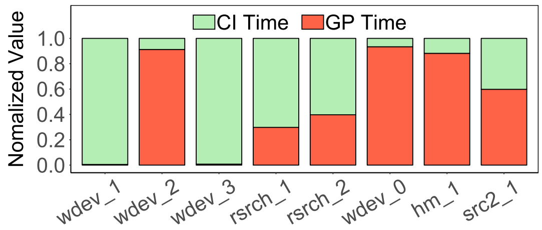



Fig. 14. Experiment 6 (Breakdown on the stripe organization time).

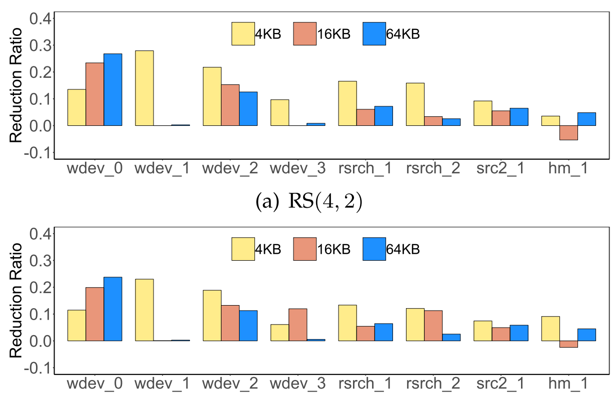
correlated data chunks are organized into stripes by following Algorithm 1 (see Section 4.1.2). Thus, the number of correlated data chunks, the number of data chunks in a stripe (i.e., $k$), and the number of correlated stripes collectively determine the graph partition time.

Finally, the stripe organization time in CASO is reasonable and acceptable. For more than half of the traces, CASO needs less than 10 seconds to organize the stripes. For the traces wdev_0, the stripe organization time will not exceed 1,326 seconds. Stripe organization is usually triggered before data is encoded, and rarely changed once being established. Therefore, it can be treated as the one-time cost in data storage. It is acceptable when given the performance gains brought by CASO in future partial stripe writes.

*Experiment 6 (Breakdown on the Stripe Organization).* We also give a further breakdown on the stripe organization time. We select $\mathrm{RS}(4,2)$ and set the analysis ratio as 0.5. The stripe organization is partitioned into two stages: *correlation identification* (CI) and *graph partition* (GP). CI will identify the correlated data chunks and their correlation degrees from a given batch of write requests. The correlated data chunks can construct a correlation graph (see Section 4.1.1). GP will further organize the correlated data chunks into stripes by partitioning the correlation graph into subgraphs based on the greedy selection (see Algorithm 1).

Fig. 14 plots the ratios of the time in CI and GP during the stripe organization. We can observe that the ratio of GP varies across different traces. For example, the time of GP only occupies 0.5 percent of the stripe organization time for the trace wdev_1 that only has 1,055 write requests. This ratio increases to 93.4 percent for the trace wdev_0, which includes 913,732 write requests.

*Experiment 7 (Impact of Chunk Sizes on Parity Updates).* We further investigate the impact of chunk sizes on parity




(a) $\mathrm{RS}(4,2)$

(b) $\mathrm{LRC}(4,2,2)$

Fig. 15. Experiment 7 (Impact of chunk sizes on parity updates).

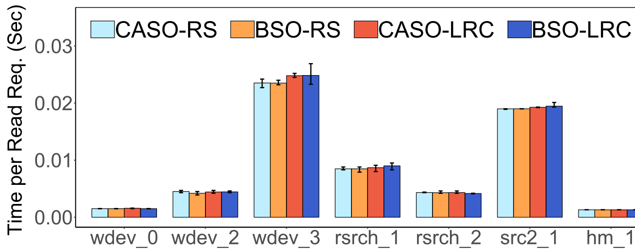



Fig. 16. Experiment 8 (Normal read time). The smaller value is better.

updates. We vary the size of a chunk from 4 KB to 64 KB, and calculate the *reduction ratios of parity updates* for $\mathrm{RS}(4,2)$ and $\mathrm{LRC}(4,2,2)$. The analysis ratio is set as 0.5. Suppose that the numbers of parity updates introduced in CASO and BSO are $t^*$ and $t$, respectively. The reduction ratio of parity updates can be derived as $1-\frac{t^*}{t}$.

Fig. 15 shows the results. First, CASO reduces 13.7, 7.0 and 7.3 percent of parity updates on average when the chunk size is set as 4 KB, 16 KB and 64 KB, respectively. Second, the performance gains introduced by CASO drop when the chunk size increases. Recall that CASO aims to pack the correlated data chunks in the same stripe to reduce the parity updates. As the average chunk size of the traces is no more than 20 KB (see Table 1), the number of correlated chunks decreases when the chunk size increases (e.g., 64 KB). Thus, it is important to configure the appropriate chunk size, to ensure that CASO can capture enough correlation for improving the write efficiency.

*Experiment 8 (Normal Read Time).* We finally evaluate the normal read time for both CASO and BSO when all the disks are healthy. As opposed to degraded reads, all the requested data in normal reads are available and can be directly retrieved from the underlying storage devices. In this experiment, we set the size of a chunk as 4 KB, and configure the analysis ratio as 0.5. As the trace wdev_1 does not have any read request, we replay the remaining seven traces when deploying CASO and BSO over $\mathrm{RS}(4,2)$ and $\mathrm{LRC}(4,2,2)$.

Fig. 16 shows the average time to complete a normal read request. First, the normal read time in CASO differs by no more than 0.5 percent with that in BSO, indicating that CASO can significantly improve the write efficiency without affecting the normal read efficiency. Second, as the traces have different I/O sizes and read patterns, the average time to complete a normal read request varies across the traces.

## 6 Conclusion

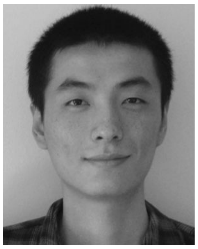
We study the optimization of partial stripe writes in erasure-coded storage from the perspectives of *data correlation* and *stripe organization*. CASO is a *correlation-aware stripe organization* algorithm that captures data correlations from a small portion of data accesses. It groups the correlated data chunks into stripes to centralize partial stripe writes, and organizes the uncorrelated data chunks into stripes to make use of the spatial locality. We show how CASO can be applied to RS codes and Azure's LRC. Experimental results show that CASO can reduce up to 29.8 percent of parity updates in partial stripe writes and reduce the write time by up to 46.7 percent, while still preserving the read performance.

## ACKNOWLEDGMENTS

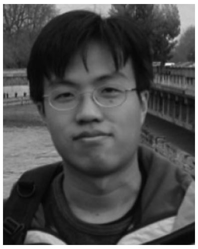
This work is supported by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G), National Natural Science Foundation of China (Grant No. 61602120, 61672159, U1705262, 61832011), and the Fujian Provincial Natural Science Foundation (Grant No. 2017J05102). A preliminary version [1] of this paper was presented at the 36th IEEE International Symposium on Reliable Distributed Systems (SRDS'17). In this journal version, we extend the algorithmic design of CASO to Azure's Local Reconstruction Codes [2] and conduct more experimental evaluation.

## REFERENCES

[1] Z. Shen, P. P. Lee, J. Shu, and W. Guo, "Correlation-aware stripe organization for efficient writes in erasure-coded storage systems," in *Proc. IEEE 36th Symp. Reliable Distrib. Syst.*, 2017, pp. 134–143.
[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, pp. 2–2.
[3] 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.*, 2007, pp. 2–2.
[4] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" presented at the *5th USENIX Conf. File Storage Technol.*, San Jose, CA, USA, 2007.
[5] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 61–74.
[6] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. Revised Papers 1st Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–338.
[7] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, and A. e. a. Dimakis, "Xoring elephants: Novel erasure codes for big data," in *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, Mar. 2013.
[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Comput. Surveys*, vol. 26, no. 2, pp. 145–185, 1994.
[9] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage," in *Proc. USENIX FAST*, 2014, pp. 163–176.
[10] Z. Shen, J. Shu, and Y. Fu, "Parity-switched data placement: Optimizing partial stripe writes in XOR-coded storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3311–3322, Nov. 2016.
[11] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: Mining block correlations in storage systems," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 173–186.
[12] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Society Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[13] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 1–1.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] Z. Shen and J. Shu, "HV code: An all-around MDS code to improve efficiency and reliability of RAID-6 systems," in *Proc. IEEE/IFIP DSN*, 2014, pp. 550–561.
[19] C. Jin, D. Feng, H. Jiang, and L. Tian, "RAID6L: A log-assisted RAID6 storage architecture with improved write performance," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–6.
[20] 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.
[21] 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, 2008.
[22] HDFS RAID. (2010). [Online]. Available: https://wiki.apache.org/hadoop/HDFS-RAID
[23] EMC symmetrix DMX - Raid 6 implementation. (2009). [Online]. Available: https://storagenerve.com/2009/02/27/emc-symmetrix-dmx-raid-6-implementation/
[24] C. Lueth, "RAID-DP: Network appliance implementation of RAID double parity for data protection," Netw. Appliance, Inc., Sunnyvale, CA, USA: Tech. Rep. TR-3298, 2004.
[25] A. Thomasian, "Reconstruct versus Read-Modify Writes in RAID," *Inf. Process. Lett.*, vol. 93, no. 4, pp. 163–168, 2005.
[26] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch," in *Proc. USENIX Annu. Tech. Conf. Proc. USENIX Annu. Tech. Conf.*, 2007, Art. no. 20.
[27] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-Aware Prefetching at the Storage Server," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 377–390.
[28] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *Proc. Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.
[29] R. Li, Y. Hu, and P. P. Lee, "Enabling efficient and reliable transition from replication to erasure coding for clustered file systems," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 148–159.
[30] A. Miranda and T. Cortes, "CRAID: Online RAID upgrades using dynamic hot data reorganization," in *Proc. USENIX FAST*, 2014, pp. 133–146.
[31] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, 2008, Art. no. 10.
[32] Q. Li, L. Shi, C. J. Xue, K. Wu, J. Cheng, Q. Zhuge, and E. H.-M. Sha, "Access characteristic guided read and write cost regulation for performance improvement on flash memory," in *Proc. USENIX FAST*, 2016, pp. 125–132.
[33] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al., "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 253–265.
[34] Z. Zhang and W. Jiang, "Native erasure coding support inside HDFS," in *Proc. Strata + Hadoop World*, 2015, https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42957
[35] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 109–116, 1988.
[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[37] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: A tool for finding copy-paste and related bugs in operating system code," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation - Vol. 6*, 2004, pp. 289–302.
[38] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
[39] HDFS Erasure Coding. [Online]. Available: https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFS ErasureCoding.html
[40] 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. Reliable Distrib. Syst.*, 2015, pp. 228–237.
[41] Z. Shen, P. P. Lee, J. Shu, and W. Guo, "Encoding-aware data placement for efficient degraded reads in XOR-coded storage systems," in *Proc. IEEE 35th Symp. Reliable Distrib. Syst.*, 2016, pp. 239–248.



**Zhirong Shen** received the BS degree from the University of Electronic Science and Technology of China, in 2010, and the PhD degree in computer science from Tsinghua University, in 2016. He is now a postdoctoral fellow with the Chinese University of Hong Kong. His current research interests include storage reliability and storage security. He is a member of the IEEE.



**Patrick P. C. Lee** received the BEng degree (first-class honors) in information engineering from the Chinese University of Hong Kong, in 2001, the MPhil degree in computer science and engineering from the Chinese University of Hong Kong, in 2003, and the PhD degree in computer science from Columbia University, in 2008. He is now an associate professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include various applied/systems topics including storage systems, distributed systems and networks, dependability, and security. He is a senior member of the IEEE.

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

**Wenzhong Guo** received the BS and MS degrees in computer science, and the PhD degree in communication and information system from Fuzhou University, Fuzhou, China, in 2000, 2003, and 2010, respectively. He is currently a full professor with the College of Mathematics and Computer Science, Fuzhou University. His research interests include intelligent information processing, sensor networks, network computing, and network performance evaluation. He is a member of the IEEE.