Correlation-Aware Stripe Organization for Efficient Writes in Erasure-Coded Storage Systems

Zhirong Shen*, Patrick P. C. Lee†, Jiwu Shu§‡, Wenzhong Guo*‡
* College of Mathematics and Computer Science, Fuzhou University
* Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou, China
†Department of Computer Science and Engineering, The Chinese University of Hong Kong
§Department of Computer Science and Technology, Tsinghua University
zhirong.shen2601@gmail.com, pclee@cse.cuhk.edu.hk, shujw@tsinghua.edu.cn, guowenzhong@fzu.edu.cn

***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 reconsider this problem from two new perspectives: data correlation and stripe organization, and propose CASO, a *correlation-aware stripe organization* algorithm. CASO captures data correlation of a data access stream. 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 accesses. By differentiating correlated and uncorrelated data in stripe organization, we show via extensive trace-driven evaluation that CASO reduces up to 25.1% of parity updates and accelerates the write speed by up to 28.4%.**

I. 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 [7], [20], [25]. 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 [32]. 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 [9] and Facebook's Hadoop Distributed File System [24].

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* [4], 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 [3], [28], 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 the parity renewal and finally result in the incorrectness of data recovery. Thus, accelerating partial stripe writes is critical for improving not only performance, but also reliability, in erasure-coded storage systems.

Our insight is that we can exploit *data correlation* [14] 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 [14]. By extracting data correlations from an accessed stream of data chunks, we can organize correlated data chunks (which are likely to be 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 of data chunks. It then accordingly classifies data chunks into either correlated or uncorrelated data chunks.

‡Corresponding author: Jiwu Shu and Wenzhong Guo. This work is supported by National Natural Science Foundation of China (Grant No. 61602120, 61672159, 61232003, 61433008, and 61571129), the Fujian Collaborative Innovation Center for Big Data Application in Governments, Fujian Provincial Natural Science Foundation (Grant No. 2017J05102), and the Technology Innovation Platform Project of Fujian Province (Grant No.2009J1007 and 2014H2005). This work is also supported by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15).

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 further accesses.

CASO is applicable for general erasure codes, such as the classical Reed-Solomon (RS) codes [23] and XOR-based erasure codes [5], [10], [27], [33]–[35]. In addition, CASO is orthogonal and complementary to previous approaches that optimize the performance of partial stripe writes at coding level [27], [28], [34] or system level [3], [11], 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 in stripe organization 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 and conduct extensive trace-driven testbed experiments. We show that CASO decreases up to 25.1% of parity updates and accelerates the average write speed by up to 28.4% compared to the baseline stripe organization technique. Furthermore, we show that CASO preserves the performance of degraded reads [12], which are critical recovery operations in erasure-coded storage.

The rest of this paper proceeds as follows. Section II presents the basics of erasure coding and reviews related work. Section III formulates and motivates our problem. Section IV presents the detailed design of CASO. Section V evaluates CASO using trace-driven testbed experiments. Finally, Section VI concludes the paper.

II. Background and Related Work

A. Basics of Erasure Coding

We first elaborate the background details of erasure coding following our discussion in Section I. 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 focus on erasure codes that are *Maximum Distance Separable (MDS)*, meaning that any k out of $k + m$ chunks of a stripe can sufficiently reconstruct the original k data chunks, while the amount of storage redundancy to achieve the fault tolerance is minimum among all possible erasure code constructions. In other words, MDS codes can tolerate any loss of at most m chunks with optimal storage efficiency.

Reed-Solomon (RS) codes [23] are one well-known family of MDS erasure codes, and they perform encoding operations based on Galois Field arithmetic [22]. RS codes support general parameters of k and m, and have been widely deployed in production storage systems, such as Google's ColossusFS [1] and Facebook's HDFS-RAID [2]. Figure 1 illustrates a stripe of the $(6, 3)$ RS code, in which there are six data chunks (i.e., $k = 6$) and three parity chunks (i.e., $m = 3$). XOR-based MDS erasure codes are another special family of MDS erasure codes that perform encoding using only XOR operations. Examples of XOR-based erasure codes include RDP Code [5], X-Code [35], STAR Code [10], HDP Code [33], H-Code [34], and HV-Code [27]. XOR-based erasure 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 erasure codes are usually used in local storage systems, such as EMC Symmetrix DMX [18] and NetApp RAID-DP [16]. Our work is applicable for both RS codes and XOR-based erasure codes.

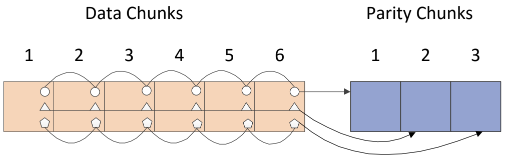


Fig. 1. Encoding of the $(6, 3)$ RS code for a stripe, in which there are six data chunks and three parity chunks. If one of the data or parity chunks is lost, any six surviving chunks within the stripe can be used to reconstruct the lost chunk.

B. 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 [31]. Since small writes dominate in real-world storage workloads [3], [28], 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 [3], 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 [34] and HV Code [27] are new erasure code constructions that associate sequential data with the same parity information, so as to favor sequential access. Shen et al. [28] develop a new data placement that attempts to arrange sequential data with the same parity information for any given XOR-based erasure code. Some approaches are based on parity logging [3], [11], 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.

C. 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 [14]. 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 [27], [28], [34] favor sequential access, yet correlated data chunks may not necessarily be sequentially placed. Previous studies [6], [14], [30] 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. Existing studies mainly focus on optimizing partial stripe writes for XOR-based erasure codes [27], [28], [34]. Nevertheless, XOR-based erasure codes often put specific restrictions on the coding parameters, while today's production storage systems often deploy RS codes for general fault tolerance (see Section II-A). Thus, optimizing partial stripe writes for RS codes is still an imperative need.

III. Motivation

This paper aims to address the following problem: *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.

A. 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 [14]. 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 [17] (see Section V for details about the traces). Each workload in the traces 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 start address of the request, and the size of the accessed data.

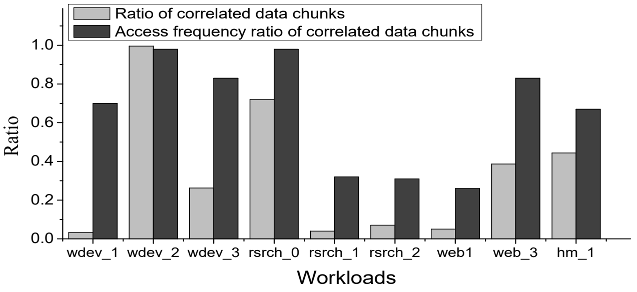


Fig. 2. An analysis on the real workloads about data correlation.

In this paper, we assume that two data chunks are said to be correlated if both of them are accessed by requests with the same timestamp value at least twice. Note that the timestamp is represented in units of ticks that correspond to 100-nanosecond intervals, yet the timestamp values are rounded to the nearest 1,000. In other words, we set the time distance as 100 microseconds and the access threshold as two.

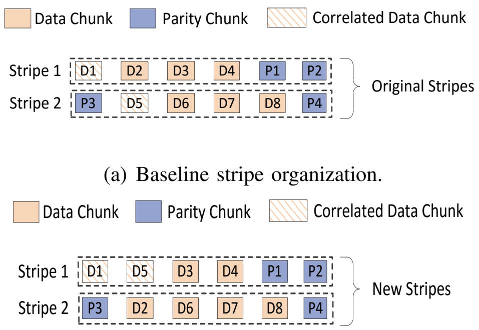
Let n_c denote the number of correlated data chunks that we infer in a workload and let f_c be the number of times accessed to these n_c correlated data chunks over the entire workload. Suppose n_a denotes the number of distinct data chunks requested in a workload, and f_a represents the number of times accessed 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 access 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 Figure 2. We make two observations.

- **The ratios of correlated data chunks vary significantly across workloads.** For example, the ratio of correlated data chunks in `wdev_2` is 98.2% and the ratio in `wdev_1` is only 3.3%.
- **Correlated data chunks receive a considerable number of data accesses.** For example, 70.0% of data accesses are issued for correlated data chunks in `wdev_1`, while in `wdev_2` the access frequency ratio of correlated data chunks reaches 98.0%.

In addition, previous work [13] reveals that most read (resp. write) requests will access read-only (resp. write-only) data chunks. As correlated data chunks exhibit similar access characteristics, a read-only (resp. write-only) data chunk is expected to be more correlated to another data chunk that is also read-only (resp. write-only).

B. Motivating Examples

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. Figure 3 shows two different stripe organization methods with the $(4, 2)$ RS code. Note that the placement of parity chunks is rotated across stripes to evenly distribute parity updates across the whole storage space, as



(a) Baseline stripe organization.

(b) New stripe organization.

Fig. 3. Motivation: Two different stripe organization methods.

commonly used in practical storage systems [21]. 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. Figure 3(a) shows a *baseline stripe organization* (BSO) methodology, which is considered for RS codes in the plugins of HDFS [36]. Specifically, BSO places sequential data chunks across $k+m$ storage devices in a round-robin fashion [19]. As shown in Figure 3(a), 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 can arrange D_1 and D_5 in the same stripe (shown in Figure 3(b)). In this case, updating both chunks only needs to renew two associated parity chunks P_1 and P_2 in Stripe 1.

Besides, correlated data chunks in a stripe are dispersed onto different disks and the access parallelism to them will be improved.

IV. Correlation-Aware Stripe Organization

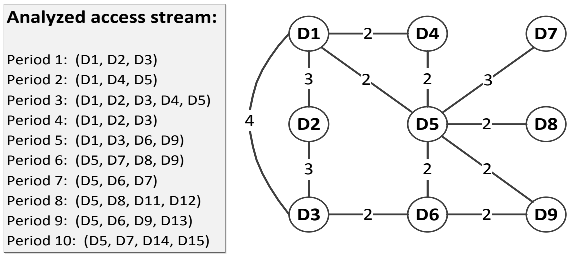
We now present CASO, a correlation-aware stripe organization algorithm. The main idea of CASO is to capture data correlations by first carefully analyzing a short period of an access stream and then separating the stripe organization for correlated and uncorrelated data chunks. Table I summarizes the major notations used in this paper and their descriptions.

A. Stripe Organization for Correlated Data Chunks

Organizing correlated data chunks is a non-trivial task and is subject to two key problems: how to identify data correlation, and 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 [14] and CP-Miner [15]. 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.

TABLE I
Major notations.

Notation	Description
k	number of data chunks in a stripe
m	number of parity chunks in a stripe
n_c	number of correlated data chunks
n_u	number of uncorrelated data chunks
$\mathcal{D}$	set of correlated data chunks $\{D_1, D_2, \cdots, D_{n_c}\}$
$\mathcal{E}$	set of connections among correlated data chunks $\mathcal{D}$
C	correlation function maps $\mathcal{E}$ to non-negative numbers
G, G_i	correlation graph over $\mathcal{D}$, the i-th correlation subgraph
λ	$\lceil \frac{n_c}{k} \rceil$, i.e., the number of correlation subgraphs
D_i	the i-th data chunk
$E(D_i, D_j)$	connection between D_i and D_j
$C(D_i, D_j)$	correlation degree between D_i and D_j
$\mathcal{S}_i$	set of data and parity chunks in the i-th stripe
$\mathcal{D}_i$	set of data chunks in G_i, set of data chunks in $\mathcal{S}_i$
$R(\cdot)$	function to calculate correlation degrees of data chunks
$\mathcal{O}$	all possible stripe organization methods


Analyzed access stream:

Period 1: (D1, D2, D3)
Period 2: (D1, D4, D5)
Period 3: (D1, D2, D3, D4, D5)
Period 4: (D1, D2, D3)
Period 5: (D1, D3, D6, D9)
Period 6: (D5, D7, D8, D9)
Period 7: (D5, D6, D7)
Period 8: (D5, D8, D11, D12)
Period 9: (D5, D6, D9, D13)
Period 10: (D5, D7, D14, D15)

Fig. 4. An example of correlation graph constructed from an access stream.

*1) **Correlation Graph**:* To theoretically 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*.

In the correlation graph $G(\mathcal{D}, \mathcal{E}, C)$, suppose that $\mathcal{D} = \{D_1, D_2, \cdots, D_{n_c}\}$ denotes the set of correlated data chunks that are identified, where 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 III-A 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.

Figure 4 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 $\mathcal{D} = \{D_1, D_2, \cdots, D_9\}$ (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 4, then we set $C(D_1, D_3) = 4$ in the figure.

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 the (k, m) RS code. Then the correlated data chunks will be organized

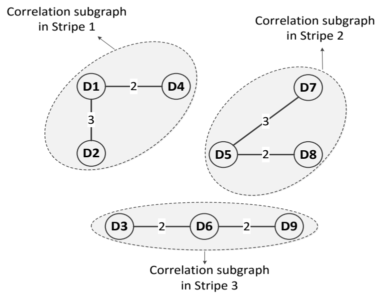

Correlation subgraph in Stripe 1
Correlation subgraph in Stripe 2
Correlation subgraph in Stripe 3
D1
D2
D3
D4
D5
D6
D7
D8
D9
2
3
3
2
2
2

Fig. 5. An example of three correlation subgraphs (suppose $k = 3$). 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.

into $\lambda = \lceil \frac{n_c}{k} \rceil$ stripes, namely $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_\lambda\}$. 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 decompose the correlation graph G into λ 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 the graph partition, the correlated data chunks in a subgraph will be organized into the same stripe. Suppose that $\mathcal{D}_i = \{D_{i_1}, D_{i_2}, \cdots, D_{i_k}\}$, 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 $\mathcal{D}_i$ can be given by

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

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 λ 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:

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

For example, we configure $k = 3$ in erasure coding and group the nine correlated data chunks in Figure 4 into three subgraphs as shown in Figure 5. 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 $\sum_{i=1}^{3} R(\mathcal{D}_i) = 14$.

*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.

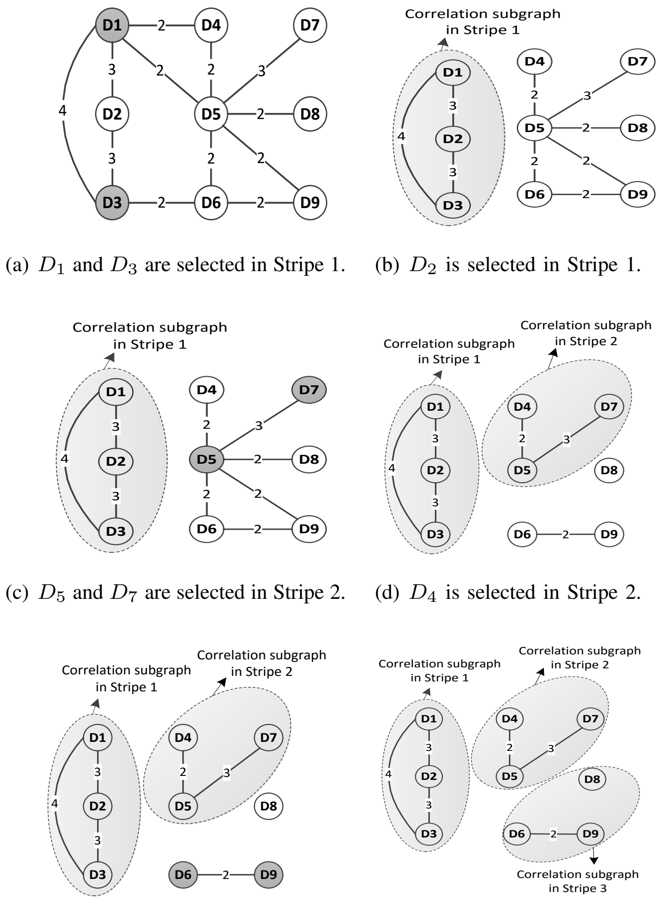
Algorithm 1: Stripe organization for correlate data chunks.

```
Input: A correlation graph $G(\mathcal{D}, \mathcal{E}, C)$.
Output: The $\lambda$ stripes that are organized.
1  Set $\mathcal{D} = \{D_1, D_2, \cdots, D_{n_c}\}$
2  Set $\mathcal{D}_i = \emptyset$ for $1 \le i \le \lambda$
3  for $i = 1$ to $\lambda - 1$ do
4      Select $D_{i_1}$ and $D_{i_2}$ with the maximum correlation degree
         in $G(\mathcal{D}, \mathcal{E}, C)$
5      Update $\mathcal{D} = \mathcal{D} - \{D_{i_1}, D_{i_2}\}$, $\mathcal{D}_i = \mathcal{D}_i \cup \{D_{i_1}, D_{i_2}\}$
6      for each data chunk $D_x \in \mathcal{D}$ do
7          Calculate $R(\mathcal{D}_i \cup \{D_x\})$
8      Find the data chunk $D_y$, where
         $R(\mathcal{D}_i \cup \{D_y\}) = \text{Max}\{R(\mathcal{D}_i \cup \{D_x\}) | D_x \in \mathcal{D}\}$
9      Set $\mathcal{D} = \mathcal{D} - \{D_y\}$, $\mathcal{D}_i = \mathcal{D}_i \cup \{D_y\}$
10     Repeat step 6∼step 9 until $\mathcal{D}_i$ includes $k$ data chunks
11     Remove the connections between the data chunks in $\mathcal{D}_i$
         and those in $\mathcal{D}$ over $G(\mathcal{D}, \mathcal{E}, C)$
12 Organize the remaining correlated data chunks into $\mathcal{D}_\lambda$
13 For each stripe, generate the corresponding parity chunks
```

In the initialization of Algorithm 1, $\mathcal{D}$ includes all the correlated data chunks. The set $\mathcal{D}_i$ $(1 \le i \le \lambda)$, which is used to include the data chunks in the stripe $\mathcal{S}_i$, is set as empty (step 1∼step 2). For the stripe $\mathcal{S}_i$ $(1 \le i \le \lambda - 1)$, 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 4). These two data chunks will be excluded from $\mathcal{D}$ and added into $\mathcal{D}_i$ (step 5). 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 $\mathcal{D}_i$ (step 6∼step 7). 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∼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 data chunks (step 12), and encodes the k data chunks in $\mathcal{D}_i$ $(1 \le i \le \lambda)$ by generating m parity chunks (step 13).

An example: We show an example in Figure 6 based on the correlation graph in Figure 4. In this example, we set $k = 3$ and thus $\lambda = \lceil \frac{n_c}{k} \rceil = 3$. At the beginning, $\mathcal{D} = \{D_1, D_2, \cdots, D_9\}$ and $\mathcal{D}_i = \emptyset$ for $1 \le i \le 3$.

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


D1
D2
D3
D4
D5
D6
D7
D8
D9
2
3
4
Correlation subgraph in Stripe 1
Correlation subgraph in Stripe 2
Correlation subgraph in Stripe 3

(a) D_1 and D_3 are selected in Stripe 1.

(b) D_2 is selected in Stripe 1.

(c) D_5 and D_7 are selected in Stripe 2.

(d) D_4 is selected in Stripe 2.

(e) D_6 and D_9 are selected in Stripe 3.

(f) D_8 is selected in Stripe 3.

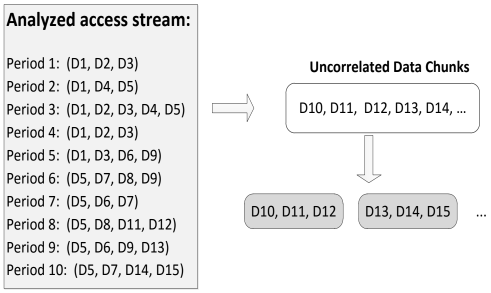
Fig. 6. An example of organizing correlated data chunks in CASO.

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 $\mathcal{D} = \{D_2, D_4, D_5, \cdots, D_9\}$ and set $\mathcal{D}_1 = \{D_1, D_3\}$ (see Figure 6(a)). 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 $C(D_1, D_2) + C(D_2, D_3) = 6$. We next turn to D_4 in $\mathcal{D}$, which only connects D_1 and has the correlation degree of $C(D_1, D_4) = 2$. 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 $\mathcal{D} = \{D_4, D_5, \cdots, D_9\}$ 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 Figure 6(b).

Following this principle, we obtain $\mathcal{D}_2 = \{D_4, D_5, D_7\}$ (see Figure 6(d)) and $\mathcal{D}_3 = \{D_6, D_8, D_9\}$ (see Figure 6(f)). We can see that $\sum_{i=1}^{3} R(\mathcal{D}_i) = 17$.

Algorithm 2: Stripe organization for uncorrelated data chunks.

```
for each uncorrelated data chunk $D_i$ do
    Organize it into the $(\lambda + \lceil \frac{i-n_c}{k} \rceil)$-th stripe
For each organized stripe, calculate the $m$ parity chunks
Store the chunks of each stripe on $k+m$ storage devices with
  only one chunk being kept on one device
```


Analyzed access stream:

Period 1: (D1, D2, D3)
Period 2: (D1, D4, D5)
Period 3: (D1, D2, D3, D4, D5)
Period 4: (D1, D2, D3)
Period 5: (D1, D3, D6, D9)
Period 6: (D5, D7, D8, D9)
Period 7: (D5, D6, D7)
Period 8: (D5, D8, D11, D12)
Period 9: (D5, D6, D9, D13)
Period 10: (D5, D7, D14, D15)

Fig. 7. An example of stripe organization for uncorrelated data chunks.

B. Stripe Organization for Uncorrelated Data Chunks

We also consider the organization of uncorrelated data chunks to optimize partial stripe writes. 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`, `wdev_3`, `rsrch_1`, `rsrch_2`, and `web_1` in Figure 2).

Therefore, we propose to organize the uncorrelated data chunks in a round-robin fashion [19]. Suppose that $\{D_{n_c+1}, D_{n_c+2}, \cdots, D_{n_c+n_u}\}$ denotes the set of uncorrelated data chunks, where n_c and n_u are the numbers of correlated and uncorrelated data chunks, respectively. For the data chunk D_i, we say i is the chunk identity of D_i. In addition, we can configure the number of uncorrelated data chunks that satisfy spatial locality based on the deployment environment. For example, for the data chunks with sequential logical chunk addresses in the direct-attached storage [8] or the data chunks that belong to the same file in distributed storage systems [36], we can make their chunk identities contiguous.

Algorithm 2 gives the main steps to organize uncorrelated data chunks. It scans the uncorrelated data chunks. For the uncorrelated data chunk D_i $(n_c+1 \leq i \leq n_c+n_u)$, it will be organized into the $(\lambda + \lceil \frac{i-n_c}{k} \rceil)$-th stripe. After determining the stripe identity that every uncorrelated data chunk belongs to, the algorithm then calculates the m parity chunks for each stripe and stores the $k+m$ chunks (including the k data chunks and m parity chunks) of each stripe on $k+m$ storage devices with only one chunk being assigned to one device.

TABLE II
CHARACTERISTICS OF SELECTED WORKLOADS.

Workloads	wdev_1	wdev_2	wdev_3	web_1	web_3	rsrch_0	rsrch_1	rsrch_2	src2_1
Number of read operations	0	189	11	87,057	10,049	99,779	42	136,364	643,669
Average read size (KB)	0	6.12	63.27	45.90	74.89	11.47	6.72	4	60.31
Number of write operations	1,055	181,076	670	73,833	21,330	948,796	13,737	71,222	14,104
Average write size (KB)	5.13	8.16	2.03	9.22	20.83	8.74	6.19	4.25	13.34

An example: We set $k = 3$ in erasure coding. Figure 7 shows an example based on the access stream in Figure 4. From Figure 4, the correlated data chunks are $\{D_1, D_2, \cdots, D_9\}$ and are organized into $\lambda = 3$ stripes. We then identify the uncorrelated ones $\{D_{10}, D_{11}, \cdots, D_{9+n_u}\}$, where n_u is the number of uncorrelated data chunks for being encoded. To organize D_{10}, it will be organized in the 4-th stripe. Following this method, we can obtain the stripes that preserve a high degree of data sequentiality as shown in Figure 7.

C. Complexity Analysis

Complexity of Algorithm 1. Algorithm 1 has to construct λ stripes. Suppose that there are n_c correlated data chunks. For each stripe, to select the first two correlated data chunks with the maximum correlation degree, the algorithm needs no more than n_c^2 trials (step 4 in Algorithm 1). To select the remaining $k-2$ correlated data chunks for each stripe, the algorithm should repeat the following steps.

- Scan no more than n_c data chunks in $\mathcal{D}$ (step 6).
- For each data chunk, calculate the sum of correlation degrees with no more than k data chunks that have been chosen in the candidate stripe (step 7).

Therefore, the complexity of determining the stripe identity for each correlated data chunks (i.e., step 1$\sim$ step 12) is $O(\lambda n_c^2 + \lambda n_c k^2)$. In step 13, as the number of stripes organized by correlated data chunks is λ and each stripe will calculate m parity chunks, the complexity of calculating parity chunks is $O(\lambda m)$. Finally, the complexity of Algorithm 1 is $O(\lambda n_c^2 + \lambda n_c k^2 + \lambda m)$.

Complexity of Algorithm 2. We then analyze the complexity of Algorithm 2. Suppose that n_u denotes the number of uncorrelated data chunks. The algorithm needs to scan every uncorrelated data chunk, so the complexity of determining the stripe identities for uncorrelated data chunks (i.e., step 1$\sim$step 2) is $O(n_u)$. The number of stripes organized by uncorrelated data chunks is $O(\lceil \frac{n_u}{k} \rceil)$ and each stripe will calculate m parity chunks, so the complexity of calculating parity chunks is $O(\lceil \frac{n_u}{k} \rceil \cdot m)$. Finally, the complexity of Algorithm 2 is $O(n_u + \lceil \frac{n_u}{k} \rceil \cdot m)$.

V. Performance Evaluation

In this section, we carry out extensive testbed experiments to evaluate the performance of CASO. We would like to answer the following questions:

- How many parity updates can be reduced by CASO for different erasure codes?
- How many parity updates can be reduced by CASO when the number of access requests analyzed for data correlation changes?
- How much write speed can be accelerated by CASO?
- Will CASO affect the performance of *degraded reads* (i.e., read operations that include temporarily unavailable data chunks)?

A. Experiment Preparation

In this evaluation, the parameter $k+m$ is configured in the range from 6 to 12, which covers typical system configurations of existing storage systems [1]. Specifically, we mainly consider three erasure codes: the $(4, 2)$ RS code, the $(6, 3)$ RS code that is employed in Google Colossus FS [1], and the $(8, 4)$ RS code.

Our evaluation is driven by real-world block-level workloads from MSR Cambridge Traces [17], which describe various access characteristics of enterprise storage servers. The workloads are collected from 36 volumes that span 179 disks of 13 servers for one week. Each workload records the start position of the I/O request and the request size. Here, we select 9 volumes, most of which have small write size (i.e., smaller than 10KB). Therefore, this selection can better evaluate the performance of partial stripe writes with small write size. Table II lists the characteristics of the selected workloads.

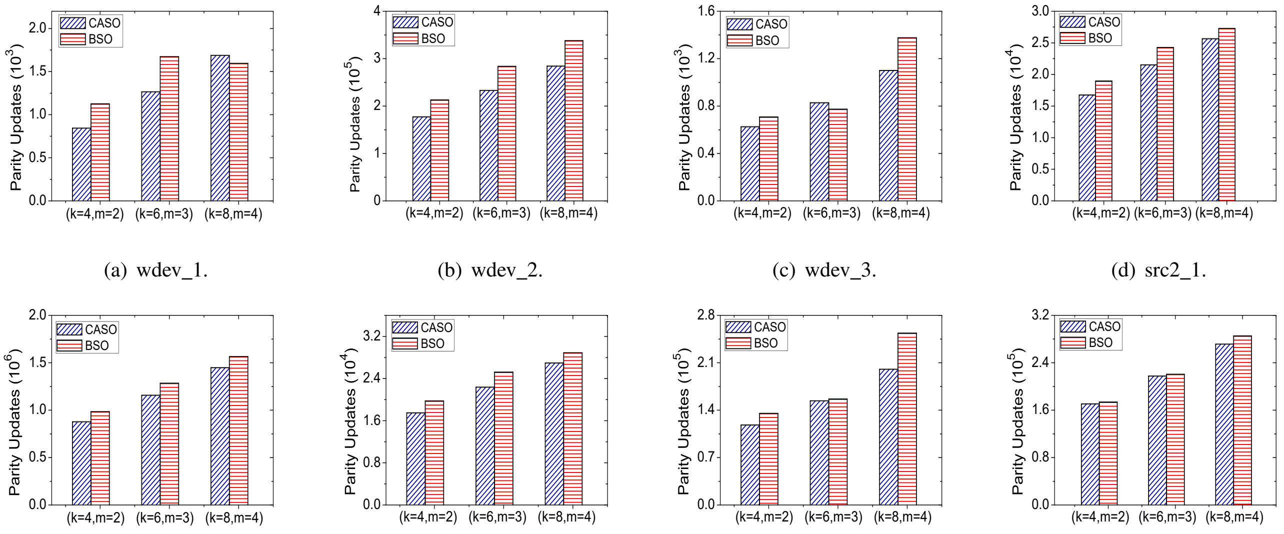
Evaluation Methods. In the evaluation, the chunk size is set as 4KB, which is consistent with the deployment of erasure codes in real storage systems [3], [29].

For each workload, we only select a small portion of access requests for correlation analysis. To describe the ratio of access requests of a workload that are analyzed in CASO, we first define the concept of "analysis ratio" as follows.

$$\text{analysis ratio} = \frac{\text{num. of analyzed access requests in CASO}}{\text{num. of access requests of a workload}}.$$

After correlation identification, we group the correlated data chunks that are inferred in the analysis (see Algorithm 1), and organize the remaining data chunks by the logical chunk addresses (see Algorithm 2). To fairly evaluate CASO, we replay the access requests that are not used in the correlation analysis for each workload. We compare CASO with *baseline stripe organization* (BSO) in the evaluation.

Evaluation Environment: The 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 300GB storage capability and 10,000 rmp. The machine and the disk



(a) wdev_1.

(b) wdev_2.

(c) wdev_3.

(d) src2_1.

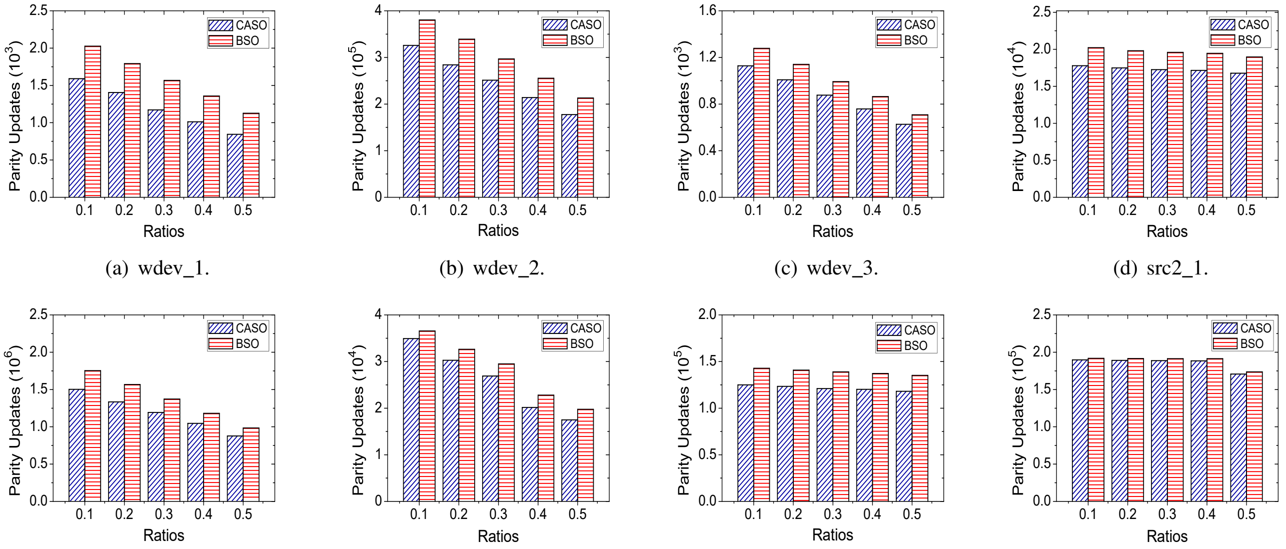
(e) rsrch_0.

(f) rsrch_1.

(g) rsrch_2.

(h) web_1.

Fig. 8. Experiment 1 (Impact of different erasure codes on parity updates). Smaller values means less parity updates are incurred.



(a) wdev_1.

(b) wdev_2.

(c) wdev_3.

(d) src2_1.

(e) rsrch_0.

(f) rsrch_1.

(g) rsrch_2.

(h) web_1.

Fig. 9. Experiment 2 (Impact of different analysis ratios on parity updates). Smaller values means less parity updates are incurred.

array are connected by a Fiber cable with the bandwidth of 800MB/sec. The selected erasure codes are realized based on Jerasure 1.2 [22].

B. Experiment Results

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 three erasure codes with different parameters: the $(4, 2)$ RS code, the $(6, 3)$ RS code, and the $(8, 4)$ RS code. The results are shown in Figure 8. We make two observations.

First, CASO can reduce 10.4% of parity updates on average for different erasure codes under different real workloads. In particular, when using the $(4, 2)$ RS code in the workload `wdev_1`, CASO reduces 25.1% 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 and the number of parity chunks to be updated is reduced.

Second, the larger value of m will generally cause more parity updates. The reason is that in addition to the data chunks being updated, a partial stripe write operation should also renew the m parity chunks in a stripe so as to promise the correctness of data recovery.

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 0.5, and

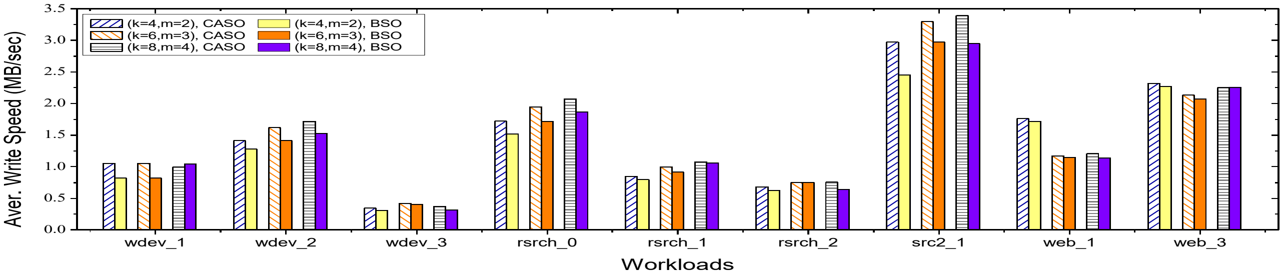
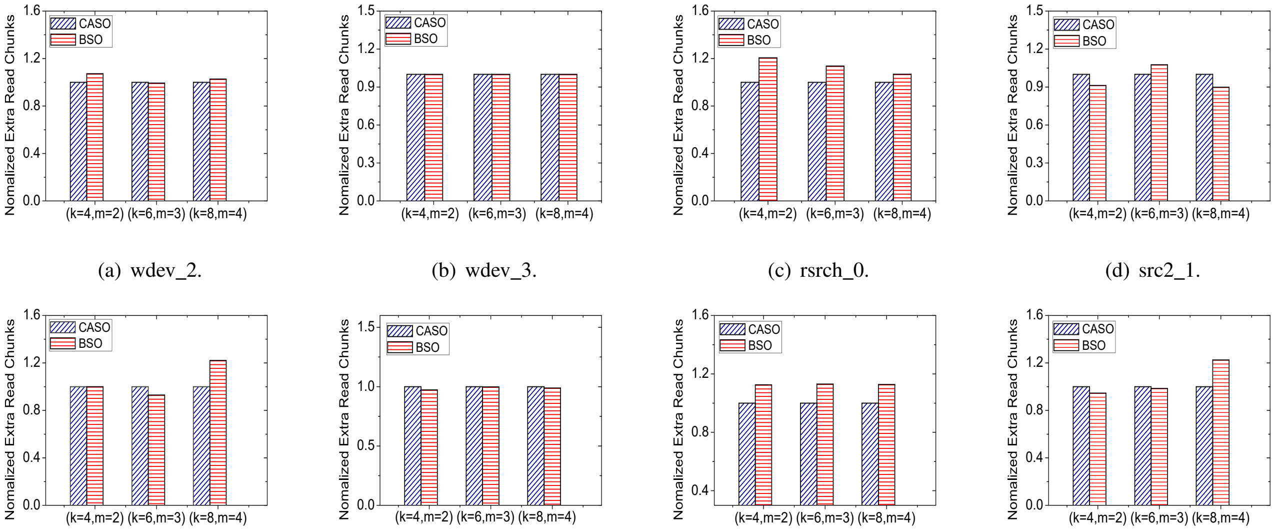


Fig. 10. Experiment 3 (Average write speed). Larger values enable faster write operations.



(a) wdev_2.

(b) wdev_3.

(c) rsrch_0.

(d) src2_1.

(e) rsrch_1.

(f) rsrch_2.

(g) web_1.

(h) web_3.

Fig. 11. Experiment 4 (Normalized ratio of additional data read in degraded reads.) Smaller values indicate less data are retrieved in degraded reads.

measure the number of resulting parity updates incurred in the $(4, 2)$ RS code for CASO and BSO. Figure 9 illustrates the results.

We observe that CASO generally reduces more parity updates when taking more access requests into correlation analysis. Take the workload wdev_1 as an example. CASO cuts down about 21.5% of parity updates when the analysis ratio is 0.1, and this reduction increases to 25.1% when the analysis ratio reaches 0.5.

In addition, as referred above, the evaluation measures the parity updates by using the remaining access requests that are not used in correlation analysis for each workload. Therefore, fewer access 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 speed). We further measure the average speed for our testbed to complete a write operation in different workloads. We set the analysis ratio as 0.5 and run the tests for different erasure codes. Each test is repeated for five runs and the results are averaged in Figure 10.

We can see that even for different erasure codes, CASO can effectively accelerate the write speed for most of the workloads. For example, when issuing the write operations in wdev_1 to the system deployed with the $(4, 2)$ RS code, CASO can accelerate the write speed by 28.4% compared to BSO. In addition, CASO can improve the write speed by 21.2% when replaying the write operations in src2_1 to the system deployed with the $(4, 2)$ RS code. This is because CASO can significantly decrease the number of parity updates in partial stripe writes. Note that the average write sizes of most workloads are smaller than 10KB. Thus, the write throughput in our test is only several megabytes per second, which is reasonable in real storage systems.

Experiment 4 (Additional I/Os in degraded reads). In this test, we evaluate the performance of degraded reads in CASO. Degraded reads [9], [12], [26] 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 or parity chunks to recover the lost chunk, and finally, the number of I/Os increases. To evaluate degraded reads, we erase the data on a disk, replay the read requests that are not used in the correlation analysis for each workload, and record the average amount of data to be additionally read in one disk's failure. We repeat this procedure for all $k + m$ disks. The average results are shown in Figure 11, which normalize the number of chunks that are additionally read in CASO as 1.

We can see that CASO will not downgrade the overall performance of degraded reads to the workloads. It can even decrease 4.2% of additional chunks on average that are retrieved in degraded reads. Specifically, CASO can decrease 7.5% of additional data on average in degraded reads for the workloads `wdev_2`, `rsrch_0`, `web_1`, and `web_3`. The reason is that Algorithm 1 in CASO can differentiate and separate read-only data chunks with write-only data chunks in stripe organization, which will help to improve the degraded read performance. Suppose that some data chunks in a read request fail. Then the storage system can reuse its correlated data chunks in the request for data reconstruction. Finally, the extra number of I/Os in degraded reads can be decreased. We expect that CASO can reach better degraded read performance when being deployed in the scenario in which the correlated data chunks are read-only and non-sequential.

VI. Conclusion

In this paper, we reconsider the optimization of partial stripe writes in erasure-coded storage systems from the perspectives of *data correlation* and *stripe organization*. We then propose CASO, a *correlation-aware stripe organization* algorithm. CASO identifies 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 spatial locality. Experimental results show that CASO can reduce up to 25.1% of parity updates in partial stripe writes and accelerate the write speed by up to 28.4% for the traces with small write size, while still preserving the performance of degraded reads.

References

[1] Colossus, successor to google file system. http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf.

[2] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling. HDFS RAID. In *Hadoop User Group Meeting*, 2010.

[3] 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. of USENIX FAST*, 2014.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of USENIX FAST*, 2004.

[6] 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. of USENIX ATC*, 2007.

[7] 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. of USENIX OSDI*, 2010.

[8] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[9] 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. of USENIX ATC*, 2012.

[10] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *Computers, IEEE Transactions on*, 57(7):889–901, 2008.

[11] C. Jin, D. Feng, H. Jiang, and L. Tian. RAID6L: A log-assisted raid6 storage architecture with improved write performance. In *Proc. of IEEE MSST*, 2011.

[12] 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. of USENIX FAST*, 2012.

[13] 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. of USENIX FAST*, 2016.

[14] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proc. of USENIX FAST*, 2004.

[15] 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 *USENIX OSDI*, 2004.

[16] C. Lueth. RAID-DP: Network appliance implementation of raid double parity for data protection. Technical report, 2004.

[17] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[18] D. Panchigar. Emc symmetrix dmxraid 6 implementation, 2009.

[19] D. A. Patterson, G. Gibson, and R. H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[20] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of USENIX FAST*, 2007.

[21] 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 *In Proc. of USENIX FAST*, 2009.

[22] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[23] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[24] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, and A. e. a. Dimakis. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, 2013.

[25] B. Schroeder and G. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *Proc. of USENIX FAST*, 2007.

[26] Z. Shen, P. Lee, J. Shu, and W. Guo. Encoding-aware data placement for efficient degraded reads in XOR-coded storage systems. In *Proc. of IEEE SRDS*, 2016.

[27] Z. Shen and J. Shu. Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems. In *Proc. of IEEE/IFIP DSN*, 2014.

[28] Z. Shen, J. Shu, and Y. Fu. Parity-switched data placement: optimizing partial stripe writes in XOR-coded storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 2015.

[29] Z. Shen, J. Shu, and Y. Fu. Seek-efficient I/O optimization in single failure recovery for XOR-coded storage systems. In *Proc. of IEEE SRDS*, 2015.

[30] G. Soundararajan, M. Mihailescu, and C. Amza. Context-aware prefetching at the storage server. In *USENIX Annual Technical Conference*, 2008.

[31] A. Thomasian. Reconstruct versus read-modify writes in raid. *Information processing letters*, 93(4):163–168, 2005.

[32] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[33] 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. of IEEE/IFIP DSN*, 2011.

[34] 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. of IEEE IPDPS*, 2011.

[35] L. Xu and J. Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[36] Z. Zhang and W. Jiang. Native erasure coding support inside hdfs. In *Strata + Hadoop World*, 2015.