Encoding-Aware Data Placement for Efficient Degraded Reads in XOR-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
[†]Key Laboratory of Spatial Data Mining and Information Sharing, Minstry of Education, 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 increasingly used by distributed storage systems to maintain fault tolerance with low storage redundancy. However, how to enhance the performance of degraded reads in erasure-coded storage has been a critical issue. We revisit this problem from two different perspectives that are neglected by existing studies: data placement and encoding rules. To this end, we propose an *encoding-aware data placement (EDP)* approach that aims to reduce the number of I/Os in degraded reads during a single failure for general XOR-based erasure codes. EDP carefully places sequential data based on the encoding rules of the given erasure code. Trace-driven evaluation results show that compared to two baseline data placement methods, EDP reduces up to 37.4% of read data on the most loaded disk and shortens up to 15.4% of read time.**

I. Introduction

Failures are prevalent in distributed storage systems [17], [23]. To maintain data availability, traditional distributed storage systems often replicate identical data copies across different disks (or storage nodes) [4], [9]. However, replication incurs substantial storage overhead, especially in the face of the unprecedented growth of today's scale of data storage. In view of this, erasure coding has been increasingly adopted by distributed storage systems in enterprises (e.g., Google [6], Microsoft [10], Facebook [21]) as a practical alternative for maintaining data availability. Erasure coding is shown to incur much lower storage redundancy, while achieving the same or even higher fault tolerance than traditional replication [26]. While there are many possible ways to construct an erasure code, practical erasure codes are often *maximum distance separable (MDS)* and *systematic*. Specifically, an erasure code can be configured by two parameters k and m. A (k, m) code treats original data as k equal-size (uncoded) *data chunks* and encodes them to form another m equal-size (coded) *parity chunks*, such that the $k+m$ dependent chunks are collectively called a *stripe*. The code is MDS if the original data chunks can be recovered from any k out of the collection of $k+m$ chunks, while incurring the minimum storage redundancy; also, the code is systematic if the k uncoded data chunks are kept in the stripe. A distributed storage system stores multiple stripes, each of which is independently encoded, and is tolerable against any m failures.

For general (k, m) codes, recovering each failed chunk needs to retrieve k available chunks of the same stripe; this differs from replication, which can recover a lost chunk by simply retrieving another available chunk replica. Thus, although erasure coding improves storage efficiency, it triggers additional I/O and bandwidth due to recovery. In particular, as opposed to permanent failures (i.e., the stored chunks are permanently lost), transient failures (i.e., the stored chunks are temporarily unavailable) account for over 90% of failure events in real-life distributed storage systems [6], possibly due to power outages, loss of network connectivity, and system reboots and maintenance. In the presence of transient failures, a storage system issues *degraded reads* to unavailable chunks, and the read performance incurs higher latency than directly reading the available chunks when no failure happens. Note that degraded reads differ from recovering the permanently lost chunks of entire disks, as the degraded read performance heavily depends on the read patterns (e.g., sequential or random access, read size, read positions). Thus, given that degraded reads trigger additional I/O and bandwidth and they are frequently performed in practice, how to improve degraded read performance becomes a critical concern when deploying erasure coding in distributed storage systems.

In this paper, we study the problem of improving degraded read performance from two specific perspectives that are neglected by previous studies (see Section II-C for related work): (i) data placement (i.e., how data is placed across disks) and (ii) encoding rules (i.e., how parity chunks are encoded from data chunks). Here, we focus on *single failures* (including a single unavailable chunk in a stripe or a single disk failure), since they are the most common failure scenarios in practice as opposed to concurrent multiple failures [6], [10], [20]. Also, our work is driven to be applicable for general *XOR-based*

§ Corresponding author: Wenzhong Guo (guowenzhong@fzu.edu.cn).
This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902, 61433008), University Grants Committee of Hong Kong (Grant No. AoE/E-02/08), VC Discretionary Fund of CUHK (Grant No. VCF2014007), and Research Committee of CUHK. This work is also supported by the Technology Innovation Platform Project of Fujian Province (Grants Nos.2009J1007, 2014H2005), the Fujian Collaborative Innovation Center for Big Data Applications in Governments.

1060-9857/16 $31.00 © 2016 IEEE
DOI 10.1109/SRDS.2016.39

erasure codes, a special class of erasure codes whose encoding and decoding operations are purely based on XOR operations. Our intuition is that by carefully examining the encoding rules of an erasure code, we can arrange the layout of data and parity chunks, so as to reduce the number of I/Os of degraded reads without violating the fault tolerance properties of the erasure code. By reducing the number of I/Os, we not only enhance the performance of degraded reads, but also reduce the amount of recovery traffic that can disturb the performance of foreground jobs [20].

To this end, we propose EDP, an *encoding-aware data placement* scheme that aims to enhance the performance of degraded reads in a single failure for any XOR-based erasure code. EDP carefully places sequential data over a stripe, so that the number of I/Os in a degraded read can be reduced. It attempts to use sequential data for parity generation so that the requested data of a degraded read can be associated with common parity information. It also adjusts the order of parity generation and refines the data placement based on the new order, so as to reduce the number of I/Os when the requested data of a degraded read is associated with different parity information. To the best of our knowledge, EDP is the first work that addresses degraded read performance for any XOR-coded storage systems through data placement designs.

Our contributions are summarized as follows.

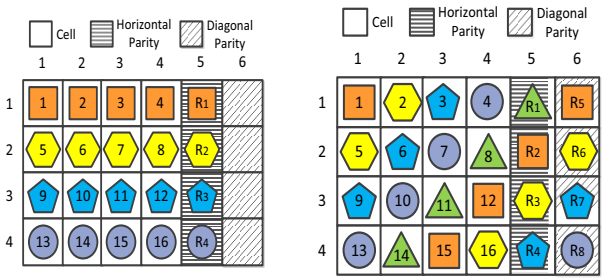
- We present EDP, a new data placement scheme that aims to improve degraded read performance for any XOR-coded storage system.
- We present a greedy algorithm for EDP that can efficiently determine how to place sequential data according to the encoding rules and how to order the parity generation. We also present an algorithm for EDP to refine data placement. Both algorithms are shown to have polynomial complexities.
- We realize EDP on a real storage systems equipped with representative erasure codes. Experiments based on real-world workloads show that compared with two baseline data placement schemes, EDP reduces up to 37.4% of read data on the most loaded disk and shortens up to 15.4% of read time.

The rest of this paper proceeds as follows. Section II will introduce the research background and related works. Section III will describe the motivating argument of this research. We will present the detailed design of EDP in Section IV and evaluate it in Section V. Finally, Section VI will conclude this paper.

II. Background and Related Work

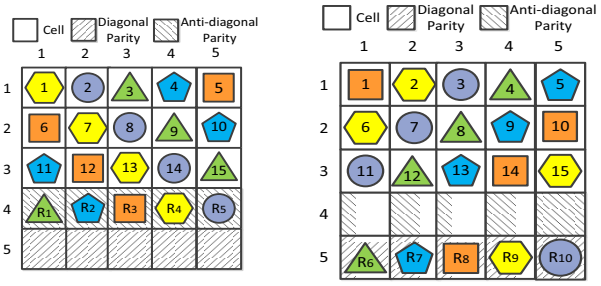
A. Basics of XOR-based Erasure Codes

XOR-based erasure codes perform purely XOR operations in encoding and decoding operations, thereby having higher computational efficiency than erasure codes that operate over finite fields (e.g., Reed-Solomon Codes [22], SD Codes [18], and STAIR Codes [14]). Existing XOR-based erasure codes support different levels of fault tolerance. They can tolerate double failures (e.g., EVENODD Code [1], RDP Code [5], X-Code [32], P-Code [12], HDP Code [28], H-Code [29], HV Code [24], D-Code [7]), triple failures (e.g., STAR Code [11] and TIP Code [34]), or a general number of failures (e.g., Cauchy Reed-Solomon Code [2]). We focus on XOR-based erasure codes that are MDS, such that their storage redundancy is minimum (see Section I).



(a) The Horizontal Parity. (b) The Diagonal Parity.

Fig. 1: Element layout of RDP Code under horizontal data placement over $p+1$ disks ($p=5$). Note that the numbers in data elements represent the logical order of how the data elements are stored, and the elements with the same shape belong to the same parity chain for a given encoding direction. We use these representations in our illustrations throughout the paper.



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

Fig. 2: Element layout of X-Code under horizontal data placement over p disks ($p=5$).

XOR-based erasure codes often configure the number of disks as a function of a prime number p. Note that the value of p may imply different numbers of disks for different codes (e.g., $p+1$ disks for RDP Code [5] and p disks for X-Code [32]). To perform encoding or decoding, XOR-based erasure codes often divide a data or parity chunk into sub-chunks called *elements* (called data elements and parity elements, respectively). In other words, each stripe contains multiple rows of elements. Since each stripe is independently encoded (see Section I), our discussion focuses on a single stripe.

To illustrate, Figure 1 shows the element layouts of RDP Code [5] for a single stripe over six disks, where $p=5$, $k=p-1=4$, and $m=2$ (see Section I for the definitions of k and m), while Figure 2 shows the element layouts of X-Code [32] for a single stripe over five disks, where $p=5$, $k=p-2=3$, and $m=2$. In our illustrations, we use *numbers* to specify the logical order of how data elements are stored on disks. Let `#i` be the i-th data element stored in a stripe based on the logical

order. We say that the data elements are *sequential* if they follow a continuous logical order. For example, in Figure 1, #1 and #2 are two sequential data elements. The logical order depends on the data placement strategy, as will be explained in Section II-B.

Each parity element is encoded (or XOR-ed) from a subset of elements of a stripe. Each XOR-based erasure code has its own *encoding rule*, which specifies the encoding direction (e.g., horizontal, diagonal, or anti-diagonal) and which elements are used for generating a parity element. Let R_i be the i-th parity element in a stripe. For example, in Figure 1(a), the parity element R_1 :=#1⊕#2⊕#3⊕#4 (where ⊕ denotes the XOR operation), implying that R_1 is encoded from the data elements of the same row in the horizontal direction. Note that a parity element may be encoded from another parity element. For example, in Figure 1(b), the parity element R_5 :=#1⊕ #15⊕ #12 ⊕ R_2, which is encoded from the data and parity elements along the diagonal direction. We define a *parity chain* as the collection of a parity element and the elements that are XOR-ed together to form the parity element. In our illustrations, we mark the elements in the same *shape* if they belong to the same parity chain for a given encoding direction. For example, the collection {#1, #2,#3, #4, R_1} forms the horizontal parity chain in Figure 1(a), and the collection {#1, #15, #12, R_2, R_5} forms the diagonal parity chain in Figure 1(b). Note that the data element #1 belongs to both of the two parity chains.

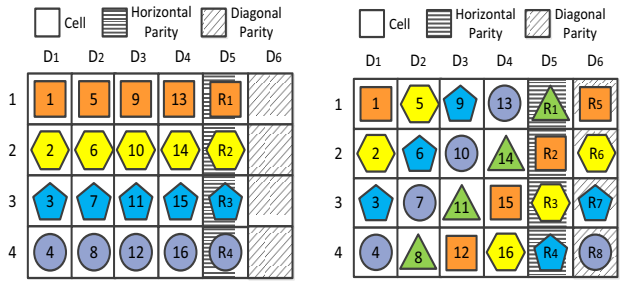
XOR-based erasure codes have different placement strategies for parity elements. They may place data and parity elements in separate disks, such as RDP Code [5] (see Figure 1), or spread parity elements across all disks, such as X-Code [32] (see Figure 2). Our work retains the same placement of parity elements for a given erasure code and hence preserves its fault tolerance. Specifically, our work focuses on a different placement strategy of data elements for more efficient degraded reads.

B. Data Placement

Data placement refers to how we place data elements across disks when they are first stored. Given a data placement, parity elements are placed accordingly based on the erasure code. To our knowledge, most existing studies do not specifically consider the data placement of XOR-coded storage systems (see Section II-C for related work). Here, we consider two baseline data placement strategies: horizontal and vertical.

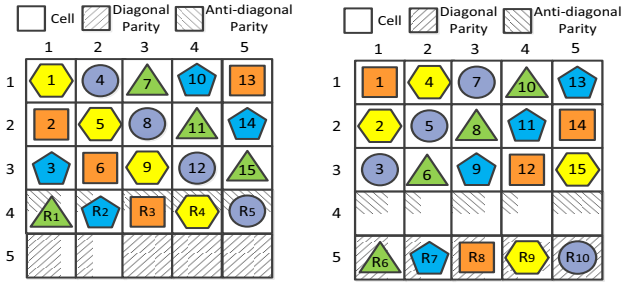
Horizontal Data Placement. Horizontal data placement proposes to place sequential data elements across disks. For example, Figures 1 and 2 illustrate the layouts of RDP Code [5] and X-Code [32] under horizontal data placement, respectively.

Horizontal data placement brings two benefits. First, it can take full advantage of parallelization [13] to reduce the access latency. For example, when a storage system requests data elements {#1,#2} in Figure 1, it can read them from disk 1 and disk 2 respectively in parallel. Second, horizontal data placement can effectively reduce the number of elements



(a) The Horizontal Parity. (b) The Diagonal Parity.

Fig. 3: Element layout of RDP Code under vertical data placement over $p+1$ disks ($p=5$).



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

Fig. 4: Element layout of X-Code under vertical data placement over p disks ($p=5$).

retrieved in degraded reads when an erasure code has horizontal parity chains. For example, in Figure 1, suppose that disk 1 fails and the storage system issues a read operation that requests elements {#1,#2,#3}. Then the storage system only needs to retrieve another two elements {#4, R_1}, such that the unavailable element #1 can be recovered through the horizontal parity chain (see Figure 1(a)).

However, the second advantage will be lost if the erasure code does not have a horizontal parity chain, such as X-Code [32]. In these codes, the sequential data elements placed on the same row may not have a common parity element. Consequently, when the storage system issues degraded reads to the sequential data elements in a same row, it may need to retrieve additional elements for data reconstruction. For example, Figure 2 illustrates the layout of X-Code where data elements are horizontally placed. Suppose that disk 1 fails and the storage system issues a read to data elements {#1,#2,#3}. To reconstruct #1, the storage system needs to read three additional elements (i.e., {#7,#13,R_4} in Figure 2(a), or {#10,#14,R_8} in Figure 2(b)).

Vertical Data Placement. Vertical data placement puts sequential data elements along the columns in a stripe. For example, Figures 3 and Figure 4 illustrate the element layouts of RDP Code and X-Code under the vertical data placement, respectively. Vertical data placement is also assumed in previous works (e.g., [36]).

However, vertical data placement has two limitations. First, it restricts parallel access. For example, reading data elements

{#1,#2,#3} will only be limited to disk 1 (shown in Figure 3). Second, vertical data placement often needs to retrieve a large number of elements in degraded reads, as the reconstructed elements residing in the same disk do not share any common parity element. For example, suppose disk 1 fails in Figure 3(a) and the storage system issues a read to the lost data {#1,#2,#3}, it needs to retrieve another 12 elements (i.e., {#5,#6,#7,#9,#10,#11,#13,#14,#15, R_1,R_2,R_3}) to recover the lost data elements and then serve the read request.

C. Related Work

We summarize existing studies on enhancing the performance of degraded reads, and also identify their limitations.

New erasure code constructions have been proposed to explicitly incorporate the optimization of degraded reads. For example, Khan et al. [13] design Rotated RS Codes, which extend Reed-Solomon Codes [22] to include additional parity elements so as to improve the performance of degraded reads across stripes. Local Reconstruction Codes [10] construct additional local parity elements so as to reduce the lengths of parity chains, so that the number of I/Os in degraded reads can be reduced. However, both Rotated RS Codes and Local Reconstruction Codes are non-MDS (see Section I), and hence incur additional storage redundancy. In addition, HV Code [24] and D-Code [7] are two RAID-6 codes (i.e., double-fault-tolerant codes) specifically designed for reducing the amount of I/Os in degraded reads. HV Code [24] proposes to shorten the parity chain lengths and place sequential data elements on horizontal parity chains, while D-Code [7] extends X-Code [32] by adding horizontal parity chains and evenly distributing parity elements across disks.

Some studies propose to optimize the recovery performance for general XOR-based erasure codes. For example, Khan et al. [13] and Zhu et al. [35] study the problem of achieving optimal single failure recovery for general XOR-based erasure codes by searching for the solution with minimum number of I/Os. Both of their approaches address the recovery of the permanently failed data in a whole-disk failure, while our work focuses on the degraded reads for general XOR-based erasure codes and pays special attention to the characteristics of read operations. Zhu et al. [36] assume vertical data placement and address the degraded read performance in heterogeneous storage systems. In contrast, our work focuses on designing encoding-aware data placement to improve degraded read performance.

Note that Shen et al. [25] also study the data placement problem for general XOR-coded storage systems. However, their proposed data placement scheme aims to improve partial-stripe write performance, while our data placement scheme aims to improve degraded read performance and hence has an inherently different design.

Some studies address degraded reads from different perspectives. Zhang et al. [33] consider the routing of degraded reads in different topologies of data centers. Li et al. [15] study the degraded read performance when MapReduce runs on erasure-coded storage, and propose a different task scheduling algorithm that allows degraded reads to exploit unused network resources. Xia et al. [30] propose to switch between erasure coding parameters so as to balance the trade-off between storage redundancy and degraded read performance. Fu et al. [8] propose a framework that improves parallelism of degraded reads for Reed-Solomon Codes [22] and Local Reconstruction Codes [10]. On the other hand, our work focuses on reducing the number of I/Os in degraded reads, and can be integrated with the above approaches for further performance gains in degraded reads.

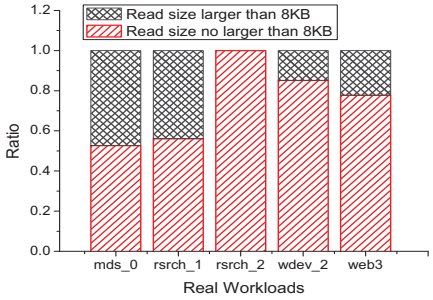


Fig. 5: Read size of real workloads in MSR Cambridge Traces [16].

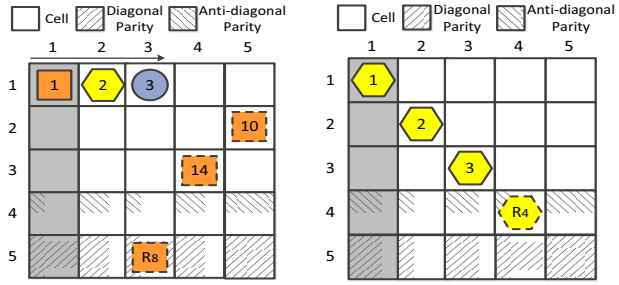
III. Problem

In this paper, our primary objective is to design a new data placement strategy that reduces the number of elements to be retrieved for degraded reads for any XOR-based erasure code. Our data placement strategy should preserve the encoding rule of the given XOR-based erasure code, so as to preserve its fault tolerance. Also, it makes no effect on normal reads, which can still access the same elements directly for systematic erasure codes.

Here, we focus on the degraded reads for a *single failure*, which is the most common failure event in practical distributed storage systems (see Section I). Note that most existing studies on enhancing the performance of degraded reads also focus on single failures (e.g., [10], [13], [21]). In addition, we assume that read requests in many scenarios are sequential and have small read sizes. For example, Figure 5 analyzes the read size distributions of several real-world I/O workloads from MSR Cambridge Traces [16] (see Section V for more details of the traces). The figure indicates that small reads are common. For example, the small reads whose read sizes are no more than 8KB account for more than 50% of all read operations.

We address the objective through three motivations. In the following, we use X-Code over $p = 5$ disks and assume that disk 1 fails as our motivating examples.

Motivation 1: Generating Parity Elements from Sequential Data Elements. We first consider how we reduce the number of elements to be retrieved in a degraded read *within a parity chain*. Our observation is that we can generate parity elements by using sequential data elements. If sequential data elements in a parity chain are requested in a degraded read, then the available elements in the request can be reused



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

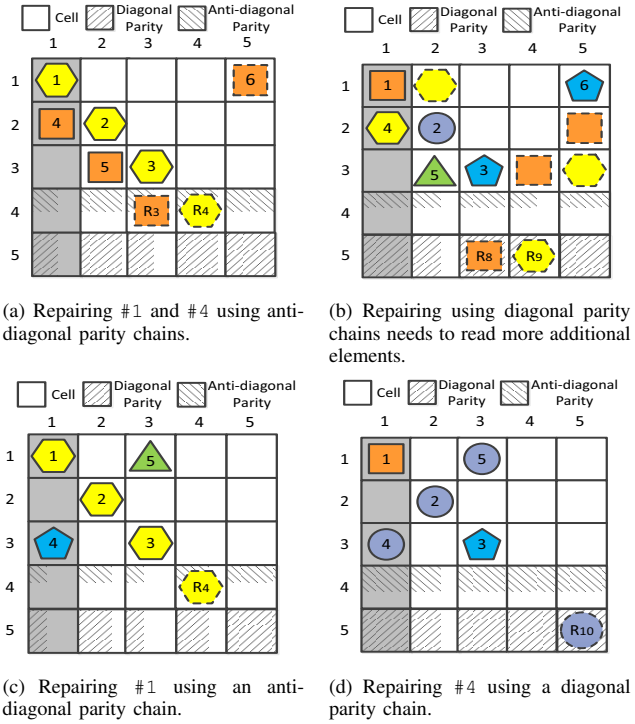
Fig. 6: Explanations of Motivation 1, based on X-Code over $p = 5$ disks. Anti-diagonal data placement (figure (b)) retrieves fewer elements than horizontal data placement (figure (a)) for the degraded read $\{\#1, \#2, \#3\}$. The shape with dashed line denotes the extra element to be read for data recovery.

to reconstruct the unavailable element in the request. This reduces the number of additional elements to be retrieved for data reconstruction.

For example, Figure 6 shows the element layouts of X-Code under both horizontal and anti-diagonal data placements. Suppose that disk 1 fails and a degraded read requests sequential data elements $\{\#1, \#2, \#3\}$. In horizontal data placement (see Figure 6(a)), the same degraded read will access three elements (i.e., $\{\#10, \#14, R_8\}$) for reconstructing the unavailable element $\#1$. On the other hand, we can place sequential data elements to generate the anti-diagonal parity element R_4, as shown in Figure 6(b). In this case, the degraded read only needs to retrieve one additional element (i.e., R_4) for the reconstruction of $\#1$.

Motivation 2: Parity Generation Orders. We also consider how to reduce the number of elements to be retrieved in a degraded read *across parity chains*. Our observation is that we can exploit the generation orders of parity elements, so as to make more elements included in different parity chains. We call the data elements that join different parity chains *overlapped data elements*, which can be used to repair many lost data elements if we read sequential data elements that span across parity chains. This further reduces the number of extra elements to be retrieved for data reconstruction.

For example, Figure 7 illustrates two data placements in X-Code, in which we follow Motivation 1 to place sequential data elements along the same parity chain if possible. Suppose that disk 1 fails and a degraded read requests data elements $\{\#1, \#2, \cdots, \#5\}$. First, we consider the data placement in which we place sequential data elements along anti-diagonal parity chains only. To repair the unavailable elements $\#1$ and $\#4$, we need to retrieve three additional elements (i.e., R_3, R_4, and $\#6$) through the anti-diagonal parity chains (see Figure 7(a)); and we do not exploit the diagonal parity chains because even more additional elements will be retrieved (another six elements to be read as shown in Figure 7(b)). Now we consider another data placement in which we first place the sequential data elements $\{\#1, \#2, \#3\}$ in an anti-diagonal chain (see Figure 7(c)), followed by placing the remaining



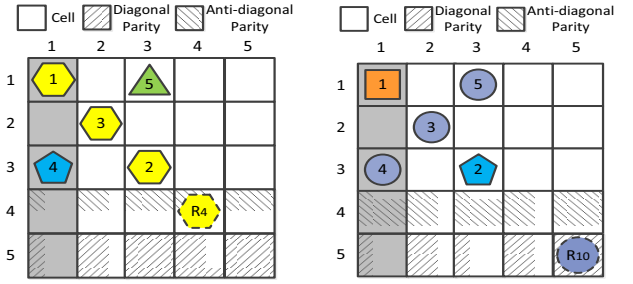
(a) Repairing $\#1$ and $\#4$ using anti-diagonal parity chains.

(b) Repairing using diagonal parity chains needs to read more additional elements.

(c) Repairing $\#1$ using an anti-diagonal parity chain.

(d) Repairing $\#4$ using a diagonal parity chain.

Fig. 7: Explanations of Motivation 2, based on X-Code over $p = 5$ disks. Anti-diagonal data placement retrieves additional elements from anti-diagonal parity chains (figure (a)), and reads more extra elements when using diagonal parity chains (figure (b)). However, if we place sequential data elements first in an anti-diagonal parity chain (figure (c)) followed by a diagonal parity chain (figure (d)), we can retrieve fewer additional elements.



(a) Repairing $\#1$ using an anti-diagonal parity chain.

(b) Repairing $\#4$ using a diagonal parity chain.

Fig. 8: Explanations of Motivation 3, based on X-Code over $p = 5$ disks. By switching the positions of $\#2$ and $\#3$, we ensure that sequential data elements are placed in both the anti-diagonal parity chain (figure (a)) and the diagonal parity chain (figure (b)).

sequential data elements $\{\#4, \#5\}$ in a diagonal parity chain (see Figure 7(d)). In this case, the overlapped data element $\#2$ can be used in the reconstruction of both $\#1$ and $\#4$. Thus, the degraded read only needs to retrieve two additional elements (i.e., R_4 in Figure 7(c) and R_{10} in Figure 7(d)).

Motivation 3: Refinement of Data Placement. Our observation is that if we only concern about how to find overlapped data elements (Motivation 2), we may not keep the sequential

data elements in the same parity chain (Motivation 1). For example, in Figure 7(d), although we have an overlapped data element #2, the diagonal parity chain to generate R_{10} holds the non-sequential data elements {#2, #4, #5}. In view of this, our third motivation is to refine the positions of data elements. Suppose that we first generate a parity element R_i, followed by R_j. We can make their overlapped data elements be the last data elements in the parity chain of R_i, so that the overlapped data elements become the first few data elements in the parity chain of R_j. This refinement ensures that we can keep the sequential data elements in each parity chain if possible.

For example, Figure 8 shows our refined data placement by switching the positions of #2 and #3. This refinement keeps the same number of overlapped data element for $\{\#1, \#2, \cdots, \#5\}$, but ensures that each of the anti-diagonal and diagonal parity chains stores the sequential data elements. Suppose that disk 1 fails and a degraded read requests {#3, #4, #5}. The original data placement in Figure 7(d) should retrieve two additional elements (i.e., #2 and R_{10}) to reconstruct the unavailable element element #4. On the other hand, our refined data placement in Figure 8 only needs to retrieve one additional element (i.e., R_{10} in Figure 8(b)).

IV. Encoding-Aware Data Placement

We propose an *encoding-aware data placement* (EDP) that addresses the problem and motivations in Section III. EDP builds on two algorithms. The first algorithm places sequential data elements in the same parity chain (Motivation 1) and exploits a *greedy* approach to select an order of generating parity elements (Motivation 2), with the goal of maximizing the number of overlapped data elements across parity chains. The second algorithm refines the positions of data elements so that the overlapped data elements lie at the intersection of parity chains (Motivation 3).

A. Greedy Parity Generation

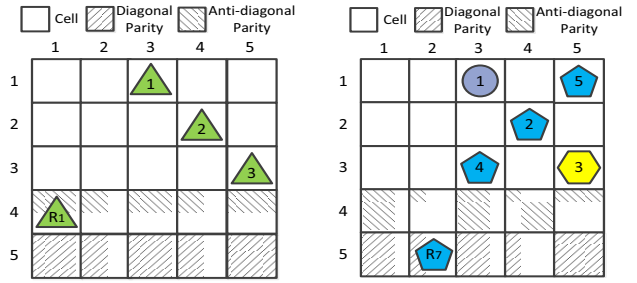
As shown in Section III, a key step of reducing the number of additional elements to be retrieved in a degraded read is to maximize the number of overlapped data elements across parity chains, by ordering the generation of parity elements. However, how to find the right generation order is a non-trivial problem. A straightforward approach is to enumerate all possible generation orders of all parity elements, yet its complexity is extremely high. For example, for X-Code with $2p$ parity elements in a stripe, the enumeration would require $(2p)!$ permutations in total.

In this paper, we propose a greedy approach to efficiently search for a generation order for parity elements, as shown in Algorithm 1. The main idea is that in each iteration, we select a parity element to be generated, such that its parity chain can produce the maximum number of overlapped data elements with respect to the parity chain of the parity elements generated in the last iteration. Here, we only need to examine the overlapped data elements in *two* parity chains (i.e., in the current and last iterations), based on the observation that our target workloads have small read sizes (see Section III) and hence most read operations span at most two parity chains.

Algorithm 1: Greedy parity generation.

Input: A given XOR-based erasure code.
Output: Parity generation order $\mathcal{O}$.

1 Set all cells of a stripe to be blank; set $\mathcal{R}$ to include all parity elements of a stripe; set $\mathcal{O} = \emptyset$
2 **for** *each candidate parity element* $R_i \in \mathcal{R}$ **do**
3 Calculate λ_i
4 Select R_j, where $\lambda_j = \max\{\lambda_i | R_i \in \mathcal{R}\}$
5 Place sequential data elements on the blank cells of the parity chain of R_j
6 Remove R_j from $\mathcal{R}$ and append R_j to $\mathcal{O}$
7 Repeat steps 2-6 until all cells in the stripe are occupied by data elements
8 Generate the remaining parity elements in $\mathcal{R}$ through placed data elements
9 Return $\mathcal{O}$



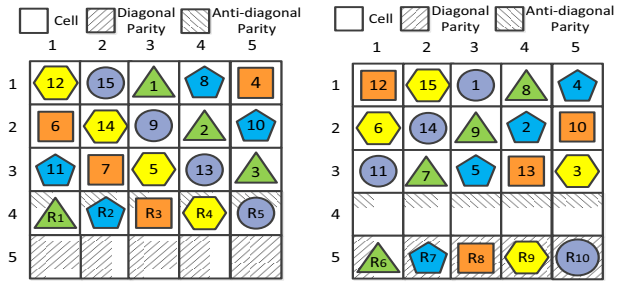
(a) R_1 is the first parity element to be generated.

(b) Generating R_7 will cause one overlapped data element (i.e., #2).

Fig. 9: An example of greedy selection for parity generation orders.

Details of Algorithm 1. Let $\mathcal{R}$ be the set of candidate parity elements that can be selected in each iteration; let $\mathcal{O}$ record the generation order of parity elements generated from sequential data elements; let λ_i be the number of overlapped data elements derived from the generation of the parity element $R_i \in \mathcal{R}$, with respect to the parity chain of the parity elements selected in the last iteration. In addition, we define a *cell* as the storage region (e.g., disk sector or block) that holds an element, and let $C_{i,j}$ be the cell whose position is at the i-th row and the j-th column in a stripe. Initially, for a given XOR-based erasure code, we first set all cells in a stripe to be *blank*, meaning that no element is stored in each cell. We also set $\mathcal{R}$ to include all parity elements of a stripe, $\mathcal{O}$ to be empty (step 1).

In each iteration, the algorithm calculates λ_i for each $R_i \in \mathcal{R}$ (steps 2-3). It selects $R_j \in \mathcal{R}$ that produces the maximum number of overlapped data elements (step 4). It places sequential data elements on the blank cells of the parity chain of R_j (step 5). The algorithm removes R_j from $\mathcal{R}$ and appends it to $\mathcal{O}$ (step 6). The algorithm repeats steps 2-6 until all cells have been occupied by data elements (step 7). It then completes the encoding by generating the remaining parity elements in $\mathcal{R}$ from the placed data elements (step 8). The



(a) Anti-diagonal parity chains. (b) Diagonal parity chains.

Fig. 10: Data placement of X-Code ($p = 5$) after running Algorithm 1.

algorithm finally returns $\mathcal{O}$ (step 9).

Example. We take X-Code ($p = 5$) as an example to show how Algorithm 1 works. As X-Code has $2p$ parity elements, we initialize $\mathcal{R} = \{R_1, R_2, \cdots, R_{10}\}$ and $\mathcal{O} = \emptyset$. In the first iteration, since $\mathcal{O}$ is empty, all parity elements have no overlapped data elements. Without loss of generality, we select R_1, so we place sequential data elements $\{\#1,\#2,\#3\}$ on the cells $\{C_{1,3}, C_{2,4}, C_{3,5}\}$, respectively (see Figure 9(a)). We then update $\mathcal{R} = \{R_2, R_3, \cdots, R_{10}\}$ and $\mathcal{O} = \{R_1\}$.

In the second iteration, we find that R_7 gives the maximum number of overlapped data elements (with $\lambda_7 = 1$). We place the sequential data elements on the cells $\{C_{3,3}, C_{2,4}, C_{1,5}\}$ (see Figure 9(b)). Finally, we obtain the data placement when Algorithm 1 finishes, as shown in Figure 10.

B. Position Refinement for Data Elements

Given the parity generation order $\mathcal{O}$ from Algorithm 1, EDP further proposes to refine the positions of data elements, as shown in Algorithm 2.

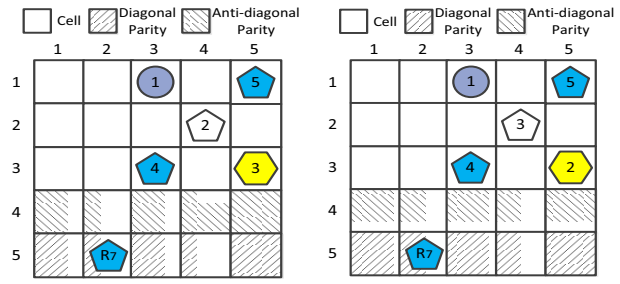
Details of Algorithm 2. Initially, the algorithm marks all data elements to as "movable", meaning that the positions of the data elements can be changed (step 1). It sets R_{cur} as the first parity element in $\mathcal{O}$ and R_{nxt} as the next one after R_{cur} (step 2). For each data element `#i` in the parity chain of R_{cur} (step 3), the algorithm checks if `#i` is movable and appears in the parity chain R_{nxt} (i.e., `#i` is an overlapped data element for the parity chains of R_{cur} and R_{nxt}) (step 4). If yes, the algorithm finds another data element `#j` that appears as the last one among all movable data elements in the parity chain of R_{cur} (step 5). It then switches the positions of `#i` and `#j` (step 6) and marks both data elements as non-movable (step 7). After that, it then tries the next pair of parity elements in $\mathcal{O}$ (step 8). The algorithm repeats steps 3-8 until reaching the last parity element in $\mathcal{O}$ (step 9).

Example. Based on the data placement in Figure 9, Figure 11 shows how we can further refine the data placement. Initially, R_1 and R_7 are the first two parity elements to be generated in $\mathcal{O}$. Thus, we set $R_{cur} = R_1$ and $R_{nxt} = R_7$. We scan the data elements $\{\#1, \#2, \#3\}$ in the parity chain of R_1, and find that `#2` involves in the parity chain of R_7 and is movable. Also, we find the data element `#3` as the last data element that is not fixed in the parity chain of R_1. We can switch the positions

Algorithm 2: Position refinement for data elements.

Input: Parity generation orders $\mathcal{O}$.
Output: A new data layout after refinement.

1 Let R_{cur} be the first parity element in $\mathcal{O}$ and R_{nxt} be the next parity element after R_{cur}
2 Mark all data elements as "movable"
3 **for** *each data element* `#i` *in the parity chain of* R_{cur} **do**
4 **if** `#i` *is movable and appears in the parity chain of* R_{nxt} **then**
5 Let `#j` be the last movable data element in the parity chain of R_{cur}
6 Switch `#i` with `#j`
7 Mark both `#i` and `#j` as non-movable
8 Set R_{cur} to be R_{nxt}, and R_{nxt} to be the next parity element after R_{cur} in $\mathcal{O}$
9 Repeat steps 3-8 until R_{cur} is the last parity element in $\mathcal{O}$.



(a) Generating R_7 by $\{\#2,\#4,\#5\}$. (b) Generating R_7 by $\{\#3,\#4,\#5\}$.

Fig. 11: An example to refine the data placement when generating R_7. In this example, $R_{cur} = R_1$ and $R_{nxt} = R_7$, where sequential data elements $\{\#1, \#2, \#3\}$ is placed to generate R_1.

of `#2` and `#3`, and a new data placement will be obtained as shown in Figure 11(b). After the refinement, both `#2` and `#3` will be non-movable in the subsequent iterations. By scanning every two parity elements whose generation orders are adjacent in $\mathcal{O}$, we can adjust the positions of overlapped data elements and finally obtain a refined data placement, as shown in Figure 12.

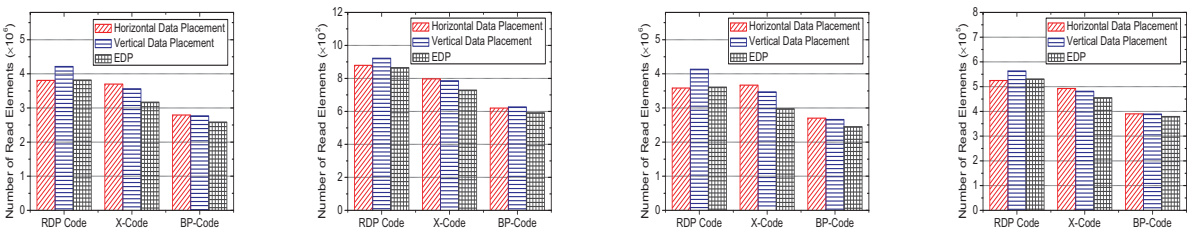
C. Complexity Analysis

Given an XOR-based erasure code, suppose that there are a total of K data elements and M parity elements in a stripe. For Algorithm 1, it scans every candidate parity element in $\mathcal{R}$ and will proceed no more than M times. Therefore, its complexity is $O(M^2)$. For Algorithm 2, it needs to scan every parity element in $\mathcal{O}$ and the associated data elements. As the number of data elements in a parity chain is no more than K, its complexity is $O(KM)$. In summary, EDP maintains a polynomial complexity.

V. Performance Evaluation

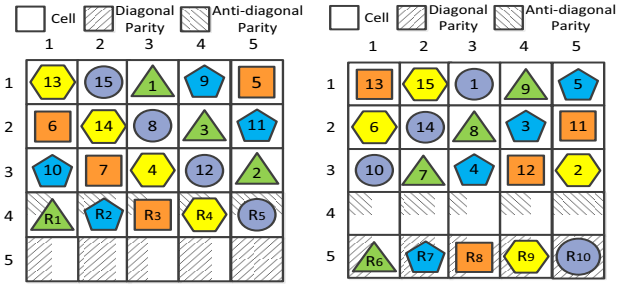
We conduct experiments to evaluate the performance of EDP and aim to address the following three questions:

1) How much reduction of I/Os on degraded reads can EDP achieve?



(a) mds_0. (b) rsrch_1. (c) web_1. (d) web_3.

Fig. 13: The number of elements to be retrieved when replaying the workloads. Smaller values are better.



(a) Anti-diagonal parity chains. (b) Diagonal parity chains.

Fig. 12: Data placement of X-Code ($p = 5$) after refinement.

2) How much reduction on the degraded read time can EDP gain?
3) Will EDP maintain its effectiveness for different XOR-based erasure codes?

Evaluation Methodology: In the evaluation, we choose three representative XOR-based erasure codes, namely RDP Code [5], X-Code [32], and Balanced P-Code (BP-Code) [31]. These three codes have different properties:

- RDP Code [5] is a RAID-6 code (i.e., double-fault-tolerant) that separates the storage of data elements and parity elements on different disks (see Figure 1). Its construction is based on horizontal parity chains and diagonal parity chains. It does not reach the optimal update efficiency.
- X-Code [32] is another RAID-6 code constructed based on diagonal parity chains and anti-diagonal parity chains (see Figure 2). Unlike RDP Code, X-Code spreads parity elements across all disks and reaches the optimal update efficiency.
- Balanced P-Code (BP-Code) [31] is a RAID-6 code designed for supercomputing data centers. Unlike RDP Code and X-Code, BP-Code is built by vertical parity chains only. It extends P-Code [12] by evenly spreading the data elements of a common parity chain across rows. It also achieves the optimal update efficiency.

In our evaluation, we select $p = 7$ for RDP Code, X-Code, and BP-Code. Note that the same p may imply different numbers of disks in a stripe, as shown in Table I. Note that our choice of p, and hence the number of disks in a stripe, is similar with the stripe size in many well-known erasure-coded storage systems [3], [6], [10]. For example, the number of disks in a stripe of Google Colossus FS is 9 [30].

TABLE I: Configurations of erasure codes with respect to p.

Coding Scheme	number of disks in a stripe	k	m
RDP Code [5]	$p+1$	$p-1$	2
X-Code [32]	p	$p-2$	2
BP-Code [31]	$p-1$	$p-3$	2

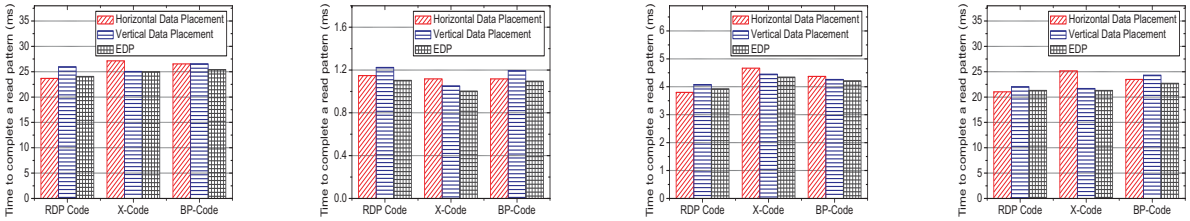
Our evaluation is driven by real-world block-level workloads from MSR Cambridge Traces [16]. The workloads are collected from 36 volumes that span 179 disks of 13 servers for one week, and describe various access characteristics of enterprise storage servers. Each workload records the start position of the I/O request and the request size. Here, we select four volumes and mainly focus on the read operations (which allow us to evaluate the impact of degraded reads). Table II lists the characteristics of our selected workloads, which show the types of workloads and the statistics of the read operations.

In the evaluation, the element size is set as 16KB. We then erase the data on one of the nodes in a stripe to simulate a single failure, and replay the read operations in the selected workloads. We run the evaluation by erasing the data for every node, repeat this evaluation, and obtain the overall average. We compare EDP with the two baseline data placement schemes (see Section II-B): horizontal and vertical data placements. In the comparison, we always choose the degraded read solution that reads less data for data recovery. For example, if an unavailable element is included in two parity chains, we will select the one that repairs the element with less elements that are additionally retrieved

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 array are connected by a Fiber cable with the bandwidth of 800MB/sec. The selected erasure codes are realized based on Jerasure 1.2 [19], a widely-used library to realize erasure coding storage systems.

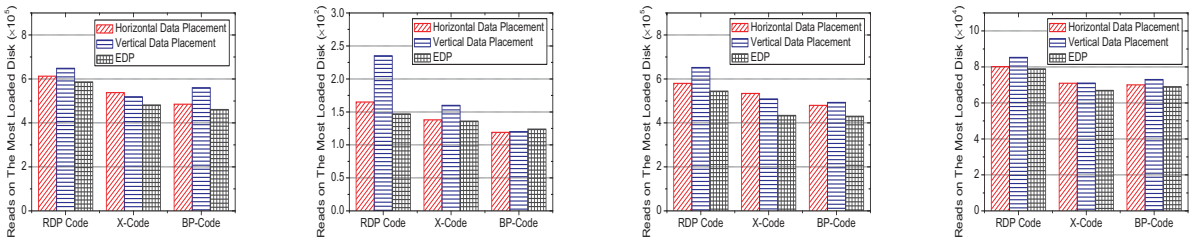
TABLE II: Characteristics of selected workloads.

Workloads	mds_0	rsrch_1	web_1	web_3
Types	Media server	Research project	Web/SQL server	Web/SQL server
Number of read operations	132,318	43	87,058	10,050
Average read size (KB)	25.3	13.9	45.9	74.9



(a) mds_0. (b) rsrch_1. (c) web_1. (d) web_3.

Fig. 14: The time to complete a read pattern. Smaller values are better.



(a) mds_0. (b) rsrch_1. (c) web_1. (d) web_3.

Fig. 15: The number of elements to be retrieved on the most loaded disk. Smaller values are better.

A. *Total number of read elements*

We first evaluate the number of elements to be read in degraded reads when replaying the read patterns in the workloads. The read elements include the originally requested elements from front-end and the additionally retrieved elements for recovery. The final results are shown in Figure 13.

We can derive two observations from Figure 13. First, EDP significantly decreases the number of elements to be read, compared with another two data placements for most of the selected workloads. Take the workload web_1 as an example, the reduction on the number of read elements brought by EDP will be up to 18.9% (compared with the horizontal data placement over X-Code). This is because EDP always tries to reuse the originally requested elements to repair the lost data, and thus reduces the number of elements to be additionally retrieved.

Second, the horizontal data placement reads fewer elements than the vertical data placement when deployed over RDP Code, and this advantage will be lost for X-Code and BP-Code. This is because for RDP Code, the requested data that are in the same row with the unavailable data, associate with a common horizontal parity element and can be reused for data recovery. For X-Code and BP-Code, in which data elements in the same row will join different parity chains, the horizontal data placement cannot reuse the requested data in the same row for data recovery and thus causes nearly the same amount of read data as vertical data placement.

B. *The time to complete a read pattern*

We further replay the four workloads and record the time to complete each read pattern. Figure 14 shows the final results.

We have two observations. First, the read time not only relates to the average read size, but also associates with the ratio of read patterns with small read size. For example, it needs about 20ms to complete a read pattern in the workload web_3, while the read time in the workload rsrch_1 is no more than 1.6ms. These two workloads have different read sizes and ratios of the read operations with small read sizes (e.g., read size no more than 8KB in Figure 5).

Second, EDP can markedly reduce the read time by up to 15.4% (in the workload web_3) when deployed over X-Code and BP-Code. It also indicates that the improvement on the degraded read speed will be up to 18.2%, as the read speed is proportional to the inverse of the read time. However, the improvement on RDP Code is not very significant. This test also implies that EDP is more suitable to the codes with the optimal update efficiency (e.g., X-Code and BP-Code).

C. *The number of read elements on the most loaded disk*

We also measure the number of read elements on the most loaded disk after replaying the workloads. Suppose the storage

system consists of N disks and the i-th disk is requested to read Q_i elements during the workload replaying. This metric can be calculated by $Q_{max} = \text{Max}\{Q_i | 1 \leq i \leq N\}$. The evaluation results are presented in Figure 15.

We can observe that EDP can well lighten the I/O burden on the most loaded disk when compared with another two data placements. Compared with the vertical data placement, EDP reduces up to 37.4% read I/Os (in the workload rsrch_1) on the most loaded disk. Besides, EDP can also lighten the read burden by up to 18.7% on the most loaded disk compared with the horizontal data placement (in the workload web_1). This test indicates that EDP should be more effective for the storage systems [27] that are sensitive to the read traffic.

VI. Conclusion

Erasure codes have been intensively used in current storage systems for their high storage efficiency. In view of the commonplace of single failure and read operations in real-world applications, this paper proposes EDP, an *encoding-aware data placement* scheme to optimize single-failure degraded reads. EDP suggests generating parity elements by using sequential data elements. It then designs an order to generate parity elements and refines the data layout to achieve further optimization. Experimental results show that compared with two baseline data placement methods, EDP reduces up to 37.4% of read data on the most loaded disk and shortens up to 15.4% of read time.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[2] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An xor-based erasure-resilient coding scheme. Technical report, 1995.

[3] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling. Hdfs raid. In *Hadoop User Group Meeting*, 2010.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. of ACM SOSP*, 2011.

[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] 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.

[7] Y. Fu and J. Shu. D-code: An efficient raid-6 code to optimize i/o loads and read performance. In *Proc. of IEEE IPDPS*, 2015.

[8] Y. Fu, J. Shu, and Z. Shen. Ec-frm: An erasure coding framework to speed up reads for erasure coded cloud storage systems. In *Proc. of IEEE ICPP*, pages 480–489, 2015.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, 2003.

[10] 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.

[11] 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.

[12] C. Jin, H. Jiang, D. Feng, and L. Tian. P-code: A new raid-6 code with optimal properties. In *Proc. of ACM ICS*, 2009.

[13] 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.

[14] M. Li and P. P. Lee. Stair codes: a general family of erasure codes for tolerating device and sector failures in practical storage systems. In *Proc. of USENIX FAST*, 2014.

[15] R. Li, P. P. Lee, and Y. Hu. Degraded-first scheduling for mapreduce in erasure-coded storage clusters. In *Proc. of IEEE/IFIP DSN*, 2014.

[16] 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.

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

[18] J. S. Plank and M. Blaum. Sector-disk (sd) erasure codes for mixed failure modes in raid systems. *ACM Transactions on Storage (TOS)*, 10(1):4, 2014.

[19] 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.

[20] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[21] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[22] 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.

[23] 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.

[24] 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.

[25] Z. Shen, J. Shu, and Y. Fu. Parity-switched data placement: Optimizing partial stripe writes in xor-coded storage systems. *To appear at IEEE Transactions on Parallel and Distributed Systems*.

[26] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[27] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.

[28] 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.

[29] 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.

[30] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in hdfs. In *Proc. of USENIX FAST*, 2015.

[31] P. Xie, J. Huang, Q. Cao, and C. Xie. Balanced p-code: A raid-6 code to support highly balanced i/os for disk arrays. In *Proc. of IEEE NAS*, 2014.

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

[33] J. Zhang, X. Liao, S. Li, Y. Hua, X. Liu, and B. Lin. Aggrecode: constructing route intersection for data reconstruction in erasure coded storage. In *Proc. of IEEE INFOCOM*, 2014.

[34] Y. Zhang, C. Wu, J. Li, and M. Guo. Tip-code: A three independent parity code to tolerate triple disk failures with optimal update complextiy. In *Proc. of IEEE/IFIP DSN*, 2015.

[35] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu. On the speedup of single-disk failure recovery in xor-coded storage systems: Theory and practice. In *Proc. of IEEE MSST*. IEEE, 2012.

[36] Y. Zhu, J. Lin, P. Lee, and Y. Xu. Boosting degraded reads in heterogeneous erasure-coded storage systems. *IEEE Transactions on Computers*, 64(8):2145–2157, 2015.