Encoding-Aware Data Placement for Efficient Degraded Reads in XOR-Coded Storage Systems: Algorithms and Evaluation

Zhirong Shen, Patrick P. C. Lee, Jiwu Shu, and Wenzhong Guo

Abstract—Modern storage systems adopt erasure coding to maintain fault tolerance with low storage redundancy. However, how to improve 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 *encoding-aware data placement (EDP)*, which mitigates the number of I/Os in degraded reads during a single failure for general XOR-based erasure codes. EDP carefully selects appropriate parity units to be generated by sequential data based on the encoding rules and establishes their generation orders. We further refine the data placement for optimizing the degraded reads to any two sequential data units. Trace-driven evaluation results show that EDP significantly reduces I/Os in degraded reads and hence shortens the read time.

1 Introduction

Failures are prevalent in storage systems [18], [24]. To maintain data availability, traditional 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 storage systems in enterprises (e.g., Google [6], Microsoft [10], Facebook [22]) as a practical redundancy alternative for maintaining data availability. Erasure coding is shown to incur much lower storage redundancy than traditional replication, while achieving the same degree of fault tolerance [28]. There are many possible ways to construct an erasure code. Nevertheless, 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 units* and encodes them to form m additional equal-size (coded) *parity units*, such that the $k+m$ dependent units are collectively called a *stripe*. The code is MDS if the original k data units can be recovered from any k out of the collection of $k+m$ units, while incurring the minimum storage redundancy. In addition, the code is systematic if the k uncoded data units are kept in the stripe and can be directly accessed by normal read operations. A large-scale storage system typically stores multiple stripes, each of which is independently encoded, and is tolerable against any m failures.

For general (k, m) codes, recovering each failed unit needs to retrieve k available units of the same stripe. This differs from replication, which can recover a lost unit by simply retrieving another available unit replica. Thus, while erasure coding improves storage efficiency, it triggers additional I/O and bandwidth during recovery. Field studies show that frequent failures can trigger substantial network traffic due to recovery in production erasure-coded storage systems [21]. As opposed to permanent failures (i.e., the stored units are permanently lost), transient failures (i.e., the stored units are temporarily unavailable) account for over 90% of failure events in real-life 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 units, whose latencies are higher than normal reads to available units when no failure happens. Note that degraded reads differ from recovering the permanently lost units of entire disks, as the degraded read performance heavily depends on the read patterns (e.g., sequential or random access, read size, read position). Thus, given that degraded reads trigger additional I/O and bandwidth and are frequently performed in practice, how to improve degraded read performance becomes a critical concern when deploying erasure coding in storage systems.

In this paper, we study the problem of achieving efficient degraded reads from two specific perspectives that are neglected by previous studies (see Section 2.3 for related work): (i) data placement (i.e., how data is placed across disks) and (ii) encoding rules (i.e., how parity units are encoded from data units). Here, we focus on *single failures* (i.e., either a single unavailable unit in a stripe, or a single

- *A preliminary version [25] of this paper was presented at the 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS'16). In this journal version, we extend the algorithmic design of EDP and conduct more experimental evaluation.*

- *Zhirong Shen and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong. (Email: zhirong.shen2601@gmail.com, pclee@cse.cuhk.edu.hk)*
- *Jiwu Shu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. (E-mails: shujw@tsinghua.edu.cn)*
- *Wenzhong Guo is with College of Mathematics and Computer Sciences, Fuzhou University, Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, and Key Laboratory of Spatial Data Mining & Information Sharing, Ministry of Education, Fuzhou 350003, China*
- *Corresponding author: Wenzhong Guo (guowenzhong@fzu.edu.cn)*

disk failure), since they are the most common failure scenarios in practice as opposed to concurrent multiple failures [6], [10], [21]. Also, our work is driven to be applicable for general *XOR-based erasure codes*, which refer to a special class of erasure codes whose encoding and decoding operations are purely based on XOR operations for computational efficiency. Our intuition is that by carefully examining the encoding rules of an erasure code, we can arrange the data and parity layouts such that the number of I/Os of degraded reads can be reduced without violating the fault tolerance 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 [21].

To this end, we propose EDP, an *encoding-aware data placement* scheme that aims to enhance the performance of degraded reads in single failures for general XOR-based erasure codes. EDP attempts to use sequential data units for parity generation, so that the requested data units of a degraded read can be associated with common parity units. Also, it specifies the selection principles for the parity units being generated by sequential data units, and carefully establishes the generation order of parity units, so as to further reduce the number of I/Os of a degraded read. To the best of our knowledge, EDP is the *first work* that exploits data placement designs to improve degraded read performance for any XOR-coded storage systems.

Our contributions can be summarized as follows.

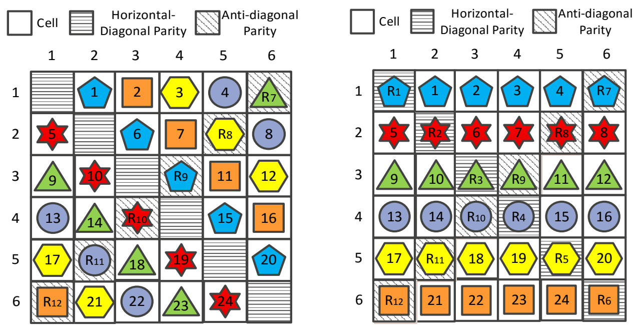
- We present EDP, a new data placement design that aims to improve degraded read performance for any XOR-coded storage system.
- We present a greedy algorithm for EDP that can efficiently determine how to place sequential data units according to the encoding rules, how to select the appropriate parity units to be generated by sequential data units, and how to establish the parity generation order. We also present an algorithm for EDP to refine data placement. Our proposed algorithms are shown to have polynomial complexities.
- We realize EDP on a real storage system equipped with representative erasure codes. Experiments based on real-world workloads show that EDP significantly reduces extra I/O caused by degraded reads and also reduces the degraded read latency.

The rest of this paper proceeds as follows. Section 2 will introduce the research background and related work. Section 3 will describe the motivating argument of this research. Section 4 will present the detailed design of EDP. Section 5 will present evaluation results. Finally, Section 6 will conclude this paper.

2 Background and Related Work

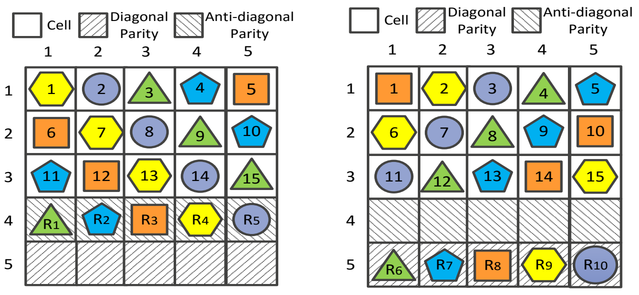
2.1 Basics of XOR-based Erasure Codes

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 [23], SD Codes [19], 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 [34], P-Code [12], Balanced P-Code [33], HDP Code [29], H-Code [30], HV Code [26], D-Code [7]), triple failures (e.g., STAR Code [11], T-Code [16], and TIP Code [36]), or a general number of failures (e.g., Cauchy Reed-Solomon Code [2]). In this paper, we mainly focus on XOR-based erasure codes (see Section 1).


Cell
Horizontal-Diagonal Parity
Anti-diagonal Parity
1 2 3 4 5 6
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12

(a) Anti-Diagonal Parity (b) Horizontal-Diagonal Parity

Fig. 1: Element layout of HDP Code under horizontal data placement over $p-1$ disks ($p=7$). 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 and color belong to the same parity chain for a given encoding direction. We use these representations in our illustrations throughout the paper.



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

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

Many 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 HDP Code [29] and p disks for X-Code [34]). To perform encoding or decoding, XOR-based erasure codes often divide a data or parity unit into sub-units, which we call *elements* (i.e., data elements and parity elements, respectively). In other words, each stripe contains multiple rows of elements. Since each stripe is independently encoded (see Section 1), our discussion focuses on a single stripe.

To illustrate, Figure 1 shows the element layouts of HDP Code [29] for a single stripe over six disks, where $p=7$, $k=4$, and $m=2$ (see Section 1 for the definitions of k and m). HDP Code is an MDS code that tolerates double disk failures. It has two kinds of parity elements termed horizontal-diagonal parity elements and anti-diagonal parity elements. It first computes the anti-diagonal parity elements using the data elements (see Figure 1(a)), and then

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842210, IEEE Transactions on Parallel and Distributed Systems

computes the horizontal-diagonal parity elements using both data elements and the computed anti-diagonal parity elements. Figure 2 also shows the element layouts of X-Code [34] for a single stripe over five disks, where $p = 5$, $k = 3$, 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 2.2.

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 anti-diagonal parity element R_9 =#1⊕#6⊕#15⊕#20 (where ⊕ denotes the XOR operation), implying that R_9 is encoded from the four data elements along the anti-diagonal direction. Following the same principle, the horizontal-diagonal parity element R_1 =#1⊕#2⊕#3⊕#4⊕R_7 as shown in Figure 1(b).

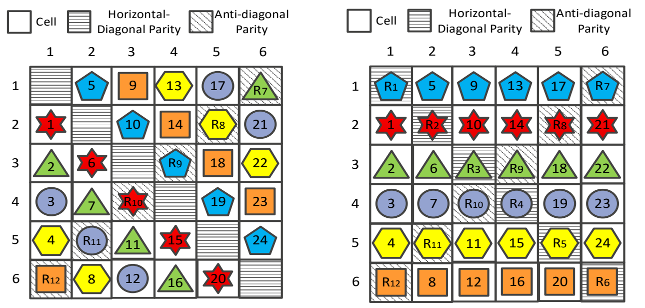
We define a *parity chain* as the collection of a parity element and the elements that are XOR-ed together to form the parity element. The number of elements in a parity chain is defined as the *length of the parity chain*. In our illustrations, we mark the elements in the same shape and color if they belong to the same parity chain for a given encoding direction. For example, the collection {#1, #6, #15, #20, R_9} forms an anti-diagonal parity chain in Figure 1(a) whose length is five, and the collection {#1, #2, #3, #4, R_7, R_1} forms the horizontal-diagonal parity chain in Figure 1(b) whose length is six. Note that the data element #1 belongs to both of the two parity chains. We can see that an erasure code may have the different lengths of parity chains. For example, in HDP code, an anti-diagonal parity chain has the length of $(p-2)$, while a horizontal-diagonal parity chain has length $(p-1)$.

Parity chains can be used for data recovery. For example, suppose that the data element `#1` fails in Figure 1. It can be repaired by an anti-diagonal parity chain by performing #1=#6⊕#15⊕#20⊕R_9 (see Figure 1(a)). Based on the same principle, it can also be recovered by the associated horizontal-diagonal parity chain in Figure 1(b).

XOR-based erasure codes have different placement strategies for parity elements. They may place data and parity elements in separate disks (e.g., RDP Code [5]) or spread parity elements across all disks (e.g., HDP Code [29] (see Figure 1) and X-Code [34] (see Figure 2)). Our work retains the same placement of parity elements for a given erasure code and hence preserves its fault tolerance, and we focus on designing a placement strategy of data elements for more efficient degraded reads.

2.2 Data Placement

Data placement refers to how we place data elements across disks when they are first stored. Given a data placement,



(a) Anti-Diagonal Parity. (b) Horizontal-Diagonal Parity.

Fig. 3: Element layout of HDP Code under vertical data placement over $p-1$ disks ($p = 7$).

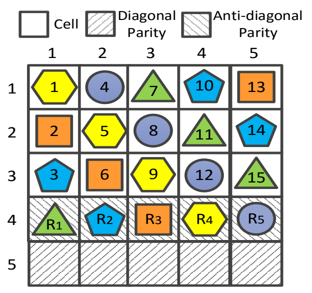
parity elements are placed accordingly based on the erasure code. To our knowledge, most existing studies do not specifically consider the data placement of XOR-coded storage systems (see Section 2.3 for related work). Here, we consider two baseline data placement strategies: horizontal and vertical.

Horizontal Data Placement. Horizontal data placement places sequential data elements across disks. For example, Figures 1 and 2 illustrate the layouts of HDP Code [29] and X-Code [34] under horizontal data placement, respectively.

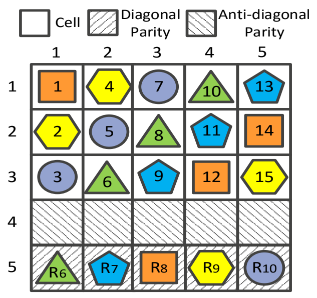
Horizontal data placement brings two benefits. First, it can take full advantage of parallelization [13] to reduce the access latency. For example, when a storage system requests data elements {#1,#2} in Figure 1, it can read them from disk 2 and disk 3 respectively in parallel. Second, horizontal data placement can effectively reduce the number of elements retrieved in degraded reads when the data elements at the same row associate with the same parity element (e.g., horizontal parity or horizontal-diagonal parity). For example, in Figure 1, suppose that disk 2 fails and the storage system issues a read operation that requests elements {#1,#2,#3}. Then the storage system can reuse the available data elements in the request (i.e., `#2` and `#3`) and only needs to retrieve another three elements {#4, R_1, R_7}, such that the unavailable element `#1` can be recovered through the horizontal-diagonal parity chain (see Figure 1(b)).

However, the second advantage will be lost if the erasure code does not have a horizontal parity chain, for example X-Code [34]. In these codes, the sequential data elements placed at the same row may not associate with a common parity element. Consequently, when the storage system issues degraded reads to the sequential data elements at the same row, it may need to retrieve *more* 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 at this time the storage system reads data elements {#1,#2,#3}. To reconstruct `#1`, the storage system needs to read three 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 4 illustrate the element layouts of HDP Code and X-Code under the vertical data placement, respectively. Vertical data placement is also assumed in



(a) The Anti-diagonal Parity.



(b) The Diagonal Parity.

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

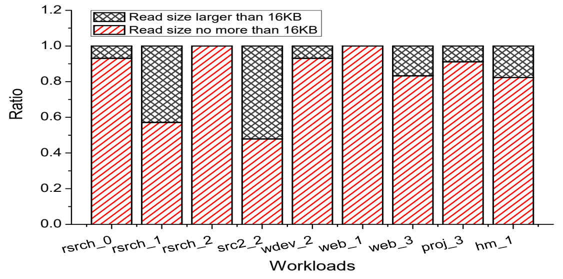


Fig. 5: Read size of workloads in MSR Cambridge Traces [17].

previous work (e.g., [38]).

However, vertical data placement has two limitations. First, it restricts parallel access. For example, reading data elements {#1,#2,#3} in HDP Code will only be limited to disk 1 (see 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 generally do not associate with any common parity chain. For example, suppose that disk 1 fails in Figure 3 and the storage system issues a read to the lost data {#1,#2,#3}. If the system chooses to repair the lost elements by using anti-diagonal parity, it needs to retrieve another 12 elements (i.e., {#6, R_{10}, #15, #20, #7, #11, #16, R_7, R_{11}, #12, #17, #21}, shown in Figure 3(a)) and then serve the read request.

2.3 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 [23] to include additional parity units so as to improve the performance of degraded reads across stripes. Local Reconstruction Codes [10] construct additional local parity units 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 1), and hence incur additional storage redundancy. In addition, HV Code [26] 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 [26] proposes to shorten the length of parity chains and place sequential data elements on horizontal parity chains, while D-Code [7] extends X-Code [34] 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. [37] 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. [38] assume vertical data placement and optimize the degraded read performance in heterogeneous storage systems. In contrast, our work focuses on designing encoding-aware data placement to improve degraded read performance.

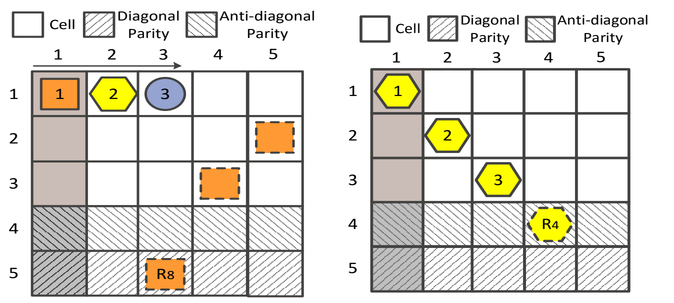
Like our work, Shen et al. [27] also study the data placement problem for general XOR-coded storage systems. However, their proposed data placement scheme aims to improve partial-stripe write performance, while our data placement scheme aims to improve degraded read performance and hence has an inherently different design.

Some studies address degraded reads from different perspectives. Zhang et al. [35] 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. [31] 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 [23] 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.

3 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 follow 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. Moreover, our data placement neither changes the decoding rule nor increases the parity units, so it does not degrade the decoding efficiency, the parity update efficiency, and storage efficiency.

Here, we focus on the degraded reads for a *single failure*, which is the most common failure event in practical storage systems (see Section 1). Note that most existing studies on enhancing the performance of degraded reads also focus on single failures (e.g., [10], [13], [22]). In addition, we assume that read requests in many scenarios are sequential and have small read sizes. For example, Figure 5 analyzes the



(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.

read size distributions of several real-world I/O workloads from MSR Cambridge Traces [17] (see Section 5 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 16KB account for a majority of all read operations.

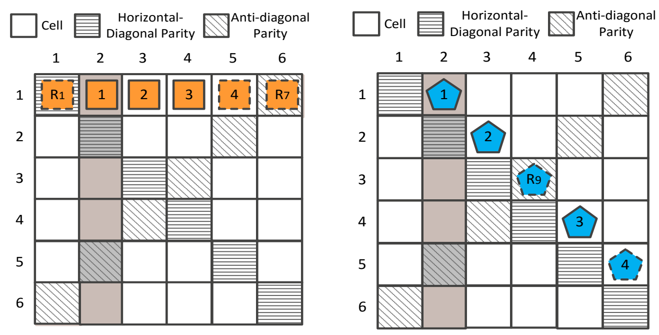
We address the objective through four motivations. In the following, we use HDP Code over $p-1$ disks ($p = 7$) and X-Code over p disks ($p = 5$) as our motivating examples.

Motivation 1: Generating Parity Elements from Sequential Data Elements. We first consider how to reduce the number of elements 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 to reconstruct the unavailable one in the request. As a result, the number of additional elements to be retrieved for data reconstruction can be reduced.

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 the horizontal data placement (see Figure 6(a)), the same degraded read should access another three elements (denoted by the dashed shape in Figure 6(a)) for reconstructing the unavailable element #1, based on the diagonal parity chain of R_8. 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.

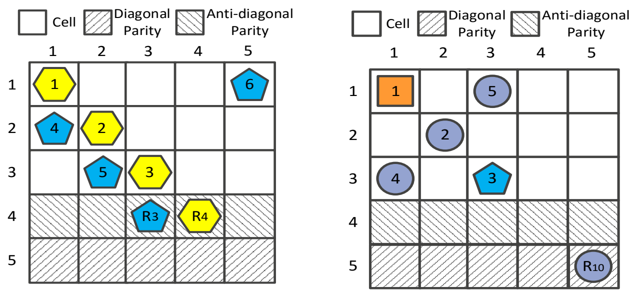
Motivation 2: Parity Selection Principles. An erasure code has multiple parity chains. Thus, the degraded read efficiency depends on which parity element is selected to be generated by sequential data elements. We propose two principles of parity selection to exploit data sequentiality.

First, we consider the erasure codes constructed over the parity chains with different lengths (e.g., HDP Code). For such kind of codes, we prefer using sequential data elements to generate the parity elements with shorter parity chains, such that fewer extra elements are needed for data recovery



(a) Repairing #1 using horizontal-diagonal parity chain (b) Repairing #1 using anti-diagonal parity chain

Fig. 7: The first selection principle of Motivation 2, based on HDP Code over $p-1$ disks ($p = 7$). A horizontal-diagonal parity chain has six elements (figure (a)), while an anti-diagonal parity chain has five elements (figure (b)). Using sequential data elements to generate anti-diagonal parity elements (figure (b)) retrieves fewer elements for the degraded read {#1, #2, #3} than to generate horizontal-diagonal parity elements (figure (a)). The shape with dashed line denotes the extra element to be read for data recovery.

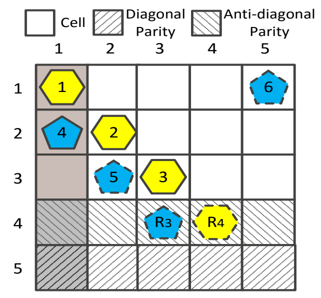


(a) Generating R_3 after R_4 (b) Generating R_{10} after R_4

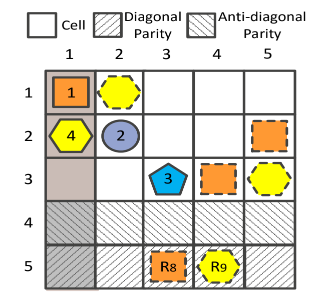
Fig. 8: The second selection principle of Motivation 2, based on X-Code over p disks ($p = 5$). Suppose that R_4 is the first parity element to be generated based on Motivation 1. The first three sequential data elements {#1, #2, #3} are accordingly placed for R_4's generation. This figure shows two examples of selecting the second parity element to be generated after R_4. Generating R_3 after R_4 can place three sequential data elements {#4, #5, #6} (figure (a)). As a comparison, generating R_{10} after R_4 can only arrange two sequential data elements {#4, #5} (figure (b)).

in degraded reads. For example, Figure 7 illustrates two cases of data layouts for HDP Code [29]. Figures 7(a) and 7(b) generate horizontal-diagonal parity elements and anti-diagonal parity elements using sequential data elements, respectively. Suppose that disk 2 fails and a degraded read requests data elements {#1, #2, #3}. The approach in Figure 7(a) needs to retrieve three elements (i.e., {R_1, #4, R_7 }) to repair #1, while that in Figure 7(b) should only read another two elements (i.e., {R_9, #4}).

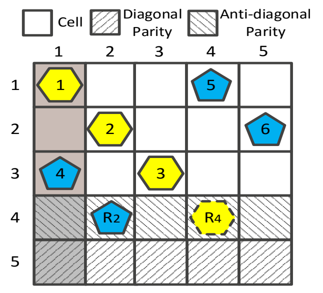
Second, for the erasure codes constructed over the parity chains with the same length (e.g., X-Code), we will select the parity element whose generation can place the most sequential data elements to make the number of sequential data elements in the same parity chain as many as possible. This placement can fully utilize the requested data elements that are available in a parity chain for data recovery. For example, suppose that R_4 is the first parity element to



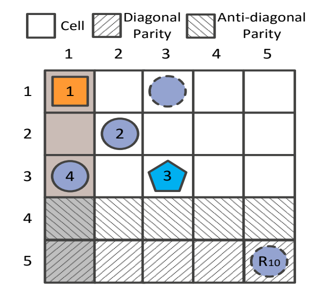
(a) Repairing `#1` and `#4` using anti-diagonal parity chains



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

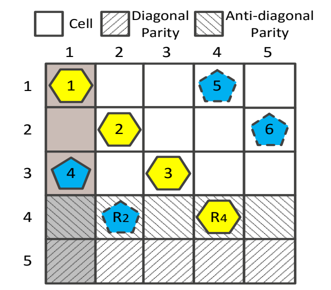


(c) Repairing `#1` using an anti-diagonal parity chain

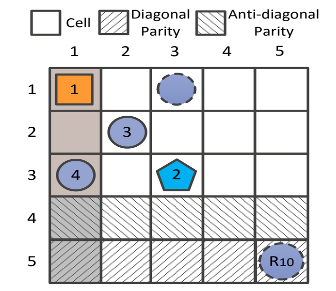


(d) Repairing `#4` using a diagonal parity chain

Fig. 9: Explanations of Motivation 3, based on X-Code over $p = 5$ disks. R_4 and R_3 are selected as the first two parity elements to be generated by sequential data elements. When `#1` and `#4` are missing, it needs four additional elements for data recovery from anti-diagonal parity chains (figure (a)), or reads even more extra elements when using diagonal parity chains (figure (b)). However, if we generate the first two parity elements in the order of R_4 and R_2, then it needs to read only three extra elements, just by repairing `#1` by an anti-diagonal parity chain (figure (c)) and recovering `#4` from a diagonal parity chain (figure (d)).



(a) Before refinement: Repairing `#4` using an anti-diagonal parity chain



(b) After refinement: Repairing `#4` using a diagonal parity chain

Fig. 10: Explanations of Motivation 4, based on X-Code over $p = 5$ disks. By switching the positions of `#2` and `#3`, `#3` can be reused for repairing `#4` for the degraded read `{#3, #4}`. Finally, repairing `#4` by the diagonal parity chain only reads two additional elements (marked in dashed line in figure (b)).

be generated based on Motivation 1 (see Figure 6). Figure 8 presents two examples of parity selection after the generation of R_4. We see that generating R_3 after R_4 (see Figure 8(a)) can place one more sequential data element in a parity chain compared to the parity selection in Figure 8(b).

Motivation 3: Parity Generation Orders. In addition to optimize degraded reads in a parity chain, we also exploit the generation orders of parity elements in order to leverage the data elements that have been placed in previous iterations. This design is to reduce the number of elements to be retrieved in a degraded read *across parity chains*. To explain this idea, we start with carefully examining the relationship between the data elements that have been placed and those to be placed when generating the next parity element.

To formalize the relationship between the placed data elements and those to be placed, we refine the concept of "overlapped data elements". In particular, suppose R_h is the parity element to be generated currently. Let `#i` be a data element to be placed in the generation of R_h and let `#j` be a data element that has been placed in previous parity generations. We call `#j` is an *overlapped data element* if `#j` and `#i` belong to a parity chain of R_q, where $R_q \neq R_h$.

Our finding is that overlapped data elements offer more choices to reduce the number of extra elements for recovery in the degraded read across parity chains. Suppose that we follow Motivation 1 and Motivation 2 to place sequential data elements along the same parity chain if possible. The placement in Figure 9(a) generates the anti-diagonal parity elements in the order of $\{R_4, R_3\}$, in which the data elements placed in the generation of R_4 are not associated with any parity element with those to be placed in the generation of R_3 (i.e., there is no overlapped data element). Suppose that disk 1 fails and a degraded read requests data elements $\{$`#1,#2,`$\cdots$`,#4`$\}$ at this time. To accomplish the recovery of `#1` and `#4`, it retrieves four additional elements (i.e., R_3, R_4, `#5`, and `#6`) through the anti-diagonal parity chains (see Figure 9(a)); or retrieves six extra elements through the diagonal parity chains (see Figure 9(b)).

As a comparison, the placement in Figure 9(c) generates the anti-diagonal parity elements in the order of $\{R_4, R_2\}$. We see that the generation of R_2 will produce two overlapped data elements (i.e., `#1` and `#2`). In particular, with respect to the data elements to be placed (i.e., `{#4, #5, #6}`) in the generation of R_2 (i.e., R_h), `#4` joins the parity chain of R_{10} (i.e., R_q) with the data element `#2` which has been placed (see Figure 9(d)). Based on the same rule, we can obtain that `#6` (to be placed) and `#1` (has been placed) join the parity chain of R_8. Suppose that disk 1 fails and a degraded read requests data elements $\{$`#1,#2,`$\cdots$`,#4`$\}$ at this time. In the placement of Figure 9(c), we first use anti-diagonal parity chain to repair `#1` by using the elements $\{$`#2,#3,` $R_4\}$ (see Figure 9(c)). As the data element `#2` is an overlapped data element that has been retrieved and it is associated with the same diagonal parity chain with the lost element `#4`, we can reuse the overlapped data element `#2` and switch to the diagonal parity chain to repair `#4` (see Figure 9(d)). Finally, only three extra elements are needed in this placement to serve the degraded read.

Motivation 4: Refinement of Data Placement. In view of the small reads account for the majority of read operations in real workloads, we finally consider a special case: the degraded reads to two sequential data elements. As Motivation 1 has covered the case when two sequential data elements are in a parity chain, here we mainly focus on another case when they are across parity chains. Our observation is to refine the positions of data elements, so as to reduce the number of elements retrieved for this kind of degraded read. Suppose that we first generate a parity element R_i, followed by R_j. We can make the last data element in the parity chain of R_i and the first one in the parity chain of R_j associate with another common parity element. When the two sequential data elements are requested and some of them fails at this time, another available data element in the request can be reused for data recovery.

For example, followed the Motivation 3, Figure 10(a) shows the first two parity elements (i.e., R_4 and R_2) to be generated by sequential data elements {`#1`, `#2`, $\cdots$, `#6`}. In this figure, the two sequential data elements across parity chains (i.e., `#3` and `#4`) are not included in any common parity chain. We then refine the data placement by switching `#2` and `#3`, such that `#3` and `#4` are in the diagonal parity chain of R_{10} (shown in Figure 10(b)).

Suppose that disk 1 fails and a degraded read requests {`#3,#4`}. The original data placement in Figure 10(a) should repair `#4` by retrieving three additional elements (i.e., `#5`, `#6`, and R_2). On the other hand, by switching the positions of `#2` and `#3`, our refined data placement in Figure 10(b) can reuse `#3` in the request and finally only needs to retrieve two additional elements by using the diagonal parity chain of R_{10}.

4 ENCODING-AWARE DATA PLACEMENT

We propose *encoding-aware data placement* (EDP) to address the problem and motivations in Section 3. EDP builds on two algorithms. The first algorithm selects the shortest parity chains (Motivation 2), places sequential data elements in the same parity chain (Motivation 1), and exploits a *greedy* approach to determine an order of generating parity elements (Motivation 3). The second algorithm refines the positions of data elements so that the two sequential data elements across parity chains associate with another common parity element (Motivation 4).

4.1 Greedy Parity Generation

As shown in Section 3, the first three key steps of reducing the number of additional elements to be retrieved in degraded read is to 1) select the parity element with the shortest parity chain in priority; 2) choose a parity element whose generation can place as many sequential data elements as possible; 3) maximize the number of overlapped data elements. However, how to realize the above three steps and find the right generation order is a non-trivial issue. 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, as shown in Algorithm 1, which incorporates the three design criteria referred above so as to efficiently search for a generation order for parity elements. The *main idea* is that in each iteration, we first incorporate Motivation 2 by choosing the parity elements with the shortest parity chain (see step 2 in Algorithm 1) and further picking out the ones whose generation can place the most sequential data elements (step 3). Given the returned parity elements, we finally follow Motivation 3 by selecting the parity element whose parity chain can produce the most overlapped data elements with respect to the data elements placed in previous iterations (step 4), and then place sequential data elements to generate the parity element (step 5) based on Motivation 1.

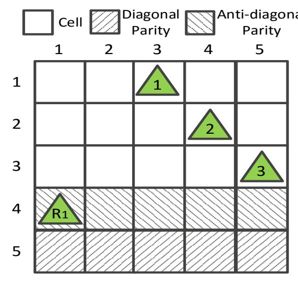
Algorithm 1: Greedy parity generation.

```
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 Select $\mathcal{L} \subset \mathcal{R}$, where $R_i \in \mathcal{L}$ has the shortest parity
     chain
3 Obtain $\mathcal{S} \subset \mathcal{L}$, such that for $R_h \in \mathcal{S}$,
     $\delta_h = \max\{\delta_i | R_i \in \mathcal{L}\}$
4 Select $R_j \in \mathcal{S}$, where $\lambda_j = \max\{\lambda_i | R_i \in \mathcal{S}\}$
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$\sim$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 $\mathcal{O}$
```

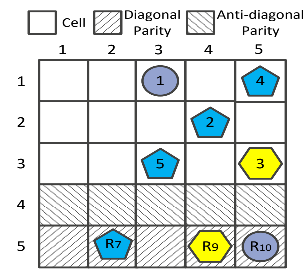
Details of Algorithm 1. Let $\mathcal{R}$ be the set of candidate parity elements that can be selected in each iteration; let $\mathcal{O}$ record the generation order of parity elements generated from sequential data elements; let δ_i be the number of sequential data elements that can be placed in the generation of the parity element $R_i \in \mathcal{R}$; let λ_i be the number of overlapped data elements derived in the generation of $R_i \in \mathcal{R}$, with respect to the data elements placed in previous iterations. 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 first extracts a set $\mathcal{L}$ from $\mathcal{R}$, such that the parity element $R_i \in \mathcal{L}$ has the shortest parity chain among those in $\mathcal{R}$ (step 2). Based on $\mathcal{L}$, the algorithm then selects the ones whose generation can place the most sequential data elements and constructs $\mathcal{S}$ (step 3). Finally, the algorithm chooses the one R_j from $\mathcal{S}$ whose generation can further produce the most overlapped data elements (step 4). After the parity selection, 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 algorithm finally returns $\mathcal{O}$ (step 9).

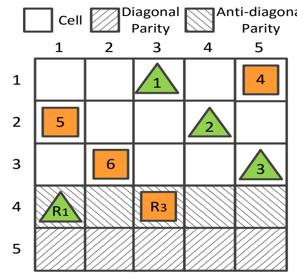
Example. We take X-Code ($p = 5$) as an example to show how Algorithm 1 works. As X-Code has $2p$ parity elements, we initialize $\mathcal{R} = \{R_1, R_2, \cdots, R_{10}\}$ and $\mathcal{O} = \emptyset$. In the first iteration, $\mathcal{O}$ is empty. As all parity elements have the same



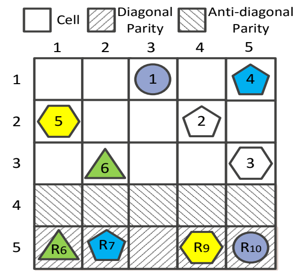
(a) Generating R_1.



(b) Generating R_7 after R_1 can only place two sequential data elements.



(c) Generating R_3 after R_1 can place three sequential data elements.



(d) Generating R_3 will produce two overlapped data elements (i.e., `#2 and #3`).

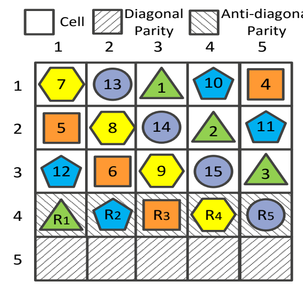
Fig. 11: An example of greedy parity selection.

length of parity chain (i.e., four, with three data elements and the generated parity element), $\mathcal{L} = \mathcal{R}$ establishes throughout this algorithm. The generation of every parity element in $\mathcal{L}$ can place three sequential data elements and produce none overlapped data element, so we select R_1 without loss of generality and place sequential data elements {`#1`,`#2`,`#3`} on the cells $\{C_{1,3}, C_{2,4}, C_{3,5}\}$, respectively (see Figure 11(a)). We then update $\mathcal{R} = \{R_2, R_3, \cdots, R_{10}\}$ and $\mathcal{O} = \{R_1\}$.

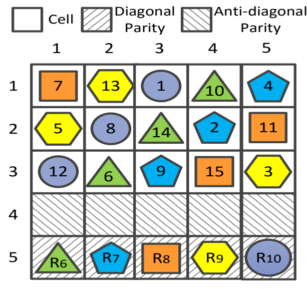
In the second iteration, as $\mathcal{L} = \mathcal{R}$, we then scan all the remaining parity elements in $\mathcal{L}$. For example, generating R_7 after R_1 can only further place two sequential data elements (i.e., `#4` and `#5` in Figure 11(b)), while generating R_3 after R_1 can lay three sequential data elements (i.e., `#4`, `#5`, and `#6` in Figure 11(c)). Following this principle, we can obtain $\mathcal{S} = \{R_2, R_3, \cdots, R_5, R_6, R_8\}$ where $\delta_i = 3$ for $R_i \in \mathcal{S}$. We further find that generating R_3 results in the most overlapped data elements (i.e., `#2` and `#3`) with $\lambda_3 = 2$ (see Figure 11(d)). Specifically, with respect to the sequential data elements to be placed in R_3's generation (i.e., {`#4`, `#5`, `#6`}), the overlapped data element `#2` that has been placed will join the parity chain of R_7 with `#4`, and another overlapped data element `#3` will be included in the parity chain of R_9 with `#5` (see Figure 11(d)). Therefore, we place the sequential data elements on the cells $\{C_{1,5}, C_{2,1}, C_{3,2}\}$ to generate R_3. Finally, we obtain the data placement when Algorithm 1 finishes, as shown in Figure 12.

4.2 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.



(a) Anti-diagonal parity chains.



(b) Diagonal parity chains.

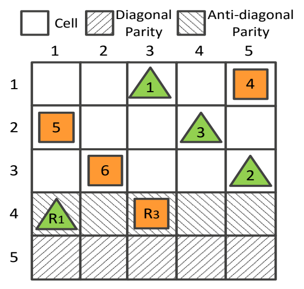
Fig. 12: Data placement of X-Code ($p = 5$) after running Algorithm 1.

Algorithm 2: Position refinement for data elements.

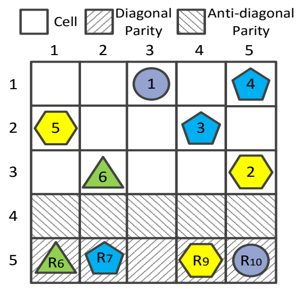
```
Input: Parity generation orders $\mathcal{O}$.
Output: A new data layout after refinement.
1  Mark all data elements as "movable"
2  Let $R_{cur}$ be the first parity element in $\mathcal{O}$ and $R_{nxt}$ be
     the next parity element after $R_{cur}$
3  for each data element #i in the parity chain of $R_{cur}$ do
4      for each data element #j in the parity chain of $R_{nxt}$ do
5          if #i and #j is movable and join a common parity
             element's generation then
6              Switch #i with the last data element in the
                 parity chain of $R_{cur}$
7              Switch #j with the first data element in the
                 parity chain of $R_{nxt}$
8              Mark both #i and #j as non-movable
9  Set $R_{cur}$ to be $R_{nxt}$, and $R_{nxt}$ to be the next parity
     element after $R_{cur}$ in $\mathcal{O}$
10 Repeat steps 3-9 until $R_{cur}$ is the last parity element in
     $\mathcal{O}$.
```

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). The algorithm then scans the data element `#i` in the parity chain of R_{cur} and `#j` in the parity chain of R_{nxt}. If both of these two data elements are movable and join the generation of a common parity element (step 5), it then switch `#i` with the last data element in the parity chain of R_{cur} and swap `#j` with the first data element in the parity chain of R_{nxt} (step 6$\sim$7). After this refinement, the last data element in the parity chain of R_{cur} and the first one in the parity chain of R_{nxt} will associate with a common parity element. Subsequently, these two data elements are marked as "fixed" and are not allowed to be moved in next refinements (step 8). The algorithm then tries the next pair of parity elements in $\mathcal{O}$ (step 9). The algorithm repeats steps 3-9 until reaching the last parity element in $\mathcal{O}$ (step 10).

Example. Based on the data placement in Figure 11(c), Figure 13 shows how we can further refine the data placement. Initially, R_1 and R_3 are the first two parity elements to be generated in $\mathcal{O}$. Thus, we set $R_{cur} = R_1$ and $R_{nxt} = R_3$. We scan the data elements {`#1`, `#2`, `#3`} in the parity chain of R_1 and {`#4`, `#5`, `#6`} in that of R_3, and find that `#2` (i.e., `#i` in Algorithm 2) and `#4` (i.e., `#j`) join the generation of

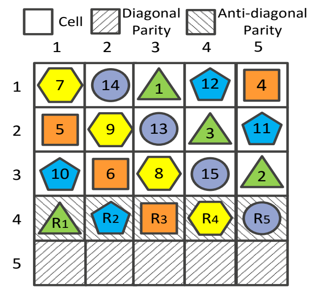


(a) After refinement: Switch the positions of #2 and #3 in Figure 11(c).

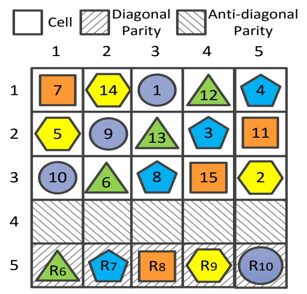


(b) After refinement: #3 and #4 associate with a common diagonal parity element R_7.

Fig. 13: An example to refine the data placement when generating R_1 and R_3. R_1 and R_3 are both anti-diagonal parity elements and generated by sequential data elements {#1, #2, #3} and {#4, #5, #6}, respectively. In this example, $R_{cur} = R_1$ and $R_{nxt} = R_3$.



(a) Anti-diagonal parity chains.



(b) Diagonal parity chains.

Fig. 14: Data placement of X-Code ($p = 5$) after refinement.

the diagonal parity element R_7 (see Figure 11(d)). We then perform the data movement by switching #2 with the last data element (i.e., #3) in the parity chain of R_1. With respect to #4, as it has been already the first data element in the parity chain of R_3, we do not perform the movement. After the refinement, the two sequential data elements #3 and #4 will associate with a common parity element (i.e., R_7) and will be non-movable in the subsequent iterations (see Figure 13). By scanning every two parity elements whose generation orders are adjacent in $\mathcal{O}$, we can obtain a refined data placement, as shown in Figure 14.

4.3 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. We first analyze the complexity of determining the parity generation orders in Algorithm 1. In Algorithm 1, to select an appropriate parity element, it need to scan every candidate parity element in $\mathcal{R}$ that includes no more than M elements. For every candidate parity element, it needs to calculate the overlapped data elements from the data elements that have been placed, which needs no more than K^2 trials. This selection will proceed no more than M times. Therefore, its complexity is $O(K^2M^2)$.

For Algorithm 2, as the number of data elements in a parity chain is less than K and the algorithm will take every two consecutive data elements of $\mathcal{O}$ in order, the complexity of Algorithm 2 is $O(K^2M)$. In summary, EDP maintains a polynomial complexity.

TABLE 1: Configurations of erasure codes with respect to p.

Coding Scheme	number of disks per stripe	k	m
RDP Code [5]	$n = p + 1$	$p - 1$	2
X-Code [34]	$n = p$	$p - 2$	2
BP-Code [33]	$n = p - 1$	$p - 3$	2
HDP Code [29]	$n = p - 1$	$p - 3$	2
V^2-Code [32]	$n = 4y - 3$, where $y \geq 2$	$n - 2$	2
T-Code [16]	$n = 3y + 1$	$n - 3$	3

5 Performance Evaluation

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

1) How many additional I/Os caused by degraded reads can EDP reduce for erasure codes with different constructions and fault tolerance degrees?
2) How much reduction on the degraded read time can EDP gain?
3) What is the scalability of EDP when the number of disk in a stripe varies?

Evaluation Methodology: In the evaluation, we choose six representative XOR-based erasure codes, namely RDP Code [5], X-Code [34], Balanced P-Code (BP-Code for short) [33], HDP Code [29], V^2-Code [32], and T-Code [16]. These six codes have different constructions and properties:

- RDP Code [5] is an MDS RAID-6 code (i.e., double-fault-tolerant) based on horizontal parity chains and diagonal parity chains. In this evaluation, we will examine whether EDP can improve the degraded read performance for the codes with horizontal parity chains.
- X-Code [34] is an MDS RAID-6 code constructed based on diagonal parity chains and anti-diagonal parity chains (see Figure 2). In this evaluation, we will study the effect when applying EDP to the codes with anti-diagonal and diagonal parity chains.
- Balanced P-Code (BP-Code) [33] is an MDS RAID-6 code designed for supercomputing data centers. Unlike RDP Code and X-Code, BP-Code is built by vertical parity chains only.
- HDP Code [29] is an MDS RAID-6 code constructed over horizontal-diagonal parity chains and anti-diagonal parity chains (see Figure 1). These two kinds of parity chains have different numbers of elements. In this evaluation, we will examine how much improvement on degraded read performance EDP can gain for the codes with different lengths of parity chains.
- V^2-Code [32] is a non-MDS RAID-6 code. It is constructed over a novel parity chain that has a letter "V" shape in geometry. In this evaluation, we would like to learn if EDP can still improve degraded read efficiency for non-MDS RAID-6 codes.
- T-Code [16] is an MDS array code that tolerates any triple failures. In this evaluation, we will show whether EDP can work for the codes with high fault tolerance.

Table 1 shows the parameters of our selected erasure codes, including the number of disks in a stripe (denoted by n) and the fault tolerance degree (denoted by m). Note that for RDP Code, X-Code, BP-Code, and HDP Code, the value of p should be a prime number, which is used to configure the number of disks in a stripe. According to the design of V^2-Code, n should be no smaller than $(4y-3)$ [32], where $y \geq 2$. For T-Code, n should be a prime number and satisfy $n = 3y + 1$ [16].

To evaluate the degraded read efficiency, we first erase the data on one of the storage devices in a stripe to simulate a single failure, and perform the read operations. We repeat the evaluation by erasing the data for each device and obtaining the overall average across all devices. 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 fewer elements. In addition, we set the element size as 16KB throughout the evaluation.

In this evaluation, we consider two versions of EDP. The first one is the preliminary version proposed in our previous conference version [25], which we term EDPv1. The second one is the version proposed in this journal version, which we term EDPv2. Compared to EDPv1, EDPv2 makes the following extensions: (1) selecting the appropriate parity elements being generated by sequential data (see Motivation 2), so as to reduce the I/Os when the requested data elements are in the same parity chain; and (2) describing a new definition of overlapped data elements (see Motivation 3) and accordingly order the parity generation for producing as many overlapped data elements as possible, so as to reduce the I/Os when the requested data elements are across parity chains. Thus, our evaluation mainly compares EDPv2 with three baseline data placement schemes: the horizontal and vertical data placements (see Section 2.2), as well as EDPv1.

Evaluation Environment: We conduct our evaluation 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. We implement our selected erasure codes via Jerasure 1.2 [20].

Evaluation Metrics: We are mainly interested in the following two metrics in our evaluation:

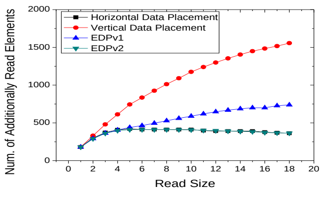
- **Total number of additionally read elements:** the total number of elements that are additionally accessed to accomplish degraded reads.
- **Average read time:** the average time to perform a constant number of read operations, including normal reads (i.e., all the read elements are available) and degraded reads.

5.1 Storage Overhead

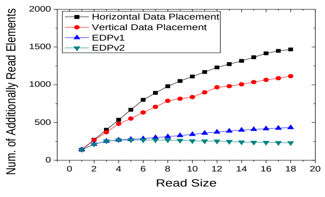
We first measure the storage overhead to keep the placement information (i.e., the mapping relationship between cells and data elements of a stripe) generated by EDPv2. We vary the number of disks in a stripe and calculate the incurred storage overhead to keep the placement information of a stripe. Table 2 presents the results. We observe that the maintenance of the data placement information only takes up marginal storage space in real storage systems. Take RDP Code as an example. When $p = 11$, the storage overhead is only 0.39KB. Note that the placement information, while being specified for a single stripe, applies to all stripes. Given the set of disks for a stripe, a storage system can first locate the stripe identity of any data element, and then use the placement information to map it to the corresponding disk and cell.

TABLE 2: Storage overhead to keep the data placement information generated by EDPv2.

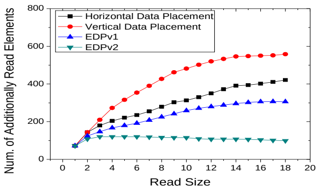
Codes	$p=5$	$p=7$	$p=11$	$n=9$	$n=13$
RDP Code	0.06KB	0.14KB	0.39KB	–	–
X-Code	0.06KB	0.14KB	0.39KB	–	–
HDP Code	0.06KB	0.14KB	0.39KB	–	–
BP-Code	0.03KB	0.07KB	0.20KB	–	–
V^2-Code	–	–	–	0.07KB	0.15KB
T-Code	–	–	–	–	0.16KB



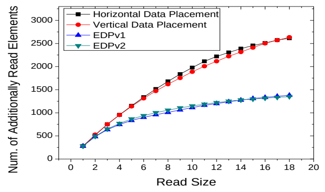
(a) RDP Code.



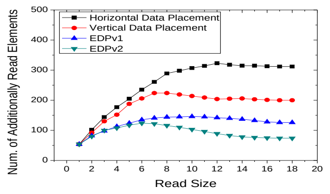
(b) X-Code.



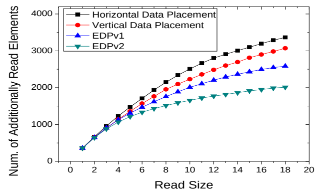
(c) HDP Code.



(d) BP-Code.



(e) V^2-Code.



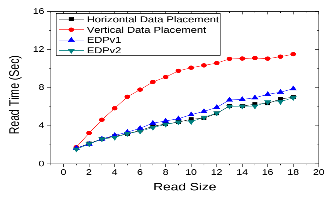
(f) T-Code.

Fig. 15: The number of additionally read elements in degraded reads under different read sizes.

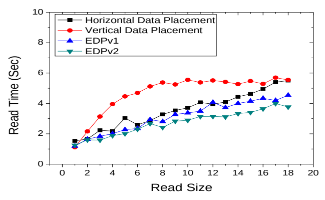
5.2 Impact of Read Size

We now evaluate the impact of read size (i.e., the number of elements in a read operation). We set $p = 7$ for RDP Code, X-Code, and HDP Code, and choose $p = 11$ for BP-Code. Under this configuration, the number of disks in a stripe of RDP Code, X-Code, HDP Code, and BP-Code is 8, 7, 6, and 10, respectively. In addition, we set the number of disks of a stripe for V^2-Code and T-Code as 9 and 13, respectively. This configuration ensures that the stripe sizes (i.e., the number of disks in a stripe) in our evaluation close to those in practical enterprise storage systems [3].

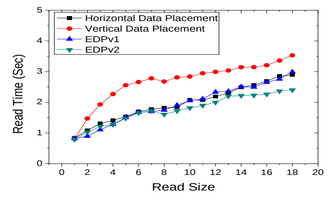
To systematically study the impact of read size, for a given read size L ($L \geq 1$), we let each data element of a



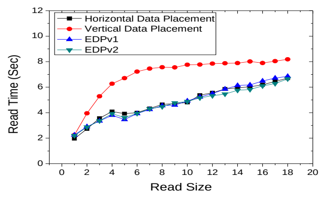
(a) RDP Code.



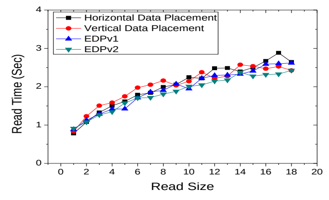
(b) X-Code.



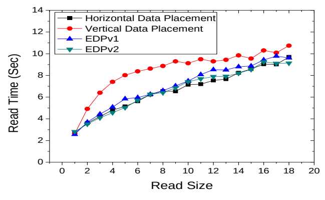
(c) HDP Code.



(d) BP-Code.



(e) V^2-Code.



(f) T-Code.

Fig. 16: Read time under different read sizes.

stripe serve as the start element, which should be read with the next $L-1$ sequential data elements in a read operation. If some of the requested data elements in a read operation is missing, the degraded read procedure will be triggered by retrieving additional elements for data recovery. Otherwise, the read operation will be served as a normal read. We vary the value of L from 1 to 18. For each given read size, we repeat the evaluation for all the six erasure codes, accumulate the number of additionally read elements, and calculate the average to complete a read operation. Figure 15 and Figure 16 show the results. We make two key findings.

First, Figure 15 shows that EDPv2 significantly reduces the number of elements to be additionally retrieved in degraded reads. In general, EDPv2 is more advantageous when the read size is larger. Take X-Code as an example. When $L=9$, EDPv2 reduces 75.0%, 67.9%, and 20.1% of extra elements to be retrieved in degraded reads compared to the horizontal data placement, vertical data placement, and EDPv1, respectively. When $L=18$, the reductions will increase to 84.2%, 79.3%, and 46.8%, respectively. Note that for RDP Code, the I/O reduction introduced by EDPv2 is marginal compared to the horizontal data placement. The reason is that the data placement established by EDPv2 for RDP Code is close to the horizontal data placement.

Second, Figure 16 shows that among the four data placement methodologies, EDPv2 needs the shortest time to serve a read operation for most of the codes. Take HDP Code as an example. EDPv2 decreases the read time by 9.0%, 32.9%, and 6.2% on average when compared to the horizontal data placement, vertical data placement, and EDPv1, respectively. Notice that the time saving introduced by EDPv2 is less significant compared to the additional I/O reduction shown in Figure 15. The reason is that for normal reads which are also evaluated in the read time test, EDP will not reduce their additional I/Os which should only be incurred when serving degraded reads. Also, as the vertical data placement usually stores sequential data elements (likely to be requested in a read operation) on a disk, it almost causes the longest average read time among all the four data placement methods.

5.3 Performance under Different Traces

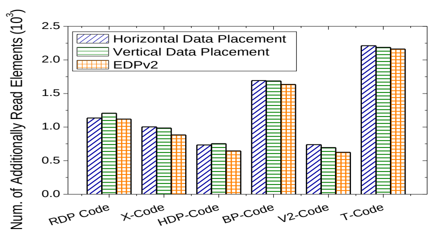
We also evaluate the performance via different real traces. We set $p=7$ for RDP Code, X-Code, and HDP Code, and choose $p=11$ for BP-Code. Also, we set the number of disks of a stripe for V^2-Code and T-Code as 9 and 13, respectively. Our evaluation is driven by real-world block-level MSR Cambridge Traces [17]. The traces are collected from 36 volumes that span 179 disks of 13 servers for one week (starting from Feb. 22, 2007), which describe the I/O requests with various access characteristics of enterprise storage servers. The total size of the 36 traces is 29GB. Each request in traces records a timestamp, the disk number, the offset to start the I/O operation (in bytes), the request size (in bytes), the request type (read or write), and the response time to complete the I/O. The 36 traces include 434 million requests, of which there are 70% of read requests. During the trace period, a total of 8.5TB (resp. 2.3TB) of data are read from (resp. write to) the trace volumes. Here, we select six volumes and mainly focus on the read operations (which allow us to evaluate the impact of degraded reads). Table 3 lists the characteristics of the selected traces, including the types of traces and the statistics of the read operations. We can see that the selected traces have different average read sizes, and it enables us to evaluate the performance of EDPv2 for the traces with varied read sizes. Figure 17 and Figure 18 show the evaluation results in terms of additional I/O and the read time, respectively.

Figure 17 indicates that among the three data placement methodologies, EDPv2 always needs the least elements that are additionally read in degraded reads for data recovery. For example, when replaying the trace `wdev_3` to the data encoded by V^2-Code (see Figure 17(b)), EDPv2 needs 48.6% (resp. 35.7%) fewer elements that are additionally read in degraded reads compared to the horizontal (resp. vertical) data placement. Moreover, EDPv2 reduces more elements when the read size increases. Take the trace `wdev_2` whose average read size is 6.12KB as an example. When being deployed over T-Code, EDPv2 can only reduce 2.3% (resp. 1.1%) of additional elements retrieved in degraded reads compared to the horizontal (resp. vertical) data placement. As a comparison, for the trace `wdev_3` with a larger average read size (i.e., 63.27KB), EDPv2 can reduce 46.3% and 38.1% of extra elements in degraded reads, when compared to the horizontal and vertical data placement.

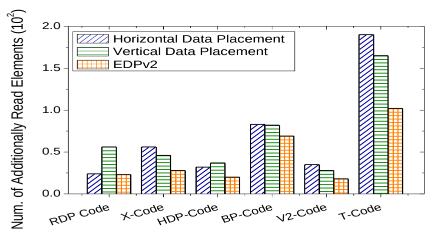
Figure 18 shows that EDPv2 decreases the time when replaying the read operations. Take the HDP Code as an example. EDPv2 reduces 16.3% (resp. 20.5%) of the read time when replaying the read operations of `rsrch_1` compared to the horizontal (resp. vertical) data placement.

5.4 Scalability

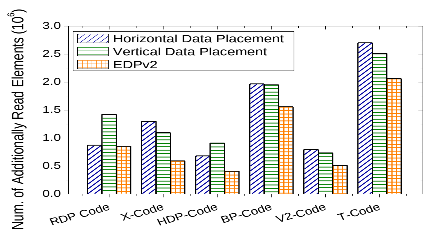
We finally evaluate the scalability of EDPv2 in terms of the number of elements to be additionally read in degraded



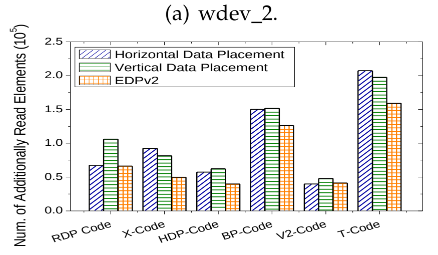
(a) wdev_2.



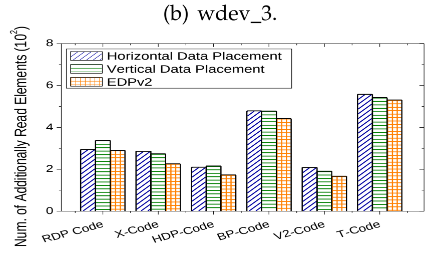
(b) wdev_3.



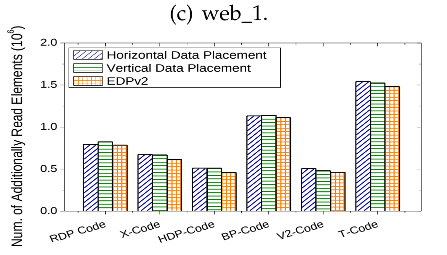
(c) web_1.



(d) web_3.

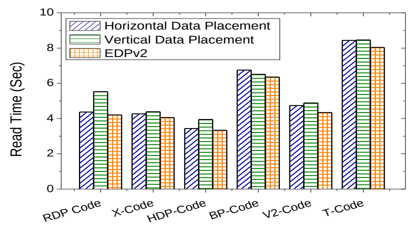


(e) rsrch_1.

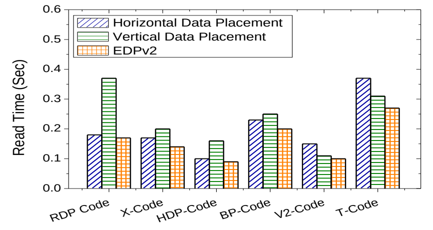


(f) rsrch_2.

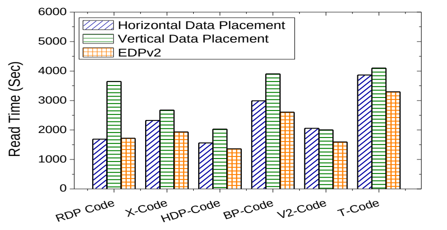
Fig. 17: The number of additionally read elements in degraded reads under different real traces.



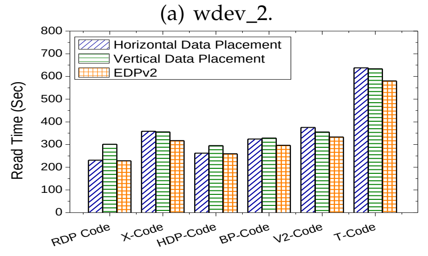
(a) wdev_2.



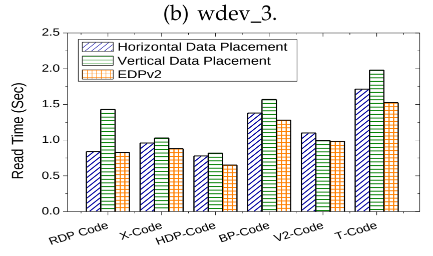
(b) wdev_3.



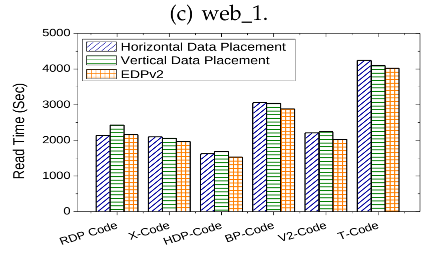
(c) web_1.



(d) web_3.



(e) rsrch_1.



(f) rsrch_2.

Fig. 18: The read time for different real traces. The smaller value is better.

TABLE 3: Characteristics of selected workloads.

Workloads	**wdev_2**	**wdev_3**	**rsrch_1**	**rsrch_2**	**web_1**	**web_3**
Types	Test web server	Test web server	Research project	Research project	Web/SQL server	Web/SQL server
Num. of read operations	189	11	43	13,6364	87,058	10,050
Average read size (KB)	6.12	63.27	13.9	4	45.9	74.9

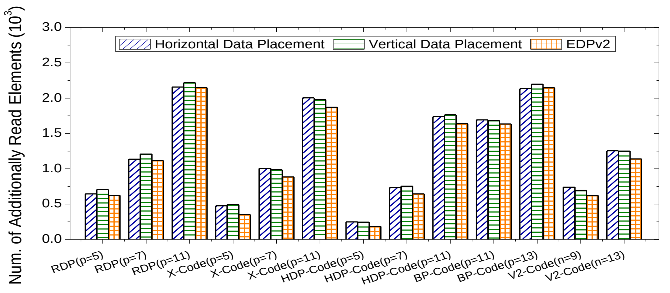
reads when the stripe size (i.e., the number of disks in a stripe) increases. For RDP Code, X-Code, HDP Code, and BP-Code, we vary the selection of p. For V^2-Code, we choose different parameters of n.

Figure 19 shows that given an erasure code and a data placement methodology, the number of elements to be additionally read generally increases with the stripe size. The reason is that when the stripe size increases, the parity chains of an erasure code will accordingly extend to preserve the given fault tolerance. Consequently, more elements should be accessed to repair a lost element in a longer parity chain.

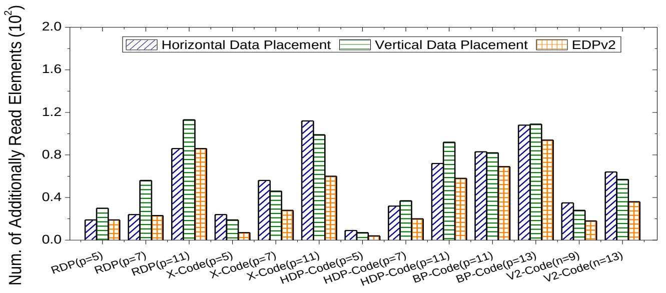
When the stripe size increases, EDPv2 will still preserve its effectiveness on reducing the number of extra elements that are accessed in degraded reads. For example, when being deployed over X-Code with $p = 5$, EDPv2 cuts down 38.6% (resp. 27.8%) of elements that are additionally read for recovery in degraded reads compared to the horizontal (resp. vertical) data placement in the trace `web_3` (see Figure 19(d)). When the scale of X-Code increases to $p = 11$, this reduction reaches 56.1% (resp. 50.6%).

6 Conclusion

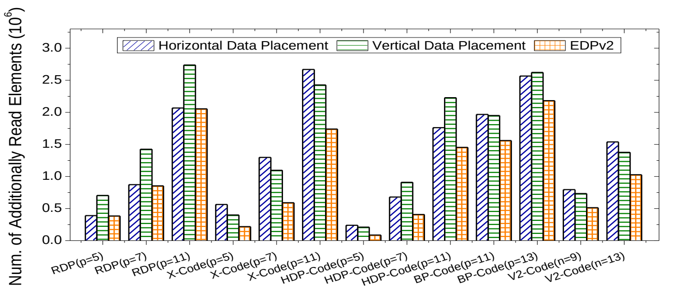
Erasure codes have been intensively used in current storage systems due to their high storage efficiency. In view of the commonplace of single failure and read operations in real-world applications, this paper proposes EDP, an *encoding-aware data placement* scheme to optimize single-failure degraded reads. EDP suggests generating parity elements by



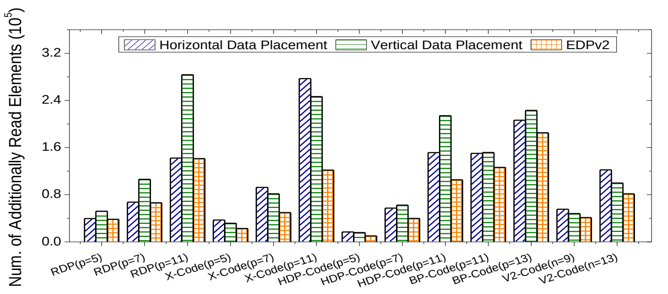
(a) wdev_2.



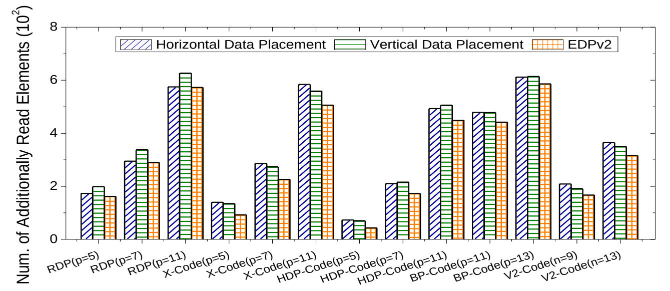
(b) wdev_3.



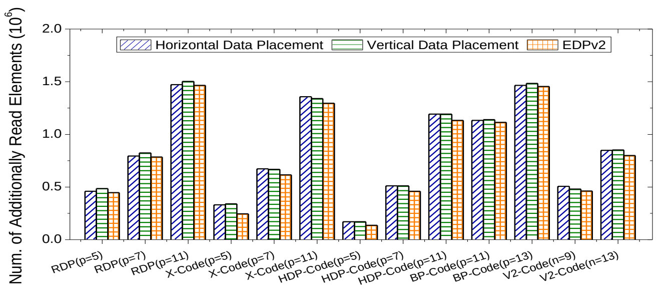
(c) web_1.



(d) web_3.



(e) rsrch_1.



(f) rsrch_2.

Fig. 19: The number of elements to be additionally read under different stripe sizes.

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 EDP can effectively decrease the number of data to be additionally retrieved in degraded reads, and shorten the read time.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61602120, 61327902, 61433008, U1435216, 61672159, U1705262), the Technology Innovation Platform Project of Fujian Province (Grant No. 2014H2005), the Fujian Collaborative Innovation Center for Big Data Application in Governments, the Fujian Engineering Research Center of Big Data Analysis and Processing, the Fujian Provincial Natural Science Foundation (Grant No. 2017J05102), and the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G).

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] S. Lin, G. Wang, D. S. Stones, X. Liu, and J. Liu. T-code: 3-erasure longest lowest-density mds codes. *IEEE Journal on Selected Areas in Communications*, 28(2), 2010.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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] 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.

[25] 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.

[26] 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.

[27] 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*.

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

[29] 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.

[30] 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.

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

[32] P. Xie, J. Huang, Q. Cao, X. Qin, and C. Xie. V 2-code: A new non-mds array code with optimal reconstruction performance for raid-6. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, 2013.

[33] 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.

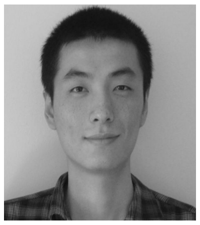
[34] L. Xu and J. Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[35] 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.

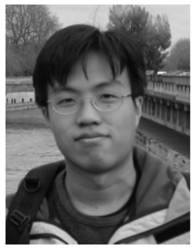
[36] Y. Zhang, C. Wu, J. Li, and M. Guo. Tip-code: A three independent parity code to tolerate triple disk failures with optimal update complextiy. In *Proc. of IEEE/IFIP DSN*, 2015.

[37] 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.

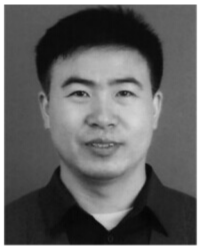
[38] Y. Zhu, J. Lin, P. Lee, and Y. Xu. Boosting degraded reads in heterogeneous erasure-coded storage systems. *IEEE Transactions on Computers*, 64(8):2145–2157, 2015.



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



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in cloud storage, distributed systems and networks, and security/resilience. He is a senior member of the IEEE.



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

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