SN Computer Science (2020) 1:126
https://doi.org/10.1007/s42979-020-00141-y

ORIGINAL RESEARCH

A New Decoding Algorithm for XOR-Based Erasure Codes

Rui Chen[1] · **Lihao Xu**[1]

Received: 8 January 2020 / Accepted: 31 March 2020 / Published online: 20 April 2020

Abstract
Data protection is essential in large-scale storage systems. Over the years, erasure codes, which provide the system ability to reconstruct data when damage occurs, have been proven effective and integrated within various large storage systems. With the emergence of new data storage technologies, such as SSD and NVMe [33, 34], the performance of erasure codes may soon become a potential bottleneck in the whole system. While encoding performance of XOR-based codes has been studied and optimized [7, 19, 20], there is a need of decoding performance to match. This paper addresses new methods in improving the decoding speed for XOR-based erasure codes. A new decoding algorithm is proposed, with which CPU cache can be utilized more efficiently. Various sets of experiments are conducted on different platforms, and the results show that, with the new decoding algorithm, general decoding speed gains considerable improvements.

Keywords Erasure codes · Performance evaluation · Data storage systems

Introduction

For the past decade, data have been increasing exponentially, especially in larger scale data storage systems. With the ever-increasing amount of data, it becomes more and more important to provide reliability when data storage systems or devices fail. Erasure codes, which provide a more economical way than mere data replication [29], have emerged and been widely distributed in various data storage systems, such as Amazon S3 [1, 23], Google file system [10], Microsoft Azure [11, 15] and Facebook Analytics Hadoop Cluster [30].

An erasure code derives redundant parity data through an encoding computation process, and recovers original data through a decoding computation process when data failures occur. Typically, an erasure code system is composed of a total of n storage units. If the n storage units consist of k data units and m parity units, the erasure code system is called a (k, m) erasure code system ($n = k + m$). Theoretically, a (k, m) erasure code is called maximum distance separable code (MDS Code), if it can tolerate up to m storage units failures [21]. Erasure codes can be categorized into MDS Codes [21], such as Reed-Solomon (RS) Code [28], Blaum-Roth (BR) Code [5], EVENODD Code [3], STAR Code [12]; and non-MDS Codes [16], such as Local Repairable Code (LRC) [25], Low-Density Parity-Check (LDPC) Code [18, 22, 24], LT Code [17], Raptor Code [31]. In this paper, we focus only on MDS Codes, since they are the most efficient in storage space usage for desired data reliability, and moreover, are used as the component codes for non-MDS codes, in practical systems [16, 19, 21].

There are mainly two categories of MDS codes: Reed-Solomon Code [28] and XOR-based Code [20] (e.g., BR Code [5] EVENODD Code [3], STAR Code [12], generalized Row-Diagonal Parity (RDP) Code [2, 8, 9]). Though Reed-Solomon Code provides more flexibility by supporting recovery of an arbitrary number of erasures, XOR-based code, which employs XOR-operations for both encoding and decoding instead of more complex finite field operations, outperforms Reed-Solomon Code significantly in both encoding and decoding speeds [27]. Most existing libraries of RS Code, such as Jerasure [26] and Intel's ISA-L [14], are implemented intuitively and directly from the code specification. We call this straightforward coding the conventional algorithm (or traditional algorithm) [19]. On the other hand, due to XOR-operation's commutativity, the order of XOR-operations is changeable. Based on this property, new XOR-scheduling algorithms have been proposed, which

✉ Rui Chen
chenrui@wayne.edu

Lihao Xu
lihao@wayne.edu

[1] Wayne State University, Detroit, MI 48202, USA

show considerable improvements in encoding performance of XOR-based codes [19, 20].

In an erasure code system, decoding operation is needed when failure occurs. Therefore, decoding speed impacts greatly on general performance of the entire system when fault happens. Besides, poor decoding performance brings great vulnerability to the system, since failures can accumulate with low recovery speed. Normally, in most traditional storage systems, decoding computation is fast enough not to become the bottleneck in the whole system. However, with the emergence of new generation of storage technologies, such as flash-based solid-state drive (SSD), non-volatile memory (NVM), and 3D XPoint technology, the throughput of I/O can easily reach the level of tens of GB/sec [33, 34]. Therefore, higher matching performance of decoding operations becomes urgent to reduce vulnerability window when faults occur and thus, to improve the overall system.

It was shown in [19], that by scheduling XORs in proper order, the encoding of XOR-based erasure codes can be improved significantly, which cannot be achieved by code optimization using compilers. While encoding operations for an XOR-based erasure code have a fixed encoding path [19] and thus can be pre-scheduled, decoding operations of such codes have variable decoding paths depending on data erasure (i.e., disk failure) patterns, corresponding XOR scheduling schemes cannot be applied directly. In this paper, a new decoding algorithm is proposed to use on-the-fly caching of decoding path for a given data erasure pattern to enable efficient XOR scheduling for decoding operations with considerable performance improvement, just as XOR encoding operations [19]. While neither efficient XOR scheduling [19] or caching is new, combining them for decoding performance improvement is new, which is the main contribution of this work.

Just as in [19], STAR code [12], an optimal erasure code that can correct up to three data erasures, is used as an example to show the detailed implementation and performance of the new decoding algorithm. Detailed benchmarks show that such a new decoding algorithm is practically feasible in required cache size, about 2–6KB for a practical storage system configuration, and can provide about 10–50% improvement in decoding throughput. While the details may vary slightly, the principle used in this new decoding algorithm (i.e, given a data erasure, its decoding path only needs to be computed once on-the-fly and then stored in a small-sized cache, thus an efficient XOR scheduling algorithm can be applied for proceeding decoding operations for better throughput) can be readily applied to other XOR-based erasure codes besides STAR code, such as the generalized EVENODD code [3], the RDP code [8] and its generalization [2, 9]. Thus another contribution of this work is to provide data storage practitioners a new practical principle of improving decoding performance of XOR-based erasure codes in their systems.

The paper is organized as follows: Sect. 2 gives some background introduction of erasure code basics; the platforms we use for evaluations are described in Sect. 3; Sect. 4 introduces the basics in decoding operation; Sect. 5 proposes a new decoding algorithm; extensive performance evaluation data of the new decoding algorithm is presented in Sect. 6; finally Sect. 7 concludes the paper and discusses the future work.

Background

Typically, a (k, m) erasure code system is composed of an array of n storage units of equal size, which can be further divided into k data units and m parity units ($n = k + m$). As shown in Fig. 1, D_0 to D_{k-1} represent k data units which contain original data, while C_0 to C_{m-1} represent m parity units which contain redundant data generated from encoding. More detailed explanation of all parameters can be found in [27]. When a storage unit fails, the lost data can be recovered by performing the decoding operations with the remaining data units and parity units.

For encoding/decoding operations, each storage unit is partitioned into several **strips** (or **blocks**), each one of which has the same stripsize (or blocksize) and consists of r **packets**. Packets within data strip D_i and parity strip C_j are labeled as $D_{i,0}, \ldots D_{i,r-1}$ and $C_{j,0}, \ldots C_{j,r-1}$, respectively. Different erasure codes typically have different constraints on r, e.g., for RDP [8], EVENODD [3] and STAR [12] Codes, $r + 1$ must be a prime number, while, for X-Code [35], r must be a prime number.

A packet is further divided into s contiguous **machine words**. A machine word is the smallest unit in this structure (e.g., one byte). In most codes, another parameter, w, is included to indicate the wordsize of one machine word, such that a collection of w bits is considered to be one machine

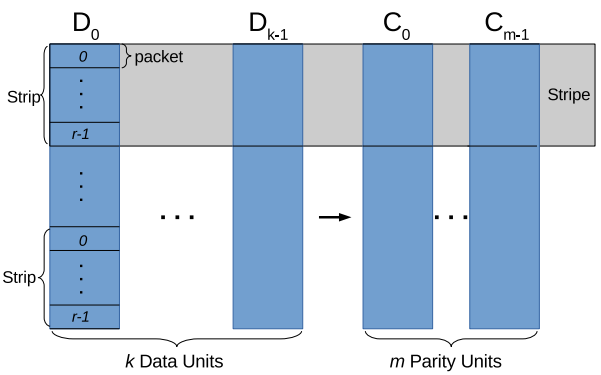


Fig. 1 A typical (k, m) erasure code system

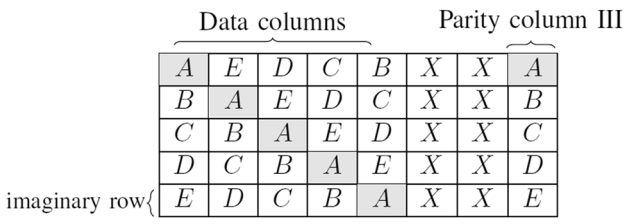
word [26]. Usually $w \in \{8, 16, 32\}$. Throughout this paper, unless specifically mentioned, we assume $w = 8$, in which case, one machine word is equal to one byte.

When encoding/decoding is performed, one strip from each storage unit is collected, together forming a **stripe** (or **codeword**). Every stripe is encoded/decoded independently. The locations of the erasures are called **erasure patterns**, for example, if data unit D_0 is damaged, erasure pattern of each stripe is $\{D_0\}$. Obviously, we have:

$$\begin{cases} \text{codewordsize} & = (k+m) \times \text{blocksize} \\ \text{blocksize} & = r \times \text{packetsize} \\ \text{packetsize} & = s \times \text{wordsize} \\ \text{wordsize} & = \frac{w}{8}\text{bytes} \end{cases} \quad (1)$$

Among the various XOR-based erasure codes, STAR code [12] stands out due to its high performance and efficiency in both encoding and decoding [6, 7]. It is a special case of the extended EVENODD with three parities [3, 4]. STAR code can tolerate up to three erasures [12], i.e., $m = 3$, while k can be any integer as system requires. The best performance is achieved when k is a prime number p [12].

Figure 2 shows the structure of one stripe of a (5,3) STAR code and an example of how parity strip III is generated. More comprehensive description and analysis of STAR code can be found in [12]. Simply speaking, the first packet in parity strip III is generated by applying XOR-operation between packets from each strip in a diagonal way. Note that the bottom row is an imaginary row, which consists of nothing but zeros. While there is no open source library of STAR code at this point, its encoding and decoding algorithms are in public domain. Through this paper, we use a (k, 3) STAR Code as an instance to illustrate our new decoding algorithm, even though this new decoding algorithm applies to any XOR-based erasure codes. All examples focus on decoding one stripe, but we will discuss how to apply this new decoding algorithm to multiple stripes in a system with strip rotation in Sect. 4.4.



A	E	D	C	B	X	X	A
B	A	E	D	C	X	X	B
C	B	A	E	D	X	X	C
D	C	B	A	E	X	X	D
E	D	C	B	A	X	X	E

Fig. 2 How parity column III is generated in a (5,3) STAR code with $r = 4$

Experiment Setup

Throughout this paper, we use two platforms, namely Lenovo ThinkCentre M900 and Intel NW200 Roke (M900 and NW200 in brief), to conduct experiments. Basic configurations of the two platforms are presented in Table 1. Both of the two platforms are equipped with 64-bit processors and 64-bit operating systems.

System Configuration

Lenovo ThinkCentre M900 has a CPU of Intel's i5-6500 (4 cores), which has 256 KB of L-1 cache, 1024 KB of L-2 cache and 6 MB of L-3 cache. It has a total memory of 4 GB. Ubuntu 16.04.3 LTS is installed on the machine, with *gcc* version of 5.4.0.

Intel NW200 Roke is equipped with a CPU of Intel's Xeon E3-1275 (8 cores), which has 512 KB of L-1 cache, 2048 KB of L-2 cache and 8 MB of L-3 cache. It has a total memory of 32 GB, with OS of Ubuntu 17.10. The *gcc* version is 7.2.0.

We stick with *gcc* as our compiler for our implementation of STAR code library. The *gcc* version is listed in Table 1. We include $-O3$ option throughout our experiments. To fully use all levels of caches and provide smoother measurements, enough iterations are executed with the results being averaged.

STAR Code is implemented in C language, in a single-thread mode. Hence, the code does not take advantage of the multiple cores on both platforms. All our experiments follow similar methodology in [27] and [19]. The performance is evaluated in terms of speed $= \frac{\text{data size}}{\text{time}}$. Programming-wise, we use *gettimeofday()* function to measure time interval with accuracy of some µs. Intel's SSE [13] is integrated in

Table 1 Test platform configurations

Platform	CPU model	L1 cache (KB)	L2 cache (KB)	L3 cache (MB)	Memory (GB)
M900	Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz	4 × 64	4 × 256	6	4
NW200	Intel(R) Xeon(R) E3-1275 v5 @ 3.60GHz	8 × 64	8 × 256	8	32

our library to accelerate *XOR* performance. No disk I/O is involved in any measurements.

Baseline Evaluation

Since speed is a relative term, we benchmark the basic hardware performance of the two platforms first. We use the speed of simple *memcpy* and *XOR* to represent this baseline. Within each stripe, simple *memcpy* or *XOR* is performed between the strips, instead of performing actual encoding or decoding. All baseline tests are conducted with 1000 stripes and k from 6 to 17. The results are shown in Fig. 3, with blocksize set to 1 KB and 2 KB, respectively, on both platforms. The x-axis represents k, and y-axis represents the speed, in unit of GB/s.

Roughly speaking, the speed of *memcpy* on M900 can reach around 25–35 GB/s and 15–20 GB/s on NW200. On the other hand, speed of *XOR* is a little slower than *memcpy* on both platforms, which is not surprising.

Basics of Decoding

For simpler illustration and evaluation, we use a toy example shown in Fig. 4 throughout this section, which presents one stripe of a (5,3) STAR Code, as a typical representative of XOR-based codes, as mentioned in Sect. 1. Based on the structure of STAR Code [12], $k = 5$, $m = 3$, $r = 4$ and the prime number $p = 5$ ($p = r + 1$) here. The last row is an imaginary row (marked as yellow), within which all packets are nothing but zeros. $D_0, D_1, \ldots, D_4$ are the data strips, while C_0, C_1, C_2 are the parity strips generated from encoding process. Each packet within the strips is labeled as $D_{0,0}$, $D_{0,1}$, etc.

Path and Computational Complexity

Normally, for a XOR-based erasure code, each packet is encoded or decoded by applying some XOR-operations among a set of packets. This set of packets are imposed only by the structure of the code itself, and thus, obviously varies from code to code. We call this set of packets a Path.

D_0	D_1	D_2	D_3	D_4	C_0	C_1	C_2
$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$	$D_{4,0}$	$C_{0,0}$	$C_{1,0}$	$C_{2,0}$
$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$	$D_{4,1}$	$C_{0,1}$	$C_{1,1}$	$C_{2,1}$
$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$	$D_{4,2}$	$C_{0,2}$	$C_{1,2}$	$C_{2,2}$
$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$	$D_{4,3}$	$C_{0,3}$	$C_{1,3}$	$C_{2,3}$
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Fig. 4 A toy example of a (5,3) STAR Code

Definition 1 In a XOR-based code, if the set of packets $P = \{p_0, p_1, \ldots, p_n\}$ is used to construct one packet p_x, then this set P is called a Path of packet p_x, so that $p_x = p_0 \oplus p_1 \oplus \cdots \oplus p_n$.

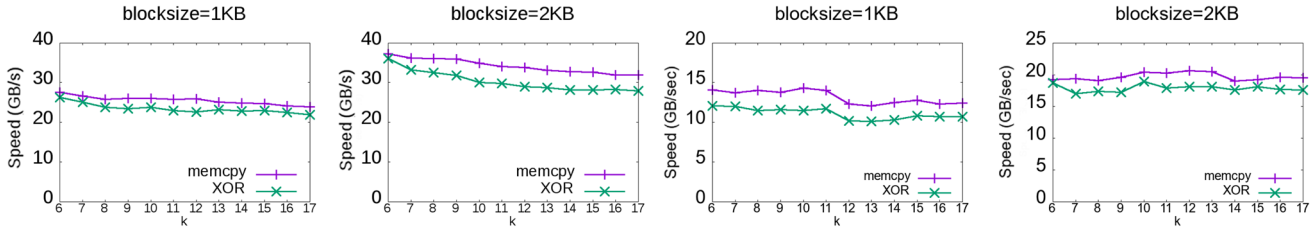
Definition 2 If this path is used for encoding, it is called an encoding path. Likewise, if this path is used for decoding, it is called a decoding path.

For example, STAR Code is designed with encoding rules of the following three equations below [12]:

$$C_{0,i} = \bigoplus_{j=0}^{p-1} D_{j,i} \tag{2}$$

$$C_{1,i} = S_1 \oplus \left(\bigoplus_{j=0}^{p-1} D_{j,<i-j>_p} \right), \quad \text{where } S_1 = \bigoplus_{j=0}^{p-1} D_{j,<p-1-j>_p} \tag{3}$$

$$C_{2,i} = S_2 \oplus \left(\bigoplus_{j=0}^{p-1} D_{j,<i+j>_p} \right), \quad \text{where } S_2 = \bigoplus_{j=0}^{p-1} D_{j,<j-1>_p} \tag{4}$$



(a) M900 (Core i5-6500). **(b)** NW200 (Xeon E3-1275).

Fig. 3 Baseline performance of testing platforms

S_1 and S_2 here are known as EVENODD adjuster and STAR adjuster, respectively. $\oplus$ is the XOR-operation. $< x >_p$ indicates x *mod* p. More details can be found in [3] and [12]. In our toy example, with $k = 5$ and $p = 5$, the encoding paths of packets $C_{0,0}, C_{1,0}, C_{2,0}$ are, respectively, given in Table 2:

XOR-operation is the only operation in XOR-based codes. Since XOR-operation provides commutativity, the order does not matter when applying a series of XOR-operations on a set of packets. Therefore, the computational complexity of applying this series of XOR-operations depends merely on the total number of XOR-operations needed, rather than the order of the XOR-operations. Note that the total number of XOR-operations needed is equal to the total number of elements to be XOR-ed minus one. Based on this, we give definition of computational complexity in XOR-based codes:

Definition 3 In a XOR-based code, the computational complexity of one packet p_x, with encoding/decoding path $P = \{p_0, p_1, \dots p_n\}$, is the total number of XOR-operations needed to construct p_x, which is equal to n.

Definition 4 In a XOR-based code, the computational complexity of one *strip* is the total number XOR-operations needed to construct all packets of the strip. Similarly, the computational complexity of one *stripe* is the total number of XOR-operations needed to construct all strips of the stripe.

Definition 5 In a XOR-based code, the computational complexity of encoding/decoding process is the total number of XOR-operations needed to encode/decode all stripes.

In our toy example, when encoding, $C_{0,0} = D_{0,0} \oplus D_{1,0} \oplus D_{2,0} \oplus D_{3,0} \oplus D_{4,0}$, hence the computational complexity of $C_{0,0}$ is 4. Similarly, the computational complexity of $C_{1,0}$ and $C_{2,0}$ are, respectively, 7 and 7, as shown in Table 2.

The encoding computational complexity is usually influenced only by the structure of the code. Generally, for STAR Code, the encoding computational complexity is imposed only by the total number of data strips, k. With k imposed, computational complexity of parity strip I is given by $(k-1)^2$, and $k^2 - k - 1$ for each parity strip II and III, as is shown in Fig. 5. Note in both of these two cases, S_1 and S_2 need to be calculated only once. Thus, computational complexity of encoding is equal to $(k-1)^2 + (k^2 - k - 1) + (k^2 - k - 1) = 3k^2 - 4k - 1$.

Table 2 Encoding paths and computational complexity in toy example of a (5,3) STAR Code

Packet	Encoding	Computational complexity
$C_{0,0}$	$D_{0,0}, D_{1,0}, D_{2,0}, D_{3,0}, D_{4,0}$	4
$C_{1,0}$	$D_{0,0}, D_{2,3}, D_{3,2}, D_{4,1}$ $D_{1,3}, D_{2,2}, D_{3,1}, D_{4,0}$ $(S_1 = D_{1,3} \oplus D_{2,2} \oplus D_{3,1} \oplus D_{4,0})$	7
$C_{2,0}$	$D_{0,0}, D_{1,1}, D_{2,2}, D_{3,3}$ $D_{1,0}, D_{2,1}, D_{3,2}, D_{4,3}$ $(S_2 = D_{1,0} \oplus D_{2,1} \oplus D_{3,2} \oplus D_{4,3})$	7

Overhead of Computing Decoding Paths

When decoding, the decoding paths of damaged packets are calculated in much more complicated fashion. The decoding computational complexity is impacted not only by the basic parameters of the code (such as k in STAR Code), but also greatly by the number of erasures, the erasure pattern and decoding algorithms accordingly. Decoding computational complexity grows as the number of erasures increases. Erasure pattern also affects the decoding computational complexity. A best case scenario is when erasure only happens on the parity strips, in which case, the decoding process becomes to re-encode the parity strips, partially or entirely.

Obviously, higher decoding computational complexity indicates more time it takes to compute decoding paths. We define the overhead of computing decoding paths, as the ratio of computing decoding paths time over total decoding time.

Definition 6 In one stripe of a XOR-based code, the **overhead** of computing decoding paths, labeled as o_d, is the percentage of the time computing decoding paths in total decoding time.

Explicitly, we have:

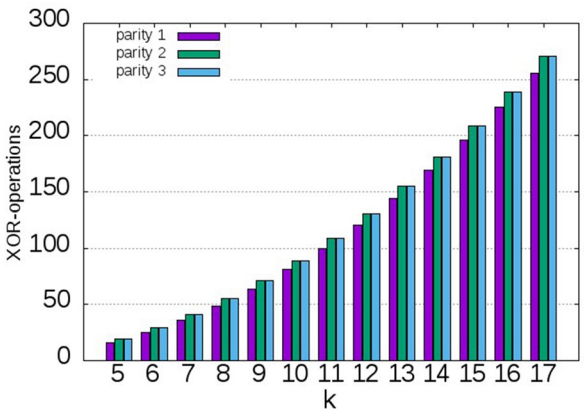


Fig. 5 Computational complexity of encoding parity strips in STAR Code

$$o_d = \frac{t_c}{t_d} \times 100\,\% \tag{5}$$

where t_c, t_d are the time of computing decoding paths and the time of entire decoding operation, respectively, in one stripe, e.g., in our toy example, if it takes 5 time units to compute decoding paths, while it takes 10 time units to decode an entire stripe, $o_d = 50\%$.

Since STAR Code can tolerate up to three erasures, we analyze and measure the overhead of computing decoding paths based on the number of erasures.

Case 1: One Erasure

In a (k, m) code, when there is only one erasure, there are in total $C_1^{k+m} = k + m$ possible erasure patterns (C_k^n is the combinatorial number here). For a $(k, 3)$ STAR Code, the best case scenario is when the erasure occurs on parity strip I or one of the data strips. The decoding computational complexity of best case scenario is equal to $(k-1)^2$.

On the other hand, a worst case is when erasure appears on parity strip II or parity strip III, either of which makes decoding become re-encoding, and decoding computational complexity is $k^2 - k - 1$. A worst case in our toy example is shown in Fig. 6, in which the erasure pattern is $\{C_1\}$. Table 3 presents decoding paths of packets in C_1. Note that EVENODD Adjuster S_1 only needs to be calculated once.

Figure 7 shows the general decoding computational complexity of case 1. It is not difficult to realize that with only one erasure, decoding is exactly the same as encoding. Obviously from Fig. 7, the computational complexities of the best and worst cases are very close and are both not too high (e.g., less than 100, for $k \leq 10$), since in one-erasure case, there are very few packets to reconstruct after all.

Table 4a and b show that, when blocksize is set to 1 KB and 2 KB, respectively, the overhead of computing decoding paths in one stripe of a $(k, 3)$ STAR Code, with k ranging from 5 to 17, on two platforms. Time of computing decoding path and total decoding time are measured and averaged over all possible erasure patterns, all one-erasures in this case (i.e., case 1), and two-erasures and three-erasures for the following experiments in case 2 and case 3. The results show that on both platforms, the time of calculating decoding paths can account for a significant portion of 13–70% of the total decoding time. Especially for smaller ks (i.e., $k \leq 8$), it consumes over 30% to even 70% of total decoding time.

Table 3 Worst case decoding path and computational complexity of case 1

Packet	Decoding path	Computational complexity
$C_{1,0}$	$D_{0,0}, D_{2,3}, D_{3,2}, D_{4,1}$ $D_{1,3}, D_{2,2}, D_{3,1}, D_{4,0}$ $(S_1 = D_{1,3} \oplus D_{2,2} \oplus D_{3,1} \oplus D_{4,0})$	7
$C_{1,1}$	$D_{0,1}, D_{1,0}, D_{3,3}, D_{4,2}\, S_1$	4
$C_{1,2}$	$D_{0,2}, D_{1,1}, D_{2,0}, D_{4,3}\, S_1$	4
$C_{1,3}$	$D_{0,3}, D_{1,2}, D_{2,1}, D_{3,0}\, S_1$	4

Case 2: Two Erasures

When there are two erasures in a (k, m) Code, there are $C_2^{k+m} = \frac{(k+m)(k+m-1)}{2!}$ possible erasure patterns in total. In our toy example, if the damaged strips are parity strip I and parity strip II or III, which is the best case scenario, it makes decoding become re-encoding the two parity strips. The decoding computational complexity is then $(k-1)^2 + (k^2 - k - 1) = 2k^2 - 3k$.

On the other hand, worst case happens when erasures occur on any two of the data strips. A worst case in our toy example is shown in Fig. 8, where the erasure pattern is $\{D_0, D_2\}$ (marked as red). Decoding paths of packets in D_0, D_2 are given in Table 5, according to decoding algorithms in [3, 12]. It can be seen from Table 5, that decoding path of one packet depends on those of other packets.

D_0	D_1	D_2	D_3	D_4	C_0	C_1	C_2
$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$	$D_{4,0}$	$C_{0,0}$	$C_{1,0}$	$C_{2,0}$
$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$	$D_{4,1}$	$C_{0,1}$	$C_{1,1}$	$C_{2,1}$
$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$	$D_{4,2}$	$C_{0,2}$	$C_{1,2}$	$C_{2,2}$
$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$	$D_{4,3}$	$C_{0,3}$	$C_{1,3}$	$C_{2,3}$
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Fig. 6 Worst case scenario of case 1 in toy example

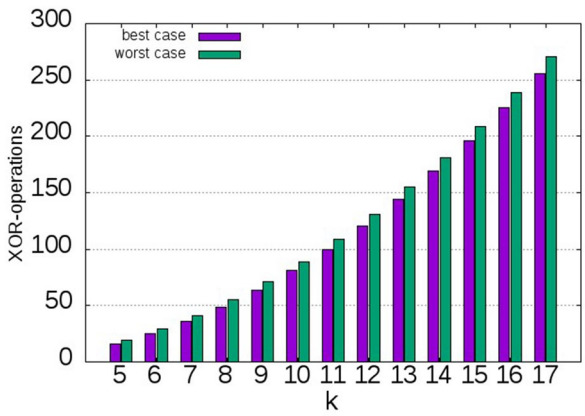


Fig. 7 Decoding computational complexity of case 1

Table 4 Overhead of computing decoding paths in case 1 (t_c and t_d are in unit of µs)

k	Blocksize = 1 KB			Blocksize = 2 KB		
	t_c	t_d	o_d (%)	t_c	t_d	o_d (%)
(a) M900 (Core i5-6500)						
5	7	13	54	14	20	70
6	29	43	67	22	92	24
7	30	72	42	29	122	24
8	39	112	35	76	199	39
9	41	151	27	76	276	28
10	47	190	25	77	353	22
11	54	259	21	77	431	18
12	65	314	21	107	540	20
13	68	369	18	108	648	17
14	93	465	20	187	836	22
15	94	560	17	188	1025	18
16	95	654	15	194	1269	15
17	95	748	13	196	1456	13
(b) NM200 (Xeon E3-1275)						
5	15	23	65	25	39	64
6	30	56	54	32	72	44
7	32	86	37	41	103	40
8	38	125	30	79	183	43
9	42	178	24	81	265	31
10	46	218	21	81	347	23
11	48	257	19	81	429	19
12	54	312	17	113	544	21
13	56	369	15	114	658	17
14	94	464	20	190	849	22
15	117	652	18	191	1041	18
16	124	783	16	192	1234	16
17	130	908	14	192	1426	13

The reconstruction of all packets cannot be performed concurrently.

Figure 9 presents general decoding computational complexity of two-erasure case. Note that unlike the best case, the decoding computational complexity of the worst case relies on the prime number p, which is the smallest prime number that is no less than k (e.g., $p = 7$ for $k = 6$). Cases with different k yet the same p share the same worst case computational complexity (e.g., cases of $k = 6$ and $k = 7$ have the same decoding computational complexity). This is imposed by the basic structure of STAR Code: for decoding, a prime number p is required to construct the stripe array [3, 12]. Also, decoding is more complicated than encoding, for example, when $k = 10$, the worst case decoding computational complexity is roughly 250, comparing to that of encoding being 170 (this encoding computational complexity is based on generating two parity strips, I and II).

Based on Definition 3, 4, 5, the encoding/decoding computational complexity indicates the size of all encoding/decoding paths. Apparently, the reason why decoding computational complexity is greater than encoding is because decoding paths are longer, which means the decoding paths involve more elements, than encoding paths.

D_0	D_1	D_2	D_3	D_4	C_0	C_1	C_2
$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$	$D_{4,0}$	$C_{0,0}$	$C_{1,0}$	$C_{2,0}$
$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$	$D_{4,1}$	$C_{0,1}$	$C_{1,1}$	$C_{2,1}$
$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$	$D_{4,2}$	$C_{0,2}$	$C_{1,2}$	$C_{2,2}$
$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$	$D_{4,3}$	$C_{0,3}$	$C_{1,3}$	$C_{2,3}$
0	0	0	0	0	0	0	0

Fig. 8 Worst case scenario of case 2 in toy example

Table 6a and b, respectively, present, when blocksize is set to 1 KB and 2 KB, the overhead of computing decoding paths in one stripe of a (k, 3) STAR Code, with k ranging from 5 to 17, on two platforms. The results clearly show that on both platforms, the time of calculating decoding paths

Table 5 Worst case decoding paths of case 2

Packet	Decoding path	Computational complexity
$D_{2,2}$	$D_{1,3}, D_{3,1}, D_{4,0}, S_1$ $(S_1 = C_{0,0} \oplus C_{0,1} \oplus C_{0,2} \oplus C_{0,3} \oplus C_{1,0} \oplus C_{1,1} \oplus C_{1,2} \oplus C_{1,3})$	10
$D_{0,2}$	$D_{2,2}, D_{1,2}, D_{3,2}, D_{4,2}, C_{0,2}$	4
$D_{2,0}$	$D_{0,2}, D_{1,1}, D_{4,3}, C_{1,2}, S_1$	4
$D_{0,0}$	$D_{2,0}, D_{1,0}, D_{3,0}, D_{4,0}, C_{0,0}$	4
$D_{2,3}$	$D_{0,0}, D_{3,2}, D_{4,1}, C_{1,0}, S_1$	4
$D_{0,3}$	$D_{2,3}, D_{1,3}, D_{3,3}, D_{4,3}, C_{0,3}$	4
$D_{2,1}$	$D_{0,3}, D_{1,2}, D_{3,0}, C_{1,3}, S_1$	4
$D_{0,1}$	$D_{2,1}, D_{1,1}, D_{3,1}, D_{4,1}, C_{0,1}$	4

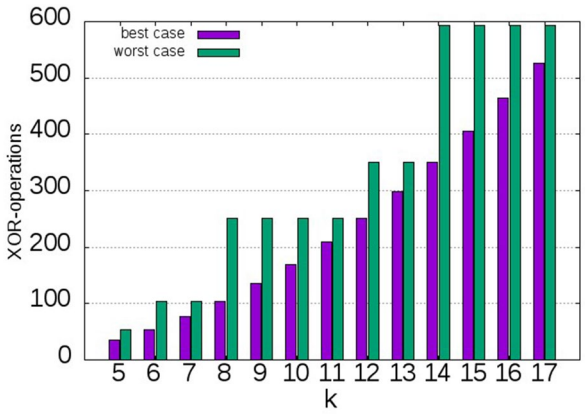


Fig. 9 Decoding computational complexity of case 2

can account for a significant portion of 9–94% of the total decoding time. Especially for smaller ks (i.e., $k \leq 8$), it consumes over 23% to even 94% of total decoding time.

Case 3: Three Erasures

Decoding process becomes much more complicated when there are three erasures in a (k, m) Code. In total, there are $C_3^{k+m} = \frac{(k+m)(k+m-1)(k+m-2)}{3!}$ possible erasure patterns. The best case scenario is when all three erasures occur on the three parity strips. In this situation, decoding simply becomes re-encoding all three parity strips I, II and III. The decoding computational complexity of best case is $3k^2 - 4k - 1$, which is equal to the encoding computational complexity.

In contrast, the worst case scenario happens when all the three erasures are on the data strips. As is shown in Fig. 10, in our toy example, data strips D_0, D_2, D_3 (marked as red) are damaged. This is also called an asymmetric case according to [12]. In an asymmetric case, the decoding paths become random and unpredictable. They depend not only on k and p, but also greatly on the erasure patterns (i.e., $\{D_0, D_2, D_3\}$ here). Thus, the total XOR-operations needed to recover data packets rises tremendously.

Figure 11 presents decoding computational complexity of three-erasure case, measured from exhaustive enumeration of all patterns of three-erasure case. Like case 2, decoding computational complexity of worst case depends on prime number p rather than k. In addition, Fig. 11 clearly indicates that decoding in three-erasures case can be extremely complicated. For example, when $k = 10$, the worst case computational complexity reach around 800, comparing to encoding complexity of 259 (based on generating all three parity strips, I, II and III). This represents long decoding paths when there are three erasures.

Again, Table 7a and b present the overhead of calculating decoding paths in one stripe of a (k, 3) STAR Code, with blocksize set to 1 KB and 2 KB, respectively, on two platforms. k varies from 5 to 17 too. The results again show that, on both platforms, pure time of computing decoding paths in three-erasure case consumes 11–97% of the total decoding time. Particularly, for smaller ks (i.e., $k \leq 8$), it can hold over 26% to even 97% of total time computing decoding paths.

Cached Decoding Path Method

An erasure code system usually involves hundreds or even thousands of stripes. Therefore, with the observation from Tables 4, 6, and 7 above, it is more than apparent that computing decoding paths results in considerable overhead for overall decoding operation, particularly when k is relatively small (i.e., $k \leq 8$).

However, erasure occurs per single data/parity unit. This means, when erasures occur, all stripes share the same erasure pattern, which further leads to the same decoding paths for data recovery. In our toy example of a (5,3) STAR Code, if one erasure occurs on data unit I, erasure pattern ($\{D_0\}$ here) in every stripe is the same. Thus, decoding path of the same packet in different stripes remains the same (e.g., packet $D_{0,0}$ in every stripe has the same decoding path of $\{D_{1,0}, D_{2,0}, D_{3,0}, D_{4,0}, C_{0,0}\}$).

With this observation, it is redundant and unnecessary to calculate decoding paths for every stripe. We propose the first improvement in our new decoding algorithm: decoding paths of a stripe is calculated only if the erasure pattern of the stripe occurs for the first time. The set of decoding paths then is stored in CPU cache and is reused for stripes with the same erasure pattern. We call this methodology of implementation cached decoding path method. With cached decoding path method, a stripe with duplicated erasure pattern, automatically obtains its decoding paths from cache, and can directly access packets needed without redundant computation overhead.

Table 6 Overhead of computing decoding paths of case 2, on two platforms (t_c, t_d are in unit of µs)

k	Blocksize = 1 KB			Blocksize = 2 KB		
	t_c	t_d	o_d (%)	t_c	t_d	o_d (%)
(a) M900 (Core i5-6500)						
5	26	30	86	49	57	86
6	43	75	57	85	144	59
7	44	120	37	85	232	37
8	103	225	46	199	434	46
9	101	327	31	199	636	31
10	102	430	24	200	839	24
11	101	533	19	200	1042	19
12	140	678	21	272	1315	21
13	138	818	17	272	1589	17
14	236	1058	22	468	2063	23
15	230	1290	18	457	2523	18
16	230	1521	15	459	2985	15
17	231	1754	13	456	3444	13
(b) NM200 (Xeon E3-1275)						
5	92	102	90	178	189	94
6	171	288	59	325	519	63
7	169	462	37	329	861	38
8	380	849	45	258	1123	23
9	400	1255	32	251	1375	18
10	133	1393	10	254	1631	16
11	133	1527	9	253	1888	13
12	171	1700	10	347	2241	15
13	172	1874	9	348	2590	13
14	289	2166	13	582	3177	18
15	288	2456	12	552	3734	15
16	289	2747	11	524	4263	12
17	287	3037	9	550	4820	11

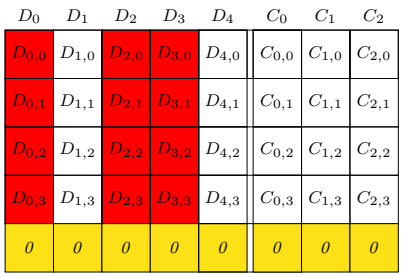


Fig. 10 Worst case scenario of case 3 in toy example

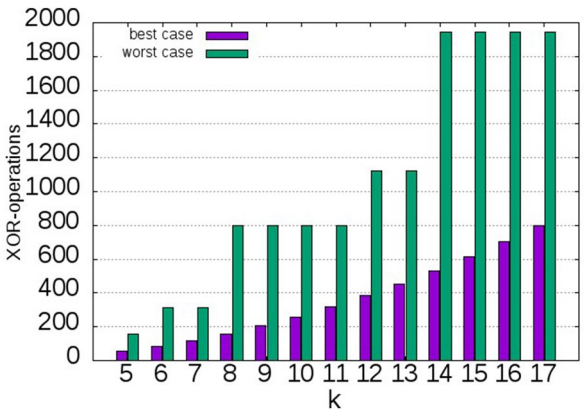


Fig. 11 Decoding computational complexity of case 3

Strip Rotation

In practical data storage systems, strip rotation is typically applied on storage units [32]. Strip rotation in data storage system provides balanced distribution among all storage units by shifting data loads. Although strip rotation causes the stripes to have different decoding paths, this rotation is periodical. Applying a (k, m) code in a storage system with N ($N \geq k+m$) total units (e.g., disks), the rotation period is N, in which case, stripe S_i and stripe S_{i+N} share the same erasure pattern. In other words, there are at most N different sets of decoding paths to be computed and cached. Real data

Table 7 Overhead of computing decoding paths of case 3, on two platforms (t_c, t_d are in unit of µs)

k	Blocksize = 1 KB			Blocksize = 2 KB		
	t_c	t_d	o_d (%)	t_c	t_d	o_d (%)
(a) M900 (Core i5-6500)						
5	78	84	92	149	157	95
6	143	229	62	276	439	63
7	136	367	37	258	702	37
8	337	707	48	658	1373	48
9	327	1040	31	647	2030	32
10	327	1371	24	647	2688	24
11	329	1704	19	646	3345	19
12	471	2178	22	918	4334	21
13	457	2639	17	900	5249	17
14	804	3451	23	1544	6798	23
15	787	4242	19	1582	8385	19
16	787	5032	16	1546	9936	16
17	802	5840	14	1571	11514	14
(b) NM200 (Xeon E3-1275)						
5	316	329	96	554	570	97
6	522	860	61	1062	1654	64
7	501	1368	37	962	2630	37
8	489	1861	26	831	3474	24
9	327	2191	15	814	4300	19
10	331	2525	13	811	5123	16
11	328	2855	11	813	5948	14
12	460	3318	14	1150	7102	16
13	446	3767	12	1101	8208	13
14	758	4528	17	1934	10152	19
15	738	5269	14	1783	11940	15
16	723	5995	11	1726	13676	13
17	723	6721	11	1663	15350	11

storage systems usually consist of hundreds or even thousands of stripes, while N is relatively small (e.g., $N \leq 24$). Therefore, with cached decoding path method, numerous computations of decoding paths can still be avoided and thus, resulting in more efficient decoding.

Figure 12 presents a simple strip rotation example of a (k, m) erasure code in a storage system with N ($N \geq k + m$) storage units in total. Each row represents one stripe, while each column represents one storage unit. The units containing the EC strips are marked as red, and the yellow labels all other idle units. Rotation starts from the second stripe S_1, in which all strips are right-shifted by one position, so that the first data strip D_0 is stored on the second storage unit, the second data strip D_1 is stored on the third storage unit, etc. Similarly, in the third stripe, the third storage unit stores data strip D_0, etc.

Since there are in total N strips in each stripe, there can be at most N shifts before the strips return to the original positions (e.g., stripe S_0 and stripe S_N are the same), and consequently, share the same erasure pattern. In this situation, up to N sets of decoding paths (i.e., decoding paths of $S_0, S_1, \ldots S_N$) need to be computed and cached, which all other stripes can benefit and directly take advantage of.

Fig. 12 Rotation example of a (k, m) Code

A New Decoding Algorithm

Once the decoding paths are calculated, the rest of decoding process becomes the same as encoding process. The remaining task is to simply reconstruct all missing packets by applying XOR-operations among the needed packets according to the decoding paths. In this section, we optimize this decoding procedure.

To simplify illustrations, we use one stripe of a (3, 3) STAR Code, with each packet consisting of two machine words (i.e., 2 bytes), as a toy example, throughout this section. Figure 13 presents this toy example.

XOR-Scheduling Algorithms

Though the order of XOR-operations does not affect correctness when reconstructing one packet with a particular decoding path, it does affect cache use and thus, decoding time, when applied on a serial of different packets. This is because their decoding paths overlap with each other. Often some packets appear multiple times in related decoding paths, and later are accessed multiple times. The order of accessing these duplicated packets makes a great difference in general encoding performance by utilizing caches differently [19, 20]. An algorithm to optimize this particular accessing order is called XOR-scheduling algorithm. In this subsection, we introduce and evaluate different XOR-scheduling algorithms for decoding operation.

There are typically four different XOR-scheduling algorithms for encoding operation [19], namely, Parity Packet Guided (PPG), Parity Words Guided (PWG), Data Packet Guided (DPG) and Data Words Guided (DWG) XOR-scheduling algorithms [19]. Among the four algorithms, PPG XOR-scheduling algorithm is known as the conventional algorithm, which has been mostly used. However, from the testing results in [19] and [20], the DWG XOR-scheduling algorithm provides better encoding performance, and this improvement cannot be achieved by a general compiler.

Based on results in [19, 20], we propose two XOR-scheduling algorithms for decoding operation: Erasure Packet Guided (EPG) and Surviving Words Guided (SWG) XOR-scheduling algorithm. We first give a set of definitions on Erasure/Surviving Strip:

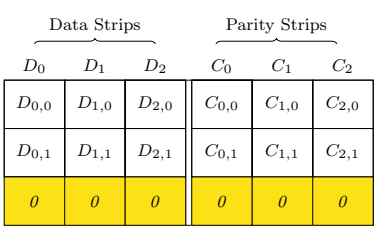


Fig. 13 Toy example of a (3,3) STAR Code

Definition 7 In one stripe of a (k, m) Code consisting of a set of strips $D = \{D_0, \ldots, D_{k+m-1}\}$, with x $(0 \le x \le m)$ erasures occurring on a subset of strips $X = \{D_{y_0}, D_{y_1}, \ldots, D_{y_{x-1}}\} \subsetneq D$ $(0 \le y_0 \le y_1 \le \cdots \le y_{x-1} \le k + m - 1)$, the set of strips with erasures (X) is called erasure strips (or erasure columns), the set of the rest strips $(D - X)$ is called surviving strips (or surviving columns).

Let us observe a two erasure case in our toy example presented in Fig. 14, where erasure occurs on D_0 and D_1. In this case, D_0, D_1 become the erasure strips E_0 and E_1 (marked as red), all other strips from D_2 to C_2 are known as the Surviving Strips, labeled as S_0 to S_3.

In addition, we define Erasure/Surviving Packet:

Definition 8 An individual packet $E_{i,j}$ within an erasure strip E_i is called an Erasure Packet. Obviously, for an erasure strip consisting of r erasure packets, we have $E_i = \{E_{i,0}, E_{i,1}, \ldots, E_{i,r-1}\}$. Similarly, a packet $S_{i,j}$ within a surviving strip S_i is called a Surviving Packet. For a surviving strip consisting of r surviving packets, we have $S_i = \{S_{i,0}, S_{i,1}, \ldots, S_{i,r-1}\}$.

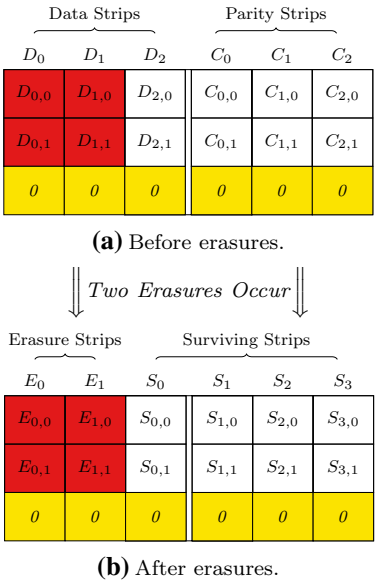
In our toy example, $E_{0,0}, E_{0,1}, E_{1,0}, E_{1,1}$ are erasure packets, while $S_{0,0}, S_{0,1}, \ldots, S_{3,1}$ are called surviving packets. Based on structure of STAR Code [12], we can easily have decoding paths of erasure packets, as shown in Table 8. Apparently, decoding paths of those packets overlap, since packets, such as $S_{0,1}$ and $S_{1,0}$, appear multiple times.

Furthermore, we give definitions of Erasure/Surviving Word:

Definition 9 An individual machine word $e_{[i,j],t}$ within an erasure packet $E_{i,j}$ is called erasure word, so that $E_{i,j} = \{e_{[i,j],0}, e_{[i,j],1}, \ldots, e_{[i,j],num-1}\}$ (num indicates number of machine words in a single packet, $\text{num} = \frac{\text{packetsize}}{\text{wordsize}}$). Similarly, a machine word $s_{[i,j],t}$ within a surviving packet $S_{i,j}$ is called surviving word, so that $S_{i,j} = \{s_{[i,j],0}, s_{[i,j],1}, \ldots, s_{[i,j],\text{num}-1}\}$ $(\text{num} = \frac{\text{packetsize}}{\text{wordsize}})$.

Since each packet consists of two machine words (2 bytes) in our toy example, erasure packet $E_{0,0} = \{e_{[0,0],0}, e_{[0,0],1}\}$, in which $e_{[0,0],0}$ and $e_{[0,0],1}$ are called erasure words. On the other hand, surviving packet $S_{0,0} = \{s_{[0,0],0}, s_{[0,0],1}\}$, in which $s_{[0,0],0}$ and $s_{[0,0],1}$ are called surviving words.

Finally, we introduce Erasure Packet Guided (EPG) and Surviving Words Guided (SWG) XOR-scheduling algorithms described in Algorithm 1 and 2, respectively:



(a) Before erasures.

(b) After erasures.

Fig. 14 Two-erasure case in toy example

Algorithm 1 Erasure Packet Guided (EPG) Scheduling

Input: Surviving strips S, each one is a *word* array
Output: Erasure strips E, each on is a *word* array

```
for each erasure strip E_j do
    for each erasure packet E_{j,i} in strip E_j do
        for each surviving packet S_{u,v} in the decoding path of E_{j,i} do
            for t=0; t<packetsize; t++ do
                e_{[j,i],t} ⊕= s_{[u,v],t}
            end for
        end for
    end for
end for
```

Algorithm 2 Surviving Words Guided (SWG) Scheduling

Input: Surviving strips S, each one is a *word* array
Output: Erasure strips E, each on is a *word* array

```
for each surviving strip S_j do
    for each surviving packet S_{j,i} in strip S_j do
        for t=0; t<packetsize; t++ do
            for each erasure packet E_{u,v} whose decoding path includes S_{j,i} do
                e_{[u,v],t} ⊕= s_{[j,i],t}
            end for
        end for
    end for
end for
```

The EPG XOR-scheduling is intuitive to implement and is commonly used, which we call the conventional or traditional XOR-scheduling. The basic principle is to focus on each erasure packet and try to finish reconstructing one erasure packet before moving to the next one. A detailed procedure of EPG in our toy example is presented in Fig. 15a, from which it is obvious that each erasure packet from $E_{0,0}$ to $E_{1,1}$ are decoded sequentially. First, all packets needed, according to $E_{0,0}$'s decoding path, are collected and used to produce $E_{0,0}$, and then $E_{0,1}$, $E_{1,0}$, etc., until all packets are decoded.

Table 8 Decoding paths of erasure packets in toy example

Packet	Decoding path
$E_{0,0}$	$S_{0,1}, S_{1,0}, S_{1,1}, S_{2,1}$
$E_{0,1}$	$S_{0,0}, S_{0,1}, S_{1,0}, S_{2,0}, S_{2,1}$
$E_{1,0}$	$S_{0,0}, S_{0,1}, S_{1,1}, S_{2,1}$
$E_{1,1}$	$S_{0,0}, S_{1,0}, S_{1,1}, S_{2,0}, S_{2,1}$

On the other hand, the SWG XOR-scheduling is called the new XOR-scheduling. It takes advantage of results from [19], in which it is shown to be more efficient. In contrast to EPG, SWG focuses on each surviving word, and tries to make the most of a single surviving word by XOR-ing it to erasure words of the erasure packets whose decoding path contains the relevant surviving packet. Figure 15b shows a concrete decoding process for our toy example code, in which surviving word $s_{[0,0],0}$ is first XOR-ed with erasure words $e_{[0,0],0}$, $e_{[1,0],0}$, $e_{[1,1],0}$, since $E_{0,0}$, $E_{1,0}$, $E_{1,1}$ all have $S_{0,0}$ in their decoding paths. Surviving word $s_{[0,0],1}$ is used next, following with $s_{[0,1],1}, \ldots, s_{[2,1],1}$ sequentially, until all packets are decoded.

EPG XOR-scheduling accesses each machine word (byte) in each packet in a logically sequential way. It thus provides a good spatial locality by having small memory distance between the data in a series of accesses [19]. On the other hand, SWG XOR-scheduling performs the XOR iteration at machine word level (e.g., per each byte of a packet), and improves temporal locality of $s_{[j,i],t}$. This provides short time period between two consecutive accesses to the same data [19], but decreases some spatial locality of both $e_{[u,v],t}$ and $s_{[j,i],t}$. In practical systems, the number of erasure strips is usually small (≤ 3), comparing to the number of surviving strips (e.g., ≥ 10 with $k = 10$), so the loss of spatial locality of $s_{[j,i],t}$ is observed to be minor than the gain from temporal locality of $s_{[j,i],t}$. Thus, in general, the overall locality is better using the SWG XOR-scheduling algorithm [19].

Caching Decoding Path

Using the cached decoding path method introduced in Sect. 4.3, total amount of XOR-operations is minimized by avoiding redundant calculation of numerous decoding paths. Now we discuss how to cache decoding paths efficiently. Our STAR library is currently written in C, within which, a Decoding Table is used and stores the decoding paths of erasure patterns. Decoding Table is implemented by a 3-D Linklist data structure. Programming-wise, three *struct*s are declared to fulfill the data structure, as presented below:

ID	XOR
1	$e_{[0,0],0} \oplus = s_{[0,1],0}$
2	$e_{[0,0],1} \oplus = s_{[0,1],1}$
3	$e_{[0,0],0} \oplus = s_{[1,0],0}$
4	$e_{[0,0],1} \oplus = s_{[1,0],1}$
5	$e_{[0,0],0} \oplus = s_{[1,1],0}$
6	$e_{[0,0],1} \oplus = s_{[1,1],1}$
7	$e_{[0,0],0} \oplus = s_{[2,1],0}$
8	$e_{[0,0],1} \oplus = s_{[2,1],1}$
9	$e_{[0,1],0} \oplus = s_{[0,0],0}$
10	$e_{[0,1],1} \oplus = s_{[0,0],1}$
11	$e_{[0,1],0} \oplus = s_{[0,1],0}$
12	$e_{[0,1],1} \oplus = s_{[0,1],1}$
13	$e_{[0,1],0} \oplus = s_{[1,0],0}$
14	$e_{[0,1],1} \oplus = s_{[1,0],1}$
15	$e_{[0,1],0} \oplus = s_{[2,0],0}$
16	$e_{[0,1],1} \oplus = s_{[2,0],1}$
17	$e_{[0,1],0} \oplus = s_{[2,1],0}$
18	$e_{[0,1],1} \oplus = s_{[2,1],1}$
19	$e_{[1,0],0} \oplus = s_{[0,0],0}$
20	$e_{[1,0],1} \oplus = s_{[0,0],1}$
21	$e_{[1,0],0} \oplus = s_{[0,1],0}$
22	$e_{[1,0],1} \oplus = s_{[0,1],1}$
23	$e_{[1,0],0} \oplus = s_{[1,1],0}$
24	$e_{[1,0],1} \oplus = s_{[1,1],1}$
25	$e_{[1,0],0} \oplus = s_{[2,1],0}$
26	$e_{[1,0],1} \oplus = s_{[2,1],1}$
27	$e_{[1,1],0} \oplus = s_{[0,0],0}$
28	$e_{[1,1],1} \oplus = s_{[0,0],1}$
29	$e_{[1,1],0} \oplus = s_{[1,0],0}$
30	$e_{[1,1],1} \oplus = s_{[1,0],1}$
31	$e_{[1,1],0} \oplus = s_{[1,1],0}$
32	$e_{[1,1],1} \oplus = s_{[1,1],1}$
33	$e_{[1,1],0} \oplus = s_{[2,0],0}$
34	$e_{[1,1],1} \oplus = s_{[2,0],1}$
35	$e_{[1,1],0} \oplus = s_{[2,1],0}$
36	$e_{[1,1],1} \oplus = s_{[2,1],1}$

(a) EPG

ID	XOR
9	$e_{[0,1],0} \oplus = s_{[0,0],0}$
19	$e_{[1,0],0} \oplus = s_{[0,0],0}$
27	$e_{[1,1],0} \oplus = s_{[0,0],0}$
10	$e_{[0,1],1} \oplus = s_{[0,0],1}$
20	$e_{[1,0],1} \oplus = s_{[0,0],1}$
28	$e_{[1,1],1} \oplus = s_{[0,0],1}$
1	$e_{[0,0],0} \oplus = s_{[0,1],0}$
11	$e_{[0,1],0} \oplus = s_{[0,1],0}$
21	$e_{[1,0],0} \oplus = s_{[0,1],0}$
2	$e_{[0,0],1} \oplus = s_{[0,1],1}$
12	$e_{[0,1],1} \oplus = s_{[0,1],1}$
22	$e_{[1,0],1} \oplus = s_{[0,1],1}$
3	$e_{[0,0],0} \oplus = s_{[1,0],0}$
13	$e_{[0,1],0} \oplus = s_{[1,0],0}$
29	$e_{[1,1],0} \oplus = s_{[1,0],0}$
4	$e_{[0,0],1} \oplus = s_{[1,0],1}$
14	$e_{[0,1],1} \oplus = s_{[1,0],1}$
30	$e_{[1,1],1} \oplus = s_{[1,0],1}$
5	$e_{[0,0],0} \oplus = s_{[1,1],0}$
23	$e_{[1,0],0} \oplus = s_{[1,1],0}$
31	$e_{[1,1],0} \oplus = s_{[1,1],0}$
6	$e_{[0,0],1} \oplus = s_{[1,1],1}$
24	$e_{[1,0],1} \oplus = s_{[1,1],1}$
32	$e_{[1,1],1} \oplus = s_{[1,1],1}$
15	$e_{[0,1],0} \oplus = s_{[2,0],0}$
33	$e_{[1,1],0} \oplus = s_{[2,0],0}$
16	$e_{[0,1],1} \oplus = s_{[2,0],1}$
34	$e_{[1,1],1} \oplus = s_{[2,0],1}$
7	$e_{[0,0],0} \oplus = s_{[2,1],0}$
17	$e_{[0,1],0} \oplus = s_{[2,1],0}$
25	$e_{[1,0],0} \oplus = s_{[2,1],0}$
35	$e_{[1,1],0} \oplus = s_{[2,1],0}$
8	$e_{[0,0],1} \oplus = s_{[2,1],1}$
18	$e_{[0,1],1} \oplus = s_{[2,1],1}$
26	$e_{[1,0],1} \oplus = s_{[2,1],1}$
36	$e_{[1,1],1} \oplus = s_{[2,1],1}$

(b) SWG

Fig. 15 EPG and SWG XOR-scheduling algorithms in our toy example

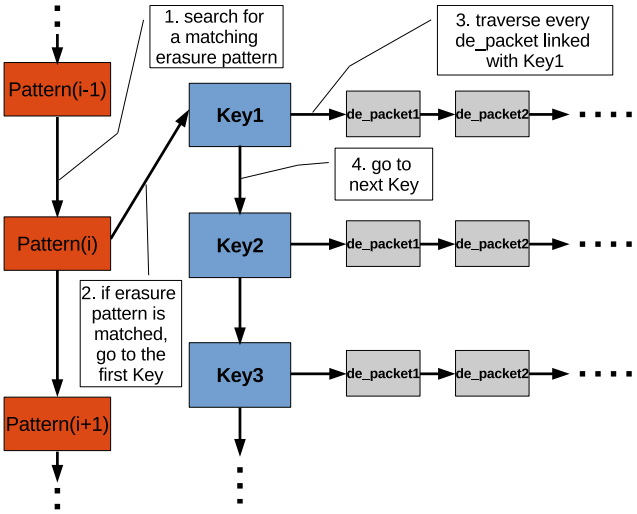


Fig. 16 Decoding table data structure

```
typedef struct pattern{
   int *erasure_pattern;
   struct pattern *next_pattern;
   struct key *first_key;
}Pattern;

typedef struct key{
   int k_idx;
   struct key *next_key;
   struct de_packet *first_p;
}Key;

typedef struct de_packet{
   int p_idx;
   struct de_packet *next_p;
}de_packet;
```

Note that, though an erasure code is structured as a 2-D array, it is implemented by a 1-D array in programming since most application data is stored in a 1-D buffer. Thus, in terms of code, the index of a packet is represented by one single integer, e.g., in our toy example shown in Fig. 14, indexes of $D_{0,0}$,$D_{0,1}$ are 0, 1, respectively.

A Pattern node indicates an erasure pattern for a set of decoding paths. int * erasure_pattern is an integer array, of which each element (i.e., an integer) marks the index of a strip that is in the erasure pattern. A Key node stores the a surviving packet p_x, whose index is represented by int *k_idx*. A de_packet node stores an erasure packet that has p_x in its decoding path, whose index is marked by int *p_idx*. Note that this structure is designed to optimize SWG XOR-scheduling algorithm for decoding. Obviously, the size of strcut pattern, struct key and struct de_packet are 8 B, 12 B and 8 B, respectively, in C.

In practical decoding situations, parameters such as k, m and the erasure pattern are known, when building the Decoding Table, values of each strip in * erasure_pattern can be easily calculated: if it is a data strip, the index is simply its strip index; if it is a parity strip, the index is its strip index plus k, e.g., in our toy example shown in Fig. 14, indexes of strip $D_0, D_1, D_2, C_0, C_1, C_2$ are 0, 1, 2, 3, 4, 5, respectively. Since there is only one erasure pattern in this case, the Decoding Table consists of only one Pattern node. As the erasure pattern is $\{D_0, D_1\}$, Pattern.erasure_pattern=$\{0, 1\}$. Pattern.first_key points to surviving packet $S_{0,0}$, which is linked with erasure packets $E_{0,1}, E_{1,0}, E_{1,1}$. The second Key $S_{0,1}$ is linked with $E_{0,0}, E_{0,1}, E_{1,0}$, etc.

When decoding is performed, one stripe ST_i with erasure pattern EP_i accesses the first Pattern node (Pattern1) in Decoding Table. If two erasure patterns match, i.e., EP_i = Pattern1.erasure_pattern, the first Key node (Key1) is accessed, from which a list of de_packets are traversed. After XOR-operations are applied among those packets, the second Key (Key2) is accessed, etc. In this way, ST_i is reconstructed. This process is shown in Fig. 16. As discussed in Sect. 4.4, in practical systems, strip rotation is often applied. For a storage system with N units (disks), the rotation period is at most N, in which case, N sets of decoding paths need to be cached, resulting in N Pattern nodes in the Decoding Table. Since N is relatively small (e.g., ≤ 24), a simple linear search of all Pattern nodes for a matching erasure pattern is efficient enough. Other multiple erasure patterns can be handled similarly.

An erasure pattern depends on both k and number of erasures. As discussed in Sect. 4.2, for a (k, m) Code, the maximum number of all possible erasure patterns is given as C_1^{k+m}, C_2^{k+m}, C_3^{k+m}, for one-erasure, two-erasure and three-erasure case, respectively. Typically, erasure patterns are different, each of which produces a different set of decoding paths.

In our toy example presented in Fig. 13, there are $C_2^6 = 15$ possible erasure patterns with two erasures. Erasure pattern of $\{D_0, D_1\}$ makes it the worst case scenario. With the decoding paths given in Table 8, the size of the Decoding Table generated is 224 ($8B + 12B \times 6 + 8B \times 18$) bytes, which is stored in CPU cache. To store decoding paths of all possible erasure patterns (i.e., 15 Pattern nodes), it needs at most $224\text{B} \times 15 = 3360\text{B}$. Therefore, considering strip rotation situations, if we cache decoding paths of all possible erasure patterns, the total size of Decoding Table is at most 3360B *times N*; on the other hand, if we cache decoding paths of only one erasure pattern, total size of the Decoding Table is at most $224\text{B} \times N$. In practice, $N \leq 24$, thus, with the former method, the Decoding Table needs at most 79 KB of cache size, while with the latter method, roughly 5 KB of cache is required.

Obviously, the latter method is better than the former, because it avoids a waste of cache and redundant computation as well as search time for matching erasure pattern, by

storing decoding paths of only one erasure pattern. Nowadays, most CPUs provide over 512 KB of L-1 cache, which is enough for storing Decoding Table with a single erasure pattern.

With all the techniques discussed above, Algorithm 3 describes the caching decoding path algorithm we use to fulfill the cached decoding path method. Basically, if the erasure pattern of a stripe is found in a Pattern node in the cached Decoding Table, the algorithm goes through the list and gets the decoding paths by accessing indexes in each Key and every de_packet linked with it.

Algorithm 3 Caching Decoding Path Algorithm

Input: Decoding Table DT and a stripe ST_t with erasure pattern EP_t
Output: A stripe ST_i with erasure strips recovered

```
for each Pattern node Pattern_i in DT do
  if EP_t = Pattern_i.erasure_pattern then
    set boolean Found = true
    for each Key node Key_j linked to Pattern_i do
      for each de_packet node de_packet_k linked to Key_j do
        XOR between two packets labeled by the indexes from Key_j and de_packet_k
      end for
    end for
  end if
end for
if Found != true then
  calculate decoding paths P_t
  decode ST_t with P_t
  cache P_t by extending DT
end if
```

A New Decoding Algorithm

We finally introduce our New Decoding Algorithm, with which, both cached decoding path method and SWG XOR-scheduling algorithm are used, as described in Algorithm 4:

Algorithm 4 A New Decoding Algorithm

Input: A stripe ST with erasure pattern EP
Output: A stripe ST with erasure strips recovered

```
for each stripe ST_i do
  if erasure pattern EP_i is found then
    decode ST_i with SWG XOR-scheduling
  else
    calculate decoding paths P_i
    decode ST_i with SWG XOR-scheduling
    cache P_i
  end if
end for
```

In comparison, if the decoding implementation includes no cached decoding path method, and uses EPG XOR-scheduling algorithm, we call it a Traditional Decoding Algorithm (or conventional decoding algorithm), as described in Algorithm 5:

Algorithm 5 A Traditional Decoding Algorithm

Input: A stripe ST with erasure pattern EP
Output: A stripe ST with erasure strips recovered

```
for each stripe ST_i do
  decode ST_i with EPG XOR-scheduling
end for
```

Performance Evaluation

In this section, we present experimental measurements to show the performance improvement in decoding brought by the new decoding algorithm for XOR-based codes. As mentioned in Sects. 1 and 2, we use STAR Code as a representative of XOR-based code for performance evaluation. Two versions of STAR Code libraries are implemented: with one using the new decoding algorithm, yet the other one using traditional decoding algorithm introduced in Sect. 5. The platform configurations are described in Sect. 3.

Typically, decoding performance of STAR Code is influenced by many factors, such as blocksize, k and total number of *stripes* [6, 7]. In this section, we first discuss the size of cache used by implementing the new decoding algorithm, then present two sets of experiment measurement results based on different blocksize and k, respectively.

Practical Cache Size

In this subsection, we discuss the practical feasibility of the new decoding algorithm by analysing the usage of cache. The data structure and implementation are introduced with

more details in Sect. 5.2. Figure 17a and b showcase the average size of cache used to store the decoding path in the two sets of tests, which are derived from practical implementation experiments. Note that in the first set of tests over blocksize, we consistently set $k = 10$, and in the second set over k, blocksize is set to 1 KB.

As shown in Fig. 17, the usage of cache is not influenced by blocksize and grows as k increases. Even with relatively large number of storage units (e.g., $k = 17$), the used cache size is at most 4 KB for two-erasure case, and 11 KB for three-erasure case. In a more general case (e.g., $k = 10$), around 3 KB and 6 KB is consumed in cache for two-erasure case and three-erasure case, respectively. Since most modern CPUs provide L-1 cache of more than 512 KB, this can be easily supported.

Blocksize

In the first set of experiments, we choose *blocksize* from 16 B to 4 KB, while keeping k equal to 10. As introduced in Sect. 2, the prime number p required for STAR Code is set to 17 consistently throughout this subsection. All experiments are conducted with 1000 and 2000 stripes, respectively. Scenarios of decoding two-erasure case and three-erasure case are tested separately. Decoding one-erasure case is omitted, since it is purely re-encoding as discussed in Sect. 4.2.1. The results are displayed in Fig. 18. The x-axis represents *blocksize*, and y-axis marks the decoding speed (in unit of GB/s). The two lines on each graph, respectively, represent decoding speed of STAR library with new decoding algorithm and STAR library with traditional decoding algorithm.

Results on both platforms consistently indicate that decoding speed is increased by around 10–50% with the new decoding algorithm, compared with traditional decoding algorithm. This boost is more obvious with the three-erasure case than the two-erasure case, since in three-erasure case, the decoding paths size is larger with higher decoding computational complexity. Thus, having cached decoding path method avoids a high load of redundant decoding path computations. On the other hand, with more stripes (2000 over 1000), more obvious improvement is achieved, since more stripes lead to more redundant operations being avoided apparently.

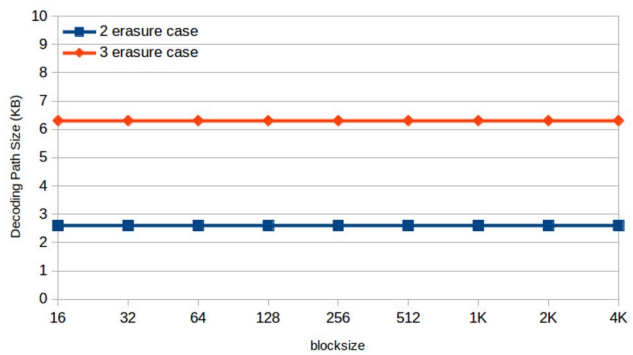
k

Next, we test impact of new decoding algorithm for various k. In this set of experiments, we keep blocksize s 1 KB and 2 KB, respectively, and change k from 6 to 17 ($p = 17$ constantly). All experiments are executed with 1000 stripes. The results are shown below in Fig. 19. The x-axis represents k, and y-axis marks the decoding speed (in unit of GB/s). The two lines on each graph, respectively, represent decoding speed of STAR library with new decoding algorithm and STAR library with traditional decoding algorithm.

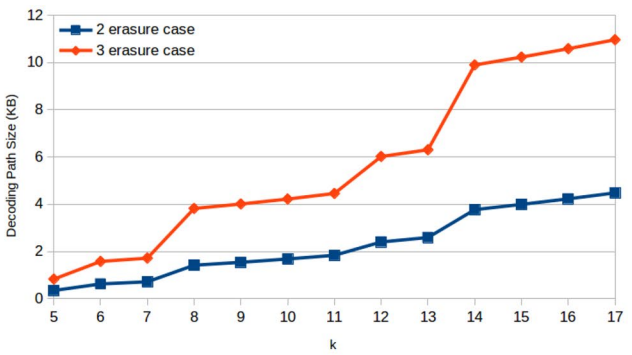
It can be observed consistently from results on both platforms that integrating new decoding algorithm improves the general decoding performance by roughly 10–50%. Similar to the observation in Sect. 6.2, three-erasure case, which needs more computation in deciding decoding paths, brings greater improvement to the decoding performance, comparing to two-erasure case. Moreover, greater improvement is achieved with more stripes (2000 over 1000) as well. This is because with more stripes, more redundant computation of decoding paths is avoided.

Conclusions and Future Work

This paper proposes a new decoding algorithm, to improve decoding performance of XOR-based erasure codes. Decoding speed of a XOR-based erasure code is typically impacted

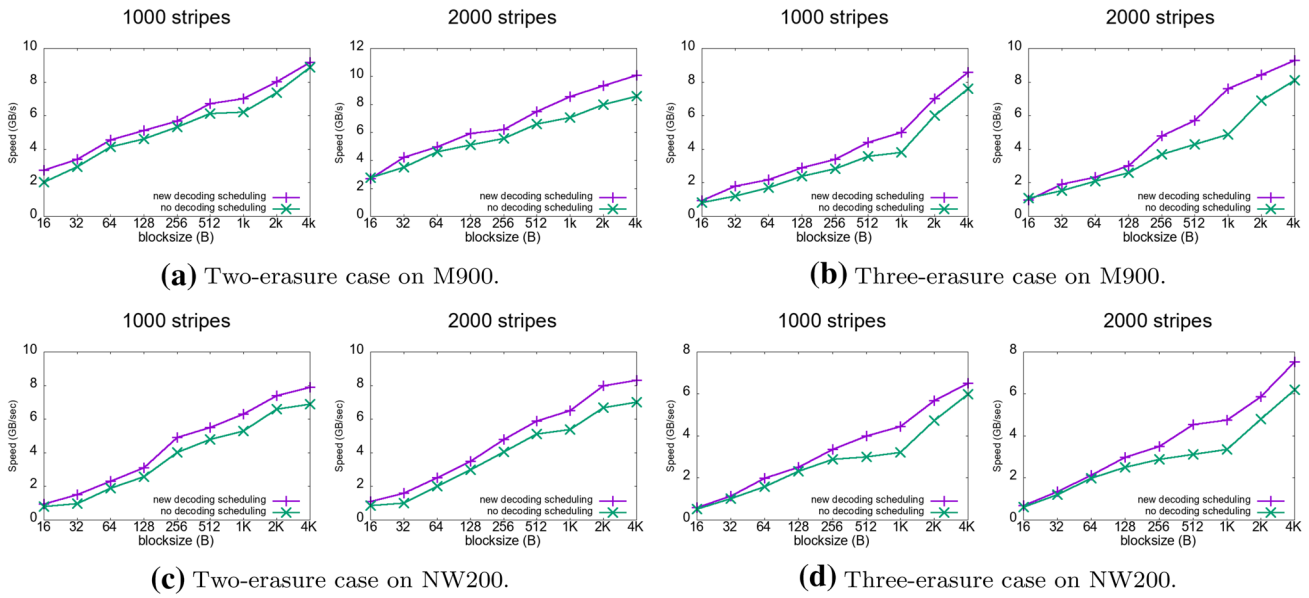


(a) Over $blocksize$ with $k = 10$.



(b) Over k with $blocksize = 1KB$.

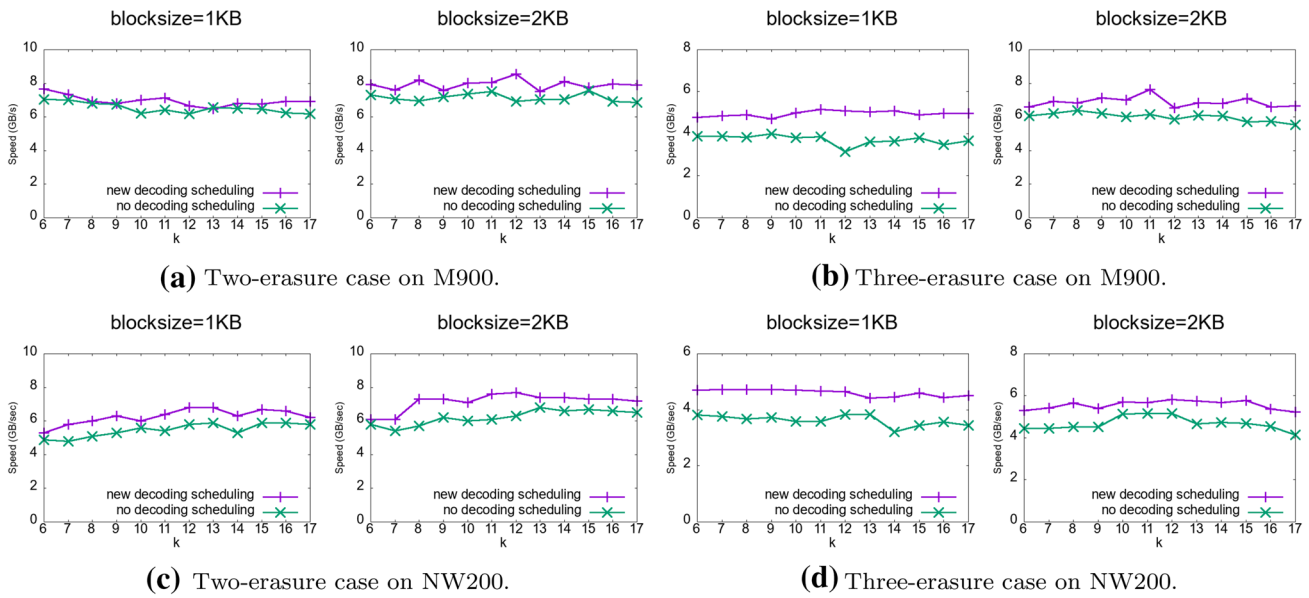
Fig. 17 Average cache size



(a) Two-erasure case on M900. **(b)** Three-erasure case on M900.

(c) Two-erasure case on NW200. **(d)** Three-erasure case on NW200.

Fig. 18 Improvement from new decoding algorithm in STAR Code for different *blocksize*



(a) Two-erasure case on M900. **(b)** Three-erasure case on M900.

(c) Two-erasure case on NW200. **(d)** Three-erasure case on NW200.

Fig. 19 Improvement from new decoding algorithm in STAR Code for different *k*

by two factors: decoding computational complexity and cache behavior. Based on these two factors, this paper:

1. studies the cached decoding path method with which decoding performance can be improved by avoiding redundant computation of decoding paths in various stripes. This cached decoding path method has been utilized by STAR code and can be further adapted by other XOR-based codes;
2. Observes different erasure patterns as well as how the size of decoding paths stored impacts CPU cache with general strip rotation, and then designs a data structure to store decoding paths in CPU. This observation enhances practical utilization of the cached decoding

path method so that it reduces both the occupied cache size and unnecessary computation;
3. Extends research in [19] and [20], in which encoding performance is optimized using a more cache-efficient XOR-scheduling algorithm. This paper proposes a similar XOR-scheduling technique to improve decoding performance using cache efficiently as well;
4. Combines both the cached decoding path method and the SWG XOR-scheduling algorithm to derive a new decoding algorithm for XOR-based erasure codes. The new decoding algorithm is implemented into STAR code, and different sets of experiments on two platforms are conducted in Sect. 6. Results on both platforms consistently indicate that, with the new decoding algorithm, decoding speed of STAR Code is improved considerably by 10–50%.

The purpose of this paper is to combine the efficient XOR scheduling and caching technique and propose a new decoding algorithm for XOR-based erasure codes, which can be readily implemented by data storage practitioners to improve the decoding performance for their systems or applications. Due to the limitation of time and equipment in our lab, the evaluation of the proposal is currently based on STAR code. However, the new decoding algorithm can be readily applied to other XOR-based codes since the principle is clear and adaptable, though some minor modifications are needed. Thus, from the consistent results shown in this paper, we can reasonably expect similar results for other XOR-based codes. Data storage practitioners and researchers are also welcomed to share their evaluation results with us.

In the future, we intend to implement into other XOR-based codes, such as RDP Code and SD Code, to benchmark the performance improvement, and we plan to separately evaluate the impact from the caching technique and the SWG scheduling on those codes. We also plan to explore the feasibility of the new decoding algorithm on Reed-Solomon Code, by integrating the proposal into practical libraries such as Jerasure and Intel's ISA-L. Since both of those libraries use an inverted Vandermonde matrix to decode, the new decoding algorithm can be applied with some modifications. In the meantime, we look forward to results from other researchers and storage system practitioners by adopting this new decoding algorithm in their systems and applications.

Compliance with Ethical Standards

Conflict of Interest Author R. Chen and author L. Xu declare that they have no conflict of interest.

References

1. Amazon EC. Amazon web services. 2015. http://aws.amazon.com/es/ec2/. Accessed Nov 2012.
2. Blaum M. A family of MDS array codes with minimal number of encoding operations. In: 2006 IEEE International Symposium on Information Theory, IEEE 2006, p. 2784–2788.
3. Blaum M, Brady J, Bruck J, Menon J. Evenodd: an efficient scheme for tolerating double disk failures in raid architectures. IEEE Trans Comput. 1995;44(2):192–202.
4. Blaum M, Bruck J, Vardy A. MDS array codes with independent parity symbols. IEEE Trans Inf Theory. 1996;42(2):529–42.
5. Blaum M, Roth RM. New array codes for multiple phased burst correction. IEEE Trans Inf Theory. 1993;39(1):66–77.
6. Chen R, Xu L. Adapting star code for non-volatile memory systems. Flash Memory Summit; 2018. The proceedings list can be found at https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/Proceedings_Chrono_2018.html. The presentation content can be found at https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180809_CTLR-301-1_Chen.pdf.
7. Chen R, Lihao X. Practical performance evaluation of space optimal erasure codes for high-speed data storage systems. SN Comput Sci. 2020;1(1):1–14.
8. Corbett P, English B, Goel A, Grcanac T, Kleiman S, Leong J, Sankar S. Row-diagonal parity for double disk failure correction. In: Proceedings of the 3rd USENIX Conference on file and storage technologies; 2004. p. 1–14.
9. Fujita H. Modified low-density MDS array codes. In: 2006 IEEE International Symposium on Information Theory, IEEE; 2006. p. 2789–2793.
10. Ghemawat S, Gobioff H, Leung S-T. The Google file system, vol. 37. New York: ACM; 2003.
11. Huang C, Simitci H, Xu Y, Ogus A, Calder B, Gopalan P, Li J, Yekhanin S. Erasure coding in windows azure storage. In: Proceedings of 2012 USENIX Annual Technial Conference, ATC'12. USENIX; 2012.
12. Huang C, Lihao X. Star: an efficient coding scheme for correcting triple storage node failures. IEEE Trans Comput. 2008;57(7):889–901.
13. Dylan B (Intel). Intel®c++ compiler 17.0 developer guide and reference, https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide, 2016. Accessed Dec 2019.
14. Thai L (Intel). Optimizing storage solutions using the intel®intelligent storage acceleration library. 2014. https://software.intel.com/en-us/articles/optimizing-storage-solutions-using-the-intel-intelligent-storage-acceleration-library. Accessed Sept 2014.
15. Khan O, Burns RC, Plank JS, Pierce W, Huang C. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In: FAST; 2012, p. 20.
16. Kiani A, Akhlaghi S. A non-MDS erasure code scheme for storage applications. *arXiv preprint* arXiv:1109.6646, 2011. Accessed Sept 2011.
17. Luby M. Lt codes. In: The 43rd Annual IEEE symposium on foundations of computer science, 2002. Proceedings IEEE; 2002. p. 271–280.
18. Luby MG, Mitzenmacher M, Shokrollahi MA, Spielman DA, Stemann V. Practical loss-resilient codes. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, ACM; 1997. p. 150–159.
19. Luo J, Shrestha M, Lihao X, Plank JS. Efficient encoding schedules for xor-based erasure codes. IEEE Trans Comput. 2014;63(9):2259–72.

20. Luo J, Xu L, Plank JS. An efficient xor-scheduling algorithm for erasure codes encoding. In: Dependable systems and networks, 2009. DSN'09. IEEE/IFIP International Conference on IEEE; 2009. p. 504–513.
21. MacWilliams FJ, Sloane NJA. The theory of error-correcting codes. Amsterdam: Elsevier; 1977.
22. Méasson C, Montanari A, Urbanke R. Maxwell construction: The hidden bridge between iterative and maximum a posteriori decoding. IEEE Trans Inf Theory. 2008;54(12):5277–307.
23. Palankar MR, Iamnitchi A, Ripeanu M, Garfinkel S. Amazon s3 for science grids: a viable solution? In: Proceedings of the 2008 international workshop on Data-aware distributed computing, ACM; 2008. p. 55–64.
24. Paolini E, Liva G, Matuz B, Chiani M. Maximum likelihood erasure decoding of ldpc codes: Pivoting algorithms and code design. IEEE Trans Commun. 2012;60(11):3209–20.
25. Papailiopoulos DS, Dimakis AG. Locally repairable codes. IEEE Trans Inf Theory. 2014;60(10):5843–55.
26. Plank JS, Greenan KM. Jerasure: a library in c facilitating erasure coding for storage applications–version 2.0. Technical report, Technical Report UT-EECS-14-721, University of Tennessee; 2014.
27. Plank JS, Luo J, Schuman CD, Xu L, Wilcox-O'Hearn Z, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. Fast. 2009;9:253–65.
28. Reed IS, Solomon G. Polynomial codes over certain finite fields. J Soc Ind Appl Math. 1960;8(2):300–4.
29. Rizzo L. Effective erasure codes for reliable computer communication protocols. ACM SIGCOMM Comput Commun Rev. 1997;27(2):24–36.
30. Sathiamoorthy M, Asteris M, Papailiopoulos D, Dimakis AG, Vadali R, Chen S, Borthakur D. Xoring elephants: novel erasure codes for big data. Proc VLDB Endow. 2013;6:325–36.
31. Shokrollahi A. Raptor codes. IEEE Trans Inf Theory. 2006;52(6):2551–67.
32. SANS Institute Stephen Lennon. Backup rotations—a final defense. 2001. https://www.sans.org/reading-room/whitepapers/sysadmin/paper/305. Accessed Aug 2001.
33. Suzuki K, Swanson S. A survey of trends in non-volatile memory technologies: 2000-2014. In: Memory Workshop (IMW), 2015 IEEE International, IEEE; 2015. p. 1–4.
34. Waddington D, Harris J. Software challenges for the changing storage landscape. Commun ACM. 2018;61(11):136–45.
35. Xu L, Bruck J. X-code: MDS array codes with optimal encoding. IEEE Trans Inf Theory. 1999;45:272–6.