Approximate Code: A Cost-Effective Erasure Coding Framework for Tiered Video Storage in Cloud Systems

Huayi Jin

ABSTRACT

Nowadays massive video data are stored in cloud storage systems, which are generated by various applications such as autonomous driving, news media, security monitoring, etc. Meanwhile, erasure coding is a popular technique in cloud storage to provide both high reliability with low monetary cost, where triple disk failure tolerant arrays (3DFTs) is a typical choice. Therefore, how to minimize the storage cost of video data in 3DFTs is challenge for cloud storage systems. Although there are several solutions like approximate storage technique for storage devices, it cannot guarantee low storage cost and high data reliability in storage systems concurrently.

To address this problem, in this paper, we propose Approximate Code, which is an erasure coding framework for video applications in cloud storage systems. The key idea of Approximate Code is distinguishing the important data and unimportant data in videos with different capabilities of fault tolerance. On one hand, for important data, Approximate Code provides triple parities to provide high reliability. On the other hand, single/double parities are applied for unimportant data, which can save the storage cost and accelerate the recovery process. To demonstrate the effectiveness of Approximate Code, we conduct several experiments in Hadoop systems. The results show that, compared to traditional 3DFTs using various erasure codes such as RS, STAR and TIP-Code, Approximate Code reduces the number of parities by up to 55%, saves the storage cost by up to 20.8%, increase the recovery speed by up to 1.25X when single disk fails, and can reconstruct the whole video data via fuzzification when triple disks fail.

KEYWORDS

Erasure Codes, Approximate Storage, Multimedia, Cloud Storage

ACM Reference Format:

Huayi Jin. 2019. Approximate Code: A Cost-Effective Erasure Coding Framework for Tiered Video Storage in Cloud Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For typical cloud storage systems such as Windows Azure [5] and Amazon AWS [1], erasure coding is a popular technique to provide both high reliability and low monetary cost [2–4, 7, 8, 10, 34, 37], where triple disk failure fault tolerant arrays (3DFTs) are widely used. Typical erasure codes can be divided into two categories, RS-based Codes [24] [15] and XOR-based codes [2, 16, 35, 39]. RS-based codes are encoded according the Galois Field Computation in Reed Solomon Code [24], which allow flexible configuration and have a little higher computation cost. XOR-based codes simplify the computation, but the scalability is a significant issue in previous literatures.

With the increasing demand on higher resolution and frame rate for video data, massive storage devices are highly desired in cloud storage systems, which makes data centers much bigger. Therefore, in this paper, we set out to answer the following question, **In a cloud storage system, how to efficiently store the tremendous video data in 3DFTs?**

To reduce the storage cost in cloud storage systems, a feasible solution is approximate storage. Approximate storage exposes unimportant data to errors, saving the overhead of redundant backups [20, 25] , thus the data reliability cannot be guaranteed. Another solution is using disk arrays like RAID-5 or RAID-6 [21], but the capability of fault tolerance would be sacrificed. Data compression[1] is also a common strategy for reducing storage overhead [9, 40, 41]. However, the compression and decompression process results in very large computational overhead, high recovery speed and high response time, which is not suitable for video applications. Therefore, existing solutions cannot provide low storage cost and high reliability simultaneously.

To address the above problem, in this paper, we propose Approximate Code[2], which is an erasure coding framework to provide comprehensive solution for video data storage in cloud systems. The key idea of Approximate Code is treating the important/unimportant data in different ways. For important data, we add additional parities to provide high capability of fault tolerance. On the other hand, the unimportant data are encoded with a minimum number of parities, which only supply the basic requirement of the recovery. When triple disks fail, the lost data can be reconstructed via a fuzzy manner.

We have the following contributions of this work,

(1) We propose Approximate Code, which a cost-effective framework to store video data in cloud storage systems.
(2) Approximate Code can be implemented by combining most erasure codes in 3DFTs, such as RS, STAR Code, TIP-Code, etc.
(3) We conduct several quantitative analysis, simulations and experiments according to different layouts of various erasure codes, and the results show that Approximate Code achieves lower storage cost and faster data recovery when single disk fails.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]We do not discuss video compression technology much in this paper because it is an application layer algorithm and can be used with our scheme.
[2]This article abbreviates "Approximate" to "Appr. " when necessary.

Table 1: The symbols used in this paper.

Symbols	Description
k	the number of data nodes in an array
r	the number of local parity nodes in an array
n	the number of nodes in an array ($n = k + r$)
h	the number of arrays
g	the number of global parity nodes
$E_{i,j}$	element at the i-th row and j-th column
LP	local parity nodes
GP	global parity nodes
ID	important data
UD	unimportant data

The rest of the paper is organized as follows. In Section 2, we introduce related work and our motivation. In Section 3, the design of Approximate Code and its encoding and decoding process will be illustrated in detail. The evaluation is presented in Section 4 and the conclusion of our work is in Section 5.

2 RELATED WORK AND OUR MOTIVATION

In this section, we introduce the background of video storage, existing solutions to reduce storage cost and the motivation of this paper. To facilitate the discussion, we summarize the symbols used in this paper in Table 1.

2.1 Basis of Video Storage

For normal HD (resolution 1280×720, 8-bit, 30 fps) video, the amount of raw video data in 1 minute is 4.63 GB, so video data is usually encoded by lossy algorithms before storage.

2.1.1 Video coding. H.264 is one of the advanced algorithms for this type of work. This coding technique is widely used on platforms such as YouTube because it has higher compression ratio and lower complexity than its predecessor. For the HD video mentioned earlier, H.264 can reduce its size by about 10 times, only 443.27MB.

H.264 classifies all frames into three different categories:

(1) I frame: A frame that does not depend on other frame data, which means it can be decoded independently of other frames.
(2) P frame: A frame holds the changes compared to the previous frame, thus saving much space by leaving out redundant information.
(3) B frame: A frame saves more space by utilizing the data of both the preceding and following frame.

A GOP consists of multiple consecutive I, P, and B frames that are independent of frames in other GOPs to prevent bit error spread. Therefore, within each GOP, the I frame has the highest importance because other frames rely on it for recovery; the importance of the P frame is second, and the B frame has the least importance. Based on this feature, a special program can be designed to distinguish the importance of video frames.

2.1.2 Video Frame Recovery. In the circumstance of video approximate storage, it's common to lose some frames and leave the video incomplete. However, the lost frames may still be recoverable with the benefit of nowadays powerful deep learning techniques. One of them is named video frame interpolation [19, 20].

Video frame interpolation is one of the basic video processing techniques, an attempt to synthetically produce one or more intermediate video frames from existing ones.This is a technique that can model natural motion within a video, and generate frames according to this modelling.

In deep learning methods, optical changes between the frames are trained in a supervised setup mapping two frames to their ground truth optical flow. Among all these, a multi-scale network[31] based on recent advances in spatial transformers and composite perceptual losses as well as a context-aware Synthesis approach[20] have so far produced the new state-of-the-art results in terms of PSNR and middlebury benchmark respectively.

2.2 Existing Erasure Codes

Reliability is a critical issue since disk failures are typical in storage systems. To improve the reliability of storage systems, several RAID forms (e.g., RAID-5, RAID-6, 3DFTs) and erasure codes are proposed by researchers. Traditional erasure codes can be categorized into two classes, Maximum Distance Separable (MDS) codes and non-MDS codes. MDS codes aim to offer data protection with optimal storage efficiency. On the other hand, non-MDS codes improve the performance or reliability by consuming extra storage space.

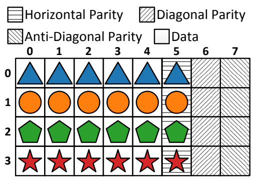
In the past two decades, several famous erasure codes are proposed for double Disk Failure Tolerant arrays (2DFTs or RAID-6), such as EVENODD code [2], RDP code [7], Blaum-Roth code[3], X-code [37], Liberation code [23], Liber8tion code [22], Cyclic [6] code, B-Code [36], Code-M [32], H-code [35], P-code [17] and HVcode [27], etc.

In Triple Disks Failure Tolerant Arrays (3DFTs), typical MDS codes include Reed-Solomon codes [24], Cauchy-RS codes [4], STAR code [16], Triple-Star code [34], Triple-Parity code [8], HDD1 code [29], RSL-code [10], RL-code [11], and so on. Typical non-MDS codes contain WEAVER codes [12], HoVercodes [13], T-code [28], HDD2 code [29], Pyramid codes [14], Local Reconstruction Codes [15], Locally Repairable Codes [26], etc.

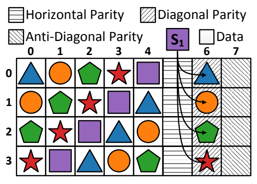
Reed Solomon Code [24] (RS code) is one kind of classic MDS codes, but its scheme requires additions and multiplications over finite fields and results in high complexity in decoding and updating. Several attempts have been made to address this issue. Local Construction Code (LRC) [15] is one of these attempts which succeeds in reducing the bandwidth and I/Os required for repair reads at the expense of higher storage overhead.

Unlike the RS code basing on additions and multiplications, another type of erasure codes is XOR-based codes and some are used in RAID (Redundant Arrays of Independent Drives) structures. This type of codes uses purely XOR operations, therefore showing a great advantage on encoding/decoding speed. XOR-based codes like EVENODD [2] and H-code [35] can tolerate 2 disk failures (EVENODD can even be designed to tolerate 3 disk failure), and they can be directly applied in RAID structures. More sophisticated ones like STAR code [16] and TIP code [39] can tolerate 3 disk failures.

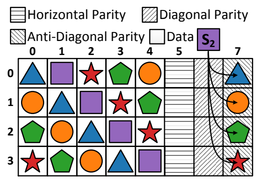
STAR code [16] is an extension of the double-erasure-correcting EVENODD [2] code, and a modification of the generalized triple-erasure-correcting EVENODD code. It employs a novice diagonal redundancy as shown in Figure 1.



(a) Horizontal parity coding (e.g., $E_{0,5} = E_{0,0} \oplus E_{0,1} \oplus E_{0,2} \oplus E_{0,3} \oplus E_{0,4}$).



(b) Diagonal parity coding (e.g., $S_1 = E_{3,1} \oplus E_{2,2} \oplus E_{1,3} \oplus E_{0,4}$ and $E_{0,6} = E_{0,0} \oplus E_{3,2} \oplus E_{2,3} \oplus E_{1,4} \oplus S_1$).



(c) Anti-diagonal parity coding (e.g., $S_2 = E_{0,1} \oplus E_{1,2} \oplus E_{2,3} \oplus E_{3,4}$ and $E_{0,7} = E_{0,0} \oplus E_{1,1} \oplus E_{2,2} \oplus E_{3,3} \oplus S_2$).

Figure 1: Encoding of STAR code ($p = 5$).

TIP-code [39] is a solution for p or p+1 disks, where p is a prime number. Compared to STAR, Tip's input of the diagonal and anti-diagonal parity is modified so they are evenly distributed among all the data blocks. This minor modification has enabled TIP-code to achieve optimal update complexity.

2.3 Approximate Storage

Approximate Storage is another way outside of traditional methods of trading off the limited resource budget with the costly reliability requirements, which recently receives more attentions since data centers are faced with storage pressure from the ever-increasing data.

Use cases for approximate storage range from transient memory to embedded settings and mass storage cloud servers. Mapping approximate data onto blocks that have exhausted their hardware error correction resources, for example, to extend memory endurance. On embedded settings, it enables the reduction of the cost of accesses and preserve battery life to loosen the capacity constraints. [25] Here, in data-center-scale video database, approximate storage can provide multiple levels of fault tolerance for data of different importance, avoiding redundant backup for the less-important data, thus saving a significant amount of space.

Approximate storage loosens the requirement of storage reliability by allowing quality loss of some specific data. Therefore, programmers can specify the importance of the data segments and assign them to different storage blocks. The critical data is still safe because they are stored and sufficiently backed up by expensive and highly reliable storage devices. Meanwhile, non-critical data is exposed to error, thus increasing storage density and saving cost.

Table 2: Comparison of storage overhead, reliability and performance between EC, Approximate Storage and Approximate Code.

Schemes		Storage Overhead	Reliability	Performance
EC	RS	high	high	medium
	RAID-6	medium	medium	high
Appr. Storage		low	low	high
Appr. Code	ID	low	high	high
	UD		medium	high

2.4 Our Motivation

Based on Table 2, either the existing erasure codes or the approximate storage methods cannot meet the requirements of video applications in the cloud storage system due to the following reasons.

First, the overhead of existing 3DFTs erasure codes is too high. Second, the 2DFTs erasure codes sacrifice part of reliability, and the reliability of approximate storage schemes are much lower since they cannot tolerate disk-level failures. Finally, existing erasure codes provide the same fault tolerance for all data without distinction, resulting in the same reliability for error-sensitive data and robust data.

Based on these reasons, we propose a scheme for tiered video storage in a highly reliable environment, Approximate Code. Approximate code is an erasure code framework that uses tiered storage[18] [33] [38] [30] and approximate storage strategies to separate an existing erasure codes into codes with different fault tolerances and apply them to important and unimportant data.

3 APPROXIMATE CODE

In this section, we introduce the Approximate Code Framework and its properties through a few simple examples.

3.1 Overview of Approximate Code Framework

The Approximate Code mainly includes three operations: code segmentation, structure selection and code generation. We use Figure 2 to illustrate our design.

3.1.1 Code Segmentation. For an input erasure code, we assume that its fault tolerance is x. The Approximate Code first splits its parity block into two parts: local and global. The former verifies all important and non-critical data, while the latter only verifies important data. Code segmentation are designed to ensure that local parities can tolerate any r node failures, so the fault tolerance of non-critical data is r. For important data, the code segmentation ensures that the parity block of important data can be completely recombined into the original xDFTs erasure code scheme.

In the example, a 3DFTs erasure code with 4 data chunks and 3 parity chunks is input, and its 3 parity chunks are separated into two local and one global parity chunks.

3.1.2 Structure Selection. The Approximate Code framework then choose the structure for the input erasure code. There are two main structures that distribute important and unimportant data in different ways. As shown in Figure 2, in *Structure I*, important data

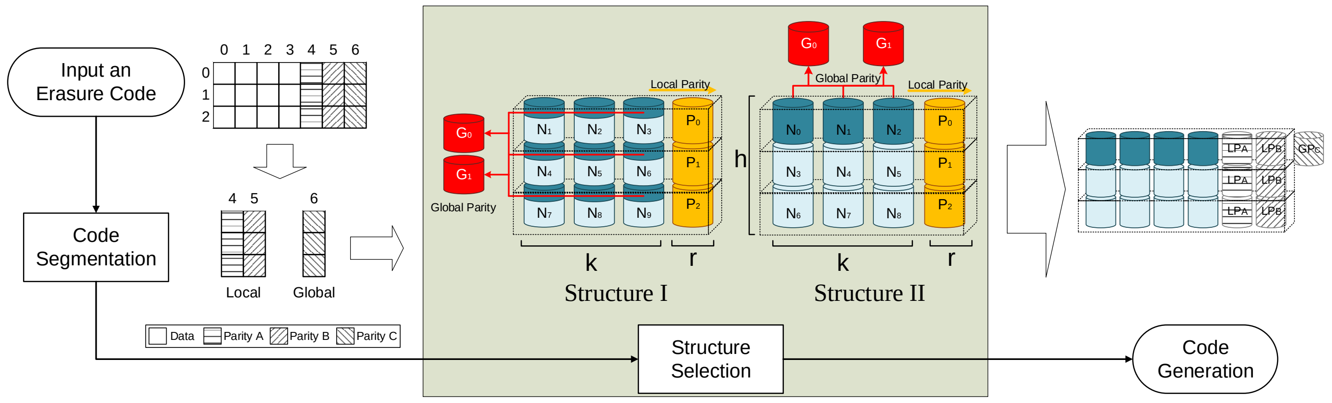


Figure 2: The framework of Approximate Code

occupies 1 block in each node, and in *Structure II*, it occupies 1 data arrays. We specify that in *Structure I*, the ratio of important data to each node is $1/h$, thereby ensuring that the number of important data in both *Structure I* and *Structure II* just fills up k nodes.

Since *Structure I* distributes important data across each node, it guarantees a more balanced load. *Structure II* concentrates important data into an array, and provides greater reliability for important data. A detailed analysis of this will be presented in Section +3.4.

3.1.3 Code Generation. After code segmentation and structure selection, the Approximate Code Framework generates an approximate form of the input erasure code.

The construction of the Approximate Code is determined by 5 parameters k, r, g, h and *Structure*. We define the naming rules of the output Approximate Code as **"Approximate XXX Code** $(k, r, g, h, Structure)$**"**, where "XXX" is the name of input erasure code. Paremeter *Structure* is I or II and it can be omitted when not focusing on the specific construction.

In a n-node array, k of them are data nodes and $r = n - k$ nodes are for local parity, where each node can be divided into multiple blocks. Approximate Code are designed for h arrays, therefore it includes $k * h$ data nodes and $r * h$ local parity nodes. Besides them, g nodes are designed for global parity, which are calculated only by the important data blocks, so the total number of nodes in Approximate Code (k, r, g, h) is $h * (k + r) + g$.

Approximate Code guarantees that the unimportant data can tolerate any r node failures and the important data can tolerate any $r + g$ node failures. Since the 3DFTs is a tipical configuration, we mainly discuss the situation of $r + g = 3$, so r is usually set to 1 or 2.

3.2 Approximate RS Code

For RS(k,p), it can be seen that it provides p independent horizontal parity nodes for k data nodes. Therefore, the code segmentation for RS is arbitrary, that is, it supports any $r = p - g (0 < g < p)$. RS also supports both two structures.

For the situation shown in Figure 2, we consider the input erasure code as RS(4,3). It is divided into two local parities and one global parities with the *Structure II* and it generate Approximate RS Code (4,2,1,3,II). As a result, the unimportant data constitutes RS(4,2), and the important data constitutes RS(4,3).

3.3 XOR-based Approximate Code

Since we mainly provide 3DFTs for important data, we prefer to construct Approximate Code with several XOR-based codes to provide faster encoding and reconstruction speed than RS. This section introduce two tipical construction of XOR-based Approximate Code, Approximate STAR Code and Approximate TIP Code.

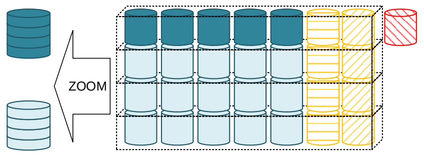


Figure 3: Construction of Approximate STAR Code (5,2,1,4,II)

3.3.1 Approximate STAR Code. Figure 3 shows the construction of Approximate STAR Code (5,2,1,4,II).

As introduced in 2.2, STAR Code [16] has 3 parity chunks and it is a tipical 3DFTs XOR-based erasure code. Since STAR is a direct extension of EVENODD[2], that is to say, the STAR that removes the anti-diagonal parity is the typical RAID-6 code EVENODD. As shown in Figure 4, in this case, the horizontal and diagonal parity are used as local parities while the anti-diagonal parity is defined as global parity. According to the construction requirements of STAR, each node is divided into 4 blocks.

Briefly, Approximate STAR Code (5,2,1,4,II) can be considered to be important data encoded by the full STAR, while the unimportant data is encoded by a portion of STAR, EVENODD. It is plain to see that important data can tolerate any 3 node failures while unimportant data can tolerate any 2 node failures.

3.3.2 Approximate TIP Code. We then intorduce the generation of Approximate TIP Code (5,1,2,6,I).

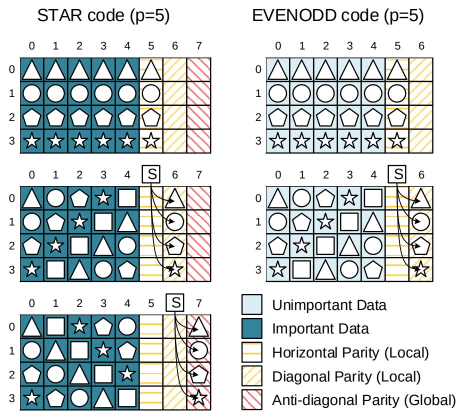


Figure 4: Encoding process of Approximate STAR Code (5,2,1,4,II), which can be divided into STAR (encode important data) and EVENODD (encode unimportant data).

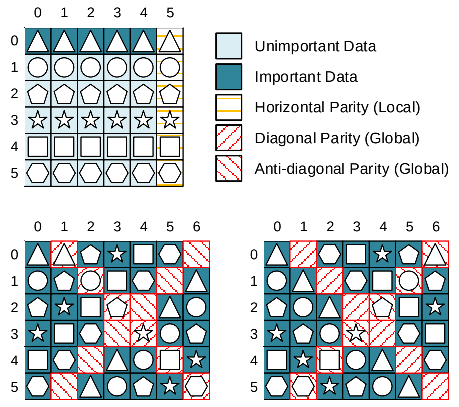


Figure 6: Encoding process of Approximate TIP Code (5,1,2,6,I), which can be divided into TIP (encode important data) and RAID-5 (encode unimportant data). (e.g., the diagonal parity block marked as a square $E_{4,5} = E_{4,0} \oplus E_{3,1} \oplus E_{2,2} \oplus E_{1,3} \oplus E_{0,4}$)

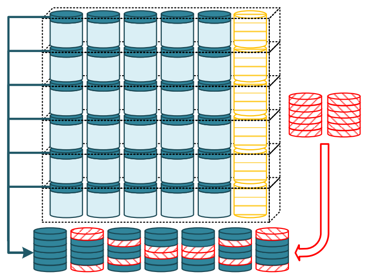


Figure 5: Construction of Approximate TIP Code (5,1,2,6,I)

The horizontal parity block of TIP code is defined as the local parity and the remaining two as the global parities.

As shown in Figure 5, 30 data nodes form 6 layers, where important data accounting for 1/6 of each node. All data nodes generate 6 local parity nodes. Important data blocks are rearranged to fit the parity form of TIP Code and generate two global parity nodes. The encoding process of Approximate TIP Code (5,1,2,6,I) is shown in Figure 6.

3.4 Reconstruction Process and Fault Tolerance

When r or less nodes fail, all the data can be reconstructed by local parities. When $r + g$ or less nodes fail, all the important data can be reconstructed by local and global parities. However, in most cases, the loss of more nodes is also recoverable. We only consider two situations: $r+1$ and $r+g+1$. The former exceeds the fault tolerance of unimportant data, while the latter exceeds the fault tolerance of important data. As mentioned in 3.1.3, $r+g$ is set to 3. Approximate Codes based on different structures have different fault tolerance and reconstruction cases, so we discuss *Structure I* and *Structure II* respectively.

3.4.1 *Structure I.*

Under *Structure I*, important data and unimportant data are distributed in the same nodes, so they suffer the same risk of device failures.

For unimportant data, it is recoverable as long as the node loss per line does not exceed r. Therefore, the unrecoverable scenarios account for

$$\frac{kh}{\binom{(k+r)h+g}{r+1}}.$$

of all $r + 1$ node failure cases. For important data, when 4 nodes fail, the unrecoverable scenarios account for

$$h * \frac{\sum_{i=0}^{g} \binom{4-i}{k+r} * \binom{i}{g}}{\binom{4}{(k+r)*h+g}} (g = 0, 1)$$

In Approximate Code (3,1,2,3,I), 90.11% 2-node failure is recoverable for unimportant data nodes and 95.50% 4-node failure is recoverable for important data nodes.

3.4.2 *Structure II.*

Important and unimportant data are storaged in different nodes under *Structure II*. When $r + 1$ nodes fail, the unrecoverable scenarios for unimportant data account for

$$\frac{k(h-1)}{\binom{(k+r)h+g}{r+1}}.$$

When 4 nodes fail, the unrecoverable scenarios for important data account for

$$\frac{\binom{k+3}{4}}{\binom{(k+r)h+g}{4}}.$$

In Approximate Code (3,1,2,3,II), 93.41% 2-node failure is recoverable for unimportant data nodes, and 98.50% 4-node failure is recoverable for important data nodes.

3.5 Properties of Approximate Code

We analyze the nature of the Approximate Code from the following aspects, and the calculation method of the relevant indicators is given in Table 3.

- Low Storage Overhead: Approximate Code reduces storage overhead by approximating and tiered storage strategies. This property is more pronounced for data with a smaller proportion of important data.
- Optimal Update Complexity: When one node is updated, Approximate RS Code only needs to write r local parity nodes and only g/h global parity nodes on average besides the write of data node.
- High reliability for important data. The Approximate Code guarantees 3DFTs for important data.
- Flexibility. The implementation of the Approximate Code can be based on RS, XOR or a mixture of the two.

Table 3: Comparison of storage overhead, fault tolerance and average single write performance between RS, STAR, TIP, Approximate RS, Approximate STAR and Approximate TIP

EC	Storage Overhead	Fault Tolerance	Single Write Performance(Average)
RS(k, r)	$(k+r)/k$	r	$r+1$
STAR$(k, 3)$	$(k+3)/k$	3	$6-\frac{4}{k}$
TIP-code$(k, 3)$	$(k+3)/k$	3	4
Appr. RS (k, r, g, h)	$\frac{(k+r)h+g}{kh}$	$r+g$	$1+r+\frac{g}{h}$
Appr. STAR $(k, 2, 1, h)$		3	$2\frac{k-h-1}{kh}+4$
Appr. TIP $(k, 1, 2, h)$		3	$2+\frac{2}{h}$

3.6 Implementation

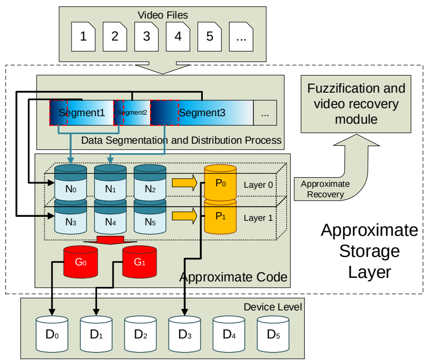


Figure 7: Overview of Approximate Storage Layer

Compared with the traditional scheme that does not consider the meaning of the upper layer data, the Approximate Code pays attention to the difference of the importance of the data, so an intermediate layer between the upper layer application and the underlying distributed storage system is necessary. We call it the approximate storage layer, which include 3 parts: data identification and allocation, encoding and decoding process of Approximate Code and the video fuzzification and recovery.

3.6.1 Data Segmentation and Distribution module. The data segmentation module splits the video file stream into multiple data segments and automatically discriminates the important data. A natural approach is to follow the GOP segmentation. According to the 2.1, the GOP of the encoded video starts with an I frame, and all the data in that GOP depends on it for decoding, so we define it as important data and divide the GOP into I frames (important part) and other data. (unimportant part). This module is also responsible for analyzing the proportion of important data and unimportant data in the data stream to select the most appropriate coding parameter configuration to ensure fault tolerance and high storage efficiency of important data.

3.6.2 Approximate Code module. The Approximate Code module is responsible for encoding and decoding important and unimportant data and completely recovering the data within fault tolerance. For data that cannot be recovered, the Approximate Code module transmits it to the video recovery module for approximate recover videos.

3.6.3 Data Fuzzification and Video Recovery. In the approximate recovery mode, the remaining data and important data are transferred to this module. Using a series of methods described in Section 2.1, frames that suffer from frame loss will be recovered using the interpolation algorithm; the corrupted frames will be processed into fuzzy images and approximately recovered by superpixel techniques.

4 EVALUATION

In this section, we conduct a series of experiments to demonstrate the efficiency of Approximate Code.

4.1 Evaluation Methodology

To compared with Approximate Code, We choose several XOR-based code, including TIP-code, EVENODD, STAR, RAID5 and RAID6, RS code and LRC codes which are widely used in cloud storage systems. Since 3DFTs are typical configurations, we set the number of parity nodes to 3 for all kinds of codes in this evaluation.

Experiments as well as mathematical analysis are used to demonstrate the performance of Approximate Code.

4.1.1 Metrics and Methods for Mathematical Analysis: We use the **Storage Efficiency**, **Fault Tolerance** and **Write(update) Cost** defined in Section 3.5 as measure. We first analyze the impact of the settings of the three parameters on these metrics, then we compare these matrics with several EC methods.

4.1.2 Metrics and Methods for Experiments: We use **Encoding Time** and **Recovery Time** as the metrics in our experiments. **Encoding Time** is the time for generating all parity nodes and **Recovery Time** is the time for recovering the lost nodes. Several codes are chosen to compared with Approximate Code with the same data nodes k. Attention that Approximate Code can only recover the important data nodes in some cases and we only keep the lost data nodes to be the same for all codes which means Approximate Code may recover less data than other codes.

The environment of our experiments are shown in Table 4. We set a Hadoop system with one NameNode and h DataNodes. Each DataNodes storage the data of one layer of Approximate Code and the global parities are distributed at each DataNodes. We set $h = 4, 6$ and node size equal $1GB$ here as two common configurations.

Our experiment can mainly be divided into five cases. In the first part, we evaluate the encoding time for all kinds of codes. In the next three parts we measure the recovery time under single, double and triple failure nodes condition respectively. Finally we conduct the experiment of frame interpolation to prove that the lost of video frames is recoverable to some extent.

Description	DELL R730 Server
CPU	Intel Xeon 3.0GHz
NIC 1Gbps	1Gbps
Memory	32GB
Disk	8TB HDD
OS	Linux 3.19
Platform	Hadoop HDFS 3.0.3

Table 4: Details of Our Evaluation Platform

4.2 Numerical Results of Mathematical Analysis

4.2.1 Storage Overhead. We compare the storage overhead between RS (k,3), Appr. RS (k,1,2,h) and Appr. RS (k,2,1,h), where h = 4 or 6. As shown in Figure 8, the results shows that Appr. RS Code has lower storage overhead than RS Code. The ratio of optimization is up to 21.4% when h=4, k=4 and 23.8% when h=6, k=4.

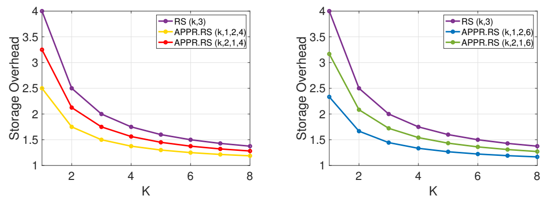
4.2.2 Fault Tolerance. All the erasure codes we evaluate in this Section are 3DFTs.

4.2.3 Single Write Performance. We compare the single write performance between RS, STAR, Appr. RS and Appr. STAR, where h = 4 or 6. As shown in Figure 9, the results shows that Appr. RS Code has lower single write performance than RS Code and STAR Code. The ratio of optimization is up to 41.3% when h=6 and 25% when h=4. As k increases, the performance of Appr. STAR is close to RS and always better than STAR.

4.3 Experimental Results Results and Analysis

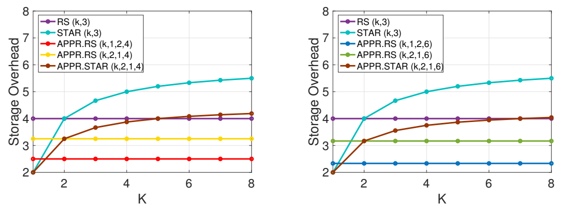
4.3.1 Encoding Time Analysis. In this part, we compared the encoding time of Approximate Code with Tip code, STAR code, RS code and LRC codes, respectively. Here we use Tip and STAR code to generate parities in Approximate Code. Thus we can cover all the code configurations from $k = 5$ to $k = 17$.

- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and STAR (k,3): Figure **??** illustrates that Approximate Code encodes faster than STAR code and the rate of optimization is up to 56.3% (between Appr. STAR (k,1,2,6) and STAR (k, 3) when k = 5).



(a) h=4 (b) h=6

Figure 8: Storage Overhead



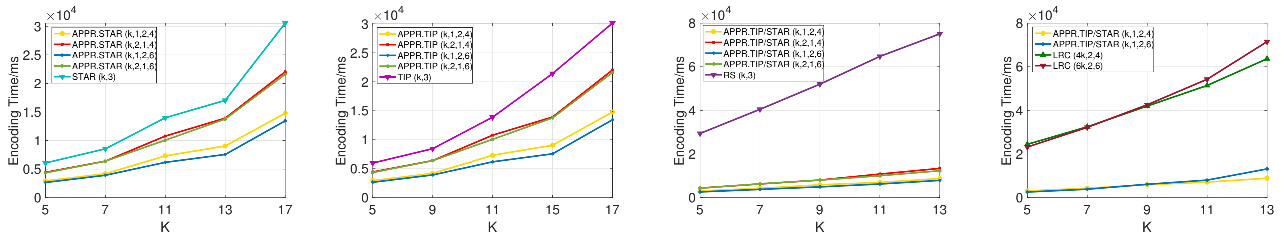
(a) h=4 (b) h=6

Figure 9: Single Write Performance

- Appr. TIP(k,1,2,4), Appr. TIP(k,2,1,4), Appr. TIP(k,1,2,6), Appr. TIP(k,2,1,6) and TIP(k,3): the effet of Tip code is similar to STAR code. From Figure 10(b), the rate of optimization is up to 58.1% (between Appr. TIP(k,1,2,6) and TIP(k, 3) when k = 9).
- Appr. TIP/STAR (k,1,2,4), Appr. TIP/STAR (k,2,1,4), Appr. TIP/STAR (k,1,2,6), Appr. TIP/STAR (k,2,1,6) and RS(k,3): From the Figure 10(c), we could find out that the AP code encodes largely faster than RS code. The rate of optimization is up to 91.1% (between Appr. TIP/STAR (k,1,2,6) and RS(k, 3) when k = 5).
- Appr. TIP/STAR (k,1,2,4), Appr. TIP/STAR (k,2,1,4), Appr. TIP/STAR (k,1,2,6), Appr. TIP/STAR (k,2,1,6) and LRC(k,4,3), LRC(k,6,3): According to Figure 10(d), we could see that Approximate Code largely reduces the encoding time, comparing to LRC codes. The rate of optimization is up to 88.9% (between Appr. TIP/STAR (k,1,2,6) and LRC(k, 6, 2) when k = 5).

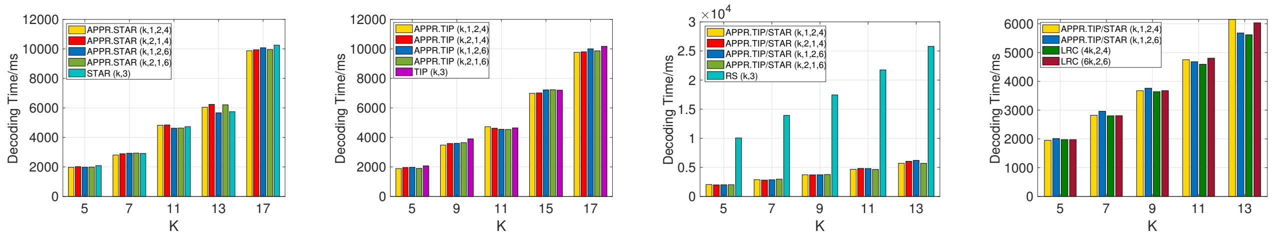
We combined the encoding time of five codes in Figure 14(a) with k=5. The results show that Approximate Code has the best encoding performance among five codes. There are two reasons for this condition:1) Approximate Code only generates 3 parities for important data which lead to the less size of total parity nodes, comparing with other codes. 2)Approximate Code use XOR-based codes (Tip and star code) which have lower computational overhead than RS -based codes.

4.3.2 Decoding Time under One Failure Node Condition Analysis. In this section, we compared the decoding time of Approximate Code with Tip code, Star code, RS code and LRC codes under single failure node condition, respectively. k changes from 5 to 17.



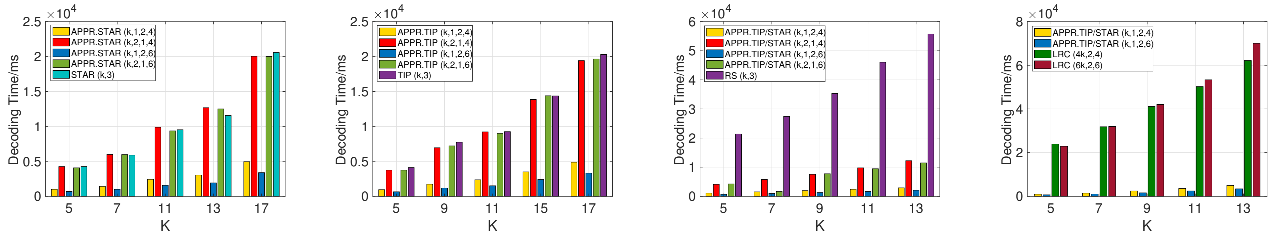
(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 10: Encoding Time Comparison between AP method and other method



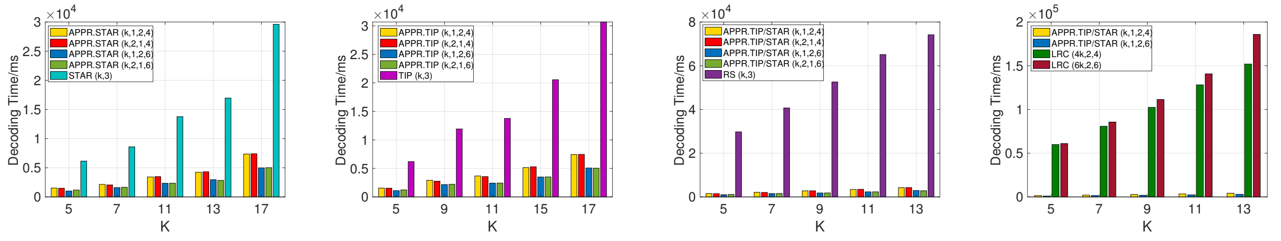
(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 11: Decoding Time Comparison in One Disk between AP method and other method



(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 12: Decoding Time Comparison in Two Disk between AP method and other method



(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 13: Decoding Time Comparison in Two Disk between AP method and other method

- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and STAR (k,3): Figure 11(a) illustrates that the decoding time of Approximate Code is almost the same as STAR code when sinlge node fails and the rate of optimization is up to 5.4% (between Appr. STAR (k,1,2,4) and STAR (k, 3) when k = 5).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and TIP(k,3): Figure 11(b) illustrates that

the decoding time of Approximate Code is almost the same as Tip code when sinlge node fails and the rate of optimization is up to 8.8% (between Appr. TIP(k,1,2,4) and TIP(k, 3) when k = 17).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and RS(k,3): From the Figure 11(c), we could find out that the Approximate Code decodes largely faster than RS code. The rate of optimization is up to 80.2% (between Appr. TIP/STAR (k,1,2,4) and RS(k, 3) when k = 5).
- Appr. TIP/STAR (k,1,2,4), Appr. TIP/STAR (k,2,1,4), Appr. TIP/STAR (k,1,2,6), Appr. TIP/STAR (k,2,1,6) and LRC(k,4,3), LRC(k,6,3): According to Figure 11(d), we could see that Approximate Code and LRC codes have no big gap when encoding one lost node. The rate of optimization is up to 5.8% (between Appr. TIP/STAR (k,1,2,6) and LRC(k, 6, 2) when k = 13).

We combined the decoding time of five codes under one failure node condition in Figure 14(b) with k=5. The results show that RS code has the worst one node decoding performance among five codes and the performance of other four codes are almost the same. The main reason is that XOR-based codes have lower computational overhead than RS-based codes.

4.3.3 *Decoding Time under Two Failure Nodes Condition Analysis.*

In this part, we compared the decoding time of Approximate Code with Tip code, Star code, RS code and LRC codes under two failure nodes condition, respectively. k changes from 5 to 17.

- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and STAR (k,3): Figure 12(a) illustrates that the rate of optimization is up to 83.9% (between Appr. STAR (k,1,2,6) and STAR (k, 3) when k = 5).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and TIP(k,3): This condition is similar to STAR code. Figure 12(b) illustrates that the rate of optimization is up to 84.8% (between Appr. TIP(k,1,2,6) and TIP(k, 3) when k = 9).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and RS(k,3): From the Figure 12(c), we could find out that the Approximate Code decodes largely faster than RS code under two failure nodes condition. The rate of optimization is up to 96.9% (between Appr. TIP/STAR (k,1,2,6) and RS(k, 3) when k = 5).
- Appr. TIP/STAR (k,1,2,4), Appr. TIP/STAR (k,2,1,4), Appr. TIP/STAR (k,1,2,6), Appr. TIP/STAR (k,2,1,6) and LRC(k,4,3), LRC(k,6,3): According to Figure 12(d), we could see that Approximate Code largely reduces the decoding time when two nodes fail, compared with LRC codes. The rate of optimization is up to 97.2% (between Appr. TIP/STAR (k,1,2,6) and LRC(k, 6, 2) when k = 5).

The combined decoding time of five codes under two failure node condition are shown in Figure 14(c) (k=5). The results show that RS code and LRC codes has the worst two failure nodes decoding performance while Appr. TIP/STAR (k,1,2,6) and Appr. TIP/STAR (k,1,2,4) perform best among all kinds of codes. The main reasons are that: On two failure nodes condition, RS code and LRC codes decode data based on RS method which have heavy computational cost while Appr. TIP/STAR (k,1,2,6) and Appr. TIP/STAR (k,1,2,4) do the approximate recovery which means they only recover important data at this time.

4.3.4 *Decoding Time under Three Failure Nodes Condition Analysis.*

In this part, we compared the decoding time of Approximate Code with Tip code, Star code, RS code and LRC codes under three failure nodes condition, respectively. k changes from 5 to 17.

- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and STAR (k,3): Figure 13(a) illustrates that the rate of optimization is up to 83.5% (between Appr. STAR (k,1,2,6) and STAR (k, 3) when k = 5).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and TIP(k,3): This condition is similar to STAR code. Figure 13(b) illustrates that the rate of optimization is up to 82.2% (between Appr. TIP(k,1,2,6) and TIP(k, 3) when k = 5).
- Appr. STAR (k,1,2,4), Appr. STAR (k,2,1,4), Appr. STAR (k,1,2,6), Appr. STAR (k,2,1,6) and RS(k,3): From the Figure 13(c), we could find out that the Approximate Code decodes largely faster than RS code under three failure nodes condition. The rate of optimization is up to 96.7% (between Appr. TIP/STAR (k,1,2,6) and RS(k, 3) when k = 5).
- Appr. TIP/STAR (k,1,2,4), Appr. TIP/STAR (k,2,1,4), Appr. TIP/STAR (k,1,2,6), Appr. TIP/STAR (k,2,1,6) and LRC(k,4,3), LRC(k,6,3): According to Figure 13(d), we could see that Approximate Code largely reduces the decoding time when three nodes fail, compared with LRC codes. The rate of optimization is up to 98.3% (between Appr. TIP/STAR (k,1,2,6) and LRC(k, 6, 2) when k = 5).

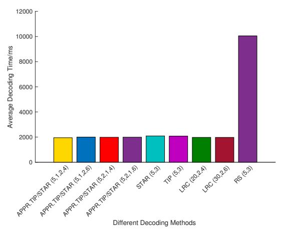
The combined decoding time of five codes under three failure node condition are shown in Figure 14(d) (k=5). The results show that LRC has the worst three failure nodes decoding performance. The main reasons are that: On three failure nodes condition, LRC codes decode data through the global parities which has a longer parity chain. For example $LRC(44, 4, 2)$ use 44 nodes to recover the three lost nodes. the computational cost of GF operations is also larger than XOR operations. On the other hand, All kinds of Approximate Code perform best among all kinds of codes because they only recover important data at this time.

4.4 Frame Recovery

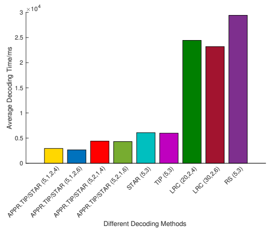
Here we present the frame recovery result by using frame interpolation [19, 20, 31]. As shown in Figure15, The top 3 figures are the original frames. When frame 2 lost, the interpolation techniques can output 15(d) by the input of 15(a) and 15(c). The recovered frame has a lower resolution and may differ from the original image, but since the unconstrained frame contains most of the context information, the video is still available, which validates the fact that the video can tolerate a certain amount of data loss.

5 CONCLUSION

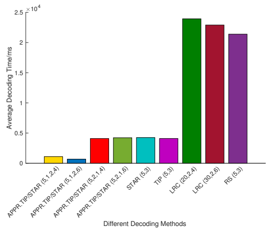
In this paper, we present the Approximate Code, which is the framework for video tiered storage in cloud systems. The Approximate Code can segment the input erasure codes and provide different



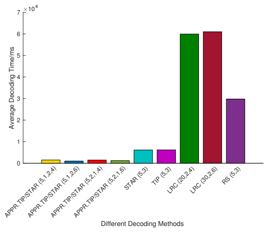
(a) Encoding Time



(b) Decoding Time in One Node Failure

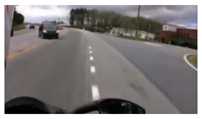


(c) Decoding Time in Two Node Failure



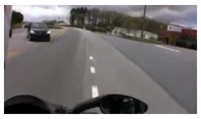
(d) Decoding Time in Three Node Failure

Figure 14: Comparison of several matrics between different methods when number of data nodes are 5

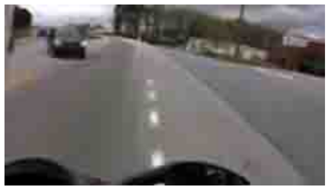


(a) Frame 1

(b) Frame 2



(c) Frame 3



(d) Recovery of Frame 2

Figure 15: Frame Recovery Result using Frame Interpolation

fault tolerance for data of different importance. For important data, the Approximate Code provides 3DFTS, and for non-critical data, the Approximate Code provides single or double parity, thus saving storage costs and speeding up recovery, while having good update write performance. We conducted several experiments in the Hadoop system and found that compared to traditional 3DFTs using various erasure codes such as RS, STAR and TIP-Code, Approximate Code reduces the number of parities by up to 55%, saves the storage cost by up to 20.8%, increase the recovery speed by up to 1.25X when single disk fails, and can reconstruct the whole video data via fuzzification when triple disks fail.

ACKNOWLEDGMENTS

REFERENCES

[1] Ignacio Bermudez, Stefano Traverso, Marco Mellia, and Maurizio Munafo. 2013. Exploring the cloud from passive measurements: The Amazon AWS case. In *2013 Proceedings IEEE INFOCOM*. IEEE, 230–234.
[2] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on computers* 44, 2 (1995), 192–202.
[3] Mario Blaum and Ron Roth. 1999. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory* 45, 1 (1999), 46–59.
[4] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. 1995. *An XOR-based Erasure-Resilient coding scheme.* Technical Report TR-95-048. International Computer Science Institute.
[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 143–157.
[6] Yuval Cassuto and Jehoshua Bruck. 2009. Cyclic Lowest Density MDS Array Codes. *IEEE Transactions on Information Theory* 55, 4 (2009), 1721–1729.
[7] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. 2004. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of the USENIX FAST'04*. San Francisco, CA.
[8] Peter Corbett and Atul Goel. 2011. Triple parity technique for enabling efficient recovery from triple failures in a storage array. US Patent 8,015,472.
[9] Peter Deutsch. 1996. *DEFLATE compressed data format specification version 1.3*. Technical Report.
[10] Gui Liang Feng, Robert Deng, Feng Bao, and J C Shen. 2005. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Trans. Comput.* 54, 9 (2005), 1071–1080.
[11] Gui Liang Feng, Robert Deng, Feng Bao, and J C Shen. 2005. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple ($\geq$ 4) disk failures. *IEEE Trans. Comput.* 54, 12 (2005), 1473–1483.
[12] James Hafner. 2005. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, Vol. 5. 16–16.
[13] James Hafner. 2006. HoVer Erasure Codes For Disk Arrays. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. 217–226.
[14] Cheng Huang, Minghua Chen, and Jin Li. 2007. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. In *Network Computing and Applications, 2007 NCA'07. Sixth IEEE International Symposium on*. IEEE, 79–86.
[15] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 15–26.
[16] Cheng Huang and Lihao Xu. 2008. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Trans. Comput.* 57, 7 (2008), 889–901.
[17] Chao Jin, Hong Jiang, Dan Feng, and Lei Tian. 2009. P-Code: A new RAID-6 code with optimal properties. In *Proc. of the ICS'09*. Yorktown Heights, NY.
[18] KR Krish, Ali Anwar, and Ali R Butt. 2014. hats: A heterogeneity-aware tiered storage for hadoop. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 502–511.
[19] Simone Meyer, Oliver Wang, Henning Zimmer, Max Grosse, and Alexander Sorkine-Hornung. 2015. Phase-based frame interpolation for video. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1410–1418.
[20] Simon Niklaus and Feng Liu. 2018. Context-aware synthesis for video frame interpolation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1701–1710.
[21] D. Patterson et al. 2008. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of the SIGMOD'88*.
[22] J. Plank. 2008. A new minimum density RAID-6 code with a word size of eight. In *Proc. of the IEEE NCA'08*. Cambridge, MA.
[23] J. Plank. 2008. The RAID-6 Liberation Codes. In *Proc. of the USENIX FAST'08*. San Jose, CA.
[24] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
[25] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2014. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)* 32, 3 (2014), 9.
[26] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 325–336.

[27] Zhirong Shen and Jiwu Shu. 2014. HV Code: An All-Around MDS Code to Improve Efficiency and Reliability of RAID-6 Systems. In *Proc. of the IEEE/IFIP DSN'14*. Atlanta, GA.

[28] Dan Tang, Xiaojing Wang, Sheng Cao, and Zheng Chen. 2008. A New Class of Highly Fault Tolerant Erasure Code for the Disk Array. In *Workshop on Power Electronics and Intelligent Transportation System, PEITS'08*. IEEE, 578–581.

[29] C. Tau and T. Wang. 2003. Efficient parity placement schemes for tolerating triple disk failures in RAID architectures. In *Proc. of the AINA'03*. Xi'an, China.

[30] Aniruddha N Udipi, Naveen Muralimanohar, Rajeev Balsubramonian, Al Davis, and Norman P Jouppi. 2012. LOT-ECC: localized and tiered reliability mechanisms for commodity memory systems. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 285–296.

[31] Joost van Amersfoort, Wenzhe Shi, Alejandro Acosta, Francisco Massa, Johannes Totz, Zehan Wang, and Jose Caballero. 2017. Frame interpolation with multi-scale deep loss functions and generative adversarial networks. *arXiv preprint arXiv:1711.06045* (2017).

[32] S. Wan et al. [n. d.]. Code-M: A Non-MDS Erasure Code Scheme to Support Fast Recovery from up to Two-Disk Failures in Storage Systems. In *Proc. of the DSN'10*.

[33] Hui Wang and Peter Varman. 2014. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*. 229–242.

[34] Y. Wang, G. Li, and X. Zhong. 2012. Triple-Star: A Coding Scheme with Optimal Encoding Complexity for Tolerating Triple Disk Failures in RAID. *International Journal of Innovative Computing, Information and Control* 8, 3 (2012), 1731–1472.

[35] Chentao Wu, Shenggang Wan, Xubin He, Qiang Cao, and Changsheng Xie. 2011. H-Code: A hybrid MDS array code to optimize partial stripe writes in RAID-6. 782–793. https://doi.org/10.1109/IPDPS.2011.78

[36] Lihao Xu, Vasken Bohossian, Jehoshua Bruck, and David Wagner. 1999. Low-Density MDS Codes and Factors of Complete Graphs. *IEEE Transactions on Information Theory* 45, 6 (1999), 1817–1826.

[37] Lihao Xu and Jehoshua Bruck. 1999. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory* 45, 1 (1999), 272–276.

[38] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. 2010. Automated lookahead data migration in SSD-enabled multi-tiered storage systems. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–6.

[39] Yongzhe Zhang, Chentao Wu, Jie Li, and Minyi Guo. 2015. Tip-code: A three independent parity code to tolerate triple disk failures with optimal update complextiy. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 136–147.

[40] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[41] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.