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 60%, saves the storage cost by up to 10%, increase the recovery speed by up to 1.5X when single disk fails, and can reconstruct the whole video data via fuzzification when triple disks fail. q

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, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For typical cloud storage systems such as Windows Azure [CITE] and Amazon AWS [CITE], erasure coding is a popular technique to provide both high reliability and low monetary cost [CITE EC], where triple disk failure fault tolerant arrays (3DFTs) are widely used. Typical erasure codes can be divided into two categories, RS-based Codes [CITE] and XOR-based codes [CITE]. RS-based codes [CITE] are encoded according the Galois Field Computation in Reed Solomon Code [CITE], which allow flexible configuration and have a little higher computation cost. XOR-based codes [CITE] 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 [CITE APPROXIMATE STORAGE], thus the data reliability cannot be guaranteed. Another solution is using disk arrays like RAID-5 or RAID-6 [CITE RAID], but the capability of fault tolerance would be sacrificed. Data compression[1] is also a common strategy for reducing storage overhead [12] [13] [2]. 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, 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.

The rest of the paper is organized as follows. In Section 2, we introduce related work and our motivation. In Section 3, the design

Conference'17, July 2017, Washington, DC, USA

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.

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
LP	local parity nodes
GP	global parity nodes
ID	important data
UD	unimportant data

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 [CITE][CITE].

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[8] based on recent advances in spatial transformers and composite perceptual losses as well as a context-aware Synthesis approach[5] 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 [CITE], RDP code [CITE], Blaum-Roth code[CITE], X-code [CITE], Liberation code [CITE], Liber8tion code [CITE], Cyclic [CITE] code, B-Code [CITE], Code-M [CITE], H-code [CITE], P-code [CITE] and HVcode [CITE], etc.

In Triple Disks Failure Tolerant Arrays (3DFTs), typical MDS codes include Reed-Solomon codes [CITE], Cauchy-RS codes [CITE], STAR code [CITE], Triple-Star code [CITE], Triple-Parity code [CITE], HDD1 code [CITE], RSL-code [CITE], RL-code [CITE], and so on. Typical non-MDS codes contain WEAVER codes [CITE], HoVer codes [CITE], T-code [CITE], HDD2 code [CITE], Pyramid codes [CITE], Local Reconstruction Codes [CITE], Locally Repairable Codes [CITE], etc.

Several classic erasure codes are illustrated in detail as below,

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. [6] 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. ("Appr." is short for "Approximate")

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[4] [9] [10] [7] 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 1 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 1, in *Structure 1*, important data occupies 1 block in each node, and in *Structure 2*, it occupies 1 data arrays. We specify that in *Structure 1*, the ratio of important data to each node is $1/h$, thereby ensuring that the number of important data in both *Structure 1* and *Structure 2* just fills up k nodes.

Since *Structure 1* distributes important data across each node, it guarantees a more balanced load. *Structure 2* concentrates important data into an array, and provides greater reliability for important data. A detailed analysis of this will be presented in 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 4 parameters k, r, g and h. 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. 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 1, we consider the input erasure code as RS(4,3). It is divided into two local parities and one global parities with the *Structure 2* and it generate Approximate RS Code (4,2,1,3,*Structure 2*). 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.

3.3.1 Approximate STAR Code. Figure 2 shows the construction of Approximate STAR Code $(5, 2, 1, 3)$ in *Structure 2*

based on EVENODD [1] and STAR [3]. EVENODD is a tipical RAID6 erasure code scheme, while STAR is a 3DFTs extension scheme of EVENODD. In this case, important data blocks are distributed in array 0, while unimportant data blocks occupy the rest arrays. $LP_{i,0}$ to $LP_{i,3}$ consist the horizontal parity nodes and $LP_{i,4}$

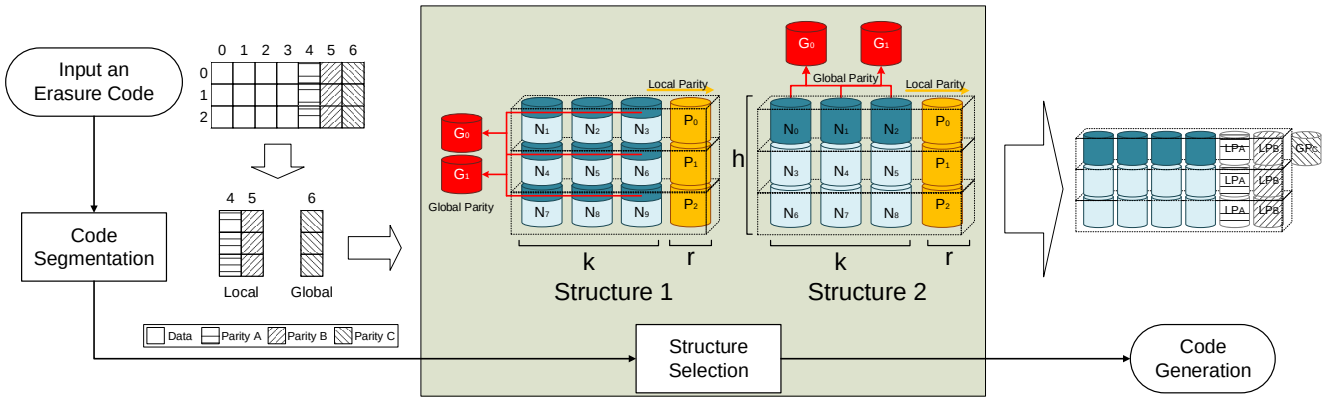


Figure 1: The framework of Approximate Code

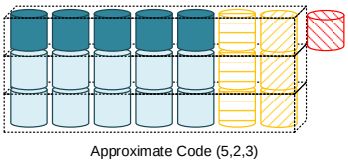


Figure 2: The XOR-based Approximate Code $(5, 2, 4)$, constructed based on EVENODD and STAR code. *LP* represent the local parity and *GP* represent the global parity.

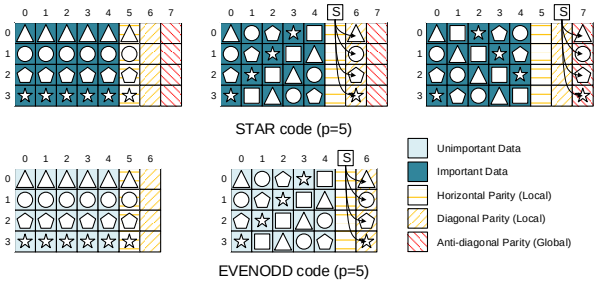


Figure 3: EVENODD only consist horizontal and diagonaol parity, while STAR add the anti-diagonal parity node based on EVENODD and can tolerate 3 device failures.

to $LP_{i,7}$ consist diagonal parity nodes for array i. So the form of every array fit the construction of EVENODD. Because the STAR code is constructed just by adding a reverse parity chain to the EVENODD code. The *GP* is the anti-diagonal parity node and guarantee important data nodes form the construction method of STAR, which ensures the 3DFTs.

3.3.2 Approximate TIP Code. Figure 4 is a tipical construction of Approximate Code $(5, 1, 6)$ in *Structure 1*, based on RAID5 and TIP-code [11]. RAID5 tolerate one node failure while TIP-code is an XOR-based 3DFTs code. In each array, local parity blocks provide

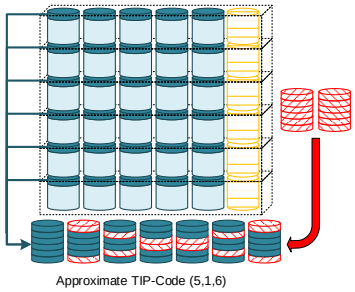


Figure 4: The XOR-based Approximate Code $(5, 1, 6$, *Structure 1*$)$, constructed based on RAID5 and TIP-code.

horizontal XOR parity, and between arrays, global parity provide diagonal parity and anti-diagonal parity.

The local (horizontal) parity elements are calculated by the following encoding equations.

$$LP_{i,0} = \bigoplus_{j=0}^{k-1} ID_{i,j} (0 \leqslant i < h) \tag{1}$$

$$LP_{i,p} (0 < p < s) = \bigoplus_{j=k(p-1)}^{kp-1} UD_{i,j} (0 \leqslant i < h) \tag{2}$$

When generating global parity blocks, the important data block, the local parity block, and the global parity block should be arranged as the encoding form of TIP-code. Figure 5 shows the coding method of diagonal and anti-diagonal parity blocks. Since the encoding process of important data blocks can be seen as RAID5 plus TIP-code, it is obvious that they achieve 3DFTs.

3.4 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.

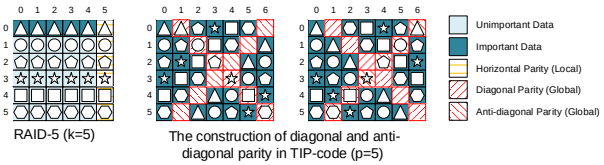


Figure 5: Encoding pf TIP-code

Table 3: Comparison of storage overhead and fault tolerance between Ap-Code (short for Approximate Code), RS and several XOR-based codes.

EC	Storage Overhead	Fault Tolerance	Single Node Update Cost
RS(k,q)	$(k+q)/k$	q	q
EVENODD$(k,2)$	$(k+2)/k$	2	2
TIP-code/STAR$(k,3)$	$(k+r)/k$	3	3
Ap-Code (k,r,h)	$\frac{(k+r)h+3-r}{k\times h}$	r to 3	$r+\frac{3-r}{h}$

- Low Storage Overhead: Approximate Code reduces storage overhead by approximating storage strategies. This property is more pronounced for data with a smaller proportion of important data.
- Low Update Overhead: When one node is updated, Approximate Code only needs to write $r \times h$ local parity nodes and only $3-r$ global parity nodes, while tipical 3DFTs EC methods should write $3 \times h$ parity nodes.
- 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.

3.5 Implementation

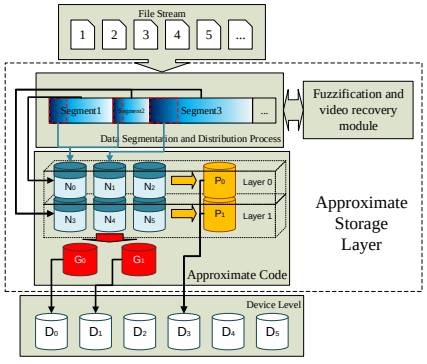


Figure 6: Overview of Approximate Code implementation

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

3.5.1 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).

The data distribution module distributes important data and unimportant data in different blocks and records the location. This location information is also defined as important data. 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.

3.5.2 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.

Approximate Code module is also responsible for allocating blocks of data to ensure that blocks of each array are distributed to different nodes, while ensuring that global check blocks and data blocks are not on one node.

3.5.3 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 in HDFS.

4.1 Evaluation Methodology

We choose the RS code to compare with several XOR-based code, including TIP-code, EVENODD, STAR, RAID5 and RAID6. Since 3DFTs are typical configurations, we set the number of check nodes for the RS to 3. 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 3.4 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.

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

4.1.2 Metrics and Methods for Experiments: We use **Encoding Time** and **Recovery Time** as the metrics in our experiments.

Our experiment includes the calculation of these two metrics and can mainly be divided into several cases. We measure the encoding time by calculating the time to generate all parity blocks, and we measure the recovery time by calculating the time to fully or approximately recover the failure nodes. Both of them should be divided into 3 situation: single, double and triple node, so we discuss them separately. Also, the encoding and recovery time consist of computation time, I/O time and transimission time in distributed file system, which should be considered separately.

The environment of our experiments are shown in Table 4. We set a Hadoop system with one NameNode and 4 DataNodes. text

4.2 Metrics and Methods of Experiment

The metric of our experiments is time. The independent variable in this experiment is the number of nodes. We change the number of data nodes with different methods to compare the operation time. The experiments are divided into two parts, as one part is for measuring the encoding time of different methods, the other part is for timing the decoding time of different methods. The unit of the time is micro-second.

There are mainly two ways to compare different methods. One is for comparing the operational time of AP code with different parameters and that of one different method as we augment the number of data nodes. The other is for a general comparison with five data nodes for the encoding time, the decoding time in one disk, the decoding time in two disks and the decoding time in three disks for every method with its different parameters.

The environment of our experiment is listed below.

4.3 Numerical Results of Mathematical Analysis

4.3.1 Encoding Time Analysis. From the Figure xx a), we could find out that the AP code encodes largely faster than RS (k, 3). The rate of optimisation between AP(k, 1, 6) and RS (k, 3) is from 976.4% (3 nodes case) to 849.2% (13 nodes case), as the encoding time of AP codes multiplies about 4.7 times and that of RS code multiplies about 4.2 times. The Figure xx b) and c) illustrate that the effet of Star code and Tip code is similar, and the difference between them and AP code is not very large. The rate of optimisation between AP (k, 1, 6) and Star (k, 3) is from 125.6% (3 nodes case) to 127.3% (17 nodes case). According to Figure xx d), we could see that AP code largely reduces the encoding time, comparing to LRC method. The rate of optimisation between AP code (k, 1, 4) and LRC (4k, 2, 4) is from 596.0% (3 nodes case) to 615.8% (13 nodes case); and the rate of optimisation between AP code (k, 1, 6) and LRC (6k, 2, 6) is from 684.0% to 443.1%. Considering the optimisation rate, AP code (k, 1, 4) is the best among the four.

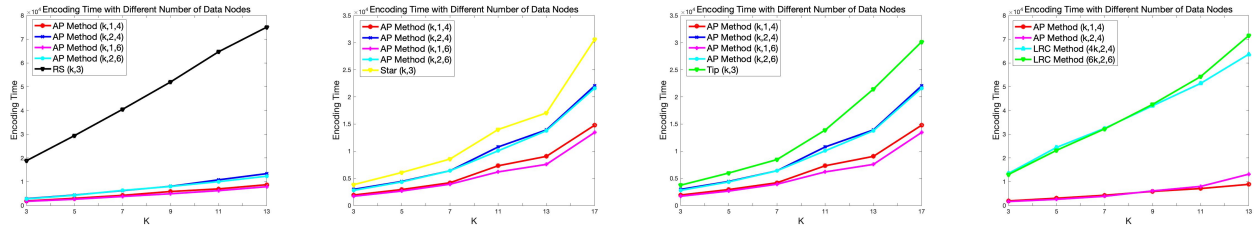
According to Figure xx, we see that RS (5, 3) has the longest encoding time among these methods, and the encoding time of LRC method is not short either. The methods whose encoding time is approaching to the AP method are Star and Tip. Among the AP method, AP (5, 1, 6) has the shortest encoding time.

4.3.2 Decoding Time in One Disk Analysis. From the Figure xx a), we could find out that the AP code decodes faster than RS (k, 3). The rate of optimisation between AP method (k, 1, 6) and RS (k, 3) is from 395.2% (3 nodesâĂŹ case) to 353.0% (13 nodesâĂŹ case). The Figure xx b) and c) illustrate that the effet of Star code and Tip code is similar, and there is nearly no difference among these three methods. The rate of optimisation between AP method (k, 2, 6) and Star (k, 3) is from 3.867% (3 nodesâĂŹ case) to 2.971% (17 nodesâĂŹ case). According to Figure xx d), we could see that LRC (4k, 2, 4) decodes a bit faster than AP code, but LRC (6k, 2, 6) decodes slower than AP code. LRC (4k, 2, 4) decodes faster from 5.941% (3 nodesâĂŹ case) to 9.542% (13 nodesâĂŹ case) than AP (1, 4); but AP (1, 6) decodes faster from 0.476% (3 nodesâĂŹ case) to 6.196% (13 nodesâĂŹ case) than LRC (6k, 2, 6).

According to Figure xx, we see that RS (5, 3) has the longest decoding time among these methods, while other methods decoding time in one disk have no significant difference.

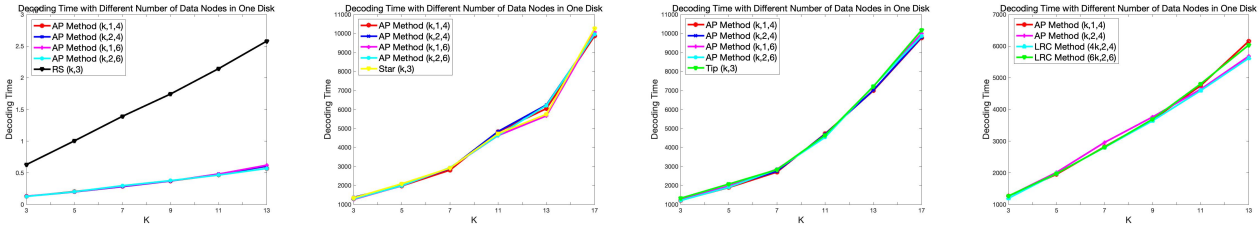
4.3.3 Decoding Time in Two Disks Analysis. From the Figure xx a), we could find out that the AP code decodes remarkably faster than RS (k, 3) in two disks. The rate of optimisation between AP method (k, 1, 6) and RS (k, 3) is from 2782.1% (3 nodesâĂŹ case) to 2593.9% (13 nodesâĂŹ case). The Figure xx b) and c) illustrate that the effet of Star code and Tip code for decoding in two disks is similar, and the difference between them and AP (k, 2, 4 or 6) is not very large. However, the difference is large when it comes to AP (k, 1, 6) The rate of optimisation between AP (k, 1, 6) and Star (k, 3) is from 520.8% (3 nodesâĂŹ case) to 508.6% (17 nodesâĂŹ case). According to Figure xx d), we could see that AP code largely reduces the decoding time in two disks in comparison to LRC method. The rate of optimisation between AP code (k, 1, 4) and LRC (4k, 2, 4) is from 1971.5% (3 nodesâĂŹ case) to 1265.5% (13 nodesâĂŹ case); and the rate of optimisation between AP code (k, 1, 6) and LRC (6k, 2, 6) is from 2941.3% to 1970.7%. Considering the optimisation rate, AP code (k, 1, 6) is the best among the four.

According to Figure xx, we see that RS (5, 3) has the longest decoding time among these methods, the LRC method is not suitable for decoding in two disks. AP code, here, largely reduces the decoding time in two disks, while the AP (5, 1, 6) keeps the shortest decoding time.



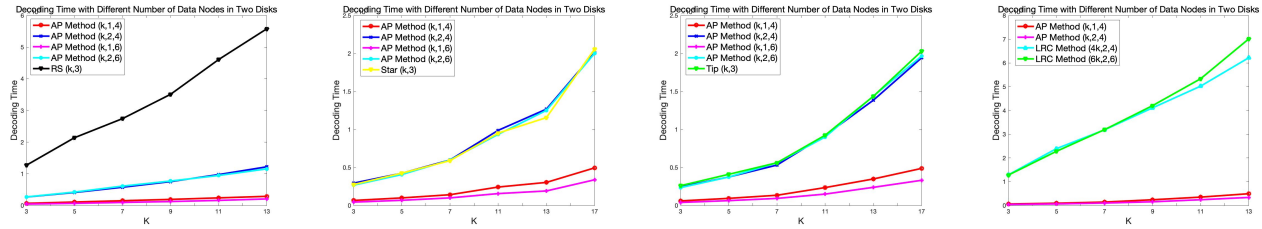
(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 7: Encoding Time Comparison between AP method and other method



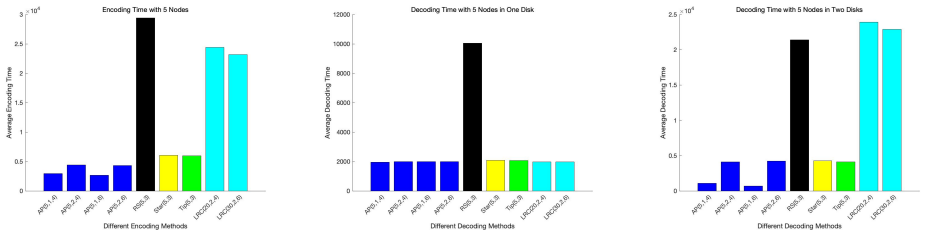
(a) AP and RS (b) AP and STAR (c) AP and TIP (d) AP and LRC

Figure 8: 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 9: Decoding Time Comparison in Two Disk between AP method and other method



(a) Encoding Time (b) Decoding Time in One Node Failure (c) Decoding Time in Two Node Failure

Figure 10: Comparison of several matrics between different methods when number of data nodes are 5

- Storage efficiency. . .
- Fault tolerance. . .
- Write overhead. . .

4.4 Encoding Performance

We use Approximate Code (4,2,4) and (5,1,6)...

4.5 Recovery Time

We use Approximate Code (4,2,4) and (5,1,6)...

4.6 Analysis

5 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

[1] 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.

[2] Peter Deutsch. 1996. *DEFLATE compressed data format specification version 1.3*. Technical Report.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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).

[9] 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.

[10] 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.

[11] 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.

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

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