STORE: Data Recovery with Approximate Minimum Network Bandwidth and Disk I/O in Distributed Storage Systems

Tai Zhou, Hui Li*, Bing Zhu, Yumeng Zhang, Hanxu Hou, Jun Chen
Shenzhen Eng. Lab. of Converged Networks Technology
Shenzhen Key Lab. of Cloud Computing Tech. and App.
Shenzhen Graduate School, Peking University, Shenzhen, 518055, China
Email: lih64@pkusz.edu.cn

Abstract—**Recently, traditional erasure codes such as Reed-Solomon (RS) codes have been increasingly deployed in many distributed storage systems to reduce the large storage overhead incurred by the widely adopted replication scheme. However, these codes require significantly high resources with respect to network bandwidth and disk I/O during recovery of missing or unavailable data. It is referred as the recovery problem.**

In this paper, we dedicate to integrating exact minimum bandwidth regenerating codes into practical systems to solve the recovery problem. We design an implementation friendly storage code with the recently proposed BASIC framework and ZigZag decodable code for saving recovery bandwidth and disk I/O. We build a system called STORE based on this code and evaluate our prototype atop a HDFS cluster testbed with 21 nodes. As shown in this paper, the recovery bandwidth achieves minimum approximately during recovery of both *data block* and *parity block* with STORE. Another attractive result is that the recovery disk I/O also achieves minimum approximately during recovery of *data block*. Due to the reduction of recovery bandwidth and disk I/O, the degraded read throughput is boosted notably.

I. Introduction

Component failures in distributed storage systems running on commodity hardware are the norm rather than the exception [1]. Therefore, distributed storage systems, such as the Google File System [1] and the Hadoop File System [2], adopt triple replication as the default storage policy to ensure data reliability and availability. Tripe replication however incurs a very large storage overhead of 200%. To reduce the storage overhead, traditional erasure codes such as Reed-Solomon codes [3] have been increasingly deployed in many distributed storage systems (e.g., [4],[5],[6]) recently.

Although erasure codes provide optimal storage efficiency, the *recovery bandwidth* (i.e., the amount of data transferred over network during data recovery) and the *recovery disk I/O* (i.e., the amount of data read from local disks during data recovery) of such erasure-coded storage systems are significantly high. Rashmi *et al.* [7] have preformed extensive measurements on Facebook's data-warehouse cluster in production, which consists of multiple thousands of nodes, and which stores multiple Petabytes of RS-encoded data. They observe that the reconstruction operations for RS-encoded data consume a large amount of **disk and cross-rack bandwidth**: a median of more than 180 Terabytes of data is transferred through the top-of-rack switches every day for this purpose. The recovery problem has already been the primary impediment towards a wider deployment of erasure codes in data centers [7].

Recently, the recovery problem has been extensively studied both in theory and practice. Many works (e.g., [8],[9],[10]) discuss how to reduce disk I/O during data recovery in erasure-coded systems. Khan *et al.* [8] conclude that finding the optimal recovery solution for arbitrary erasure codes is NP-hard.

With the purpose of reducing recovery bandwidth, large amount of study (e.g., [7],[11],[12],[13],[14],[15],[16],[17], [18],[19],[20],[21]) focuses on constructing novel codes for distributed storage systems. Xorbas [16], which adopts locally repairable codes (LRC), reduces the recovery bandwidth and the disk I/O during data block recovery to half of those in HDFS RAID [5] with 17% more storage space. While not increasing the storage cost, Hitchhiker [22] reconstructs the missing or unavailable data block with 35% less bandwidth and disk I/O.

Among all these novel codes, regenerating codes [23] achieve the best trade-off between storage overhead and recovery bandwidth. Jiekak *et al.* [24] provide an analysis of regenerating codes for practitioners to grasp the various trade-offs. To our knowledge, previous works on regenerating codes from the system perspective (e.g., [25],[26]) mainly focus on exploring how to deploy minimum storage regenerating codes in distributed storage systems. In this paper, we dedicate to integrating exact minimum bandwidth regenerating codes into practical systems for solving the recovery problem. Although there exist many construction of exact minimum bandwidth regenerating codes such as [12],[15], we desgin our code which is implementation friendly while maintains the main properties of exact minimum bandwidth regenerating codes.

Our Contribution: We design a implementation friendly exact minimum bandwidth regenerating code called $\underline{B}$inary $\underline{M}$inimum $\underline{B}$andwidth $\underline{R}$egenerating code (BMBR) inspired by the recently proposed BASIC framework [12] and ZigZag decodable code [27]. Analysis shows that the recovery bandwidth, during recovery of both *data block* and *parity block* with the BMBR code, approaches minimum approximately. Another attractive result is that the recovery disk I/O, during recovery of *data block* with the BMBR code, also achieves minimum approximately. We build a complete system called STORE based on the BMBR code, which recovers missing data with byte-wise $\underline{s}$hif$\underline{t}$ and exclusive $\underline{o}$$\underline{r}$ operations $\underline{e}$xactly. We evaluate our STORE prototype on a HDFS [2] testbed with 21 nodes and the results are fairly consistent with our analysis. Compared to a currently deployed HDFS module

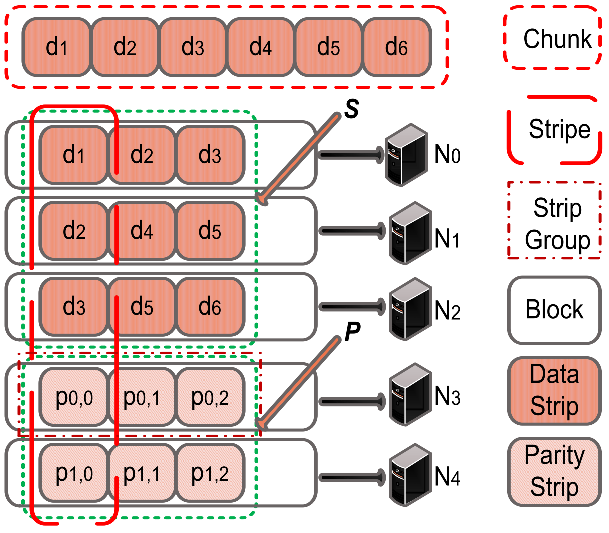


Fig. 1. A distributed storage system which implements $(5, 3)$ BMBR code.

called HDFS-RAID [5], STORE significantly reduces the recovery bandwidth and recovery disk I/O, hence boosts the degraded read throughput.

The remainder of the paper proceeds as follows. Section II illustrates terminologies and formulates our BMBR code. Section III describes the design of STORE and presents our analysis. Section IV presents prototype evaluation. Section V concludes this paper.

II. Binary Minimum Bandwidth Regenerating Code

We design an implementation friendly storage code called Binary Minimum Bandwidth Regenerating (BMBR) code inspired by the recently proposed BASIC framework and ZigZag decodable code. The BASIC framework is a kind of the matrix-production construction framework for design regenerating codes, which is able to achieve two specific points (i.e., minimum-bandwidth and minimum-storage regenerating points) on the storage and repair bandwidth trade-off curve, using only binary addition and shift operations in the coding and repair processes. The ZigZag decodable code is a new vector code that allows fast encoding and decoding, which operate over GF(2). It allows the source blocks to be recovered by using a fast algorithm called ZigZag decoding [27].

BMBR code is attractive for practical storage systems because of its two properties. One is termed (n, k) property [13], i.e., BMBR code stores information in n storage nodes and is able to tolerate up to $(n-k)$ failures without data loss. The other one is the so called efficient exact repair [14], i.e., BMBR code exactly regenerates the missing data with shift and exclusive or operations. In this section, we start by a simple example to define some terminologies and illustrate the idea behind BMBR code.

A. Example

Consider a distributed storage system consists of a collection of *nodes*, each stores a number of *blocks*. Five nodes are randomly picked out and depicted in Fig. 1, labeled by $N_0, N_1, \cdots, N_4$. A *block* is the basic unit of read/write operations in distributed storage systems, such as HDFS [2]. It is termed a *data block* if it holds *data strips*, or a *parity block* if it holds *parity strips*. The original file is divided into *chunks*, each of which is further divided into *data strips*. As shown in Fig.1, the original data chunk is split into 6 strips, labeled by d_1, d_2, $\cdots$, d_6.

Firstly, rearrange these 6 data strips into the upper-triangular part of a symmetric 3×3 matrix S in a row-major order. The lower-triangular part is filled by reflection along the diagonal. Secondly, encode all data strips in a column vector of S to generate the corresponding *parity strips* with the encoding algorithm of a vector code such as the ZigZag decodable code. All strips involved in the encoding process constitute a *stripe*. For example, encode $\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix}^{\mathrm{T}}$ to generate $\begin{bmatrix} p_{0,0} & p_{1,0} \end{bmatrix}^{\mathrm{T}}$ and $d_1, d_2, d_3, p_{0,0}, p_{1,0}$ constitute a stripe. All parity strips constitute the parity matrix P as show in Fig. 1. A *data-strip group* contains all data strips in a row vector of S, while a *parity-strip group* constains all parity strips in a row vector of P.

It is easy to check that the $(5, 3)$ BMBR code has the $(5, 3)$ property and exact repair property. For example, assuming that the strip groups $\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix}$ and $\begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} \end{bmatrix}$ is corrupt or unavailable due to the departure of node N_0 and N_3, a recovery daemon downloads d_2, d_3, $p_{1,0}$ from N_1, N_2, N_4, then performs the ZigZag decoding algorithm to recover d_1. At this point, d_1, d_2 and d_3 are all regenerated and identical to the missing ones. Now that we have all the data-strip groups, the $p_{0,0}, p_{0,1}, p_{0,2}$ can be reconstructed with the encoding algorithm of the ZigZag decodable code.

B. Code Construction

Given an original data chunk C of size L, we cut it into $B = \frac{k(k+1)}{2}$ data strips each of size $L^{'} = \frac{L}{B}$, labeled by d_1, d_2, $\cdots$, d_B. We assume that the size of C exactly equals $B \times L^{'}$ to simplify the presentation; larger files are separated into chunks and each chunk is encoded separately.

BMBR code achieves the efficient exact repair and (n, k) property simultaneously by separating the two problems. Hence, the encoding process includes the following two steps:

Step 1: place these B data strips into a symmetric $k \times k$ matrix S. Entries in the upper-triangular part are filled with data strips $d_i\,(i = 1, 2, \cdots, B)$ in a row-major order, while those in the lower-triangular part are filled by reflection along the diagonal. Let $S_{i,j}$ be an data strip positioned in line i and column j, where $i = 0, 1, \cdots, k-1$ and $j = 0, 1, \cdots, k-1$.

Step 2: encode k data strips in each column vector of S, i.e., S_j $(j = 0, 1, \cdots, k-1)$ independently with the vector code named ZigZag decodable code to generate corresponding parity vector P_j $(j = 0, 1, \cdots, k-1)$, each contains $(n-k)$ parity strips. All P_js constitute an $(n-k) \times k$ matrix P.

Let G be an $(n-k) \times k$ matrix, of which all entries are positive integers while entries in each row and each column are in increasing order. Such a matrix G ensures the decodability of the ZigZag decodable code. The ZigZag decoding algorithm is detailed in [27]. If $r = G_{n-k-1,k-1}$ (i.e., r is the max element in G and $r \ll L^{'}$ in practical systems), it is easy to formulate step 2 in the matrix-production form as $P = G \star S$. Herein, operator $\star$ is defined by two operations:

(a) *shift* (l, s), which shifts all bytes in data strip s to right l bytes and returns a new slightly larger strip $s^{'}$ size $L^{'} + r$, of which left l bytes and right $(r - l)$ bytes are padded with zero.

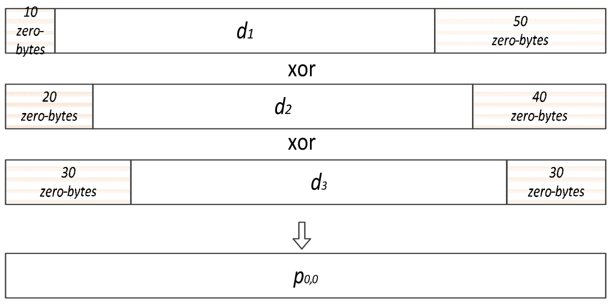


Fig. 2. Generation for a parity strip. $P_{0,0} = G_0 \star S_0^T$

.

(b) $xor\left(s_0^{'}, s_1^{'}, \cdots, s_{k-1}^{'}\right)$, which calculates $s_0^{'} \oplus s_2^{'} \oplus \cdots \oplus s_{k-1}^{'}$ and returns a parity strip size $L^{'} + r$. $\oplus$ represents exclusive or.

Therefore, a parity strip $P_{i,j} = G_i \star S_j^T$ can be generated with the following expression.

$$P_{i,j} = xor_{0 \leq k' < k}\left(shift_{0 \leq i < (n-k), 0 \leq j < k}\left(G_{i,k'}, S_{k',j}\right)\right) \quad (1)$$

.

For example, let

$$G = \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{pmatrix} \quad S = \begin{pmatrix} d_1 & d_2 & d_3 \\ d_2 & d_4 & d_5 \\ d_3 & d_5 & d_6 \end{pmatrix}$$

hence $r = 60$, we illustrate how to generate $p_{0,0}$ in Fig. 2.

C. Some Properties

According to the code construction, the code rate R, defined as the ratio of the total amount of useful stored data to the total amount of data that is stored, of the (n, k) BMBR code satisfies $R = \frac{L}{k \times k \times L' + (n-k) \times k \times \left(L' + r\right)}$.

The ZigZag decodable code, which generate $n-k$ parity strips out of k data strips, stores information in n storage nodes and is able to tolerate up to $(n-k)$ failures without data loss. When the amount of missing strip groups, including data-strip group and parity-strip group, does not exceed $n-k$, it is easy to **exactly** recover each of the missing strip groups by regenerating strips in it one by one. Therefore, our BMBR code possesses the (n, k) property.

III. Design of STORE

We build a complete system called STORE aiming at minimizing the recovery bandwidth and recovery disk I/O during recovery of up to $(n-k)$ failures. In this section, we start by describing the system architecture of STORE.

A. System Architecture

STORE, which is implemented atop of Apache Hadoop [28], consists of several components. Among all the components, the Master, Slave, BlockFixer and BMBRFS are the most relevant here. They all rely on an underlying library which implements BMBR encoding and decoding algortihms, referred to as encoder and decoder.

Similar to the RAID-Node of HDFS-RAID, the Master manages all operations related to the use of BMBR in HDFS. It has a list of files that are to be converted from the replicated state to the BMBR-coded state, and periodically preforms encoding of these files via locally running encoding process

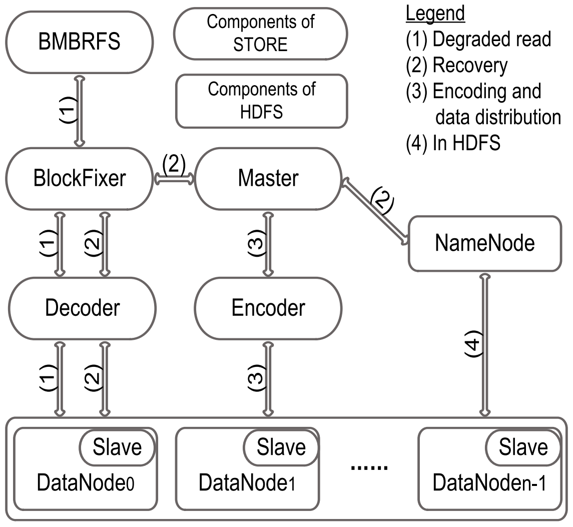


Fig. 3. Relevant components in STORE. The communication flow for encoding, degraded read and recovery operations are shown.

in small scale or, via MapReduce [29] jobs in large scale. The Master also handles recovery operations, i.e., reconstructing missing blocks in order to ensure the reliability of the system. The Master periodically asks NameNode, which manages the metadata of the running HDFS, for corrupt or lost blocks and delegates the recovery task to the BlockFixer. The BlockFixer periodically goes through those missing blocks and reconstructs them with locally recovery process in small scale or, via MapReduce jobs in large scale. The Slaves runs on the storage nodes (DataNodes in HDFS) serving the decoder's read requests when the system is recovering a missing parity block.

The BMBRFS, which wraps HDFS and acts as a general file system, serves the requests for a *degraded read* (i.e., a read request for data which is currently unavailable). The requested unavailable data is reconstructed on the fly by the BlockFixer via a locally running recovery process or a MapReduce job.

All the relevant components of STORE and the communication flows for relevant operations are depicted in Fig. 3.

B. Encoding and Data Distribution

For simplicity, we inspect the encoding process which locally runs on the Master. When the Master decides to encode a file, i.e., converts a file from the replicated state to BMBR-coded state, it launches the encoder. The encoder initially divides the file into chunks with $L = B \times L^{'}$ bytes. Depending on the size of the file, the last chunk may contains fewer than L bytes. Incomplete chunk is considered as "zero-padded" full-chunk as far as the parity calculation is concerned. The encoder iteratively loads a chunk, splits it into B data strips, places these B data strips into the symmetric matrix S logically, then encodes the data strips in every column vector $S_j \ (j = 0, 1, \cdots, k-1)$ with (n, k) ZigZag decodable code to generate corresponding parity vector $P_j \ (j = 0, 1, \cdots, k-1)$ (S_j and P_j comprise a stripe), finally groups all row vectors of S and P into strip groups.

During each iteration, the encoder stores k data-strip groups into k local data files. When the size of a local data file achieves the size of a data block, all local data files are appended to a file that resides in HDFS and then are truncated to length of zero. The encoder processes all parity-strip groups

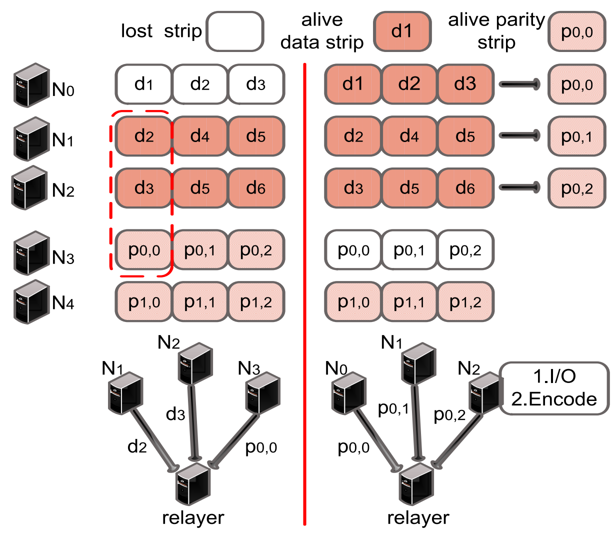


Fig. 4. Left part illustrates recovery of a data block. Right part illustrates recovery of a parity block.

in a similar way. All blocks stored in the underlying HDFS are spread across the cluster according to a configurable block placement policy. The default policy randomly places blocks at storage nodes, avoiding collocating strips of the same stripe.

Obviously, multiple files can be encoded sequentially via locally encoding process running on the Master, while it is easy to be scaled out via MapReduce jobs which contain only the Map tasks.

C. Data Recovery

Data recovery is involved in two use cases. One is termed *periodically fixing*. The Master periodically asks NameNode for corrupt or lost blocks and delegates the recovery task to the BlockFixer. Once there are blocks need to be recoverred, the BlockFixer starts a recovery process locally or launchs a MapReduce job. The other one is called *degraded read*, which happens when a client requests a file with unavailable blocks. When degraded read happens, a BMBRFS instance starts the recovery process. To simplify discussion, we utilize the name *relayer* to represent the BlockFixer or a BMBRFS instance. Before going any further, we first define two performance metrics:

(1) *recovery bandwidth*: the total amount of data being downloaded to the *relayer* during recovery of a block.

(2) *recovery disk I/O*: the total amount of data read from local disks on storage nodes incurred by recovery of a block.

To analyze the recovery bandwidth and recovery disk I/O, we further divide failures into two situations:

Left part of Fig. 4 illustrates the situation where the relayer needs to recover a lost or corrupted *data block*. The relayer recovers all data-strip groups in the lost data block one by one. As shown in the left part of Fig. 3, a data-strip group, i.e.,$[\ d_1 \quad d_2 \quad d_3\ ]$, stored in node N_0 is unavailable. The relayer downloads d_2, d_3, $p_{0,0}$ from N_1, N_2, N_3, then performs the ZigZag decoding algorithm [27] to recover d_1. The regenerated data-strip group is identical to the lost one. It is evident that, in this situation, the recovery bandwidth and recovery disk I/O are both approximately equal to (in fact slightly larger than) the size of a data block.

Right part of Fig. 4 depicts the situation where the relayer needs to recover a lost or corrupted *parity block*. Similar to recovering a data block, all parity-strip groups are recovered one by one. As shown in the right part of Fig. 3, a parity-strip group, i.e., $[\ p_{0,0} \quad p_{0,1} \quad p_{0,2}\ ]$, stored in node N_3 is unavailable. The relayer requests to download $p_{0,0}$, $p_{0,1}$, $p_{0,2}$ from N_0, N_1, N_2 respectively. A Slave daemon running on node N_0 reads a data-strip group, i.e., $[\ d_1 \quad d_2 \quad d_3\ ]$, from its local disk, generates $p_{0,0}$ using $(5, 3)$ ZigZag decodable code [27], then responses the relayer's request. Slaves running on N_1 and N_2 generate $p_{0,1}$ and $p_{0,2}$ respectively in a similar way. The relayer concatenates the downloaded parity strips to exactly recover the lost parity-strip group. Obviously, the recovery bandwidth is equal to the size of a parity block, while the recovery disk I/O is approximately equal to (in fact slightly smaller than) $3\times$ the size of a parity block.

D. Analysis

In this subsection, we study three fault-tolerant storage schemes in the context of practical systems, mainly focusing on three metrics: storage overhead, recovery bandwidth, recovery disk I/O. Recovery bandwidth and recovery disk I/O are defined in section III.C. *Storage overhead* is defined here as the additional data stored to provide fault-tolerant capacity. Further discussion based on an assumption that an block B_{lost} that contains L_{lost} bytes is unavailable.

Storage systems based on 3-way replication, such as GFS and HDFS, achieve both the minimum recovery bandwidth and minimum recovery disk I/O which are both equal to L_{lost}. However, these systems require a large storage overhead of $2\times$.

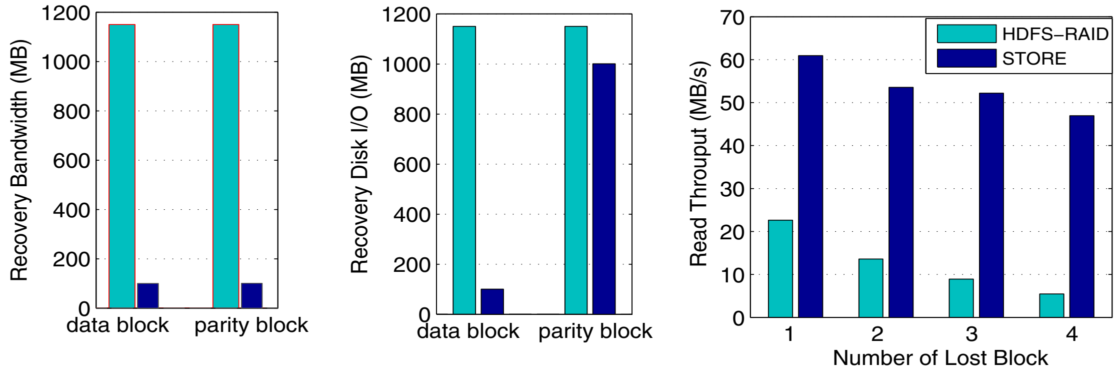
HDFS-RAID [5], which implements (n, k) Reed-Solomon (RS) encoding and decoding over Apache Hadoop [28], offers an alternative for maintaining fault-tolerant storage. Typically, users of HDFS-RAID configure (n, k) to $(14, 10)$, i.e., 4 parity blocks are created for every 10 data blocks. The code rate of RS code is $R_{RS} = \frac{k}{n}$. Such a configuration results in a storage overhead of $0.4\times$. In the currently deployed HDFS-RAID implementation, even when a single block is unavailable, its component named BlockFixer opens streams to all other available blocks of the stripe. The amount of data downloaded varies from $L_{lost} \times 13$ down to $L_{lost} \times 10$ depending on the number of available blocks in the stripe. Both the recovery bandwidth and recovery disk I/O are at least $L_{lost} \times 10$.

In order to provide reliability at the same level with HDFS-RAID, we study $(14, 10)$ BMBR code implemented in our STORE. According to the code rate expression given in subsection II.C, the storage overhead of $(14, 10)$ BMBR code is $1.55\times$. After making a similar analysis in subsection III.C, we conclude that the recovery bandwidth during **recovering a data block or a parity block** is approximately equal to L_{lost}. Similarly, the recovery disk I/O during **recovering a data block** is also approximately equal to L_{lost}.

All the results are summarized into TABLE I.

	3-way replication	HDFS-RAID	STORE
storage overhead	$2\times$	$0.4\times$	$1.55\times$
recovery bandwidth	$1\times$	$10\times$ (at least)	$\approx 1\times$
recovery disk I/O	$1\times$	$10\times$ (at least)	$\approx 1\times$ (data) $\approx 10\times$ (parity)

TABLE I. Comparison of three fault-tolerant storage schemes



(a) Recovery Bandwidth ***(b) Recovery Disk I/O*** ***(c) Degraded Read Throuput***

Fig. 5. Evaluation results. (a) results for recovery bandwidth, (b) results for recovery disk I/O, (c) results for degraded read throughput.

IV. Evaluation

We deployed HDFS-RAID and STORE on a 21-machine test cluster in order to verify BMBR's end to end functionality and evaluate their performance. Each machine is equipped with 4GB RAM, 3.6GHz CPU, 1Gb/s Ethernet card. All the machines are interconnected over a 1Gb/s Ethernet switch, therefore, all of them belong to a single rack. The size of blocks, including data blocks and parity blocks are configured to 100MB in HDFS-RAID [5]. In our STORE, we configure the size of data blocks to 100MB and that of parity blocks to 101MB (i.e., $r = \frac{1}{100}L^{'}$) because of parity block is slightly larger than data block. For both the HDFS-RAID and STORE, we configure (n, k) to $(14, 10)$.

We rely on the following metrics to evaluate STORE against HDFS-RAID: recovery bandwidth, recovery disk I/O, degraded read throughput. Recovery bandwidth and recovery disk I/O are defined in subsection III.C. They are measured under the situation where we manually delete a block, ether a data block or a parity block, to simulate failure events. Now that both HDFS-RAID and STORE recover data at the granularity of block, multiple missing blocks caused by disk failures, machine crashes and power failures, can be recovered individually, therefore, the simulation is reasonable for verifying our theoretical analysis. Degraded read throughput is defined as the amount of data being requested divided by the response time. We set the amount of lost blocks with 1, 2, 3, 4 respectively and measure degraded read throughput. All these results are averaged over five runs.

Fig. $5(a)$ depicts the recovery bandwidth during recovering a block. STORE significantly reduces the recovery bandwidth, the averaged gain is about $11\times$ while the smallest gain is about $10\times$ (not depicted in the figure). In fact, the recovery bandwidth of STORE is slightly larger than 100MB which is the minimum bandwidth cost to provide fault-tolerant storage.

Fig. $5(b)$ illustrates the recovery disk I/O during recovering a block. STORE dramatically reduces the recovery disk I/O during the recovery of **a data block**, the averaged gain is about $11\times$ too. According to a similar observation, we conclude that the recovery disk I/O achieves minimum approximately when using STORE to recover a missing data block. Results in Fig. $5(a)$ and Fig. $5(b)$ are fairly consistent with our analysis in subsection III.D.

Fig. $5(c)$ shows the degraded read throughput. Owning to the reduction of recovery bandwidth and recovery disk I/O, STORE boosts the degraded read performance notably.

The storage cost of files in HDFS-RAID is smaller than that in STORE, therefore we normalize the gains with the factor

$$F = \frac{storage\ cost\ in\ STORE}{storage\ cost\ in\ HDFS-RAID} = \frac{2.55}{1.4} \quad (2)$$

, i.e., the normalized gain of average recovery bandwidth is about $6\times$, the normalized gain of average recovery disk I/O during recovery of missing data block is also about $6\times$.

In order to verifying the normalized gains, we conduct another evaluation. The testbed cluster is reconfigured with 1 NameNode, 14 DataNodes and a block placement policy class, which ensures blocks belong to the same stripe are distributed to different machines. Firstly, we formated the underlying HDFS (i.e., all the files stored in HDFS are deleted) and deployed HDFS-RAID on the running HDFS. We created multiple files with a block size of 100MB and a total size of 20GB. After all the files are erasure-coded with HDFS-RAID, we randomly picked one DataNode and deleted all blocks stored in it to simulate multiple missing block caused by disk failure or machine crash. Once the RaidNode in HDFS-RAID obtains the list of missing blocks, it triggers the recovery. During the recovery process, the statistics utility logged the statistics into a file. On the other hand, we timed the recovery process. For comparison, we repeated the above experiment with STORE running on a formatted HDFS. All the statistics are summarized into TABLE II.

	HDFS-RAID	STORE	real gains
missing data	2GB	3.65GB	–
recovery bandwidth	26GB	3.65GB	$\approx 6.8\times$
recovery disk I/O	26GB	11.72GB	$\approx 2.2\times$
recovery time	227s	83s	$\approx 2.73\times$

TABLE II. Statistics and real gains

As shown in TABLE II, the gain of recovery bandwidth is approximately equal to the normalized analysis. The gain of recovery disk I/O is measured under the situation where the picked DataNode stores both data blocks and parity blocks, therefore the real gain is smaller than the normalized gain, which is obtained only taking the missing data blocks into account. Obviously, the reduction of recovery bandwidth and disk I/O translate into the reduction of recovery time, the real gain is about $2.73\times$.

V. Conclusions

In this paper, we present the Binary Minimum Bandwidth Regenerating code using the recently proposed BASIC framework and ZigZag decodable code and build a system called STORE based on BMBR code. We evaluate our prototype atop a HDFS testbed with 21 machines. Additionally, we make normalized gain analysis and verify the analysis with evaluations atop a HDFS testbed with 15 machines. As shown in this paper, the recovery bandwidth achieves minimum approximately during recovery of both *data block* and *parity block* with STORE. Another attractive result is that the recovery disk I/O also achieves minimum approximately during recovery of *data block*. Due to the reduction of recovery bandwidth and disk I/O, the degraded read throughput is boosted notably.

VI. Acknowledgment

This work is supported by the National Basic Research Program of China (973 Program) (No. 2012CB315904), the National Natural Science Foundation of China (No. NSFC61179028), the Natural Science Foundation of Guangdong Province (No. NSFGD S2013020012822) and the Basic Research of Shenzhen (No. SZ JCYJ20130331144502026).

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 29–43.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[3] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial & Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[4] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 6–10.

[5] "HDFS RAID wiki," [EB/OL], http://wiki.apache.org/hadoop/HDFS-RAID.

[6] "Quantcast File System," [EB/OL], http://quantcast.github.io/qfs/.

[7] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: a study on the facebook warehouse cluster," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX, 2013.

[8] O. Khan, R. Burns, J. Plank, and C. Huang, "In search of i/o-optimal recovery from disk failures," in *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*. USENIX Association, 2011, pp. 6–6.

[9] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads," in *Proc. of USENIX FAST*, 2012.

[10] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu, "On the speedup of single-disk failure recovery in xor-coded storage systems: Theory and practice," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–12.

[11] S. El Rouayheb and K. Ramchandran, "Fractional repetition codes for repair in distributed storage systems," in *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*. IEEE, 2010, pp. 1510–1517.

[12] H. Hou, K. W. Shum, M. Chen, and H. Li, "Basic regenerating code: Binary addition and shift for exact repair," in *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 1621–1625.

[13] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple regenerating codes: Network coding for cloud storage," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2801–2805.

[14] K. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, "Explicit construction of optimal exact regenerating codes for distributed storage," in *Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on*. IEEE, 2009, pp. 1243–1249.

[15] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction," *Information Theory, IEEE Transactions on*, vol. 57, no. 8, pp. 5227–5239, 2011.

[16] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proceedings of the 39th international conference on Very Large Data Bases*. VLDB Endowment, 2013, pp. 325–336.

[17] B. Zhu, K. Shum, H. Li, and H. Hou, "General fractional repetition codes for distributed storage systems," *IEEE Communications Letters*, vol. 18, no. 4, pp. 660–663, 2014.

[18] B. Zhu, H. Li, H. Hou, and K. W. Shum, "Replication-based distributed storage systems with variable repetition degrees," in *Communications (NCC), 2014 Twentieth National Conference on*. IEEE, 2014, pp. 1–5.

[19] K. W. Shum, H. Hou, M. Chen, H. Xu, and H. Li, "Regenerating codes over a binary cyclic code," in *Information Theory Proceedings (ISIT), 2014 IEEE International Symposium on*. IEEE, 2014.

[20] H. Hou, K. W. Shum, and H. Li, "Construction of exact-basic codes for distributed storage systems at the msr point," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 33–38.

[21] H. Hou, H. Li, and K. W. Shum, "General self-repairing codes for distributed storage systems," in *Communications (ICC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 4358–4362.

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

[23] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *Information Theory, IEEE Transactions on*, vol. 56, no. 9, pp. 4539–4551, 2010.

[24] S. Jiekak, A.-M. Kermarrec, N. L. Scouarnec, G. Straub, and A. V. Kempen, "Regenerating codes: A system perspective," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*. IEEE, 2012, pp. 436–441.

[25] R. Li, J. Lin, and P. P. Lee, "Core: Augmenting regenerating-coding-based recovery for single and concurrent failures in distributed storage systems," *arXiv preprint arXiv:1302.3344*, 2013.

[26] X. Huang, H. Li, T. Zhou, Y. Zhang, H. Guo, H. Hou, H. Zhang, and K. Lei, "Minimum storage basic codes: A system perspective," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 39–43.

[27] C. W. Sung and X. Gong, "A zigzag-decodable code with the mds property for distributed storage systems," in *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 341–345.

[28] "Hadoop," [EB/OL], http://hadoop.apache.org.

[29] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.