Efficient Storage Scaling for MBR and MSR Codes

XIAOYANG ZHANG1,2, (Member, IEEE), AND YUCHONG HU1,2,3,4, (Member, IEEE)

1Key Laboratory of Information Storage System, Ministry of Education, Wuhan 430074, China
2School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
3Wuhan National Laboratory for Optoelectronics, Wuhan 430074, China
4Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518000, China

Corresponding author: Yuchong Hu (yuchonghu@hust.edu.cn)

This work was supported in part by the Shenzhen Knowledge Innovation Program under Grant JCYJ20170307172447622.

ABSTRACT Due to the explosive growth of storage demands, distributed storage systems need to support storage scaling efficiently. Recent work optimizes scaling in a decentralized manner for Reed-solomon coded storage systems. In this paper, we focus on storage scaling for storage systems with regenerating codes and design two efficient scaling algorithms for minimum bandwidth regenerating (MBR) and minimum storage regenerating (MSR) codes. We integrate these two scaling algorithms into Hadoop Distributed File System (HDFS), and the experiments on Amazon EC2 show that the scaling bandwidth can be reduced up to 75% and 43.8% over the centralized scaling.

INDEX TERMS Storage scaling, erasure codes, regenerating codes, distributed file systems.

I. INTRODUCTION

Many distributed storage systems adopt erasure coding (e.g., Reed-solomon codes [22]) against node (or server) failures with low redundancy. Erasure coding is typically constructed by two parameters \( n \) and \( k \) as an \((n, k)\) code. An \((n, k)\) coded storage system organizes data as fixed-size blocks, and encodes every \( k \) blocks (called data blocks) into \( n \) \( k \) coded blocks (called parity blocks), such that the original \( k \) data blocks can be reconstructed from any \( k \) blocks out of the \( n \) data and parity blocks. Erasure coding can significantly achieve higher reliability than replication at the same storage overhead [27], and has been widely adopted in distributed storage systems [8], [15], [23] and practical cloud storage systems, e.g., Azure [15] and Facebook [17].

These storage systems often need to add new storage nodes to increase both storage space and service bandwidth for accommodating the increasing storage demands. Especially for cloud storage systems, they can scale the storage capacity to balance performance and monetary costs. For example, Amazon Web Services provides an in-memory storage service called ElastiCache [2] that can effectively add in-memory nodes to meet increasing user demands based on an automatical scaling mechanism [3]. Therefore, many researchers have focused on storage scaling [13], [16], [19], [28], [29], [31]–[34], in which a storage system re-locates existing stored data and re-computes erasure-coded data across all existing and newly added nodes, while maintaining the balanced data layout.

Storage scaling for erasure-coded storage inevitably triggers substantial scaling bandwidth (i.e., the amount of data transferred during scaling) [16], [29], [33], as the scaling process needs to migrate data blocks to different nodes (data migration) and additionally transfer blocks to update parity blocks based on the new data block layout (parity update). We notice that the scaling process is inherently different from the repair process which mainly aims to minimize the repair bandwidth (i.e., the amount of transferred data for repairing a lost block). This is because, although both scaling and repair aim to minimize bandwidth, they build on different settings that cause different analyses of bandwidth. For example, the repair process assumes that the total number of nodes stays the same after repair; however, for the scaling process, the total number of nodes varies after scaling. Figure 1 illustrates a repair process and a scaling process under \((4, 2)\) code, and show that the former maintains four nodes while the latter increases from four nodes to five nodes. Note that, the repair process keeps the same parameters \((n, k)\) and the data size of each node, while the scaling process changes parameters \((n, k)\) and the data size of each node, as illustrated in Figure 1. Thus, the scaling process is significantly different from repair process.

Many studies [16], [19], [28], [29], [31], [32], [34] focus on how to perform storage scaling efficiently for traditional erasure codes (i.e., Reed-Solomon codes) to reduce the scaling bandwidth (see Section II in more detail). Some recent
Our contributions include:

- We present novel scaling methods for E-MBR and Butterfly codes, and utilize parity locally update and features of the two codes to reduce the scaling bandwidth (Section 3 and 4).

- We implement our two scaling methods atop Facebook’s HDFS [4] and conduct experiments on Amazon EC2. We show that the scaling bandwidth can be reduced up to 75% and 43.8% compared with traditional scaling methods (Section 5 and 6).

The rest of this paper is organized as follows. Section II surveys the related work of storage scaling schemes. Section III introduces the difference of repair and scaling, and motivates this paper. Section IV and Section V design two effective scaling methods for E-MBR codes and MSR codes respectively. Section VI presents the implementation details of the two scaling methods on HDFS. Section VII shows evaluation results. Finally, we conclude our work in Section VIII.

II. RELATED WORK

In cloud storage, there are many studies focus on two operations, repair (e.g., [15], [17], [18], [21], [24], [26], [30]) and scaling (e.g., [16], [19], [28], [29], [31], [32], [34]), which cost significant bandwidth traffic.

A. REPAIR FRIENDLY CODES

Previous work have mostly been focused on repair friendly codes which can mitigate the repair bandwidth bottleneck in erasure coding based cloud storage. Locally Repairable Codes (LRC) [14] are a kind of repair friendly codes which can effectively reduce repair bandwidth, and LRC have been demonstrated in cloud storage [15], [23], [30]. Unlike LRCs which requires extra storage overhead, a new family of repair friendly codes — regenerating codes (RGCs) [7], can also reduce repair bandwidth without increasing storage overhead.

Since regenerating codes [7] were proposed, there is a large volume of published studies describing how to minimize the repair bandwidth and storage overhead. Rashmi et al. [21] proposed practical constructions for MBR and MSR codes to repair failed nodes efficiently by using a product-matrix framework. Hu et al. [10] presented a functional MSR code which can significantly reduce the repair bandwidth. Pamies-Juarez et al. [18] presented a practical MSR code called Butterfly code which can achieve the optimal repair performance of MSR codes, but it can only tolerate up to 2 node failures. Vajha et al. [26] presented Clay codes which optimize the repair bandwidth and disk I/O, and it can tolerate multiple node failures. Furthermore, some recent works (e.g., [9], [11], [25]) distinguish the repair bandwidth into cross-rack and intra-rack bandwidth, and aiming to minimize the cross-rack bandwidth. However, much of the current literature on regenerating codes pays particular attention to minimizing the repair bandwidth. Our work focuses on minimizing the scaling bandwidth in the storage scaling, and presents efficient storage scaling methods for both MBR and MSR codes.

B. STORAGE SCALING

In recent years, there has been an increasing amount of literature on storage scaling. Many studies address storage scaling in RAID arrays [28], [31], [34], which aims to...
minimize the data migration while keep the same RAID level. Some previous studies address storage scaling in Reed and Solomon [22] coded storage systems [16], [19]. The most closely related work to ours is by Zhang et al. [33], which applied network coding to Reed-solomon coded storage scaling to minimize the scaling bandwidth, but only support limited scaling cases. Hu et al. [13] generalized the scaling cases in [33]. Since the code construction of traditional erasure codes are totally different from regenerating codes, the above scaling approaches can not directly applied into regenerating coded storage systems. Our work focuses on optimal storage scaling for regenerating coded systems and is complementary with previous Reed-solomon coded storage scaling.

III. BACKGROUND

A. REGENERATING CODES

Given an \((n, k)\) code, the repair locality \(d\) is introduced for repairing a lost block via connecting to \(d\) nodes. Let \(X_i\) be the \(i^{th}\) existing node, where \(1 \leq i \leq n\).

1) MBR CODES

This paper focuses on explicit MBR codes where \(d = n - 1\), called E-MBR codes [24], [21]. For \((n, k)\) E-MBR code with \(d = n - 1\) in [24], the \(n\) storage nodes store \((n - 1)n\) blocks, which are composed of \(k(2n - k - 1)/2\) data blocks, \((n - k)(n - k - 1)/2\) parity blocks and their duplicated ones. The code construction can be modeled via a complete graph, which has \(n\) graph vertices which represent all the existing storage nodes and \((n - 1)n/2\) edges which represent all the blocks. Each vertices stores the blocks which are assigned to the \(n - 1\) edges incident on the vertices. Figure 2 illustrates the construction and layout of E-MBR codes with \((n, k) = (4, 2)\) and \(d = 3\). Five data blocks (i.e., \(1, \ldots, 5\)), one parity block \(P\) (generated by XOR-summing of the five data blocks), and their duplicates, are distributed with the complete graph.

2) MSR CODES

This paper also focuses on an explicit MSR codes which tolerate two failures (i.e., \(n - k = 2\)), called Butterfly Codes [18], which has a recursive code construction and only requires XOR operations for encoding/decoding. For \((n, k)\) Butterfly code, its code construction is based on a data block set \(D_k\) and two parity block sets \(P_k\) and \(Q_k\), where \(D_k\) is a set of \(k \times 2^{k-1}\) data blocks, and \(P_k\) and \(Q_k\) are two sets of \(2^{k-1}\) parity blocks. \(D_k\) can be expressed as a matrix which contains four components:

\[
D_k = \begin{bmatrix}
    a_{k-1} & D^1_{k-1} \\
    b_{k-1} & D^2_{k-1}
\end{bmatrix},
\]

where \(D^1_{k-1}\) and \(D^2_{k-1}\) are two \((k - 1) \times 2^{k-2}\) matrices, and \(a_{k-1}\) and \(b_{k-1}\) are column vectors of \(2^{k-2}\) data blocks. \(P_k\) and \(Q_k\) are generated via two functions such that \(P_k = \mathcal{P}(D_k)\) and \(Q_k = \mathcal{Q}(D_k)\):

\[
\mathcal{P}(D_k) = \begin{bmatrix}
    a_{k-1} \oplus D^1_{k-1} \\
    b_{k-1} \oplus D^2_{k-1}
\end{bmatrix},
\]

\[
\mathcal{Q}(D_k) = \begin{bmatrix}
    b_{k-1} \oplus D^1_{k-1} \\
    a_{k-1} \oplus D^1_{k-1} \oplus D^2_{k-1}
\end{bmatrix},
\]

where \(P_k\) represents a matrix where the counter-diagonal elements are one and the rest are zero, such that the left-multiplication of a matrix by \(P_k\) flips the matrix vertically. Figure 3 illustrates Butterfly codes with \((n, k) = (5, 3)\). Note that the layout of the \((5, 3)\) Butterfly code can be composed of \((4,2)\) Butterfly code and the bold data blocks, which implies the recursiveness of Butterfly codes.

B. REGENERATING CODES BASED STORAGE SCALING

An \((n, k, s)\) scaling process is to transform \((n, k)\)-coded blocks stored in \(n\) existing nodes into \((n + s, k + s)\)-coded blocks that will be stored in \(n + s\) nodes, including the \(n\) existing nodes and the \(s\) new nodes. Let \(Y_j\) be the \(j^{th}\) new node after scaling, where \(1 \leq j \leq s\).

1) SCALING PROBLEM

As stated in Section I, the scaling problem aims to minimize the bandwidth for relocating existing data and recomputing new erasure-coded data, which is totally different from the repair. Figure 4 depicts examples of repair and scaling for E-MBR codes, and the repair process keeps the same parameters (\(4, 2\)) and the 3 blocks of each node, while the scaling process changes parameters from \((4, 2)\) to \((5, 3)\) and increases the data blocks of each node from 3 to 4. The examples of repair and scaling for Butterfly codes is similar and omitted.
achieve optimal or near-optimal scaling bandwidth.

and every set of old group codes from \((n, k)\) to \((n', k')\) E-MBR (Feature 1:) All the blocks of each node have duplicates across \(X_2, X_3\) and \(X_4\) in Figure 2), so the blocks of the new node can be transferred from the old nodes, while maintaining the layout of the old nodes. (Feature 2:) The parity blocks always stay in the specific nodes (e.g., the parity blocks stay in \(X_3\) and \(X_4\) after scaling in Figure 2(b)), so the parity blocks can be updated locally during scaling similarly to [33].

**BUTTERFLY** (Feature 1:) The construction process of Butterfly codes is a recursive process, so we can update parity blocks by utilizing the old parity blocks instead of encoding new blocks in \(G'_i\) by \(N'_D\). Clearly, we have \(N'_D = k(2n' - k' - 1)/2\) and \(N'_D = k'(2n' - k' - 1)/2\) (see Section III-A.1).

**A. LOWER BOUND**

Clearly, the scaling bandwidth is lower bounded by the blocks stored of \(s\) new nodes [33]. Based on Section III-A.1, for a new group of \((n', k')\) E-MBR Codes after scaling, each of \(s\) new nodes has \(n' - 1\) blocks. Thus the lower bound of the scaling bandwidth per new group formed for E-MBR codes (denoted by \(\gamma_{mbr}^{opt}\)) is:

\[
\gamma_{mbr}^{opt} = s(n' - 1).
\]

**B. CENTRALIZED E-MBR SCALING**

We first give a centralized scaling approach for E-MBR codes from \((n, k)\) to \((n', k')\) operated by a controller which is commonly used in RAID arrays [28], [31], [34]. The centralized approach also assumes that all the data blocks maintain their orders [32]. To form a new group, the controller first collects \(N'_D\) data blocks from \(n\) old nodes to generate parity blocks, and then sends \(n' - 1\) to each of \(s\) new nodes. Note that we do not consider the data transfer to reorganize the data blocks for old nodes, so the centralized E-MBR scaling needs to transfer at least:

\[
\gamma_{central} = N'_D + s(n' - 1),
\]

where \(\gamma_{central}\) indicates the necessary scaling bandwidth for centralized E-MBR scaling.

Figure 5 illustrates a centralized E-MBR scaling with \((n, k, s) = (4, 2, 1)\). We find that the controller needs to download nine blocks and sends at least four blocks to the new node \(Y_1\).

**C. SCALING APPROACH**

The scaling process of EMBRScale borrows the idea in [33] which leverages computational resources of storage nodes to update parity blocks locally in a decentralized way.

To perform scaling, EMBRScale operates on a collection of \(N'_D\) old groups \((G_1, \ldots, G_{N'_D})\) before scaling, which have \(N_D \cdot N'_D\) data blocks in total. This collection of blocks can be re-organized into \(N_D\) new groups \((G'_1, \ldots, G'_N)\) after scaling. Similar to [28], for the \(N'_D\) old groups, the first \(N_P\) old groups are denoted by \(RR\) (retained region), in which all the blocks are not migrated during scaling; the remaining \(N'_D - N_P\) old groups which have \((N'_D - N_P) \cdot N_D\) data blocks in total are denoted by \(TR\) (transferred region), in which all the data blocks are transferred to the new groups.
To form the $i^{th}$ $(1 \leq i \leq N_D)$ new group $G_i$, EMBRScale operates on the $i^{th}$ old group in RR and $(N_D - N_D')$ data blocks in TR, described by the following two steps.

**STEP 1 (DATA MIGRATION):** EMBRScale sends the data blocks to the new nodes in the scaling process, as shown in Algorithm 1. It first lets each of $n$ nodes transfers its $s$ distinct data blocks of TR to the $s$ new nodes according to the layout of E-MBR codes (line 1-2). So the scaling bandwidth of Case 1 caused by Step 1 is:

$$γ_{mbr}^{case1} = s(n' - 1).$$

Figure 6 shows that EMBRScale with $(n, k, s) = (3, 2, 1)$ has no parity blocks.

**STEP 2 (PARITY UPDATE):** After migration, EMBRScale computes new parity blocks for new groups, as shown in Algorithm 2. EMBRScale updates parity blocks in three different cases, specified as follows. We denote the scaling bandwidth of EMBRScale per new group by $γ_{mbr}^{case1}, γ_{mbr}^{case2}, γ_{mbr}^{case3}$, respectively.

1) CASE 1, $n - k = 1$
Due to $n - k = 1$, there is no parity blocks (e.g., (3,2) E-MBR codes in Figure 6), EMBRScale do not need to update parity blocks (i.e., Step 2 does nothing) (line 1-2). So the scaling bandwidth of Case 1 caused by Step 1 is:

$$γ_{mbr}^{case1} = s(n' - 1).$$

2) CASE 2, $n - k = 2$ AND $N_D' | N_D''$
Due to $n - k = 2$, there are one parity block (e.g., (4,2) E-MBR codes in Figure 7) and its duplicate. Due to $N_D' | N_D''$, the $N'_D$ data blocks of a new group can be divided into several sets of $N_D$ data blocks of an old group. Based on the feature 2 of E-MBR codes (see Section III-B.3), the parity block (and its duplicate) of the new group can be generated only via XOR-summing of all the parity blocks of the old groups locally at the same node (i.e., no scaling bandwidth incurred in Step 2) (line 3-4). So the scaling bandwidth of Case 2 caused by
Step 1 is:

\[ \gamma_{\text{mbr}}^{\text{case2}} = s(n' - 1). \]  

Figure 7 illustrates \textbf{EMBRScale} with \((n, k, s) = (4, 2, 1)\), where the new parity block \(Q_1\) and its duplicate residing at \(X_3\) and \(X_4\) can be locally updated by \(Q_1 = P_1 \oplus P_2 \oplus P_3 \oplus P_4\) at the same node \(X_3\) and \(X_4\), thereby causing no scaling bandwidth for parity updates.

3) CASE 3, \(n - k = 2\) AND \(N_D \nmid N_D'\)

Due to \(N_D \nmid N_D'\), the \(N_D'\) data blocks of a new group can not be exactly divided into several sets of \(N_D\) data blocks of an old group, so the new parity blocks can not be updated locally as Case 2. Thus, to generate the new parity block (which is XOR-summing of the \(N_D'\) data blocks), we can first select a local old parity block (which is XOR-summing of \(N_D\) data blocks) in \(RR\) and \(s\) local data blocks (see Step 1) and then transfer \((N_D' - N_D - s)\) data blocks in \(TR\). In this way, the new parity block can be generated from the local old parity block, \(s\) local data blocks and \((N_D' - N_D - s)\) transferred data blocks. So can the duplicated parity block (line 5-7). Thus, the scaling bandwidth of Case 3 caused by Step 1 and 2 is

\[ \gamma_{\text{mbr}}^{\text{case3}} = s(n' - 1) + 2 \cdot (N_D' - N_D - s) = s(2n' + n - 4). \]  

The illustration for Case 3 is omitted due to limited space.

\textbf{Theorem 1:} \textbf{EMBRScale} achieves the optimal scaling bandwidth in Case 1 and 2.

\textbf{Proof:} In Case 1 and 2, we find that the scaling bandwidth of \textbf{EMBRScale} (see Equation (6), (7)) are equal to the lower bound of the scaling bandwidth for E-MBR codes (see Equation (4)). Thus, the theorem holds. \qed

\textbf{CONCLUSION:} Figure 8 illustrates the flowchart of \textbf{EMBRScale} for the \(i\)th \((1 \leq i \leq N_D)\) new group \(G_i\). In general, step 1 and 2 leverage feature 1 and 2 of E-MBR codes (see Section III-B.3) respectively. In this way, for data migration, we can just transfer data blocks to the new nodes without re-organizing the data blocks among old nodes; for parity update, we can update parity blocks locally. Therefore, \textbf{EMBRScale} achieves the optimal scaling bandwidth in Case 1 and 2 (see Theorem 1), but does not achieve the optimal scaling bandwidth in Case 3 due to the cross-node bandwidth for parity updates (see Equation (4), (8)). We pose the optimization of Case 3 as future work.

\textbf{D. NUMERICAL ANALYSIS}

We compare the scaling bandwidth of \textbf{EMBRScale} with the centralized E-MBR scaling. Based on Equation (4), (5), (6), (7) and (8), the scaling bandwidths are calculated as the total number of blocks transferred during scaling per new group. Figure 9 shows that the scaling bandwidth of \textbf{EMBRScale} can be reduced by up to 75% over the centralized E-MBR scaling when \((n, k, s) = (5, 4, 1)\).

\textbf{V. SCALING FOR BUTTERFLY CODES}

This section presents an \((n, k, s)\) scaling approach of Butterfly codes from \((n, k)\) to \((n', k')\) where \(s = n' - n = k' - k\), called \textbf{ButterflyScale}, which utilizes the features of Butterfly codes to reduce the scaling bandwidth.

Based on the Butterfly code construction in Section III-A.2, we also define every set of the blocks of \(D_k\) and its parity blocks before scaling as an \textit{old group} and every set of the blocks of \(D_{k+s}\) and its parity blocks after scaling as a \textit{new group}. We denote the \(i\)th old group by \(G_i\), and the \(i\)th new group by \(G'_i\), similar to \textbf{EMBRScale}.

\textbf{A. LOWER BOUND}

Since Butterfly codes belong to Maximum Distance Separable (MDS) codes [7], the scaling bandwidth for Butterfly codes (denoted by \(\gamma_{\text{msr}}\)) per new group is lower bounded by

\[ \gamma_{\text{msr}}^{\text{opt}} = s \cdot 2^{k' - 1}, \]  

which is given by [13].
B. CENTRALIZED BUTTERFLY SCALING

Similar to E-MBR scaling, we also give the centralized scaling approach for Butterfly codes from \((n, k)\) to \((n', k')\) via a controller. To form a new group, the controller first downloads \(k' \cdot 2^{k-1}\) data blocks to generate \(2 \cdot 2^{k-1}\) parity blocks, sends \(2^{k-1}\) data blocks to each of \(s\) new nodes for data migration, and sends \(2 \cdot 2^{k-1}\) parity blocks to two nodes for parity update. Note that we do not consider the data transfer to re-organize the data blocks for old nodes, so the centralized Butterfly scaling needs to transfer at least:

\[
y_{msr}^{central} = (k' + 2 + s) \cdot 2^{k-1},
\]

where \(y_{msr}^{central}\) indicates the necessary scaling bandwidth for the centralized Butterfly scaling.

Figure 10 illustrates a centralized Butterfly scaling with \((n, k, s) = (3, 2, 1)\). We find that the controller needs to download 12 blocks (i.e., 1, \ldots, 12) and sends 8 parity blocks to \(X_3\) and \(X_4\), and sends 4 blocks (i.e., 3, 6, 9, 12) to the new node \(Y_1\).

C. SCALING APPROACH

Similar to EMBRScale, ButterflyScale also uses the computational resources of storage nodes to update parity blocks locally in a decentralized way. Due to the complexity of code construction of Butterfly codes, we only focus on the scaling case of \(s = 1\) (i.e., adding a new node), and pose the scaling case of \(s > 1\) as future work.

To perform scaling, ButterflyScale operates on a collection of \(2k + 2\) old groups \((G_1, \ldots, G_{2k+2})\) before scaling, each of which has \(2^{k-1} \cdot k\) data blocks. This collection of blocks can be re-organized into \(k\) new groups \((G'_1, \ldots, G'_k)\) after scaling, each of which has \(2^k\cdot (k + 1)\) data blocks.

To form the \(i^{th}\) \((1 \leq i \leq k)\) new group \(G'_i\), ButterflyScale operates on the data blocks of the groups \(G_{2k+1}^i\) and \(G_{2k+2}^i\) at node \(X_i\), and the two old groups \(G_{2i-1}\) and \(G_{2i}\), described by the following two steps, similar to EMBRScale.

**STEP 1 (DATA MIGRATION):** ButterflyScale sends the data blocks to the new nodes in the scaling process, as shown in Algorithm 3. It lets node \(X_i\) transfer its data blocks which are in the groups \(G_{2k+1}\) and \(G_{2k+2}\) to the new node according to the layout of Butterfly codes. So the data transfer for data migration is \(2 \times 2^{k-1}\) (see Section III-A.2), which satisfies that the new node has \(2^k\) data blocks after scaling.

Figure 11 shows that to form the new group \(G'_1\), ButterflyScale with \((n, k, s) = (4, 2, 1)\) lets node \(X_1\) transfer data blocks 17, 19, 21, 23 in \(G_5\) and \(G_6\) to the new node \(Y_1\) for data migration.

**STEP 2 (PARITY UPDATE):** ButterflyScale does the parity updates from the two old parity block sets \(P_k\) and \(Q_k\) to the two new parity block sets \(P_{k+1}\) and \(Q_{k+1}\), specified as follows. Based on the code construction of Butterfly codes in Section III-A.2, we have

\[
D_{k+1} = \begin{bmatrix} a_k & b_k \\ & & \end{bmatrix}, \quad D^1_{k+1} = \begin{bmatrix} a_{k-1} & b_{k-1} \\ & & \end{bmatrix}.
\]

Based on Equation (2) and (11), we can infer that the parity block set \(P_{k+1}\) is

\[
\mathcal{P}(D_{k+1}) = \begin{bmatrix} a_k \\ b_k \end{bmatrix} \oplus \begin{bmatrix} \mathcal{P}(D^1_{k+1}) \\ \mathcal{P}(D^2_{k+1}) \end{bmatrix}.
\]

By Equation (12), we find that the first new parity block set \(P_{k+1}\) can be generated by XOR-summing of the data blocks of the bold part \((a_k\) and \(b_k\)\) as well as the two old parity block sets \(\mathcal{P}(D^1_{k+1})\) and \(\mathcal{P}(D^2_{k+1})\) of \(P_k\). Thus, to update parity from \(P_k\) to \(P_{k+1}\), ButterflyScale only needs to transfers

\[
2 \times 2^{k-1} \quad \text{(corresponding to } a_k \text{ and } b_k)\]

data blocks to the node where \(P_k\) resides in.

---

**Algorithm 3 Data Migration of ButterflyScale**

1. \(X_i (1 \leq i \leq k)\) sends \(2^{k-1}\) data blocks of the groups \(G_{2k+1}\) to the new node.
2. \(X_i (1 \leq i \leq k)\) sends \(2^{k-1}\) data blocks of the groups \(G_{2k+2}\) to the new node.
Algorithm 4 Parity Update of ButterflyScale

1: **ButterflyScale** sends the $2^k$ data blocks $(a_k, b_k)$ from the new node to $P_k$. 
2: **ButterflyScale** sends $2^{k-1}$ blocks $(P_k b_k)$ from the new node to $Q_k$. 
3: **ButterflyScale** sends $2^{k-1}$ blocks $(P_k(a_k \oplus P(D_k^{1}))$ from $P_{k+1}$ to $Q_k$. 
4: **ButterflyScale** sends $(k - 1) \times 2^{k-1}$ blocks $(P_{k-1}(D_k^{2}))$ and $P_{k-1}(P(D_k^{1}))$ from old data nodes to $Q_k$. 
5: **ButterflyScale** uses the received blocks to update the $P_k$, $Q_k$ to $P_{k+1}$, $Q_{k+1}$.

Based on Equation (3) and (11), we can infer that the parity block set $Q_{k+1}$ is

$$Q(D_{k+1}) = \begin{bmatrix} P_k b_k \\ P_k(a_k \oplus P(D_k^{1})) \oplus P_{k-1}(D_k^{2}) \\ P_{k-1}(P(D_k^{1})) \end{bmatrix} \oplus \begin{bmatrix} Q(D_k^{1}) \\ Q(D_k^{2}) \end{bmatrix}. \quad (14)$$

By Equation (14), we find that the second new parity block set $Q_{k+1}$ can be generated by XOR-summing of the data blocks of the bold part, as well as the two old parity block sets $Q(D_k^{1})$ and $Q(D_k^{2})$ of $Q_k$. Thus, to update the parity block set from $Q_k$ to $Q_{k+1}$, **ButterflyScale** only needs to transfer

$$
\begin{align*}
& \left(2^{k-1} \right) \quad \text{(corresponding to $P_k b_k$)}, \\
& \left(2^{k-1} \right) \quad \text{(corresponding to $P_k(a_k \oplus P(D_k^{1}))$)}, \\
& \left((k - 1) \times 2^{k-1} \right) \quad \text{(corresponding to $P_{k-1}(D_k^{2})$)}, \\
& \left((k - 1) \times 2^{k-2} \right) \quad \text{(corresponding to $P_{k-1}(P(D_k^{1}))$)},
\end{align*}
$$

blocks to the node where $Q_k$ resides in. Note that the coded blocks $a_k \oplus P(D_k^{1})$ can be transferred from $P_{k+1}$ to $Q_{k+1}$. We describe the parity update step in Algorithm 4.

From Step 1 and 2 (Equation (13) and (15)), the scaling bandwidth of **ButterflyScale** (denoted by $\gamma_{msr}^{butterflyscale}$) per new group formed after scaling is:

$$\gamma_{msr}^{butterflyscale} = (k + 5) \cdot 2^{k-1}. \quad (16)$$

Figure 11 illustrates **ButterflyScale** with $(n, k, s) = (4, 2, 1)$, where six old groups $G_1, \ldots, G_6$ are changed into two new groups $G_1', G_2'$. For example, $G_1'$ can be formed by first transferring four data blocks of $G_5$ and $G_6$ at node $X_1$ (i.e., 17, 19, 21, 23) to the new node $Y_1$, then transferring four data blocks (i.e., 17, 19, 21, 23) to node $X_3$ to update $P$, and last transferring six blocks (i.e., 5, 7, 21, 23, (19 + 4 + 3), (17 + 2 + 1)) to $X_4$ to update $Q$.

CONCLUSION: Figure 12 illustrates the flowchart of **ButterflyScale** for the $i^{th}$ group $(1 \leq i \leq k)$ new group $G_i'$. In general, step 2 leverages feature 1 and 2 of Butterfly codes (see Section III-B-3). In this way, **ButterflyScale** can use old local parity blocks to generate new parity blocks in a recursive and decentralized manner, thereby reducing the scaling bandwidth caused by parity update.

D. NUMERICAL ANALYSIS

We compare the scaling bandwidth of **ButterflyScale** with the centralized Butterfly scaling. Based on Equation (9), (10) and (16), the scaling bandwidths are calculated as the total number of blocks transferred during scaling per new group. Figure 13 shows that the scaling bandwidth of **ButterflyScale** can be reduced by up to 43.8% over the centralized E-MBR scaling when $(n, k, s) = (6, 4, 1)$.

VI. IMPLEMENTATION

We implement EMBRScal and **ButterflyScale** on Facebook’s Hadoop Distributed File System (HDFS) [4].

A. FACEBOOK’S HDFS OVERVIEW

Facebook’s HDFS has a single NameNode which managing metadata and multiple DataNodes which storing data, and it integrates HDFS-RAID [5] as a layer atop the original HDFS to support the erasure coding operations. HDFS-RAID
brings a RaidNode to handle the erasure coding operations and manage erasure coded blocks. Take the encoding operation for a stripe as an example, the RaidNode first obtains the data blocks of the stripe from several DataNodes by using the metadata which are received from the NameNode. Then the RaidNode encodes these data blocks to generate parity blocks, and distribute these blocks to DataNodes.

B. INTEGRATION
We now explain how we integrate EMBRScale and ButterflyScale into Facebook’s HDFS. We first implement E-MBR codes and Butterfly codes into HDFS-RAID as the preparation before scaling, and then we implement the scaling operations on these two types of codes. We describe the details of the implementation as follows:

ERASURE CODES: For the computation part of the scaling operation and the encode/decode operation of the E-MBR codes and the Butterfly codes, we implement it in C++ based on the Intel’s ISA-L [6], and this computation part is connected to HDFS through Java Native Interface (JNI). We mainly use the ec_init_tables API of ISA-L to initialize the coding matrix and use the ec_encode_data API of ISA-L to perform the coding operations.

PLACEMENT ALGORITHMS: The original HDFS-RAID only supports random distributions of data and parity blocks, whereas each of the data and parity blocks in the E-MBR codes and the Butterfly codes need to be placed into a specified node. Thus, we add the placement algorithms into the RaidNode to ensure these two codes are implemented correctly.

SCALING PROCESSES: Both of the two scaling processes need to transfer multiple blocks among multiple nodes. So we modify the RaidNode to collect the metadata of the blocks which are participated in the scaling operations from NameNode and then distribute the metadata to the DataNodes. We also modify the DataNodes to execute the specific scaling operations with the received metadata, such as the operation of transfer blocks and the operation of update parity blocks.

VII. EVALUATION
In this section, we present evaluation results for the scaling performances of EMBRScale and ButterflyScale. We compare EMBRScale with the centralized E-MBR scaling (denoted by E-Central), and we also compare ButterflyScale with the centralized Butterfly scaling (denoted by B-Central).

A. CLOUD EXPERIMENTS SETUP
We conduct the experiments on a cluster of 8 machines. Each machine is a m4.4xlarge instance located in the US East (North Virginia) region in Amazon EC2 [1]. We assign a dedicated instance to serve both the HDFS NameNode and the HDFS RaidNode. To prevent the variation of network in the cloud impacts the performance of the experiments, We need to maintain a stable network in the cluster. We first assign a dedicated instance to mimic a network core, such that any traffic between every pair of instances must traverse the dedicated instance. We then control the outgoing bandwidth of the dedicated instance by using the Linux traffic control command tc. Such that, we can maintain the specific bandwidth in the cluster.

We measure the scaling performances for EMBRScale and ButterflyScale, and the scaling performance is measured by the scaling time per 1 GB of data. To obtain a stable scaling performance, we execute the scaling processes on around 1000 data blocks and the corresponding parity blocks.

B. EXPERIMENT 1: IMPACT OF BANDWIDTH
We first evaluate the scaling performances of EMBRScale and ButterflyScale under different bandwidth settings. We fix the block size as 64 MB, and we vary the cluster bandwidth from 200 Mb/s up to 2 Gb/s.

Figure 14 shows the scaling time results of EMBRScale and the centralized E-MBR scaling (E-Central). We measure 3 parameter settings from 3 cases of EMBRScale, and compared them with the cases of E-Central. We find that the empirical results in Figure 14 are consistent with the numerical results in Figure 9, and EMBRScale reduces the scaling time of the E-Central. Take the E-MBR scaling with \((n, k, s) = (3, 2, 1)\) for example, EMBRScale incurs 66.67% less scaling bandwidth than the E-Central as shown in the numerical results, while in the evaluation results, EMBRScale incurs 66.49%, 66.29%, 66.1% and 65.65% less scaling time than E-Central when the cluster bandwidth is 200 Mb/s, 500 Mb/s, 1 Gb/s, and 2 Gb/s, respectively. Note that the ratio of the scaling time between EMBRScale and E-Central is slightly decreased when the bandwidth is higher,
mainly because the HDFS RaidNode collect metadata from the NameNode takes a fixed time, and this takes a higher proportion in the scaling time when the cluster bandwidth is higher.

Figure 15 shows the scaling time results of ButterflyScale and the centralized Butterfly scaling (B-Central). Similar to Figure 14, We also find that the empirical results in Figure 15 are consistent with the numerical results in Figure 13, and the ButterflyScale reduces the scaling time of the B-Central.

C. EXPERIMENT 2: IMPACT OF BLOCK SIZE
We now evaluate the scaling time versus the block size. We fix the cluster bandwidth as 1 Gb/s, and we vary the block size from 1 MB to 64 MB. We depict the evaluation results in Figure 16. We find that the scaling times of E-MBR codes and Butterfly codes are stable across different block sizes, and the EMBRScale and the ButterflyScale reduce the scaling time of the centralized scaling in all cases.

VIII. CONCLUSION AND FUTURE WORK
We present scaling approaches for two regenerating codes: E-MBR with \( n - k = 1 \) and \( n - k = 2 \), and Butterfly codes with \( s = 1 \). We utilize the network coding and the features of these two codes to optimize the scaling processes, and both of the approaches reduce the scaling bandwidth compared to the centralized scaling approaches. In the future, we will focus on the optimization of the scaling processes for E-MBR codes with \( n - k > 2 \), and Butterfly codes with \( s > 1 \).

REFERENCES
[1] Amazon Elastic Compute Cloud (EC2). Accessed: Mar. 11, 2020. [Online]. Available: http://aws.amazon.com/ec2/
[2] Amazon Elasticache. Accessed: Mar. 11, 2020. [Online]. Available: https://docs.aws.amazon.com/elasticache
[3] AWS Autoscaling. Accessed: Mar. 11, 2020. [Online]. Available: https://aws.amazon.com/autoscaling
[4] Facebook’s Hadoop 20. Accessed: Mar. 11, 2020. [Online]. Available: https://github.com/facebookarchive/hadoop-20
[5] HDFS-RAID. Accessed: Mar. 11, 2020. [Online]. Available: http://wiki.apache.org/hadoop/HDFS-RAID
[6] Intel ISA-L. Accessed: Mar. 11, 2020. [Online]. Available: https://01.org/zh/intel%C2%AE-storage-acceleration-library-open-source-version
[7] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” IEEE Trans. Inf. Theory, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
[8] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in Proc. USENIX Symp. Oper. Syst. Design Implement., 2010, pp. 1–14.
[9] H. Hou, P. Lee, K. Shum, and Y. Hu, “Rack-aware regenerating codes for data centers,” IEEE Trans. Info. Theory, vol. 65, no. 8, pp. 4730–4745, Aug. 2019.
[10] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, “Ncloud: Applying network coding for the storage repair in a cloud-of-clouds,” in Proc. USENIX Conf. File Storage Technol., 2012, p. 21.
[11] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng, “Optimal repair layering for erasure-coded data centers: From theory to practice,” in ACM Trans. Storage, vol. 13, no. 4, pp. 1–24, 2017.

[12] Y. Hu, C.-M. Yu, Y. K. Li, P. P. C. Lee, and J. C. S. Lui, “NCFS: On the practicality and extensibility of a Network-Coding-Based distributed file system,” in Proc. Int. Symp. Netw. Coding, Jul. 2011, pp. 1–6.

[13] Y. Hu, X. Zhang, P. P. C. Lee, and P. Zhou, “Generalized optimal storage scaling via network coding,” in Proc. IEEE Int. Symp. Inf. Theory (ISIT), Jun. 2018, pp. 956–960.

[14] C. Huang, M. Chen, and J. Li, “Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems,” ACM Trans. Storage, vol. 9, no. 1, pp. 1–28, Mar. 2013.

[15] C. Huang, H. Simiuci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure coding in windows azure storage,” in Proc. USENIX Annu. Techn. Conf., 2012, pp. 15–26.

[16] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie, “Scale-RS: An efficient scaling scheme for RS-coded storage clusters,” IEEE Trans. Parallel Distrib. Syst., vol. 26, no. 6, pp. 1704–1717, Jun. 2015.

[17] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, “f4: Facebook’s Warm Blob Storage System,” in Proc. USENIX Symp. Oper. Syst. Design Implement., 2014, pp. 383–398.

[18] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyout, E. E. Gad, and Z. Bandic, “Opening the chrysalis: On the real repair performance of MSR codes,” in Proc. USENIX Conf. File Storage Technol., 2016, pp. 81–94.

[19] B. K. Rai, V. Dhoorjati, L. Saini, and A. K. Jha, “On adaptive distributed storage systems,” in Proc. IEEE Int. Symp. Inf. Theory (ISIT), Jun. 2015, pp. 1482–1486.

[20] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, “Explicit construction of optimal exact regenerating codes for distributed storage,” in Proc. 47th Allerton Conf. Commun., Control, Comput. (Allerton), Sep. 2009, pp. 1243–1249.

[21] 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,” IEEE Trans. Inf. Theory, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.

[22] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” J. Soc. Ind. Appl. Math., vol. 8, no. 2, pp. 300–304, Jun. 1960.

[23] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “XORing elephants: Novel erasure codes for big data,” Proc. VLDB Endowment, vol. 6, no. 5, pp. 325–336, Mar. 2013.

[24] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, “Distributed storage codes with Repair-by-Transfer and Non-Achievability of interior points on the storage-Bandwidth tradeoff,” IEEE Trans. Inf. Theory, vol. 58, no. 3, pp. 1837–1852, Mar. 2012.

[25] J. Sohn, B. Choi, S. Yoon, and J. Moon, “Capacity of clustered distributed storage,” IEEE Trans. Inf. Theory, vol. 65, no. 1, pp. 81–107, May 2019.

[26] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Loboa, B. Sasidharan, P. Kumar, A. Burg, M. Ye, and S. Narayananmorthy, “Clay codes: Mouling MDS codes to yield an MSR code,” in Proc. USENIX Conf. File Storage Technol., 2018, pp. 139–154.

[27] H. Weatherspoon and J. D. Kubiatowicz, “Erasure coding vs. Replication: A quantitative comparison,” in Proc. IPTPS, Mar. 2002, pp. 328–337.