Toward Optimal Storage Scaling via Network Coding: From Theory to Practice

Xiaoyang Zhang[1], Yuchong Hu[1], Patrick P. C. Lee[2], and Pan Zhou[1]
[1]Huazhong University of Science and Technology, [2]The Chinese University of Hong Kong

***Abstract*—To adapt to the increasing storage demands and varying storage redundancy requirements, practical distributed storage systems need to support *storage scaling* by relocating currently stored data to different storage nodes. However, the scaling process inevitably transfers substantial data traffic over the network. Thus, minimizing the bandwidth cost of the scaling process is critical in distributed settings. In this paper, we show that optimal storage scaling is achievable in erasure-coded distributed storage based on network coding, by allowing storage nodes to send encoded data during scaling. We formally prove the information-theoretically minimum scaling bandwidth. Based on our theoretical findings, we also build a distributed storage system prototype NCScale, which realizes network-coding-based scaling while preserving the necessary properties for practical deployment. Experiments on Amazon EC2 show that the scaling time can be reduced by up to 50% over the state-of-the-art.**

I. Introduction

Distributed storage systems provide a scalable platform for storing massive data across a collection of storage nodes (or servers). To provide reliability guarantees against node failures, they commonly stripe data redundancy across nodes. Erasure coding is one form of redundancy that significantly achieves higher reliability than replication at the same storage overhead [21], and has been widely adopted in production distributed storage systems [8], [10], [19].

To accommodate the increasing storage demands, system operators often regularly add new nodes to storage systems to increase both storage space and service bandwidth. In this case, storage systems need to re-distribute (erasure-coded) data in existing storage nodes to maintain the balanced data layout across all existing and newly added nodes. Furthermore, system operators need to re-parameterize the right redundancy level for erasure coding to adapt to different trade-offs of storage efficiency, fault tolerance, access performance, and management complexity. For example, storage systems can reduce the repair cost by increasing storage redundancy [5], or dynamically switch between erasure codes of different redundancy levels for different workloads to balance between access performance and fault tolerance [27].

This motivates us to study *storage scaling*, in which a storage system relocates existing stored data to different nodes and recomputes erasure-coded data based on the new data layout. Since the scaling process inevitably triggers substantial data transfers, we pose the following *scaling problem*, in which we aim to minimize the *scaling bandwidth* (i.e., the amount of transferred data during the scaling process). Note that the scaling problem inherently differs from the classical *repair problem* [5], which aims to minimize the amount of transferred data for repairing lost data. Although both scaling and repair problems aim to minimize bandwidth, they build on different problem settings that lead to different analyses and findings.

In this paper, we study the scaling problem from both theoretical and applied perspectives. Our contributions include:

- We prove the information-theoretically minimum scaling bandwidth using the information flow graph model [3], [5]. To minimize the scaling bandwidth, we leverage the information mixing nature of network coding [3], by allowing storage nodes to send the combinations of both uncoded and coded data that is currently being stored. Note that existing scaling approaches (e.g., [11], [23], [24], [26], [28]) cannot achieve the minimum scaling bandwidth. To our knowledge, *our work is the first formal study on applying network coding to storage scaling*.
- We design a distributed storage system called NCScale, which realizes network-coding-based scaling by leveraging the available computational resources of storage nodes. NCScale achieves the minimum, or near-minimum, scaling bandwidth depending on the parameter settings, while preserving several properties that are necessary for practical deployment (e.g., fault tolerance, balanced erasure-coded data layout, and decentralized scaling).
- We have implemented a prototype of NCScale and conducted experiments on Amazon EC2. We show that NCScale can reduce the scaling time of Scale-RS [11], a state-of-the-art scaling approach for Reed-Solomon codes, by up to 50%. Also, we show that the empirical performance gain of NCScale is consistent with our theoretical findings.

II. Problem

We first present the basics of erasure coding. We then define the scaling problem and provide a motivating example. Table I summarizes the major notation used in this paper.

A. Erasure Coding Basics

Erasure coding is typically constructed by two configurable parameters n and k, where $k < n$, as an (n, k) code as follows. Specifically, we consider a distributed storage system (e.g., HDFS [20]) that organizes data as fixed-size units called *blocks*. For every group of k blocks, called *data blocks*, the storage system encodes them into additional $n-k$ equal-size blocks, called *parity blocks*, such that any k out of the n data and parity blocks suffice to reconstruct the original k data blocks. We call the collection of the n data and parity blocks a *stripe*, and the n blocks are stored in n different

TABLE I
MAJOR NOTATION USED IN THIS PAPER.

Notation	Descriptions
Defined in Section II	
(n, k)	erasure coding parameters
s	number of new nodes after scaling
X_i	i^{th} existing node, where $1 \leq i \leq n$
Y_j	j^{th} new node after scaling, where $1 \leq j \leq s$
D_*	a data block
P_*	a parity block before scaling
Q_*	a parity block after scaling
Defined in Section III	
M	original file size
β	bandwidth from X_i $(1 \leq i \leq n)$ to Y_j $(1 \leq j \leq s)$
$\mathcal{G}$	information flow graph
S	virtual source of $\mathcal{G}$
T	data collector of $\mathcal{G}$
Λ	capacity of a cut
Defined in Section IV	
$\mathbf{PG}$	a group of $nk(n+s)$ stripes for computing new parity blocks
$\mathbf{DG}$	a group of $ns(n+s)$ stripes for generating parity delta blocks
$\mathbf{D}_w$	w^{th} set of s data blocks of an existing node $X_{w \bmod n}$ in $\mathbf{DG}$, where $1 \leq w \leq nk(n+s)$ and $w' = \lceil \frac{w}{n} \rceil$
$\Delta_{i,j}$	parity delta block generated in node X_i for updating a parity block in X_j, where $1 \leq i, j \leq n$

nodes to tolerate any $n-k$ failures (either node failures or lost blocks). A storage system contains multiple stripes, which are independently encoded. The code construction has two properties: (i) *maximum distance separable (MDS)*, i.e., the fault tolerance is achieved through minimum storage redundancy, and (ii) *systematic*, i.e., the k data blocks are kept in a stripe for direct access. Reed-Solomon codes [18] are one well-known example of erasure codes that can achieve both MDS and systematic properties, and have been adopted by production systems (e.g., [8], [13]).

Most practical erasure codes (e.g., Reed-Solomon codes) are linear codes, in which each parity block is formed by a linear combination of the data blocks in the same stripe based on Galois Field arithmetic. In this paper, we focus on Vandermonde-based Reed-Solomon codes [15], whose encoding operations are based on an $(n-k) \times k$ Vandermonde matrix $[V_{i,j}]_{(n-k)\times k}$, where $1 \leq i \leq n-k$, $1 \leq j \leq k$, and $V_{i,j} = j^{i-1}$. For example, in a $(4, 2)$ code, we can compute two parity blocks, denoted by P_1 and P_2, through a linear combination of two data blocks, denoted by D_1 and D_2, over the Galois Field as follows:

$$\begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \end{bmatrix}. \quad (1)$$

Suppose that we now scale from the $(4, 2)$ code to the $(6, 4)$ code with two new data blocks D_3 and D_4. Then the two new parity blocks, denoted by P_1' and P_2', can be computed as:

$$\begin{bmatrix} P_1' \\ P_2' \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} D_3 \\ D_4 \end{bmatrix}. \quad (2)$$

Note that the Vandermonde matrix for the $(4, 2)$ code is a sub-matrix of the Vandermonde matrix for the $(6, 4)$ code. We see that each new parity block can be computed by adding an existing parity block with a *parity delta block*, which is a linear combination of the new data blocks only. In general, this holds for Vandermonde-based Reed-Solomon codes if we scale from an (n, k) code to an (n', k') code, where $n-k = n'-k'$. We leverage this feature in our scaling design.

While erasure coding incurs much less redundancy than replication [21], it is known to trigger a significant amount of transferred data when repairing a failure. For example, Reed-Solomon codes need to retrieve k blocks to repair a lost block. Thus, extensive studies focus on the *repair problem* (see survey [6]), which aims to minimize the repair bandwidth and hence improve the repair performance. In particular, based on network coding [3], regenerating codes [5] are special erasure codes that provably achieve the optimal trade-off between repair bandwidth and storage redundancy, by allowing non-failed nodes to encode their stored data during repair. On the other hand, *our work applies network coding to storage scaling, which fundamentally differs from the repair problem.*

B. Scaling

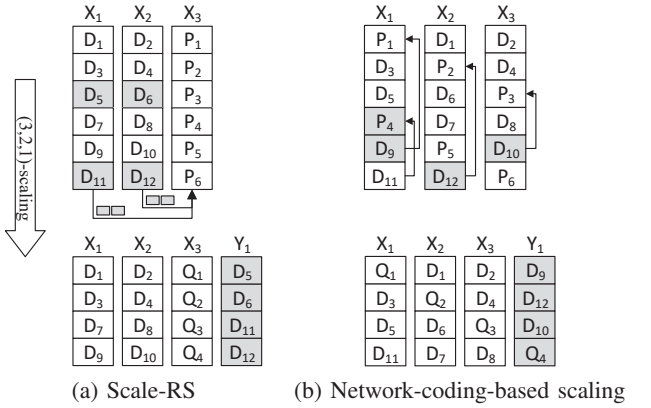
We now formalize the scaling problem.

(n, k, s)-scaling: For any $s > 0$, we 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 s new nodes, such that the following properties are achieved:

- **P1 (MDS and systematic):** The new $(n+s, k+s)$-coded stripe remains MDS and systematic, while tolerating the same number of $n-k$ failures as the original (n, k)-coded stripe.
- **P2 (Uniform data and parity distributions):** The respective proportions of data and parity blocks across multiple stripes should be evenly distributed across nodes before and after scaling. This ensures parity updates are load-balanced across nodes (assuming a uniform access pattern).
- **P3 (Decentralized scaling):** The scaling operation can be done without involving a centralized entity for coordination. This eliminates any single point of failure or bottleneck.
- **Goal:** We aim to minimize the scaling bandwidth, defined as the amount of transferred data during the scaling operation, while preserving all properties P1–P3, which are critical for practical deployment of erasure-coded storage.

We fix the number of tolerable failures (i.e., $n-k$) before and after scaling, as in existing scaling approaches for RAID (e.g., [23], [24], [28]) and distributed storage (e.g., [11], [26]), and we do not consider the variants of the scaling problem for varying $n-k$ [17]. We also discuss how we address $s<0$ (i.e., scale-down) in Section IV-E.

We address the scaling problem via network coding. We motivate this via an example of $(3, 2, 1)$-scaling shown in Figure 1. Let X_i be the i^{th} existing node of a stripe before scaling, where $1 \leq i \leq n$, and Y_j be the j^{th} new node after scaling, where $1 \leq j \leq s$. Also, let D_*, P_*, and Q_* be a data block, a parity block before scaling, and a parity block after scaling, respectively, for some index number $*$.



(a) Scale-RS (b) Network-coding-based scaling

Fig. 1. Scale-RS vs. network-coding-based scaling in (3,2,1)-scaling (i.e., from the $(3, 2)$ code to the $(4, 3)$ code). Scale-RS needs to transfer a total of eight blocks to X_3 and Y_1, while network-coding-based scaling transfers only four blocks to Y_1. Note that blocks in each node need not be stored in a contiguous manner in distributed storage systems.

We first consider Scale-RS [11] (see Figure 1(a)), which applies scaling to Reed-Solomon codes for general (n, k). Scale-RS performs scaling in two steps. The first step is *data block migration*, which relocates some data blocks from existing nodes to new nodes. For example, from Figure 1(a), the data blocks D_5, D_6, D_{11}, and D_{12} are relocated to the new node Y_1. The second step is *parity block updates*, which compute parity delta blocks in the nodes that hold the relocated data blocks and send them to the nodes that hold the parity blocks for reconstructing new parity blocks. For example, from Figure 1(a), the data blocks D_5, D_6, D_{11}, and D_{12} are used to compute the parity delta blocks, which are then sent to node X_3, where the parity blocks are stored. X_3 forms the new parity blocks Q_1, Q_2, Q_3, and Q_4, respectively. In this example, Scale-RS needs to transfer *eight blocks*. Note that property P2 is violated here, since the parity blocks are stored in a dedicated node.

We now consider how network-coding-based scaling can reduce the scaling bandwidth, as shown in Figure 1(b). Our key idea is to couple the steps of both data block migration and parity block updates, by allowing each existing node to perform local computations before relocating blocks. Specifically, each parity block before scaling is computed as the XOR operations of the data blocks in the same row as in RAID-5 [14]; for example, $P_1 = D_1 \oplus D_2$, where $\oplus$ is the XOR operator. During scaling, X_1 can locally compute the new parity block $Q_1 = P_1 \oplus D_9$. Similarly, X_2 and X_3 can compute the new parity blocks $Q_2 = P_2 \oplus D_{12}$ and $Q_3 = P_3 \oplus D_{10}$, respectively. Also, X_1 also locally computes $Q_4 = P_4 \oplus D_{11}$. Now, the scaling process relocates D_9, D_{10}, D_{12}, and the locally computed Q_4 to the new node Y_1. Thus, we now only need to transfer *four blocks*, and this amount is provably minimum (see Section III). In addition, all properties P1–P3 are satisfied.

Our idea is that unlike Scale-RS, in which data blocks and their encoded outputs (i.e., parity delta blocks) are transferred, we now make existing storage nodes send the encoded outputs of *both data blocks and parity blocks*. Since parity blocks are linear combinations of data blocks, we now include more information in the encoded outputs, thereby allowing less scaling bandwidth without losing information. This in fact follows the information mixing nature of network coding [3].

III. Analysis

We analyze (n, k, s)-scaling using the information flow graph model [3], [5]. We derive the lower bound of the scaling bandwidth, and show that the lower bound is tight by proving that there exist random linear codes whose scaling bandwidth matches the lower bound for general (n, k, s). Note that random linear codes are non-systematic, in which each stripe contains coded blocks only (i.e., P1 is violated). In Section IV, we address all P1–P3 in our scaling design.

A. Information Flow Graph Model

To comply with the information flow graph model in the literature (e.g., [5]), we assume that erasure coding operates on a per-file basis. Specifically, in order to encode a data file of size M, we divide it into k blocks of size $\frac{M}{k}$ each, encode the k blocks into n blocks of the same size, and distribute the n blocks across n nodes. Then the (n, k, s)-scaling process for the data file can be decomposed into four steps:

1) Each existing node X_i $(1 \leq i \leq n)$ encodes its stored data of size $\frac{M}{k}$ into some encoded data.
2) Each new node Y_j $(1 \leq j \leq s)$ downloads the encoded data from each X_i $(1 \leq i \leq n)$.
3) Each existing node X_i $(1 \leq i \leq n)$ deletes $\frac{M}{k} - \frac{M}{k+s}$ units of its stored data and only stores data of size $\frac{M}{k+s}$.
4) Each new node Y_j $(1 \leq j \leq s)$ encodes all its downloaded data into the stored data of size $\frac{M}{k+s}$.

Let β denote the bandwidth between any existing node X_i to any new node Y_j; in other words, each Y_j downloads at most β units of encoded data from X_i. To minimize the scaling bandwidth, our goal is to minimize β, while ensuring that the data file can be reconstructed from any k nodes.

We construct an information flow graph $\mathcal{G}$ for (n, k, s)-scaling as follows (see Figure 2):

Nodes in $\mathcal{G}$:

- We add a virtual source S and a data collector T as the source and destination nodes of $\mathcal{G}$, respectively.
- Each existing storage node X_i $(1 \leq i \leq n)$ is represented by (i) an input node X_i^{in}, (ii) a middle node X_i^{mid}, (iii) an output node X_i^{out}, (iv) a directed edge $X_i^{in} \rightarrow X_i^{mid}$ with capacity $\frac{M}{k}$, i.e., the amount of data stored in X_i before scaling, and (v) a directed edge $X_i^{mid} \rightarrow X_i^{out}$ with capacity $\frac{M}{k+s}$, i.e., the amount of data stored in X_i after scaling.
- Each new storage node Y_j $(1 \leq j \leq s)$ is represented by (i) an input node Y_j^{in}, (ii) an output node Y_j^{out}, and (iii) a directed edge $Y_j^{in} \rightarrow Y_j^{out}$ with capacity $\frac{M}{k+s}$, i.e., the amount of data stored in node Y_j.

Edges in $\mathcal{G}$:

- We add a directed edge $S \rightarrow X_i^{in}$ for every i $(1 \leq i \leq n)$ with an infinite capacity for data distribution.

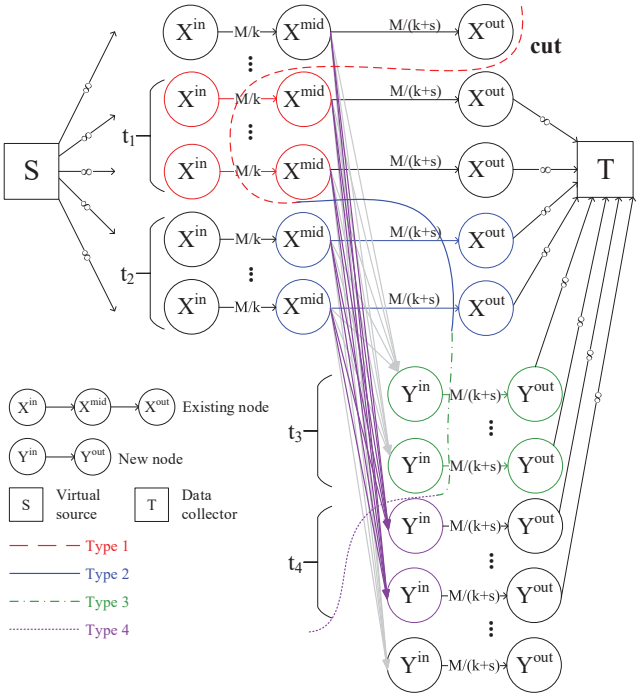


Fig. 2. Information flow graph $\mathcal{G}$ for (n, k, s)-scaling.

- We add a directed edge $X_i^{mid} \rightarrow Y_j^{in}$ for every i ($1 \leq i \leq n$) and j ($1 \leq j \leq s$) with capacity β.
- We select any k output nodes and add a directed edge from each of them to T with an infinite capacity for data reconstruction.

The following lemma states the *necessary condition* of the lower bound of β.

Lemma 1. *β must be at least $\frac{M}{n(k+s)}$.*

Proof: Clearly, each new storage node Y_j ($1 \leq j \leq s$) must receive at least $\frac{M}{k+s}$ units of data from all existing storage nodes X_i's ($1 \leq i \leq n$) over the links with capacity β each. Thus, we have $n\beta \geq \frac{M}{k+s}$. The lemma follows. □

B. Existence

To show the lower bound in Lemma 1 is tight, we first analyze the capacities of all possible min-cuts of $\mathcal{G}$. A *cut* is a set of directed edges, such that any path from S to T must have at least one edge in the cut. A *min-cut* is the cut that has the minimum sum of capacities of all its edges. Due to the MDS property, there are $\binom{n+s}{k+s}$ possible data collectors. Thus, the number of variants of $\mathcal{G}$, and hence the number of possible min-cuts, are also $\binom{n+s}{k+s}$.

Lemma 2. *Suppose that β is equal to its lower bound $\frac{M}{n(k+s)}$. Then the capacity of each possible min-cut of $\mathcal{G}$ is at least M.*

Proof: Let $(\mathcal{C}, \bar{\mathcal{C}})$ be some cut of $\mathcal{G}$, where $S \in \mathcal{C}$ and $T \in \bar{\mathcal{C}}$. Here, we do not consider the cuts that have an edge directed either from S or to T, since such an edge has an infinite capacity. For the remaining cuts, we can classify the storage nodes into four types based on the nodes in $\bar{\mathcal{C}}$:

- *Type 1*: Both X_i^{mid} and X_i^{out} are in $\bar{\mathcal{C}}$ for some $i \in [1, n]$;
- *Type 2*: Only X_i^{out} is in $\bar{\mathcal{C}}$ for some $i \in [1, n]$;
- *Type 3*: Only Y_j^{out} is in $\bar{\mathcal{C}}$ for some $j \in [1, s]$; and
- *Type 4*: Both Y_j^{in} and Y_j^{out} are in $\bar{\mathcal{C}}$ for some $j \in [1, s]$.

We now derive the capacity of each possible cut for each data collector. Suppose that T connects to t_i nodes of Type i, where $1 \leq i \leq 4$, for data reconstruction, such that:

$$t_1 + t_2 + t_3 + t_4 = k + s. \tag{3}$$

Let $\Lambda(t_1, t_2, t_3, t_4)$ denote the capacity of a cut. We derive Λ as follows:

- Each storage node of Type 1 contributes $\frac{M}{k}$ to Λ;
- Each storage node of Type 2 contributes $\frac{M}{k+s}$ to Λ;
- Each storage node of Type 3 contributes $\frac{M}{k+s}$ to Λ; and
- Each storage node of Type 4 contributes $(n - t_1)\beta$ to Λ.

Figure 2 illustrates the details. Thus, we have:

$$\Lambda = t_1 \cdot \frac{M}{k} + t_2 \cdot \frac{M}{k+s} + t_3 \cdot \frac{M}{k+s} + t_4 \cdot (n - t_1)\beta. \tag{4}$$

By Lemma 1 and Equation (3), we reduce Equation (4) to:

$$\Lambda \geq M + M \cdot \frac{t_1 \cdot (n \cdot s - k \cdot t_4)}{k(k+s)n}. \tag{5}$$

Since $n > k$ and $s \geq t_4$ (Type 4 only has new storage nodes), the right hand side of Equation (5) must be at least M. The lemma holds. □

Lemma 3 ([5])**.** *If the capacity of each possible min-cut of $\mathcal{G}$ is at least the original file size M, there exists a random linear network coding scheme guaranteeing that T can reconstruct the original file for any connection choice, with a probability that can be driven arbitrarily high by increasing the field size.*

Theorem 1. *For (n, k, s)-scaling and an original file of size M, there exists an optimal functional scaling scheme, such that β is minimized at $\frac{M}{n(k+s)}$ while the MDS property of tolerating any $n - k$ failures is preserved.*

Proof: It follows from immediately Lemmas 2 and 3. □

Theorem 1 implies that optimal scaling occurs when the amount of transferred data to the new nodes is equal to the size of the data being stored in new nodes. Thus, we have the following corollary.

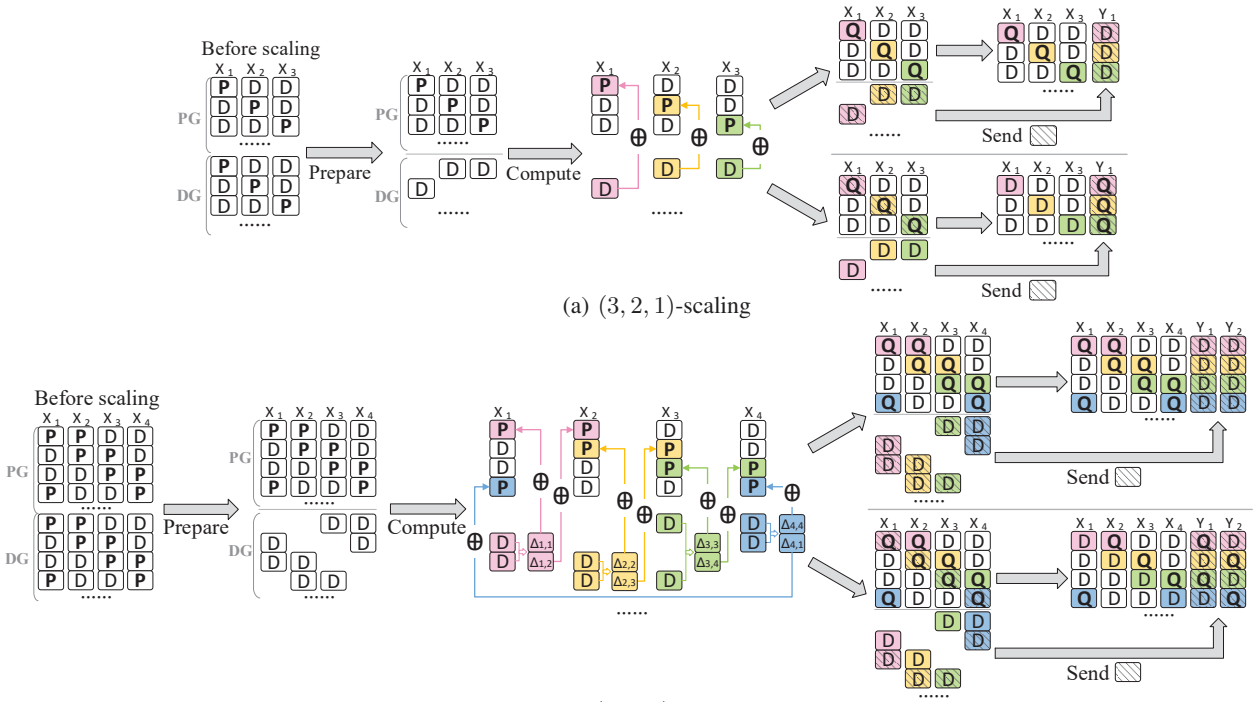
Corollary 1. *For distributed storage systems that organize data in fixed-size blocks, the minimum scaling bandwidth is s blocks per new stripe formed for any (n, k, s)-scaling.*

IV. NCScale

We present NCScale, a distributed storage system that realizes network-coding-based storage scaling. NCScale satisfies properties P1–P3 (see Section II-B) and achieves the minimum, or near-minimum, scaling bandwidth (see Section III).

A. Main Idea

NCScale operates on (systematic) Reed-Solomon codes [18], such that all blocks before and after scaling are still encoded by Reed-Solomon codes. Before scaling, each existing



(a) $(3, 2, 1)$-scaling

(b) $(4, 2, 2)$-scaling

Fig. 3. Scaling in NCScale. Note that $(3, 2, 1)$-scaling achieves the minimum scaling bandwidth (see Corollary 1), while $(4, 2, 2)$-scaling does not.

node independently computes parity delta blocks, which are then merged with existing parity blocks to form new parity blocks for the new stripes after scaling. Finally, NCScale sends some of the data blocks and new parity blocks to the new nodes, while ensuring that the new stripes have uniform distributions of data and parity blocks across nodes.

NCScale can achieve the minimum scaling bandwidth when $n - k = 1$ (i.e., each stripe has one parity block). In this case, the new parity block of each stripe can be computed *locally* from the parity delta block generated from the same node. In other words, the blocks that are sent over the network by NCScale are only those that will be stored in the new nodes. From Corollary 1, the scaling bandwidth of NCScale matches the optimal point. Figure 3(a) shows an example of $(3, 2, 1)$-scaling in NCScale.

On the other hand, NCScale cannot achieve the optimal point for $n-k > 1$ (i.e., each stripe has more than one parity block). Each existing node now not only generates a parity delta block for locally computing a new parity block, but also sends parity delta blocks for computing new parity blocks of the same stripe in different nodes. Nevertheless, the number of parity delta blocks that are sent to other nodes remains limited, as we only use one parity delta block to update each new parity block (see Section IV-C for details). Figure 3(b) shows an example of $(4, 2, 2)$-scaling in NCScale.

One constraint of NCScale is that its current algorithmic design requires $s \leq \frac{n}{n-k-1}$; if $n-k = 1$, s can be of any value (see Section IV-C). Nevertheless, we believe that the range of s is sufficiently large in practice, as n is often much larger than $n - k$ to limit the amount of storage redundancy.

B. Preliminaries

We now provide definitions for NCScale in (n, k, s)-scaling and summarize the steps of NCScale. We also present the scaling bandwidth of NCScale.

To perform scaling, NCScale operates on a collection of $n(k+s)(n+s)$ stripes in n nodes that have $nk(k+s)(n+s)$ data blocks in total. We assume that the nodes that hold the parity blocks in a stripe are circularly rotated across stripes [16], so as to keep the uniform distributions of data and parity blocks over the n nodes; formally, the $n - k$ parity blocks of the w^{th} stripe are stored in $X_i, \cdots X_{(i+n-k-1) \bmod n}$ for some $w \geq 1$ and $i = w \bmod n$. After scaling, NCScale forms $nk(n+s)$ stripes over $n+s$ nodes, with the same number of $nk(k+s)(n+s)$ data blocks in total.

NCScale classifies the $n(k+s)(n+s)$ stripes into two *groups*. The first group, denoted by $\mathbf{PG}$, contains the first $nk(n+s)$ stripes, in which their parity blocks will be updated from parity delta blocks to form new parity blocks. The second group, denoted by $\mathbf{DG}$, contains the remaining $ns(n+s)$ stripes, in which we use their data blocks to generate parity delta blocks for updating the parity blocks in $\mathbf{PG}$. Note that the number of stripes in $\mathbf{PG}$ is also equal to the number of stripes after scaling.

Parity delta blocks are formed by the linear combinations of the new data blocks in a stripe based on a Vandermonde matrix (see Section II-A). Let $\Delta_{i,j}$ be a parity delta block generated from an existing node X_i for updating a parity block in an existing node X_j, where $1 \leq i, j \leq n$ (when $i = j$, the new parity block is computed locally). NCScale ensures that for each of the $nk(n+s)$ stripes in $\mathbf{PG}$, the $n-k$ parity blocks of

Algorithm 1 Prepare

1: $\mathbf{PG}$ = first $nk(n+s)$ stripes
2: $\mathbf{DG}$ = next $ns(n+s)$ stripes
3: **for** $w = 1$ to $nk(n+s)$ **do**
4: $\quad \mathbf{D}_w = w'^{th}$ set of s data blocks of $X_{w \bmod n}$ in $\mathbf{DG}$, where $w' = \lceil \frac{w}{n} \rceil$
5: **end for**

the new stripe can be computed from parity delta blocks that are all generated by the same node. One of the parity blocks can retrieve a parity delta block locally, while the remaining $n-k-1$ parity blocks need to retrieve a total of $n-k-1$ parity delta blocks over the network. In other words, there will be a total of $nk(n+s)(n-k-1)$ parity delta blocks transferred over the network.

In addition, NCScale sends $nk(n+s) \times s$ blocks to the s new nodes. In general, the scaling bandwidth of NCScale per new stripe formed after scaling is:

$$\frac{nk(n+s)(n-k-1+s)}{nk(n+s)} = n-k-1+s. \quad (6)$$

C. Algorithmic Details

We now present the algorithmic details of (n,k,s)-scaling in NCScale. Figure 3 illustrates the algorithmic steps.

• **Prepare:** NCScale prepares the sets of data and parity blocks to be processed in the scaling process, as shown in Algorithm 1. It first identifies the groups $\mathbf{PG}$ and $\mathbf{DG}$ (lines 1-2). It then divides the data blocks in $\mathbf{DG}$ into different sets $\mathbf{D}_w$'s, where $1 \le w \le nk(n+s)$ (lines 3-5), by collecting and adding s data blocks from X_1 to X_n into $\mathbf{D}_w$ in a round-robin fashion. Specifically, $\mathbf{DG}$ has $ns(n+s)$ stripes, and hence $nsk(n+s)$ data blocks, in total. We divide the data blocks of each existing node X_i $(1 \le i \le n)$ in $\mathbf{DG}$ into $k(n+s)$ sets of s data blocks, and add the w'^{th} set of s data blocks of each existing node $X_{w \bmod n}$ into $\mathbf{D}_w$, where "mod" denotes the modulo operator and $w' = \lceil \frac{w}{n} \rceil$ (line 4).

• **Compute, Send, and Delete:** After preparation, NCScale computes new parity blocks for the new stripes, sends blocks to the s new nodes, and deletes obsolete blocks in existing nodes. Algorithm 2 shows the details. NCScale operates across all $nk(n+s)$ stripes in $\mathbf{PG}$. To compute the new parity blocks, each existing node X_i $(1 \le i \le n)$ operates on the w^{th} stripe for $i = w \bmod n$ (line 2). Recall that the parity blocks are stored in $X_i, \cdots, X_{(i+n-k-1) \bmod n}$. For $1 \le j \le n-k$ and $j' = (j+w-1) \bmod n$, X_i computes a parity delta block $\Delta_{i,j'}$ and sends it to $X_{j'}$, which adds $\Delta_{i,j'}$ to the j^{th} parity block of the w^{th} stripe in $\mathbf{PG}$ (lines 3-7). Note that when $j = 1$, X_i updates the parity block locally.

After computing the new parity blocks, NCScale sends blocks to the new nodes (lines 8-12). We find that if $w \le nk(n-s(n-k-1))$, X_i sends all s data blocks in $\mathbf{D}_w$ to the s new nodes; otherwise, X_i sends the locally updated parity block and any $s-1$ data blocks in $\mathbf{D}_w$ to the s new nodes (we assume that the parity block is rotated over the s nodes across different stripes to evenly place the parity blocks). For example, Figure 3 shows that the last step of scaling is split into two cases. Finally, X_i deletes all obsolete blocks, including the blocks that are sent to the new nodes and the parity blocks in $\mathbf{DG}$ (line 13). By doing so, we can guarantee uniform distributions of data and parity blocks after scaling (see Section IV-D).

Algorithm 2 Compute, Send, and Delete

1: **for** $w = 1$ to $nk(n+s)$ **do**
2: $\quad i = w \bmod n$
3: $\quad$ **for** $j = 1$ to $n-k$ **do**
4: $\quad\quad j' = (j+w-1) \bmod n$
5: $\quad\quad X_i$ generates $\Delta_{i,j'}$ from the s data blocks in $\mathbf{D}_w$ for the j^{th} parity block in the w^{th} stripe of $\mathbf{PG}$
6: $\quad\quad X_i$ sends $\Delta_{i,j'}$ to $X_{j'}$, which adds $\Delta_{i,j'}$ to the j^{th} parity block in the w^{th} stripe of $\mathbf{PG}$
7: $\quad$ **end for**
8: $\quad$ **if** $w \le nk(n-s(n-k-1))$ **then**
9: $\quad\quad X_i$ sends all s data blocks in $\mathbf{D}_w$ to the s new nodes
10: $\quad$ **else**
11: $\quad\quad X_i$ sends the locally updated parity block and any $s-1$ data blocks in $\mathbf{D}_w$ to the s new nodes
12: $\quad$ **end if**
13: $\quad X_i$ deletes all obsolete blocks
14: **end for**

Remark: Algorithm 2 requires $s \le \frac{n}{n-k-1}$, so that the right side of the inequality in line 8 is a positive number.

D. Proof of Correctness

Theorem 2. *NCScale preserves P1–P3 after scaling.*

Proof: Consider P1. According to Algorithm 2, the $k+s$ data blocks in the w^{th} new stripe, where $1 \le w \le nk(n+s)$, are composed of the k data blocks of the w^{th} existing stripe in $\mathbf{PG}$ and s data blocks in $\mathbf{D}_w$. Each new parity block is computed by adding (i) the existing parity block of the w^{th} stripe and (ii) the parity delta block that is formed by the linear combinations of the s data blocks in $\mathbf{D}_w$. Due to the property of the Vandermonde matrix (see Section II-A), the new parity blocks become encoded by Vandermonde-based Reed-Solomon codes over $k+s$ data blocks. Thus, both MDS and systematic properties are maintained. P1 holds.

Consider P2. We count the number of parity blocks stored in each node after scaling. Algorithm 2 moves $nk(n+s) - nk(n-s(n-k-1)) = nks(n-k)$ parity blocks from existing nodes to the s new nodes (line 11). Thus, each of the s new nodes has $\frac{1}{s} \cdot nks(n-k) = nk(n-k)$ parity blocks, while each of the n existing nodes has $\frac{1}{n} \cdot (nk(n+s)(n-k) - nks(n-k)) = nk(n-k)$ parity blocks. Thus, all $n+s$ nodes have the same number of parity blocks (and hence data blocks). P2 holds.

Consider P3. According to Algorithm 2, each existing node X_i can independently generate and send parity delta blocks of the w^{th} stripe, simply by checking if i is equal to $w \bmod n$. No centralized coordination across all existing nodes is necessary. P3 holds. □

E. Discussion

We thus far focus on the scale-up operation of adding new nodes ($s > 0$). For scale-down ($s < 0$), we reverse the steps

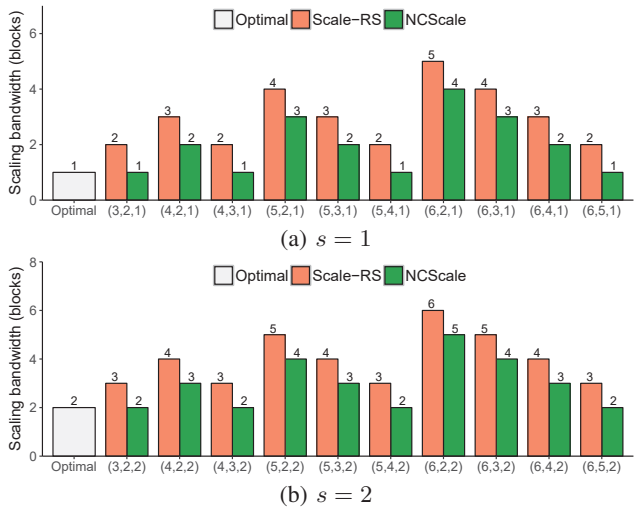


Fig. 4. Numerical results of scaling bandwidth (in units of blocks) per new stripe formed after scaling.

in Algorithm 2, by relocating data blocks in s nodes and distributing them to n nodes, computing parity delta blocks, and updating the parity blocks in $\mathbf{PG}$. Also, we need to reconstruct new parity blocks for the stripes in $\mathbf{DG}$. We do not claim the optimality of the scale-down case, and we pose it as future work.

V. Evaluation

In this section, we present evaluation results of NCScale. We compare it with Scale-RS [11], which represents the state-of-the-art scaling scheme for Reed-Solomon codes in distributed storage systems. We conduct both numerical analysis and cloud experiments, and aim to address two key questions: (i) Can NCScale improve the scaling performance by mitigating the scaling bandwidth? (ii) Is the empirical performance of NCScale consistent with the numerical results?

A. Numerical Analysis

In our numerical analysis, we calculate the scaling bandwidth as the total number of blocks transferred during scaling normalized to the total number of stripes after scaling. We summarize the numerical results below:

- **Optimal:** The information-theoretically minimum scaling bandwidth is given by s blocks (per new stripe) for any (n, k) (see Corollary 1).
- **Scale-RS:** To form a new stripe after scaling, Scale-RS first sends s data blocks from existing nodes to s new nodes for data block migration, followed by $n-k$ parity delta blocks for parity block updates (see Figure 1(a) for an example). Thus, the scaling bandwidth of Scale-RS is $s+n-k$ blocks (per new stripe).
- **NCScale:** From Equation (6), the scaling bandwidth of NCScale is $s+n-k-1$ blocks (per new stripe).

Figure 4 shows the numerical results of scaling bandwidth (in units of blocks) per new stripe formed after scaling. Here, we focus on $s = 1$ and $s = 2$, and vary (n, k). In summary, the percentage reduction of scaling bandwidth of NCScale over Scale-RS is higher for smaller $n-k$ or smaller s. For example, for $(6, 5, 1)$, the reduction is 50%, while for $(6, 4, 1)$ and $(6, 5, 2)$, the reduction is 33.3%. NCScale matches the optimal point when $n-k = 1$, and deviates more from the optimal point when $n-k$ increases (e.g., by three blocks more for $(6, 2, 2)$). Nevertheless, NCScale always has less scaling bandwidth than Scale-RS by one block (per new stripe).

TABLE II
Experiment 1: Time breakdown of NCScale for $(n, k, s) = (6, 4, 2)$ and block size 64 MB.

	Steps		
	Compute	Send	Delete
200 Mb/s	0.093s	21.89s	0.0040s
500 Mb/s	0.090s	8.70s	0.0040s
1 Gb/s	0.094s	4.33s	0.0039s
2 Gb/s	0.095s	2.17s	0.0040s

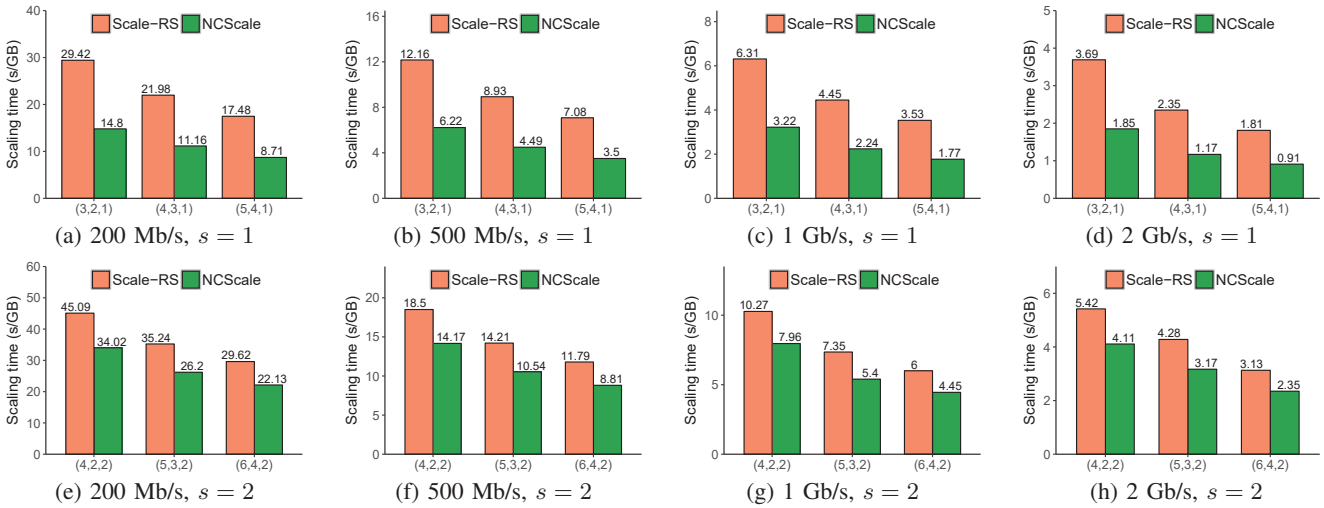
B. Cloud Experiments

We implemented NCScale as a distributed storage system prototype and evaluated its scaling performance in real-world environments. NCScale is mostly written in Java, while the coding operations of Reed-Solomon codes are written in C++ based on Intel ISA-L [2]. We also implemented Scale-RS based on our NCScale prototype for fair comparisons under the same implementation settings. Each storage node runs as a server process. In both of our NCScale and Scale-RS implementations, the storage nodes perform the scaling steps independently and in parallel.

Setup: We conduct our experiments on Amazon EC2 [1]. We configure a number of `m4.4xlarge` instances located in the US East (North Virginia) region. The number of instances varies across experiments (see details below), and the maximum is 14. Each instance represents an existing storage node (before scaling) or a new storage node (after scaling). To evaluate the impact of bandwidth on scaling, we configure a dedicated instance that acts a *gateway*, such that any traffic between every pair of instances must traverse the gateway. We then use the Linux traffic control command `tc` to control the outgoing bandwidth of the gateway. In our experiments, we vary the gateway bandwidth from 200 Mb/s up to 2 Gb/s.

Methodology: We measure the scaling time per 1 GB of data blocks (64 MB each by default). Recall that NCScale operates on collections of $n(k+s)(n+s)$ stripes (see Section IV-B). In each run of experiments, depending on the values of (n, k, s), we generate around 1,000 data blocks (and the corresponding parity blocks), so as to obtain a sufficient number of collections of stripes for stable scaling performance. We report the average results of each experiment over five runs. We do not plot the deviations, as they are very small across different runs.

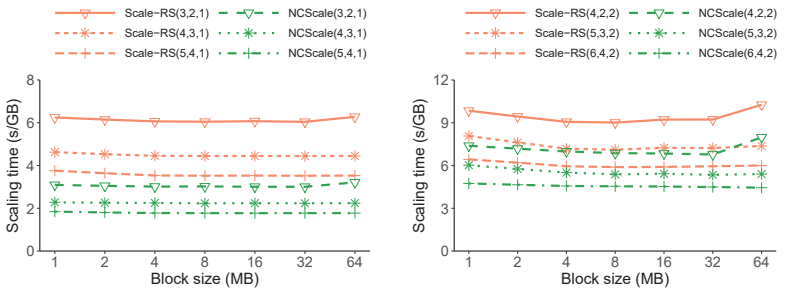
Experiment 1 (Time breakdown): We first provide a breakdown of the scaling time and identify the bottlenecked step in scaling. We decompose Algorithm 2 into three steps that are carried out by existing nodes: (i) *compute*, which refers to the generation of parity delta blocks and computation of new parity blocks, (ii) *send*, which refers to the transfers of blocks



(a) 200 Mb/s, $s = 1$ (b) 500 Mb/s, $s = 1$ (c) 1 Gb/s, $s = 1$ (d) 2 Gb/s, $s = 1$

(e) 200 Mb/s, $s = 2$ (f) 500 Mb/s, $s = 2$ (g) 1 Gb/s, $s = 2$ (h) 2 Gb/s, $s = 2$

Fig. 5. Experiment 2: Scaling time (per GB of data blocks), in seconds/GB, under different gateway bandwidth settings.



(a) $s = 1$ (b) $s = 2$

Fig. 6. Experiment 3: Scaling time (per GB of data blocks), in seconds/GB, versus block size.

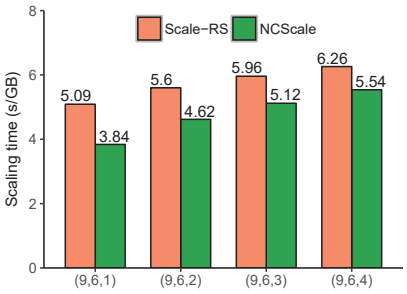


Fig. 7. Experiment 4: Scaling time (per GB of data blocks), in seconds/GB, versus s.

during the scaling process, and (iii) *delete*, which describes the deletion of obsolete blocks after scaling. Since all existing nodes perform scaling in parallel, we pick the one that finishes last and obtain its time breakdown. Here, we consider $(6, 4, 2)$-scaling. We fix the block size as 64 MB and vary the gateway bandwidth from 200 Mb/s to 2 Gb/s.

Table II shows the breakdown. We observe that the send time dominates (over 95% of the overall time), especially when the available network bandwidth is limited (while the compute and delete times stay fairly constant). This justifies our goal of minimizing the scaling bandwidth to improve the overall scaling performance.

Experiment 2 (Impact of bandwidth): We now compare NCScale and Scale-RS under different gateway bandwidth settings. Figure 5 shows the scaling time results, in which the block size is fixed as 64 MB. We find that the empirical results are consistent with the numerical ones (see Figure 4) in all cases, mainly because the scaling performance is dominated by the send time. Take $(6, 4, 2)$-scaling for example. From the numerical results (see Figure 4), NCScale incurs 25.0% less scaling bandwidth than Scale-RS (i.e., three versus four blocks, respectively), while in our experiments, Scale-RS incurs 25.29%, 25.28%, 25.83%, and 24.92% more scaling time than NCScale when the gateway bandwidth is 200 Mb/s, 500 Mb/s, 1 Gb/s, and 2 Gb/s, respectively (see Figures 5(e)-5(h)). Note that the scaling time increases with the redundancy $\frac{n}{k}$ (e.g., $(3, 2, 1)$ has higher scaling time than $(4, 3, 1)$ and $(5, 4, 1)$). The reason is that the number of stripes per GB of data blocks also increases with the amount of redundancy, so more blocks are transferred during scaling.

Experiment 3 (Impact of block size): We study the scaling time versus the block size. We fix the gateway bandwidth as 1 Gb/s and vary the block size from 1 MB to 64 MB. Figure 6 shows the results. We see that the scaling times of NCScale and Scale-RS are fairly stable across different block sizes, and NCScale still shows performance gains over Scale-RS.

Experiment 4 (Impact of s): Finally, we study the scaling time versus s (the number of new nodes). Here, we fix the gateway bandwidth as 1 Gb/s and the block size as 64 MB. We also fix $(n, k) = (9, 6)$, which is a default setting in production [13]. Figure 7 shows the results. Both NCScale and Scale-RS need to transfer more blocks as s increases, and the difference of their scaling times decreases. Overall, NCScale reduces the scaling time of Scale-RS by 11.5-24.6%.

VI. Related Work

Scaling approaches have been proposed for RAID-0 (i.e., no fault tolerance) [29], [32], RAID-5 (i.e., single fault tolerance)

[9], [23], [30], [31], and RAID-6 [24], [25], [28] (i.e., double fault tolerance). Such scaling approaches focus on minimizing data block migration and parity block updates (e.g., GSR [23] for RAID-5, and MDS-Frame [24] and RS6 [28] for RAID-6), while keeping the same RAID configuration and tolerating the same number of failures. However, they are tailored for RAID arrays and cannot tolerate more than two failures.

The most closely related work to ours is Scale-RS [11], which addresses the scaling problem in distributed storage systems that employ Reed-Solomon codes [18] to provide tolerance against a general number of failures. Wu et al. [26] apply scaling for Cauchy Reed-Solomon codes [4], but use a centralized node to coordinate the scaling process. In contrast, both Scale-RS and NCScale perform scaling in a decentralized manner. However, existing RAID scaling approaches and Scale-RS cannot minimize the scaling bandwidth.

Some studies address the efficient transitions between redundancy schemes. AutoRAID [22] leverages access patterns to switch between replication for hot data and RAID-5 for cold data. DiskReduce [7] and EAR [12] address the transition replication to erasure coding in HDFS [20]. HACFS [27] extends HDFS to support switching between two erasure codes to trade between storage redundancy and access performance. Rai et al. [17] present adaptive erasure codes for switching between the erasure coding parameters (n, k), but they only describe a few scaling cases without formal analysis. Our work emphasizes how network coding can help achieve the optimality of storage scaling, using both analysis and implementation.

VII. Conclusions

We study how network coding is applied to storage scaling from both theoretical and applied perspectives. We prove the minimum scaling bandwidth via the information flow graph model. We further build NCScale, which implements network-coding-based scaling for distributed storage. Both numerical analysis and cloud experiments demonstrate the scaling efficiency of NCScale.

Acknowledgments

This work was supported by National Natural Science Foundation of China (grants 61502191, 61502190, 61602197, 61772222), Fundamental Research Funds for the Central Universities (grants 2017KFYXJJ065, 2016YXMS085), Hubei Provincial Natural Science Foundation of China (grants 2016CFB226, 2016CFB192), Key Laboratory of Information Storage System Ministry of Education of China, and Research Grants Council of Hong Kong (grants GRF 14216316 and CRF C7036-15G). The corresponding author is Yuchong Hu.

References

[1] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[2] Intel ISA-L. https://01.org/zh/intel%C2%AE-storage-acceleration-library-open-source-version.

[3] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, 2000.

[4] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[5] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep 2010.

[6] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A Survey on Network Codes for Distributed Storage. *Proceedings of the IEEE*, 99(3):476–489, Mar 2011.

[7] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for Data-Intensive Scalable Computing. In *Proc. of ACM PDSW*, Nov 2009.

[8] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[9] S. R. Hetzler. Data Storage Array Scaling Method and System with Minimal Data Movement, Aug. 7 2012. US Patent 8,239,622.

[10] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, Jun 2012.

[11] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie. Scale-RS: An Efficient Scaling Scheme for RS-coded Storage Clusters. *IEEE Trans. on Parallel and Distributed Systems*, 26(6):1704–1717, 2015.

[12] R. Li, Y. Hu, and P. P. C. Lee. Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2015.

[13] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proceedings of the VLDB Endowment*, 2013.

[14] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of ACM SIGMOD*, 1988.

[15] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, Sep 1997.

[16] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. OHearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.

[17] B. K. Rai, V. Dhoorjati, L. Saini, and A. K. Jha. On Adaptive Distributed Storage Systems. In *Proc. of IEEE ISIT*, 2015.

[18] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[19] 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 VLDB Endowment*, pages 325–336, 2013.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[21] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[22] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. on Computer Systems*, 14(1):108–136, Feb 1996.

[23] C. Wu and X. He. GSR: A Global Stripe-based Redistribution Approach to Accelerate RAID-5 Scaling. In *Proc. of IEEE ICPP*, 2012.

[24] C. Wu and X. He. A Flexible Framework to Enhance RAID-6 Scalability via Exploiting the Similarities Among MDS Codes. In *Proc. of IEEE ICPP*, 2013.

[25] C. Wu, X. He, J. Han, H. Tan, and C. Xie. SDM: A Stripe-based Data Migration Scheme to Improve the Scalability of RAID-6. In *Proc. of IEEE CLUSTER*, 2012.

[26] S. Wu, Y. Xu, Y. Li, and Z. Yang. I/O-Efficient Scaling Schemes for Distributed Storage Systems with CRS Codes. *IEEE Trans. on Parallel and Distributed Systems*, 27(9):2639–2652, Sep 2016.

[27] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A Tale of Two Erasure Codes in HDFS. In *Proc. of USENIX FAST*, 2015.

[28] G. Zhang, K. Li, J. Wang, and W. Zheng. Accelerate RDP RAID-6 Scaling by Reducing Disk I/Os and XOR Operations. *IEEE Trans. on Computers*, 64(1):32–44, 2015.

[29] G. Zhang, J. Shu, W. Xue, and W. Zheng. SLAS: An Efficient Approach to Scaling Round-robin Striped Volumes. *ACM Trans. on Storage*, 3(1):3, 2007.

[30] G. Zhang, W. Zheng, and K. Li. Rethinking RAID-5 Data Layout for Better Scalability. *IEEE Trans. on Computers*, 63(11):2816–2828, 2014.

[31] G. Zhang, W. Zheng, and J. Shu. ALV: A New Data Redistribution Approach to RAID-5 Scaling. *IEEE Trans. on Computers*, 59(3):345–357, 2010.

[32] W. Zheng and G. Zhang. FastScale: Accelerate RAID Scaling by Minimizing Data Migration. In *Proc. of USENIX FAST*, 2011.