

NCScale: Toward Optimal Storage Scaling via Network Coding

Yuchong Hu¹, Member, IEEE, Xiaoyang Zhang, Patrick P. C. Lee², Senior Member, IEEE, and Pan Zhou¹

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 for both scale-out and scale-in cases. Based on our theoretical findings, we also build a distributed storage system prototype NCScale based on Hadoop Distributed File System, so as to realize 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.

Index Terms—Erasure coding, storage scaling.

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 [32], and has been widely adopted in production distributed storage systems [11], [15], [29].

To accommodate the increasing storage demands, system operators often regularly add new nodes to storage systems

Manuscript received November 24, 2020; revised July 2, 2021 and August 17, 2021; accepted August 17, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor H. Seferoglu. Date of publication August 26, 2021; date of current version February 17, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61872414, in part by the Key Laboratory of Information Storage System Ministry of Education of China, and in part by the Research Grants Council of Hong Kong under Grant AoE/P-404/18. An earlier version of this paper appeared at [DOI: 10.1109/INFOCOM.2018.8485961]. In this extended version, we extend the design and analysis of NCScale for the scale-in case and implement NCScale based on Hadoop Distributed File System. We also add new evaluation results. (Corresponding author: Pan Zhou.)

Yuchong Hu and Xiaoyang Zhang are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: yuchonghu@hust.edu.cn; zhangxiaoyang1993@gmail.com).

Patrick P. C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (e-mail: plee@cse.cuhk.edu.hk).

Pan Zhou is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: panzhou@hust.edu.cn).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TNET.2021.3106394>, provided by the authors.

Digital Object Identifier 10.1109/TNET.2021.3106394

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, so as to exploit the maximum possible parallelization among all nodes. Also, 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, the repair costs of erasure-coded storage systems can be reduced by increasing storage redundancy [8]. Storage systems may dynamically switch between erasure codes of different redundancy levels to balance between access performance and fault tolerance in response to different access patterns of workloads [39], or to balance between storage efficiency and fault tolerance as the disk reliability varies over the disk lifetime [17].

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* [8], which aims to minimize the amount of transferred data for repairing lost data. Although both scaling and repair problems aim to minimize bandwidth, scaling changes the coding parameters and the number of storage nodes, while repair keeps them unchanged. Thus, the scaling and repair problems 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 [6], [8]; in particular, we consider both scale-out and scale-in cases (defined in Section II-B). To minimize the scaling bandwidth, we leverage the information mixing nature of network coding [6], 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., [16], [34], [35], [38], [40]) 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 aims to achieve the 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 based on Hadoop Distributed File System (HDFS) [30] and conducted experiments on Amazon EC2. We show that NCScale reduces the scaling time of Scale-RS [16], a state-of-the-art scaling approach, by up to 50%. Also, the empirical performance gain of NCScale is consistent with our theoretical findings.

The source code of our NCScale prototype is available for download at: <https://github.com/yuchonghu/ncscale>.

II. PROBLEM

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 [30]) 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 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 (RS) codes [28] 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., [11], [22]).

Most practical erasure codes (e.g., RS 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 RS codes [24], 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 $V_{i,j} = i^{j-1}$. 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:

$$\begin{bmatrix} P'_1 \\ P'_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} D_3 \\ D_4 \end{bmatrix}. \quad (2)$$

The Vandermonde matrix for the $(4, 2)$ code is a sub-matrix of the Vandermonde matrix for the $(6, 4)$ code. Each new parity block can be computed by adding an existing parity block with a *parity delta block*, defined as the change between

the existing parity block and the new parity block. For example, when P_1 is updated to P'_1 , we have $P'_1 = P_1 + \Delta P_1$, where ΔP_1 is the parity delta block corresponding to existing parity block P_1 . Note that the parity delta block can be expressed as a linear combination of the new data blocks only (e.g., $\Delta P_1 = D_3 + D_4$ as in Equation (2)); this holds for Vandermonde-based RS 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.

Note that systematic codes built on Vandermonde matrices generally do not preserve the MDS property under finite fields [18], [25]. Intel's Intelligent Storage Acceleration Library (ISA-L) [5] also supports Vandermonde-based RS codes (in the function `gf_gen_rs_matrix`), and provides a program called `gen_rs_matrix_limits` (since ISA-L version 2.19) for finding the valid parameters that satisfy the MDS property of RS codes under the Galois Field $\text{GF}(2^8)$. If we focus on small ranges of (n, k) and (n', k') (e.g., $n, n' \leq 20$ and $n - k \leq 4$), the MDS property of the Vandermonde-based RS codes still holds under $\text{GF}(2^8)$ [2]. In practice, the parameters (n, k) in real deployment often fall into this range [13].

While erasure coding incurs much less redundancy than replication [32], it triggers a significant amount of transferred data in failure repair. For example, RS codes retrieve k blocks to repair a lost block. Thus, many studies focus on the *repair problem* [9], which aims to minimize the repair bandwidth. Regenerating codes [8] are special erasure codes that build on network coding [6] and provably achieve the optimal trade-off between repair bandwidth and storage redundancy, by allowing non-failed nodes to encode their stored data during repair. In contrast, *our work applies network coding to storage scaling, which fundamentally differs from the repair problem.*

B. Scaling

We consider two types of scaling, namely *scale-out*, in which new nodes are added to the storage system, and *scale-in*, in which existing nodes are removed from the storage system:

- **(n, k, s) -scaling (i.e., scale-out):** For any $s > 0$, we transform (n, k) -coded blocks in n 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.
- **$(n, k, -s)$ -scaling (i.e., scale-in):** For any $s > 0$, we transform (n, k) -coded blocks in n nodes into $(n - s, k - s)$ -coded blocks that will be stored in the $n - s$ surviving nodes.

Goal: Our goal is to minimize the scaling bandwidth, defined as the amount of transferred data during the scaling operation, while preserving all properties P1–P4 as stated below.

- **P1 (MDS):** The new coded stripe remains MDS, while tolerating the same number of $n - k$ failures as the original (n, k) -coded stripe.
- **P2 (Systematic):** The original data blocks are kept in the new coded stripe after scaling.
- **P3 (Uniform data and parity distributions):** The respective proportions of data and parity blocks across multiple stripes are evenly distributed across nodes before and after scaling.

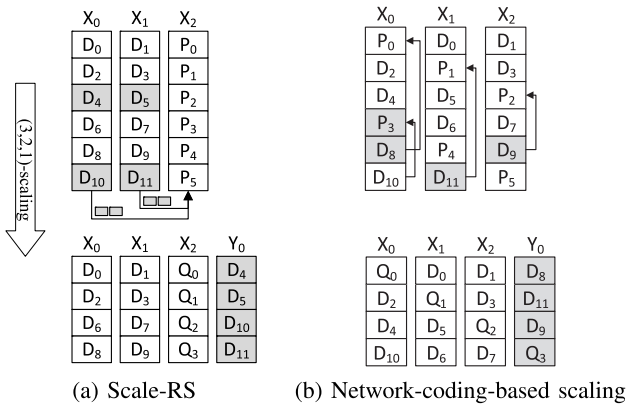


Fig. 1. Scale-RS vs. network-coding-based scaling in (3, 2, 1)-scaling (i.e., from the (3, 2) code to the (4, 3) code). Scale-RS needs to transfer a total of eight blocks to X_2 and Y_0 , while network-coding-based scaling transfers only four blocks to Y_0 . Note that blocks in each node need not be stored in a contiguous manner in distributed storage systems.

- **P4 (Decentralized scaling):** The scaling operation can be done without involving a centralized entity for coordination.

The scaling problem focuses on minimizing the scaling bandwidth subject to the MDS property (i.e., P1), and includes the properties P2–P4 in practical storage deployment with the following implications: P2 always keeps original data blocks, such that the read operations can directly access them without encoding/decoding; P3 ensures that the parity updates are load-balanced across nodes, assuming a uniform access pattern; P4 eliminates any single point of failure or bottleneck in scaling.

We fix the number of tolerable failures (i.e., $n - k$) before and after scaling, as in existing scaling approaches for RAID (e.g., [34], [35], [40]) and distributed storage (e.g., [16], [38]), and we do not consider the variants of the scaling problem for varying $n - k$.

We address the scaling problem via network coding. We motivate this via a scale-out example of (3, 2, 1)-scaling in Figure 1. Let X_i be the i^{th} existing node of a stripe before scaling, where $0 \leq i \leq n - 1$, and Y_j be the j^{th} new node after scaling, where $0 \leq j \leq s - 1$. 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 $*$.

We first consider Scale-RS [16] (Figure 1(a)), which applies scaling to RS 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_4 , D_5 , D_{10} , and D_{11} are relocated to the new node Y_0 .

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_4 , D_5 , D_{10} , and D_{11} are used to compute the parity delta blocks, which are then sent to node X_2 , where the parity blocks are stored. X_2 forms the new parity blocks Q_0 , Q_1 , Q_2 , and Q_3 , respectively. In this example, Scale-RS needs to transfer *eight blocks*. Note that property P3 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, 1) before scaling, each parity block is computed as the XOR operations of the data blocks in the same row as in RAID-5 [23]; for example, $P_0 = D_0 \oplus D_1$, where \oplus is the XOR operator; 2) during scaling, X_0 can locally compute the new parity block $Q_0 = P_0 \oplus D_8$. Similarly, X_1 and X_2 can compute the new parity blocks $Q_1 = P_1 \oplus D_{11}$ and $Q_2 = P_2 \oplus D_9$, respectively. Also, X_0 also locally computes $Q_3 = P_3 \oplus D_{10}$. Now, the scaling process relocates D_8 , D_9 , D_{11} , and the locally computed Q_3 to the new node Y_0 . Thus, we now only need to transfer *four blocks*, and this amount is provably minimum (Section III). In addition, all properties P1–P4 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 and parity blocks*; this is feasible as each storage node stores both data and parity blocks due to the property of uniform data and parity distributions (i.e., P3). 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 follows the information mixing nature of network coding [6]. In addition, the decentralized scaling property (i.e., P4) lets each storage node compute the encoded outputs without the help of a centralized entity. This follows the relay-node-encoding feature of network coding, and allows us to model the storage nodes as relay nodes in the information flow graph analysis (Section III).

C. Discussion of Existing Work on Properties P1–P4

Table I shows how existing scaling approaches (see Section IX for details) address the properties P1–P4. FastScale [45], GSR [34], MDS-Frame [35], and RS6 [40] are designed for RAID arrays rather than distributed storage systems. Thus, for P1, they either provide no fault tolerance or are tolerable against at most two failures; for P4, they rely on the RAID controller to download all original data blocks for computing the new parity blocks. Wu *et al.* [38] apply scaling to Cauchy RS codes [7] and do not consider P4, as they use a centralized node for controlling and downloading data blocks for parity updates during scaling. Rai *et al.* [27] only provide functional code constructions (i.e., all blocks are in coded form) in the theoretical context, so both P2 and P3 are violated. Scale-RS [16] only focuses on a RAID-4-like layout, in which the data and parity blocks reside in dedicated nodes, so P3 is violated. In particular, Scale-RS requires the nodes that store parity blocks to download data blocks for computing new parity blocks, thereby incurring substantial scaling bandwidth. In contrast, NCScale satisfies all the properties P1–P4.

In addition, we compare the scaling bandwidth for different schemes. FastScale [45] achieves the minimum scaling bandwidth in RAID-0 (i.e., no parity blocks are involved). The schemes that do not satisfy P4 (i.e., [34], [35], [38], [40]) need to upload and download data blocks for computing parity blocks via the RAID controller or a centralized node, so they incur at least twice the amount of transferred data (i.e., at least twice the minimum scaling bandwidth). Rai *et al.* [27] treat the scaling problem as the repair problem, but the two

TABLE I
COMPARISONS OF EXISTING SCALING PROPOSALS

	P1	P2	P3	P4	Scaling bandwidth
FastScale [45]	No, RAID-0	Yes	Yes	No	Optimal
GSR [34]	Yes, only one failure tolerable	Yes	Yes	No	$\geq 2\times$ optimal
MDS-Frame [35] RS6 [40]	Yes, only two failures tolerable	Yes	Yes	No	$\geq 2\times$ optimal
Wu et al. [38]	Yes, any $n-k$ failures tolerable	Yes	Yes	No	$\geq 2\times$ optimal
Rai et al. [27]	Yes, any $n-k$ failures tolerable	No	No	Yes	–
Scale-RS [16]	Yes, any $n-k$ failures tolerable	Yes	No	Yes	One more block per stripe than NCScale
NCScale	Yes, any $n-k$ failures tolerable	Yes	Yes	Yes	Optimal when $n-k=1$; $n-k-1$ more blocks per stripe than optimal when $n-k > 1$

problems are inherently different (Section I). The study [27] does not formally prove the optimality of scaling or provide the minimum scaling bandwidth (see Section IX for details). Scale-RS [16] always transfers one more block per stripe during scaling than NCScale (Section VII), while NCScale achieves the minimum scaling bandwidth when $n-k=1$ and has $n-k-1$ more blocks per stripe than the minimum scaling bandwidth when $n-k > 1$ (Corollary 3).

III. MODEL

We analyze both (n, k, s) -scaling and $(n, k, -s)$ -scaling using the information flow graph model [6], [8]. We derive their respective lower bounds of scaling bandwidth, and show that the lower bounds are tight by proving that there exist random linear codes whose scaling bandwidth matches the respective lower bounds for general (n, k, s) and $(n, k, -s)$. Note that random linear codes are non-systematic (i.e., P2 is violated). In Sections IV and V, we address all P1–P4 in our design.

A. Model for (n, k, s) -Scaling

We first consider (n, k, s) -scaling. To comply with the information flow graph model in the literature (e.g., [8]), 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 ($0 \leq i \leq n-1$) encodes its stored data of size $\frac{M}{k}$ into some encoded data.
- 2) Each new node Y_j ($0 \leq j \leq s-1$) downloads the encoded data from each X_i ($0 \leq i \leq n-1$).
- 3) Each existing node X_i ($0 \leq i \leq n-1$) removes $\frac{M}{k} - \frac{M}{k+s}$ units of its stored data and only stores data of size $\frac{M}{k+s}$.
- 4) Each new node Y_j ($0 \leq j \leq s-1$) 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 $k+s$ nodes.

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

1) 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 ($0 \leq i \leq n-1$) 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 ($0 \leq j \leq s-1$) 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 .

2) Edges in \mathcal{G} :

- We add a directed edge $S \rightarrow X_i^{in}$ for every i ($0 \leq i \leq n-1$) with an infinite capacity for data distribution.
- We add a directed edge $X_i^{mid} \rightarrow Y_j^{in}$ for every i ($0 \leq i \leq n-1$) and j ($0 \leq j \leq s-1$) with capacity β .
- We select any $k+s$ 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: For (n, k, s) -scaling, β must be at least $\frac{M}{n(k+s)}$.

Proof: Clearly, each new storage node Y_j ($0 \leq j \leq s-1$) must receive at least $\frac{M}{k+s}$ units of data from all existing storage nodes X_i 's ($0 \leq i \leq n-1$) over the links with capacity β each. Thus, we have $n\beta \geq \frac{M}{k+s}$. The lemma follows. \square

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: For (n, k, s) -scaling, 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 $i \in [0, n-1]$;
- *Type 2:* Only X_i^{out} is in $\bar{\mathcal{C}}$ for some $i \in [0, n-1]$;
- *Type 3:* Only Y_j^{out} is in $\bar{\mathcal{C}}$ for some $j \in [0, s-1]$; and
- *Type 4:* Both Y_j^{in} and Y_j^{out} are in $\bar{\mathcal{C}}$ for some $j \in [0, s-1]$.

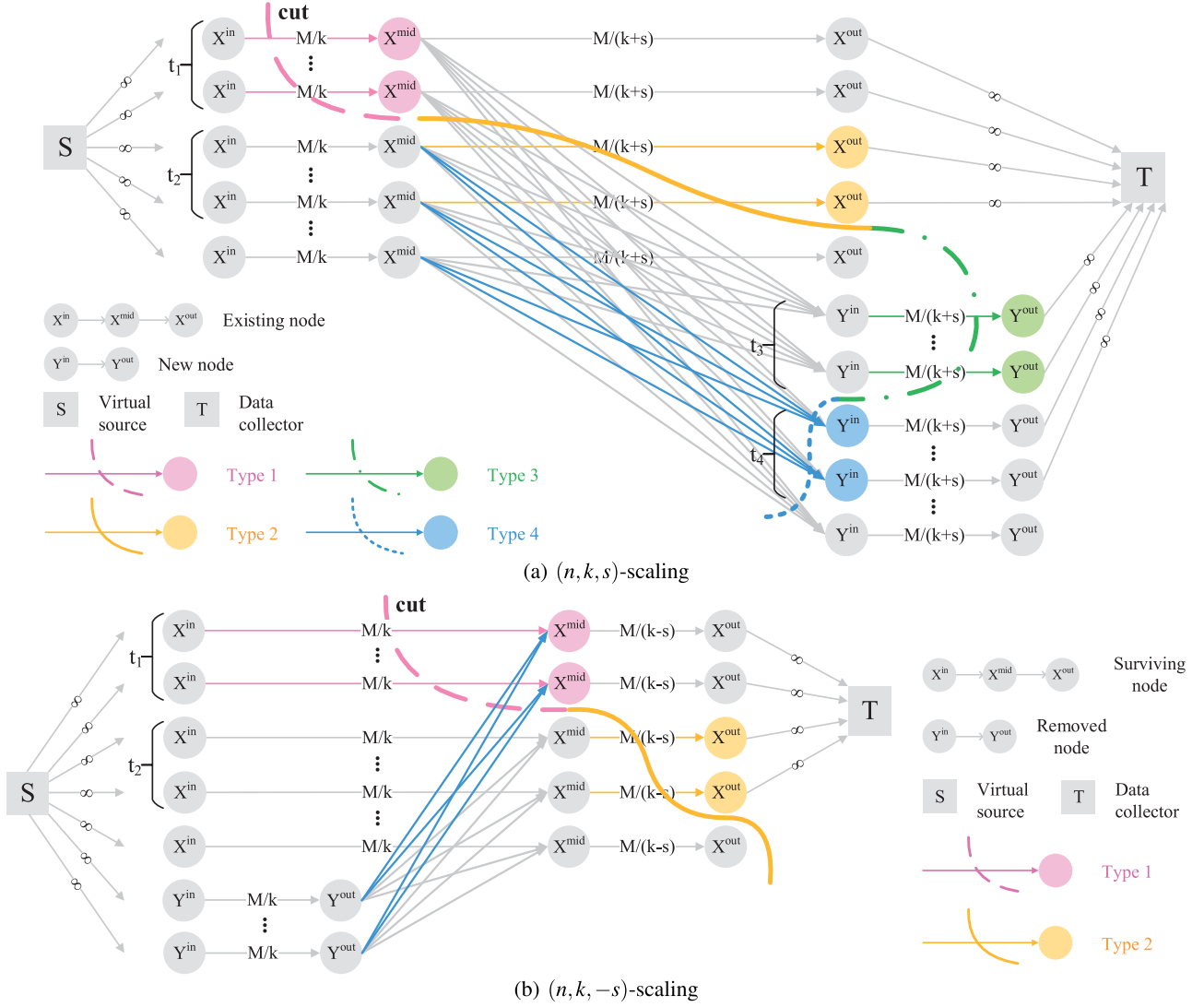


Fig. 2. Information flow graphs for (n, k, s) -scaling and $(n, k, -s)$ -scaling.

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:

$$t_1 + t_2 + t_3 + t_4 = k + s. \quad (3)$$

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(a) illustrates the details. Thus, we have:

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

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

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

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. \square

Lemma 3 [8]: If the capacity of each possible min-cut of the flow graph 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. \square

Theorem 1 implies that the minimum scaling bandwidth per new stripe is $n \cdot s \cdot \beta = \frac{s \cdot M}{k+s}$, i.e., optimal scaling occurs when the amount of transferred data to the new nodes is equal to the size of the data being stored in the s new nodes. Since each new node stores one block, optimal scaling requires s blocks for each stripe of size being scaled from n blocks to $n + s$ blocks during (n, k, s) -scaling. 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 $(n + s, k + s)$ -coded stripe for (n, k, s) -scaling.

Note that the above corollary presents the scaling bandwidth per stripe instead of the total scaling bandwidth, so as to show easily whether a scaling design is optimal and support fair comparisons in our numerical analysis (Section VII).

B. Model for $(n, k, -s)$ -Scaling

For $(n, k, -s)$ -scaling, we also start with an (n, k) -coded data file of size M as in Section III-A. The main difference here is that scale-in transfers data among removed nodes and surviving nodes, while scale-out transfers data between existing nodes and new nodes. Let X_i be the i^{th} surviving node of a stripe, where $0 \leq i \leq n - s - 1$, and Y_j be the j^{th} removed node of a stripe, where $0 \leq j \leq s - 1$. Then the $(n, k, -s)$ -scaling process for the data file can be decomposed into four steps:

- 1) Each removed node Y_j ($0 \leq j \leq s - 1$) encodes its stored data of size $\frac{M}{k}$ into some encoded data.
- 2) Each surviving node X_i ($0 \leq i \leq n - s - 1$) downloads the encoded data from each Y_j ($0 \leq j \leq s - 1$).
- 3) Each removed node Y_j ($0 \leq j \leq s - 1$) removes all of its stored data.
- 4) Each surviving node X_i ($0 \leq i \leq n - s - 1$) encodes its stored data before scaling and its downloaded data during scaling into the stored data of size $\frac{M}{k-s}$ after scaling.

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

We now construct an information flow graph \mathcal{G} for $(n, k, -s)$ -scaling (Figure 2(b)). Note that scale-in transfers data between removed nodes and surviving nodes, so we add a directed edge $Y_j^{\text{out}} \rightarrow X_i^{\text{mid}}$ for every i ($0 \leq i \leq n - s - 1$) and j ($0 \leq j \leq s - 1$) with capacity β . The following lemma states the necessary condition of the lower bound of β .

Lemma 4: For $(n, k, -s)$ -scaling, β must be at least $\frac{M}{k(k-s)}$.

Proof: Similar to Lemma 1, each surviving storage node X_i ($0 \leq i \leq n - s - 1$) must receive at least $\frac{M}{k-s} - \frac{M}{k}$ units of data from all removed storage nodes Y_j 's ($0 \leq j \leq s - 1$) over the links with capacity β each. Thus, we have $s\beta \geq \frac{Ms}{k(k-s)}$. The lemma follows. \square

Similar to the scale-out case, the lower bound in Lemma 4 is tight, achievable by random linear network coding (Lemma 3). We can deduce the following theorem.

Theorem 2: 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 $\frac{M}{k(k-s)}$ while the MDS property of tolerating any $n - k$ failures is preserved.

Theorem 2 implies that optimal scaling occurs when the amount of transferred data to the surviving nodes is equal to the size of the data being removed of the s removed nodes. Thus, we have the following corollary.

Corollary 2: For distributed storage systems that organize data in fixed-size blocks, the minimum scaling bandwidth is s blocks per new (n, k) -coded stripe for $(n, k, -s)$ -scaling.

C. Discussion

We discuss the relevance of our analysis in this section with the properties P1–P4. Theorems 1 and 2 ensure that

the scaling process under random linear coding maintains the MDS property (i.e., P1 is satisfied) and can be performed without a centralized entity (i.e., P4 is satisfied). However, random linear coding does not keep the original data blocks, so both P2 and P3 are violated. Thus, we proceed to design explicit coding schemes (Sections IV and V) that aim to minimize the scaling bandwidth, while satisfying P1–P4.

IV. NCSCALE: SCALE-OUT

We present NCScale, a distributed storage system that realizes network-coding-based storage scaling for both scale-out and scale-in cases. In this section, we focus on scale-out in NCScale, which satisfies P1–P4 (Section II-B) with the goal of achieving the minimum scaling bandwidth (Section III-A).

A. Main Idea

NCScale operates on (systematic) RS codes [28], such that all blocks before and after scaling are still encoded by RS codes. Before scaling, each existing 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 (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 (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 [26], so as

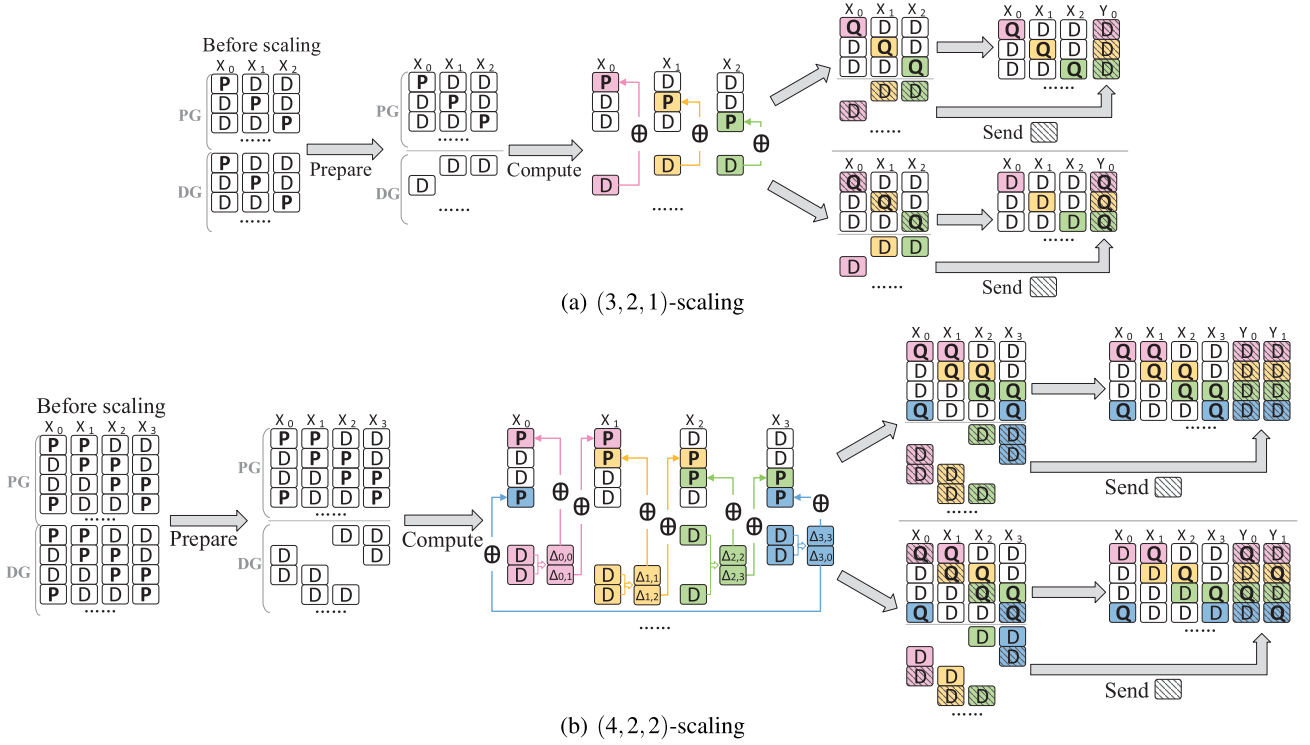


Fig. 3. Scale-out in NCScale. Note that (3, 2, 1)-scaling achieves the minimum scaling bandwidth (Corollary 1), while (4, 2, 2)-scaling does not.

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 $X_i, \dots, X_{(i+n-k-1) \bmod n}$ for some $w \geq 0$ 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 above $n(k + s)(n + s)$ stripes into two groups. The first group is denoted by **PG**. It contains the first $nk(n + s)$ stripes, whose parity blocks will be updated to new parity blocks based on parity delta blocks. The second group is denoted by **DG**. It contains the remaining $ns(n + s)$ stripes, whose data blocks will be used to generate parity delta blocks for updating the parity blocks in the first group **PG**. Note that the number of stripes in **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 (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 $0 \leq i, j \leq n - 1$ (when $i = j$, the new parity block is computed locally). NCScale ensures that for each of the $nk(n + s)$ stripes in **PG**, the $n - k$ parity blocks of 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:

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

Algorithm 1 Prepare

- 1: **PG** = first $nk(n + s)$ stripes
- 2: **DG** = next $ns(n + s)$ stripes
- 3: **for** $w = 0$ to $nk(n + s) - 1$ **do**
- 4: $\mathbf{D}_w = w^{\text{th}}$ set of s data blocks of $X_{w \bmod n}$ in **DG**, where $w' = \lceil \frac{w}{n} \rceil$
- 5: **end for**

C. Algorithmic Details

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

1) *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 **PG** and **DG** (lines 1-2). It then divides the data blocks in **DG** into different sets \mathbf{D}_w 's, where $0 \leq w \leq nk(n + s) - 1$ (lines 3-5), by collecting and adding s data blocks from X_0 to X_{n-1} into \mathbf{D}_w in a round-robin fashion. Specifically, **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 ($0 \leq i \leq n - 1$) in **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).

2) *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 **PG**. To compute the new parity blocks, each existing node X_i ($0 \leq i \leq n - 1$) operates on the w^{th} stripe for $i = w \bmod n$ (line 2). Recall that the parity blocks are stored in $X_i, \dots, X_{(i+n-k-1) \bmod n}$. For $0 \leq j \leq n - k - 1$ and $j' = (j + w) \bmod n$, X_i computes a parity delta

Algorithm 2 Compute, Send, and Delete

```

1: for  $w = 0$  to  $nk(n + s) - 1$  do
2:    $i = w \bmod n$ 
3:   for  $j = 0$  to  $n - k - 1$  do
4:      $j' = (j + w) \bmod n$ 
5:      $X_i$  generates  $\Delta_{i,j'}$  from the  $s$  data blocks in  $\mathbf{D}_w$ 
       for the  $j^{\text{th}}$  parity block in the  $w^{\text{th}}$  stripe of  $\mathbf{PG}$ 
6:      $X_i$  sends  $\Delta_{i,j'}$  to  $X_{j'}$ , which adds  $\Delta_{i,j'}$  to the  $j^{\text{th}}$ 
       parity block in the  $w^{\text{th}}$  stripe of  $\mathbf{PG}$ 
7:   end for
8:   if  $w \leq nk(n - s(n - k - 1)) - 1$  then
9:      $X_i$  sends all  $s$  data blocks in  $\mathbf{D}_w$  to the  $s$  new nodes
10:  else
11:     $X_i$  sends the locally updated parity block
       and any  $s - 1$  data blocks in  $\mathbf{D}_w$  to the  $s$ 
       new nodes
12:  end if
13:   $X_i$  deletes all obsolete blocks
14: end for

```

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 $j = 0$, 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 $w \leq nk(n - s(n - k - 1)) - 1$, 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 (Section IV-D).

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

D. Proof of Correctness

Theorem 3: NCScale preserves P1–P4 after scaling.

Proof: See Appendix A in the digital supplementary file.

Corollary 3: In (n, k, s) -scaling, when $n - k = 1$, NCScale achieves the lower bound in Corollary 1; when $n - k > 1$, the gap between the lower bound and NCScale is $n - k - 1$ blocks per new stripe.

V. NCSALE: SCALE-IN

We extend NCScale for scale-in, in which the generation of parity blocks for the new stripes is different from that in scale-out. We focus on two cases in scale-in: $n - k = 1$ and $n - k > 1$. Both cases still satisfy P1–P4 (Section II-B), with the goal of achieving minimum scaling bandwidth for $n - k = 1$ (Section III-B).

A. Scale-In for $n - k = 1$

When $n - k = 1$, each stripe has one parity block, so it is easy to update the existing parity block of a stripe locally by

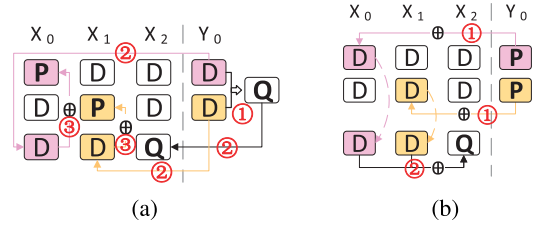


Fig. 4. Steps for $(4, 3, -1)$ -scaling: (a) scaling of stripes that have only data blocks in the removed node; (b) scaling of stripes that have parity blocks in the removed node.

transferring the removed data blocks of the stripe to the node where the parity block resides, while preserving P3 (uniform data and parity distributions). Figure 4 depicts the scale-in steps for $(4, 3, -1)$ -scaling. In particular, NCScale classifies the stripes involved in the scaling process into two cases:

- **Stripes that have only data blocks in the removed node (Figure 4(a)):** NCScale first generates new parity blocks for the new stripes from the data blocks in the removed node (step 1). It then sends these data blocks and their generated parity blocks to the surviving nodes as the new stripes, while these data blocks are transferred to the nodes that have parity blocks within the same existing stripe (step 2). It uses these data blocks to locally update the parity blocks in the existing stripes (step 3).
- **Stripes that have parity blocks in the removed node (Figure 4(b)):** NCScale first sends the existing parity blocks to surviving nodes, in which the data blocks within the existing stripe update these parity blocks locally (step 1). It then generates the new parity blocks for the new stripes by collecting these data blocks that are used to update existing parity blocks (step 2).

NCScale can reduce the scaling bandwidth when $n - k = 1$ significantly. For the stripes in the first case, the new parity block of each new stripe is sent as a coded block that is generated by the data blocks from the same removed node, and the existing parity block of each existing stripe can be updated locally. In other words, the blocks that are sent over the network by NCScale are only those that will be added to the surviving nodes. From Corollary 2, the scaling bandwidth of NCScale matches the optimal point. For the stripes in the second case, the existing parity block of each existing stripe can also be updated locally, but the new parity block of each new stripe is generated from the data blocks residing in other surviving nodes. Nevertheless, the number of data blocks that are transferred for the generation of the new parity blocks remains limited, since the parity blocks only occupy $\frac{1}{n}$ of all blocks of the stripes in the second case.

Appendix B (see the digital supplementary file) elaborates the scale-in process for $n - k = 1$. We can derive the corresponding scaling bandwidth of NCScale per new stripe as:

$$\frac{s((n-s)(n+1)-2)}{nk}. \quad (7)$$

B. Scale-In for $n - k > 1$

When $n - k > 1$, each stripe has more than one parity block, so it becomes difficult for scale-in to update each existing parity block locally while preserving P3. NCScale

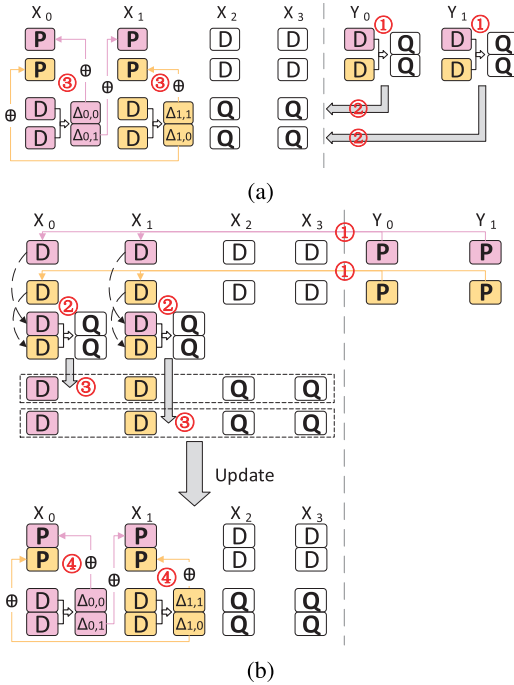


Fig. 5. Steps of (6, 4, -2)-scaling: (a) scaling of stripes that have only data blocks in the removed nodes; (b) scaling of stripes that have parity blocks in the removed nodes.

incurs additional scaling bandwidth to maintain P3, such that some of the existing parity blocks can be updated locally, while the remaining existing parity blocks need to be updated via the parity delta blocks generated from other surviving nodes. Figure 5 depicts the scale-in steps for (6, 4, -2)-scaling.

Before scaling, NCScale identifies the stripes with the same parity layout (i.e., the parity blocks reside in the same nodes), and then performs scale-in operations in a set of stripes with the same parity layout. It classifies the stripes into the following two cases:

- **Stripes that have only data blocks in the removed nodes (Figure 5(a)):** The scaling steps of this case are similar to the first case in scale-in for $n - k = 1$.
- **Stripes that have parity blocks in the removed nodes (Figure 5(b)):** NCScale first sends the existing parity blocks to the surviving nodes (step 1). In the surviving nodes, it uses the data blocks that are to be replaced by the parity blocks to generate new parity blocks for the new stripes (step 2). It sends these data blocks and their generated parity blocks to the surviving nodes as new stripes (step 3). Finally, it generates parity delta blocks from these data blocks to update the existing parity blocks in the existing stripes.

Appendix C (see the digital supplementary file) elaborates the scaling-in process for $n - k > 1$. We can derive the corresponding scaling bandwidth of NCScale per new stripe as (8), as shown at the bottom of the next page:

Similar to Section IV-D, we also provide the following corollary on the tightness of NCScale in $(n, k, -s)$ -scaling.

Corollary 4: In $(n, k, -s)$ -scaling, when $n - k = 1$, the gap between the lower bound in Corollary 2 and NCScale is $\frac{s((n-s)(n+1)-nk-2)}{nk}$ blocks per new stripe; when $n - k > 1$, the gap between the lower bound and NCScale is $\frac{(n-s)((n-k)(nk+sk-2s^2-s)-ks^2-2)-\min(n-k, k-s)((n-s)(k-s)+n)}{nk(n-s)}$ blocks per new stripe.

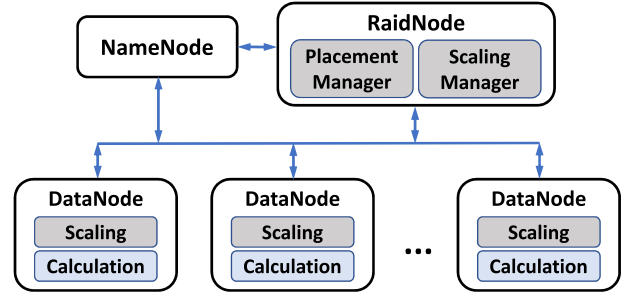


Fig. 6. Architecture of HDFS-RAID integrated with NCScale.

VI. IMPLEMENTATION

We have implemented a prototype of NCScale based on HDFS [30]. In this section, we first provide an overview of HDFS and its extension, called HDFS-RAID [4], for erasure coding. We then describe how we integrate NCScale into HDFS-RAID to support storage scaling for erasure-coded storage.

A. HDFS Overview

HDFS [30] is a widely used distributed storage system in both industrial and academic deployments. It provides reliable storage for massive volumes of data across multiple nodes in a large cluster. It stores each file in units of fixed-size blocks that are replicated for fault tolerance. It consists of two types of nodes: a single *NameNode* that stores metadata and coordinates storage operations, and multiple *DataNodes* that provide actual data storage. HDFS-RAID [4] is an extension of HDFS and employs erasure coding to provide low-redundancy fault tolerance. It introduces a *RaidNode* to perform erasure coding operations, such as encoding, decoding, and repair. To perform encoding for a stripe, the *RaidNode* first obtains the metadata of k data blocks from the *NameNode*, and uses the metadata to collect the k data blocks from k different *DataNodes* to generate the parity blocks that are then written to HDFS.

B. Integration of NCScale Into HDFS-RAID

We implement a NCScale prototype atop Facebook's HDFS-RAID [3]. Figure 6 depicts the HDFS-RAID architecture with NCScale. Our NCScale prototype is written in Java, with around 7.5K LoC. Specifically, we implement NCScale as four distinct modules, as described below.

1) *Placement Manager Module:* We implement this module to manage block placements, in which the distributions of data blocks and parity blocks are uniform before scaling (Section IV-B). Since the original HDFS-RAID only supports random distributions of data and parity blocks, we realize the placement algorithms for RS codes in the placement manager module, which is now added to the *RaidNode*. When the data blocks are encoded into parity blocks, the *RaidNode* will call the placement manager module to choose a specific *DataNode* for each block, which is then sent to the *DataNode*. Thus, the placement manager module ensures a uniform placement for the scaling operation.

2) *Scaling Manager Module:* We implement this module in the *RaidNode* to manage scaling operations. When a scaling operation is triggered, the scaling manager module chooses the

corresponding scaling algorithm based on different parameter settings. It then obtains metadata from the NameNode for the collection of blocks that are involved in a scaling operation from NameNode. It sends the metadata and the scaling type to the DataNodes via sockets, such that all the DataNodes can perform the scaling operation in parallel.

3) *Scaling Module*: We implement this module in each DataNode to perform a scaling operation. It uses the metadata from the RaidNode to read the locally stored blocks, and then uses these blocks to perform the scaling operation. The scaling module exports three APIs for a scaling operation: (i) *Compute*, which computes new parity blocks or parity delta blocks from its locally stored data blocks and received data blocks by calling the calculation module (see below); (ii) *Send*, which sends data blocks, parity blocks, and parity delta blocks to another DataNode via sockets; and (iii) *Delete*, which deletes all obsolete blocks. After a scaling operation, the scaling module updates the metadata of the NameNode and notifies the RaidNode of the completion of the scaling operation.

4) *Calculation Module*: We implement this module in each DataNode to perform encoding and decoding operations. We realize the calculation module in C++ based on Intel's Intelligent Storage Acceleration Library (ISA-L) [5]. We link the calculation module with HDFS (written in Java) via the Java Native Interface. We mainly use two ISA-L APIs: *ec_init_tables* to initialize coding coefficients, and *ec_encode_data* to execute coding operations.

VII. NUMERICAL ANALYSIS

We present numerical analysis results of NCScale. We compare it with Scale-RS [16], which represents the state-of-the-art scaling scheme for RS codes in distributed storage systems. 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.

A. Scale-Out

We first consider the scale-out case. Specifically, we consider the scaling bandwidth of three different schemes.

- *Optimal*: The information-theoretically minimum scaling bandwidth is given by s blocks (per new stripe) for any (n, k) (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 (e.g., Figure 1(a)). 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 7 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

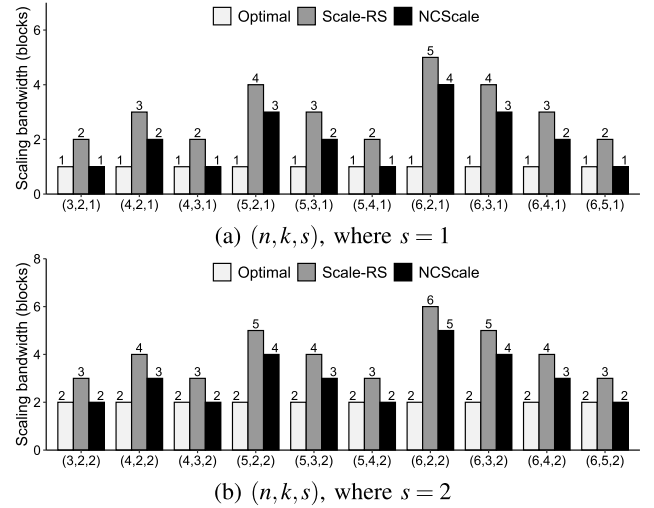


Fig. 7. Scale-out: numerical results of scaling bandwidth (in units of blocks) per new stripe formed after scaling.

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).

B. Scale-In

We now consider the scale-in case. Similar to scale-out, we consider the optimal scheme, Scale-RS, and NCScale as follows.

- *Optimal*: Since there will be $\frac{M}{k-s}$ new stripes after scaling for a data file of size M , the information-theoretically minimum scaling bandwidth is given by $\frac{s(n-s)}{k}$ blocks (per new stripe) for any (n, k) (Corollary 2).
- *Scale-RS*: From the Scale-RS [16], we derive the scaling bandwidth as follows. To form k new stripes after scaling, Scale-RS first sends $s(n-s)$ data blocks and new parity blocks from s removed nodes to $n-s$ surviving nodes as new stripes, followed by $(k-s)(n-k)$ parity delta blocks for parity block updates. Thus, the scaling bandwidth of Scale-RS is $\frac{s(n-s) + (k-s)(n-k)}{k}$ blocks (per new stripe).
- *NCScale*: From Equations (7) and (8), the scaling bandwidth of NCScale per new stripe is: (i) $\frac{s((n-s)(n+1)-2)}{nk}$ blocks for $n - k = 1$, and (ii) the equation can be derived, as shown at the bottom of next page blocks for $n - k > 1$.

Figure 8 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 $n - k = 1$ and small s . For example, for $(7, 6, -1)$, the reduction is 38.9%, while for $(7, 5, -1)$ and $(7, 6, -2)$, the reduction is 7.1% and 21.7%, respectively. NCScale nears the optimal point when $n - k = 1$. Nevertheless, NCScale always has less scaling bandwidth than Scale-RS.

$$\frac{(n-s)((n-k)(nk+sk-2s^2-s)+sk(n-s)-2)-\min(n-k, k-s)((n-s)(k-s)+n)}{nk(n-s)} \quad (8)$$

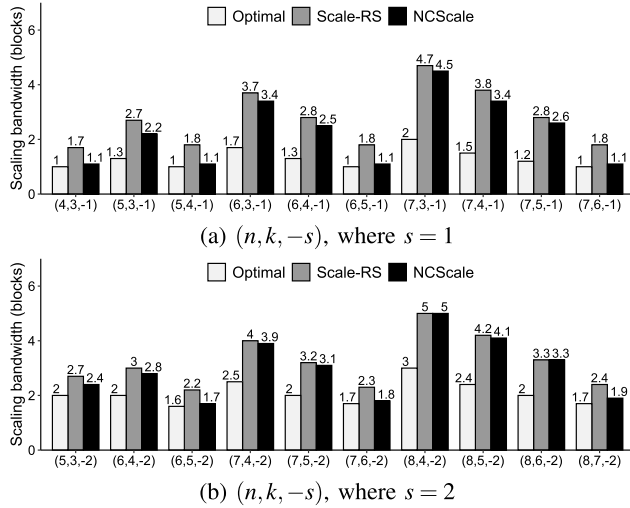


Fig. 8. Scale-in: numerical results of scaling bandwidth (in units of blocks) per new stripe formed after scaling.

VIII. PERFORMANCE EVALUATION

In this section, we present performance evaluation results of NCSCALE. We 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. Setup

We implemented NCSCALE as an extension to HDFS-RAID [4] (Section VI) and evaluated its scaling performance in real-world environments. We also implemented Scale-RS as an extension to HDFS-RAID for fair comparisons under the same implementation settings. In both of our NCSCALE and Scale-RS implementations, the storage nodes perform the scaling steps independently and in parallel.

Note that for the scale-out in Scale-RS, the parity nodes do not participate in the scale-out process, and hence Scale-RS cannot fully utilize the resources of all nodes. Thus, we rotate the parity placements for the scale-out of Scale-RS to involve all nodes in the scale-out process. For the scale-in of Scale-RS, it cannot handle the situation in which the parity blocks reside in the removed nodes. Thus, we follow the original Scale-RS [16] to organize all data blocks and parity blocks in a RAID-4-like layout (i.e., all the parity blocks reside in dedicated nodes), and require that the removed nodes must be the nodes that hold data blocks. However, the RAID-4-like layout reduces the scale-in performance of Scale-RS (see Experiment 2).

1) *Testbed*: 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. For scale-out, each instance represents an existing storage node (before scaling) or a new storage node (after scaling), while for scale-in, each instance represents a

TABLE II

EXPERIMENT 1: TIME BREAKDOWN OF NCSCALE FOR $(n, k, s) = (6, 4, 2)$ (SCALE-OUT) AND $(n, k, -s) = (8, 6, -2)$ (SCALE-IN); BLOCK SIZE IS 64 MB

	$(n, k, s) = (6, 4, 2)$			$(n, k, -s) = (8, 6, -2)$		
	Compute	Send	Delete	Compute	Send	Delete
200 Mb/s	14.99s	1654.15s	1.34s	31.70s	2568.07s	1.29s
500 Mb/s	14.96s	694.39s	1.34s	32.04s	1104.63s	1.32s
1 Gb/s	14.43s	378.43s	1.32s	31.92s	620.77s	1.31s
2 Gb/s	14.31s	222.70s	1.33s	33.98s	390.82s	1.32s

surviving node or a removed node. 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 `tcc` 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.

2) *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 (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.

B. Results

Experiment 1 (Time Breakdown): We provide a breakdown of the scaling time and identify the bottlenecked step in scaling. We decompose a scaling operation into three steps that are carried out by existing nodes: (i) *compute*, which refers to the computation of new parity blocks or parity delta blocks, (ii) *send*, which refers to the transfers of blocks 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. 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 results. We first consider scale-out, and focus on $(6, 4, 2)$ -scaling. We observe that the send time dominates (over 93% 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. We next consider scale-in, and focus on $(8, 6, -2)$ -scaling. We again observe that the send time also dominates (over 91% of the overall time), especially under limited available network bandwidth.

Experiment 2 (Impact of Bandwidth): We compare NCSCALE and Scale-RS under different gateway bandwidth settings. We first consider scale-out. Figure 9 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

$$\frac{(n-s)((n-k)(nk+sk-2s^2-s)+sk(n-s)-2)-\min(n-k, k-s)((n-s)(k-s)+n)}{nk(n-s)}$$

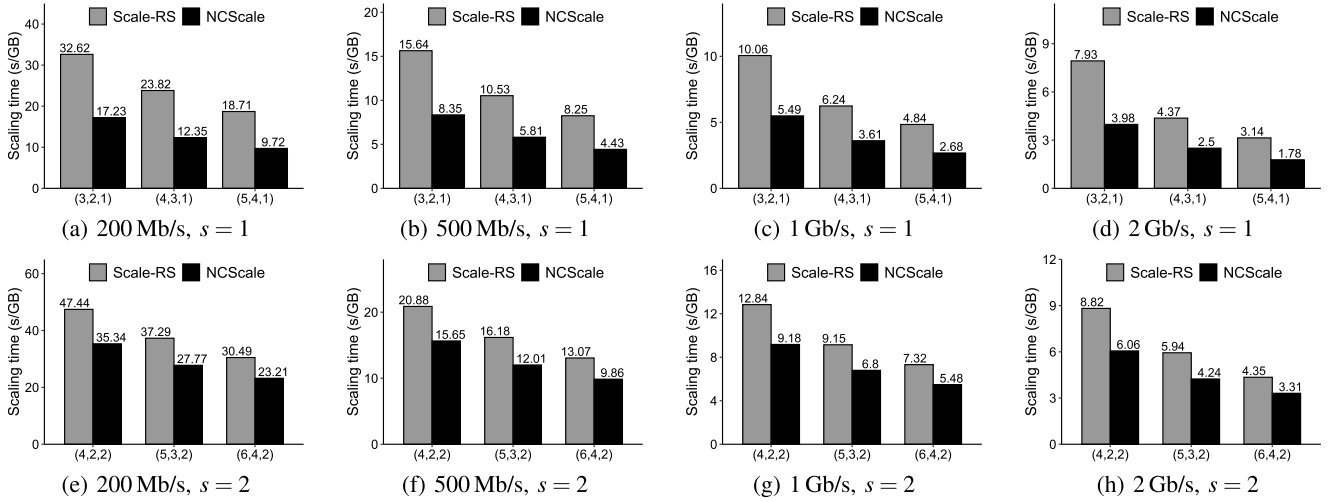


Fig. 9. Experiment 2 (Scale-out): Scaling time (per GB of data blocks), in seconds/GB, under different gateway bandwidth settings.

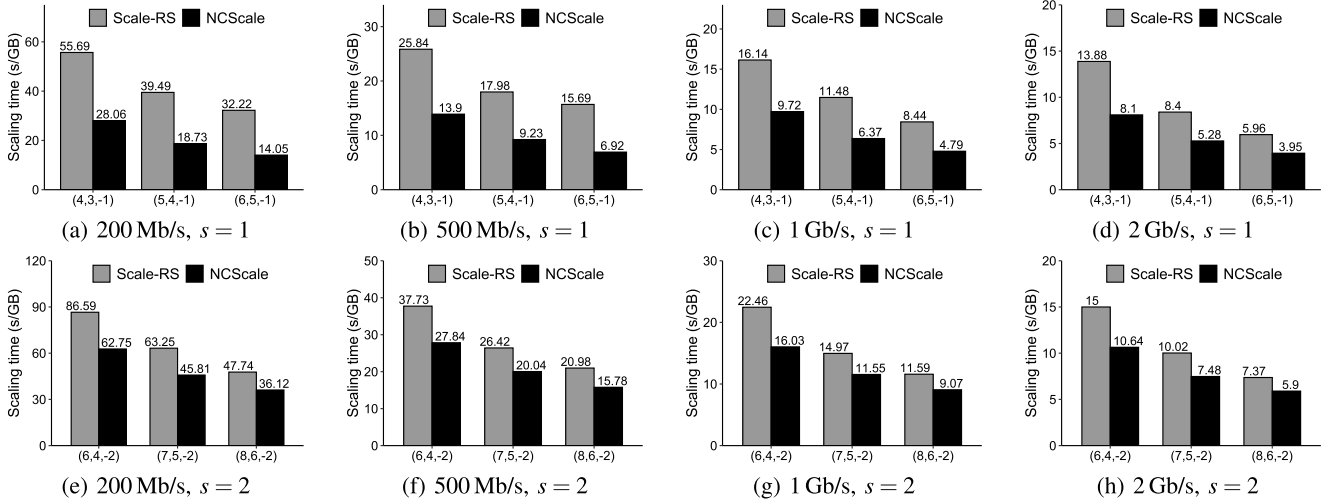


Fig. 10. Experiment 2 (Scale-in): Scaling time (per GB of data blocks), in seconds/GB, under different gateway bandwidth settings.

(Figure 7) 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 (Figure 7), NCScale incurs 25.0% less scaling bandwidth than Scale-RS (i.e., three versus four blocks, respectively), while in our experiment, Scale-RS incurs 23.88%, 24.56%, 25.14%, and 23.90% more scaling time than NCScale when the gateway bandwidth is 200 Mb/s, 500 Mb/s, 1 Gb/s, and 2 Gb/s, respectively (Figures 9(e)-9(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.

We also compare NCScale and Scale-RS in scale-in. Figure 10 shows the scaling time results, in which the block size is fixed as 64 MB. Unlike scale-out, whose empirical results are consistent with the numerical ones, the empirical results of scale-in are higher than the numerical ones (Figure 8) in all cases. In particular, the scale-in of Scale-RS is implemented in a RAID-4-like layout where the parity nodes of Scale-RS do not participate the scaling process (Section VIII-A), thereby leading to performance drops. Take $(8, 6, -2)$ -scaling for example. From the numerical results (Figure 8), NCScale has almost identical scaling bandwidth

to Scale-RS, while our experiment shows that Scale-RS incurs 24.34%, 24.79%, 21.74%, and 19.95% more scaling time than NCScale when the gateway bandwidth is 200 Mb/s, 500 Mb/s, 1 Gb/s, and 2 Gb/s, respectively (Figures 10(e)-10(h)). Similar to scale-out, the scaling time increases with the redundancy $\frac{n}{k}$ (e.g., $(4, 3, -1)$ has higher scaling time than $(5, 4, -1)$ and $(6, 5, -1)$). 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.

Figures 11(a) and 11(b) show the results for scale-out. 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. Figures 11(c) and 11(d) show the results for scale-in. The observations are similar as in scale-out.

Experiment 4 (Impact of s): Finally, we study the scaling time versus s (the number of new nodes). 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 [22].

Figure 12(a) shows the results for scale-out. Both NCScale and Scale-RS need to transfer more blocks as s increases,

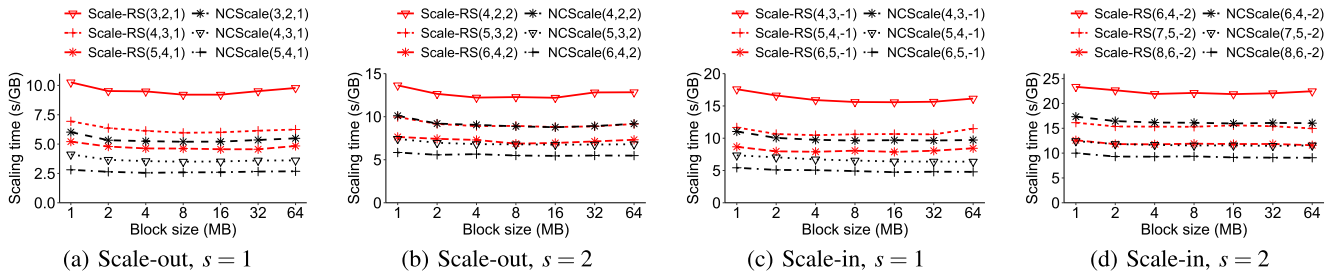


Fig. 11. Experiment 3: Scaling time (per GB of data blocks), in seconds/GB, versus block size.

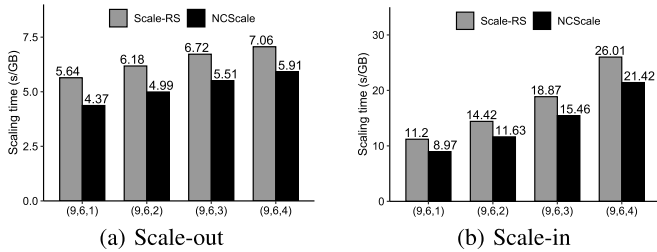


Fig. 12. Experiment 4: Scaling time (per GB of data blocks), in seconds/GB, versus s .

and the difference of their scaling times decreases. Overall, NCScale reduces the scaling time of Scale-RS by 16.3-22.5%. Figure 12(b) shows the results for scale-in. Again, the performance trends are similar as in scale-out, and NCScale reduces the scaling time of Scale-RS by 17.65-19.91%.

IX. RELATED WORK

Scaling approaches have been proposed for RAID arrays, including RAID-0 (i.e., no fault tolerance) [41], [45], RAID-5 (i.e., single fault tolerance) [12], [34], [42], [43], and RAID-6 [35], [36], [40] (i.e., double fault tolerance). Such scaling approaches focus on minimizing data block migration and parity block updates (e.g., GSR [34] for RAID-5, and MDS-Frame [35] and RS6 [40] 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 [16], which addresses the scaling problem in distributed storage systems that employ RS codes [28] to provide tolerance against a general number of failures. Wu *et al.* [38] apply scaling for Cauchy RS codes [7], 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 [33] leverages access patterns to switch between replication for hot data and RAID-5 for cold data. DiskReduce [10] and EAR [19] address the transition replication to erasure coding in HDFS [30]. HACFS [39] extends HDFS to support switching between two erasure codes to trade between storage redundancy and access performance. Ring [31] and Elastic RS codes [37] address the transitions between redundancy schemes for in-memory key-value stores, and mitigate I/O costs by decoupling block-to-node mappings.

On the theoretical side, Rai *et al.* [27] present adaptive erasure codes for switching between the erasure coding parameters (n, k) , and attempt to apply network coding to storage scaling. However, the study [27] does not provide any formal information flow graph analysis. Instead, it treats the scaling problem as the repair problem and solves the scaling problem with regenerating codes [8] (which achieve optimal repair). However, both the repair and scaling problems are different, as the scaling problem changes the erasure coding parameters (e.g., k increases for scale-out and decreases for scale-in), while the repair problem keeps the erasure coding parameters unchanged. Thus, the amount of stored data in each node (i.e., M/k) changes after scaling, but remains unchanged after repair. This fundamental difference implies that the optimal scaling problem cannot be directly mapped to the optimal repair problem as in [27]. In contrast, our work is the first formal study on applying network coding to storage scaling and presents the formal information flow graph analysis on the minimum scaling bandwidth that is achievable by random linear codes. Furthermore, NCScale addresses the practical perspective by presenting systematic code constructions and prototype evaluation, both of which are not addressed in [27]. Maturana *et al.* [20], [21] study *code conversion*, which is a similar problem that involves changing n and k in a distributed storage system. The studies [20], [21] focus on access optimality that minimizes disk accesses, while we focus on minimizing network bandwidth.

To summarize, we show how network coding can help achieve the optimality of storage scaling in terms of network bandwidth, using both analysis and implementation. Our follow-up work [14] extends the theoretical analysis of the paper for more parameters by generalizing (n', k') from the condition $n' - k' = n - k$ (in this work) to any (n', k') (in [14]). Realizing the findings of [14] in NCScale is our future work.

X. CONCLUSION

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.

REFERENCES

[1] Amazon Elastic Compute Cloud (EC2). Accessed: Aug. 2021. [Online]. Available: <http://aws.amazon.com/ec2/>
 [2] Corrupted Fragment on Decode #10. Accessed: Aug. 2021. [Online]. Available: <https://github.com/intel/isa-l/issues/10>.

- [3] *Facebook's Hadoop 20*. Accessed: Aug. 2021. [Online]. Available: <https://github.com/facebookarchive/hadoop-20>
- [4] *HDFS RAID*. Accessed: Apr. 2018. [Online]. Available: <http://wiki.apache.org/hadoop/HDFS-RAID>
- [5] *Intel ISA-L*. Accessed: Aug. 2021. [Online]. Available: <https://github.com/intel/isa-l>
- [6] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.
- [7] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," Int. Comput. Sci. Inst., UC Berkeley, Berkeley, CA, USA, Tech. Rep. TR-95-048, Aug. 1995.
- [8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
- [9] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," *Proc. IEEE*, vol. 99, no. 3, pp. 476–489, Mar. 2011.
- [10] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "DiskReduce: RAID for data-intensive scalable computing," in *Proc. ACM PDSW*, Nov. 2009, pp. 6–10.
- [11] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. USENIX OSDI*, Oct. 2010, pp. 1–14.
- [12] S. R. Hetzler, "Data storage array scaling method and system with minimal data movement," U.S. Patent 8239622, Aug. 7, 2012.
- [13] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen, "Exploiting combined locality for wide-stripe erasure coding in distributed storage," in *Proc. USENIX FAST*, 2021, pp. 233–248.
- [14] Y. Hu, X. Zhang, P. P. C. Lee, and P. Zhou, "Generalized optimal storage scaling via network coding," in *Proc. IEEE ISIT*, Jun. 2018, pp. 956–960.
- [15] C. Huang *et al.*, "Erasure coding in Windows azure storage," in *Proc. USENIX ATC*, Jun. 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. Kadekodi, K. Rashmi, and G. R. Ganger, "Cluster storage systems gotta have HeART: Improving storage efficiency by exploiting disk-reliability heterogeneity," in *Proc. USENIX FAST*, 2019, pp. 345–358.
- [18] J. Lacan and J. Fimes, "Systematic MDS erasure codes based on Vandermonde matrices," *IEEE Commun. Lett.*, vol. 8, no. 9, pp. 570–572, Sep. 2004.
- [19] 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. IEEE/IFIP DSN*, Jun. 2015, pp. 148–159.
- [20] F. Maturana, V. S. C. Mukka, and K. V. Rashmi, "Access-optimal linear MDS convertible codes for all parameters," in *Proc. IEEE ISIT*, Jun. 2020, pp. 577–582.
- [21] F. Maturana and K. V. Rashmi, "Convertible codes: New class of codes for efficient conversion of coded data in distributed storage," in *Proc. ITCS*, 2020, pp. 1–26.
- [22] M. Ovsianikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
- [23] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM SIGMOD*, 1988, pp. 109–116.
- [24] J. S. Plank, "A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems," *J. Softw. Pract. Exp.*, vol. 27, no. 9, pp. 995–1012, Sep. 1997.
- [25] J. S. Plank and C. Huang, "Tutorial: Erasure coding for storage applications," presented at the 11th USENIX Conf. File Storage Technol. (FAST), Feb. 2013.
- [26] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. USENIX FAST*, 2009, pp. 253–265.
- [27] B. K. Rai, V. Dhoorjati, L. Saini, and A. K. Jha, "On adaptive distributed storage systems," in *Proc. IEEE ISIT*, Jun. 2015, pp. 1482–1486.
- [28] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, Jun. 1960.
- [29] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
- [30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE MSST*, May 2010, pp. 1–10.
- [31] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proc. ACM EuroSys*, Apr. 2018, pp. 1–14.
- [32] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. IPTPS*, Mar. 2002, pp. 328–337.
- [33] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, Feb. 1996.
- [34] C. Wu and X. He, "GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling," in *Proc. IEEE ICPP*, Sep. 2012, pp. 460–469.
- [35] C. Wu and X. He, "A flexible framework to enhance RAID-6 scalability via exploiting the similarities among MDS codes," in *Proc. IEEE ICPP*, Oct. 2013, pp. 542–551.
- [36] 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. IEEE CLUSTER*, Sep. 2012, pp. 284–292.
- [37] S. Wu, Z. Shen, and P. P. C. Lee, "Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed–Solomon codes," in *Proc. IEEE SRDS*, Sep. 2020, pp. 246–255.
- [38] S. Wu, Y. Xu, Y. Li, and Z. Yang, "I/O-efficient scaling schemes for distributed storage systems with CRS codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2639–2652, Sep. 2016.
- [39] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *Proc. USENIX FAST*, 2015, pp. 213–226.
- [40] G. Zhang, K. Li, J. Wang, and W. Zheng, "Accelerate RDP RAID-6 scaling by reducing disk I/Os and XOR operations," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 32–44, Jan. 2015.
- [41] G. Zhang, J. Shu, W. Xue, and W. Zheng, "SLAS: An efficient approach to scaling round-robin striped volumes," *ACM Trans. Storage*, vol. 3, no. 1, p. 3, Mar. 2007.
- [42] G. Zhang, W. Zheng, and K. Li, "Rethinking RAID-5 data layout for better scalability," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2816–2828, Nov. 2014.
- [43] G. Zhang, W. Zheng, and J. Shu, "ALV: A new data redistribution approach to RAID-5 scaling," *IEEE Trans. Comput.*, vol. 59, no. 3, pp. 345–357, Mar. 2010.
- [44] X. Zhang, Y. Hu, P. P. C. Lee, and P. Zhou, "Toward optimal storage scaling via network coding: From theory to practice," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1808–1816.
- [45] W. Zheng and G. Zhang, "FastScale: Accelerate RAID scaling by minimizing data migration," in *Proc. USENIX FAST*, 2011, pp. 149–161.

Yuchong Hu (Member, IEEE) received the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2010. He is currently a Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests are dependability in distributed systems, including cloud computing and data centers.

Xiaoyang Zhang received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology in 2020. His research interests include erasure codes, storage scaling, and network coding.

Patrick P. C. Lee (Senior Member, IEEE) received the Ph.D. degree in computer science from Columbia University in 2008. He is currently an Associate Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests are in various applied/systems topics, including storage systems, distributed systems and networks, and cloud computing.

Pan Zhou received the Ph.D. degree from the School of Electrical and Computer Engineering, Georgia Institute of Technology, in 2011. He is currently a Professor with the School of Cyber Science and Engineering, Huazhong University of Science and Technology. His research interests are security and privacy, big data analytics, machine learning, and information networks.