StripeMerge: Efficient Wide-Stripe Generation for Large-Scale Erasure-Coded Storage

Qiaori Yao[†], Yuchong Hu[†], Liangfeng Cheng[†], Patrick P. C. Lee[‡], Dan Feng[†], Weichun Wang[*], Wei Chen[*]
[†]*Huazhong University of Science & Technology* [‡]*The Chinese University of Hong Kong* [*]*HIKVISION*

***Abstract*—Erasure coding has been widely deployed in modern large-scale storage systems for storage-efficient fault tolerance by storing stripes of data and parity chunks. Recently, enterprises explore the notion of *wide stripes* to suppress the fraction of parity chunks in each stripe to achieve extreme storage savings. However, how to efficiently generate wide stripes remains a non-trivial issue. In particular, re-encoding the currently stored stripes (termed narrow stripes) into wide stripes triggers substantial bandwidth overhead in relocating and regenerating the chunks for wide stripes. We propose StripeMerge, a wide-stripe generation mechanism that selects and merges narrow stripes into wide stripes, with the primary objective of minimizing the wide-stripe generation bandwidth. We prove the existence of an optimal scheme that does not incur any data transfer for wide-stripe generation, yet the optimal scheme is computationally expensive. To this end, we propose two heuristics that can be efficiently executed with only limited wide-stripe generation bandwidth overhead. We prototype StripeMerge and show via both simulations and Amazon EC2 experiments that the wide-stripe generation time can be reduced by up to 87.8% over a state-of-the-art storage scaling approach.**

I. Introduction

Enterprises deploy large-scale clustered [11], [18], [33] or geo-distributed [5], [7], [26] storage systems to manage petabytes of data to cope with the tremendous growth of data in the wild [20]. One major deployment challenge is that as storage systems grow in scale, they become more susceptible to transient and permanent failures [11]. To maintain data durability, it is thus important to store data with *redundancy*. Traditional storage systems adopt *replication* to provide fault tolerance due to its simplicity. However, the storage overhead of replication is significant and poses scalability concerns.

Erasure coding is now a widely adopted redundancy technique, as an alternative to replication, for achieving low-cost durability guarantees in large-scale storage systems. Among many erasure coding constructions, Reed-Solomon (RS) codes [34] are the most popular erasure codes deployed in production [11], [23], [26], [33], [40]. RS codes can be constructed by two configurable integer parameters k and m. A (k,m) RS code encodes k original fixed-size chunks, called *data chunks*, into m coded chunks of the same size, called *parity chunks*, such that any k out of the $k+m$ data/parity chunks can reconstruct the k data chunks (see Section II-A for details). A collection of $k+m$ data/parity chunks is called a *stripe*, in which the $k+m$ chunks are distributed across $k+m$ storage nodes for tolerating any m node failures. Large-scale storage systems store data with multiple stripes, each of which is independently encoded. We can readily observe that the redundancy overhead of erasure coding (i.e., $\frac{k+m}{k}\times$) is much lower than that of replication (i.e., $(m+1)\times$) to tolerate any m node failures. For example, to tolerate any four node failures, Facebook [26] uses the $(10,4)$ RS code with a redundancy of $1.4\times$, while replication incurs a redundancy of $5\times$ for tolerating the same number of node failures. In general, erasure coding is proven to incur significantly lower redundancy overhead than replication with the same durability guarantees, measured in terms of mean-time-to-data-loss [39].

Although erasure coding effectively mitigates storage redundancy, storage practitioners are interested in further redundancy reduction in erasure coding for *extreme* storage savings; even a small fraction of redundancy reduction in erasure coding can have significant cost benefits. For example, Azure reportedly saves millions of dollars in production by moving from the $(6,3)$ RS code to the $(14,4)$ RS code, or equivalently a redundancy reduction of 14% [32].

To achieve extreme storage savings, we study *wide stripes* in erasure coding [4], [16], which refer to the stripes that have a very large k, while keeping a small m for fault tolerance as in conventional erasure coding deployment. Wide stripes are suitable for large-scale storage systems that have sufficient nodes to support a very large k, while extensively suppressing the redundancy to almost equal to one. For example, VAST [4] considers $(k,m)=(150,4)$, thereby making the redundancy to only $1.027\times$. Given the ever-increasing storage demands in the wild [20], wide stripes are particularly attractive for achieving extreme storage savings in *cold* storage systems [1], [6], which emphasize more on low-cost storage durability than high data access performance.

Despite the promises of wide stripes, how to generate wide stripes remains an unexplored yet non-trivial issue. We note that when node failures happen, erasure coding requires that any lost chunk in the failed nodes be reconstructed by retrieving multiple available chunks from non-failed nodes, thereby triggering substantial bandwidth costs in data transfers [9]. Such reconstruction bandwidth costs are shown to increase with k [9], and become prohibitive especially for wide stripes. To balance the trade-off between storage efficiency and performance, we argue that erasure coding should be parameterized differently via a *tiered* approach with respect to the data age, based on the observation that data chunks tend to be accessed more frequently when they are newly written, but becomes less frequently accessed as they age [26]. In the tiered approach, the newly written data chunks are first encoded into the stripes with a small k, referred to as *narrow stripes*, for high performance

as in conventional erasure coding deployment. As the data chunks age, the narrow stripes are later re-encoded into wide stripes for highly storage-efficient durability.

Clearly, re-encoding narrow stripes into wide stripes inevitably relocates data chunks and regenerates parity chunks, leading to the substantial bandwidth overhead in data transfers. Several studies have examined how to mitigate the bandwidth overhead in a related problem called *storage scaling* [8], [19], [42]–[45], [49], in which new nodes are added to a storage system for capacity expansion and new stripes are re-computed across the existing and newly added nodes. However, existing storage scaling solutions still incur data transfers for relocating chunks to new nodes (Section II-C).

Our key insight is that the generation of wide stripes often occurs in large-scale storage systems that have already hosted numerous nodes and stripes. Given the currently stored narrow stripes across a large pool of nodes, it is highly likely that we can select two narrow stripes that can be merged into a new wide stripe, such that *both data and parity chunks can be locally generated*, thereby incurring no data transfer in wide-stripe generation.

To this end, we present StripeMerge, a wide-stripe generation mechanism that aims to mitigate data transfers during wide-stripe generation by carefully selecting and merging narrow stripes in large-scale erasure-coded storage systems. Our contributions include:

- We are the *first* to formally model the wide-stripe generation problem. We prove the existence of an optimal scheme that exploits the *perfect merging* property (defined in Section II-D) of narrow stripes without incurring any data transfer for wide-stripe generation. However, we also point out that the algorithmic complexity of the optimal scheme is prohibitive.
- We design two practical heuristics for StripeMerge: (i) a greedy heuristic that reduces the algorithmic complexity and (ii) a parity-aligned heuristic that further enhances the greedy heuristic by selectively merging narrow stripes based on the placements of parity chunks.
- We prototype and evaluate StripeMerge via both simulations and Amazon EC2 experiments. Evaluation results show that StripeMerge significantly reduces data transfers for wide-stripe generation by up to 87.8% compared to NCScale [49], a state-of-the-art storage scaling scheme.

Our StripeMerge prototype is now open-sourced at: **https://github.com/yuchonghu/stripe-merge**.

II. Background and Motivation

We present the basics of erasure coding (Section II-A) and describe the wide-stripe generation problem (Section II-B). We show the limitations of existing storage scaling solutions in addressing the problem (Section II-C). Finally, we motivate via examples our main idea and state the challenges (Section II-D).

A. Erasure Coding

We elaborate the definitions and properties of erasure coding from Section I. Many erasure codes have been proposed in the literature (see surveys [10], [32] and Section VI), among which *Reed-Solomon (RS) codes* [34] remain the most popular erasure codes and are widely deployed in production [11], [23], [26], [33], [40]. A (k,m) RS code encodes k fixed-size data chunks, denoted by $D_1, D_2, \cdots, D_k$, into m coded parity chunks of the same size, denoted by $P_1, P_2, \cdots, P_m$. Each set of $k+m$ data/parity chunks is called a *stripe*, and the $k+m$ chunks are stored in $k+m$ nodes. In practice, large-scale storage systems store multiple stripes that are independently encoded and distributed different sets of $k+m$ nodes. Also, each chunk is often configured with a large size (e.g., 64 MiB [12] or 256 MiB [33]) to mitigate the I/O seek overhead.

RS codes are *Maximum Distance Separable (MDS)*, in which any k out of $k+m$ data/parity chunks of a stripe can reconstruct the original k data chunks; in other words, any m failed chunks can be tolerated. The MDS property also implies storage optimality, where the storage redundancy $\frac{k+m}{k}$ is the minimum (i.e., the redundancy of any (k,m) code is at least $\frac{k+m}{k}$ to tolerate any m failures). Furthermore, we consider *systematic* RS codes, meaning that the k data chunks are kept in a stripe after encoding for direct access.

Mathematically, each parity chunk P_i $(1 \leq i \leq m)$ of RS codes is formed by a linear combination of the k data chunks $D_1, D_2, \cdots, D_k$ of the same stripe over finite fields (based on Galois Field arithmetic). In this work, we focus on *Vandermonde-based RS codes* [30], in which each parity chunk is linearly encoded by the data chunks as follows:

$$P_i = \sum_{j=1}^{k} j^{i-1} D_j, \text{ for } 1 \leq i \leq m, \tag{1}$$

where the encoding coefficient j^{i-1} $(1 \leq i \leq k$ and $1 \leq j \leq m)$ is the (i,j)-th entry of the $m \times k$ Vandermonde matrix. Note that systematic Vandermonde-based RS codes in general do not exist for all parameters of k and m in small finite fields [2], [21], [31]. Nevertheless, for small m, we can still find the feasible constructions for a wide range of k [2].

B. Wide-Stripe Generation

Motivation of wide stripes: To achieve extreme storage savings in erasure-coded storage, we store erasure-coded data in *wide stripes*, defined as the stripes that have a very large k, while m (i.e., the number of tolerable failures) remains small. In this case, the redundancy $\frac{k+m}{k}$ approaches one as k increases.

We assume that wide stripes are used for *cold* data that is rarely accessed, such as backup and archival data [1], [6] or binary large objects (BLOBs) whose access frequency drops as they age. Note that wide stripes have high repair penalty, as the repair bandwidth (i.e., the amount of data transfers during repair) increases with k. We assume that repair-efficient techniques are deployed for fast recovery, such as by parallelizing repair operations [22], [25].

Wide-stripe generation problem: Recall from Section I that the generation of wide stripes from narrow stripes incurs substantial bandwidth overhead in data transfers. To show how the data transfers are triggered, we consider a specific wide-stripe generation process that converts two (k,m) narrow

stripes that are RS-coded into a new $(2k,m)$ wide stripe that is also RS-coded. Then the wide-stripe generation process comprises the following steps:

- *Step 1:* Re-distributing the $2k$ data chunks of the two narrow stripes, such that they are stored in $2k$ different nodes.
- *Step 2:* Migrating some of the data and parity chunks of the two narrow stripes to some nodes that are responsible for generating m new parity chunks of the wide stripe; the m parity chunks are later distributed across m different nodes.

We see that there exist significant data transfers due to the re-distribution of data chunks and the generation of parity chunks from narrow stripes. Thus, our primary objective is to minimize the *wide-stripe generation bandwidth* (i.e., the amount of data transfers during wide-stripe generation).

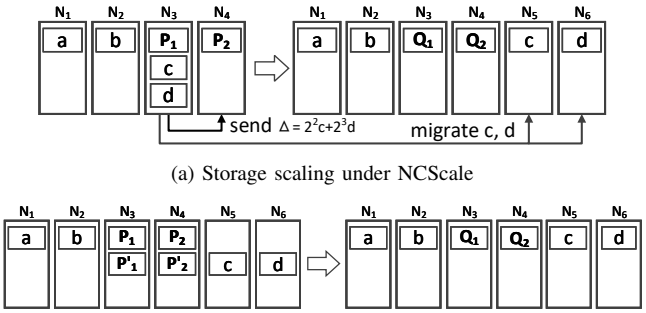
The general problem of wide-stripe generation is difficult to solve due to the numerous possible combinations of parameters. Here, we focus on the following problem based on the above specific case: *Given a set of* (k,m) *narrow stripes in a storage system, how do we generate* $(2k,m)$ *wide stripes, such that the wide-stripe generation bandwidth is minimized?*

We use an example to describe the wide-stripe generation problem. Consider two $(2,2)$ narrow stripes, denoted by $\{a,b,P_1,P_2\}$ and $\{c,d,P_1',P_2'\}$, where a, b, c, and d are data chunks, and $P_1=a+b$, $P_2=a+2b$, $P_1'=c+d$, and $P_2'=c+2d$ are the corresponding parity chunks. We merge the narrow stripes into a new $(4,2)$ wide stripe $\{a,b,c,d,Q_1,Q_2\}$, where $Q_1=a+b+c+d$ and $Q_2=a+2b+2^2c+2^3d$ are parity chunks. Note that all parity chunks are formed by the linear combination of all data chunks in the same stripe based on Vandermonde-based RS codes, and the '+' operator denotes the bitwise-XOR operation. Our goal is to minimize the wide-stripe generation bandwidth by merging the above two narrow stripes to the new wide stripe.

C. Storage Scaling

Existing *storage scaling* approaches for erasure-coded storage [8], [19], [42]–[45], [49] provide efficient, yet sub-optimal, solutions to the wide-stripe generation problem. They consider a scenario of adding s new nodes to a storage system for capacity expansion. To re-distribute the erasure-coded chunks across the existing and newly added nodes, they study how to convert (k,m) stripes into $(k+s,m)$ stripes, with the objective of minimizing the scaling bandwidth (i.e., the amount of data transfers during scaling). We can apply storage scaling to wide-stripe generation, by setting $s=k$.

However, existing storage scaling solutions cannot completely eliminate wide-stripe generation bandwidth. To justify, we consider a state-of-the-art storage scaling solution NCScale [49], which applies the idea of *network coding* in storage scaling. It shows via information-theoretic analysis that the scaling bandwidth is minimized. Its idea is to allow local computations of parity chunks. For example, Figure 1(a) shows how NCScale can be applied in wide-stripe generation. The parity chunk of the wide stripe $Q_1=a+b+c+d$ in node N_3 can be locally computed by the parity chunk $P_1=a+b$ and the data chunks c and d, all of which all reside in N_3 (note that in this example, the data chunks c and d may belong to different narrow stripes before wide-stripe generation). Also, the other parity chunk of the wide stripe $Q_2=a+2b+2^2c+2^3d$ in node N_4 can be computed by the local parity chunk $P_2=a+2b$ and a *delta chunk* $Q_2-P_2=2^2c+2^3d$ from N_3. NCScale still needs to relocate the data chunks to N_5 and N_6, respectively, leading to non-zero wide-stripe generation bandwidth.



(a) Storage scaling under NCScale

(b) Perfect merging

Fig. 1: The generation of a $(4,2)$ wide stripe from the $(2,2)$ narrow stripes over nodes N_1,N_2,N_3,N_4, N_5, and N_6.

Storage scaling generally triggers non-zero data transfers as it relocates data chunks (e.g., c and d in Figure 1(a)). However, wide-stripe generation is different from storage scaling, as it does not assume the addition of new nodes. Such a subtle difference motivates us to design an optimal solution specifically for wide-stripe generation.

D. Our Idea

Perfect merging: We observe that practical large-scale storage systems often store numerous stripes that are distributed across numerous nodes. To solve our problem of converting (k,m) narrow stripes into $(2k,m)$ wide stripes, our main idea is to select two suitable (k,m) narrow stripes from the currently stored numerous narrow stripes, such that the two selected stripes can be merged into a new $(2k,m)$ wide stripe *without any wide-stripe generation bandwidth*. We motivate this idea via an example in Figure 1(b), in which the two $(2,2)$ narrow stripes $\{a,b,P_1,P_2\}$ and $\{c,d,P_1',P_2'\}$ can be merged into a new $(4,2)$ wide stripe $\{a,b,c,d,Q_1,Q_2\}$, without any data transfer.

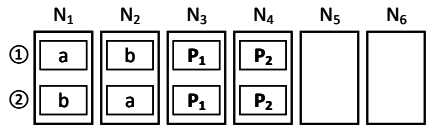
The two narrow stripes in Figure 1(b) have two properties, which we collectively call *perfect merging*, such that the wide-stripe generation bandwidth can be completely eliminated: (i) both of their data chunks reside in different nodes; and (ii) both of their parity chunks have identical encoding coefficients and reside in the same nodes.

Note that perfect merging leverages the critical feature of Vandermonde-based RS codes (Section II-A), in which the new parity chunks can be locally computed from the existing parity chunks. Specifically, based on Equation (1), we have

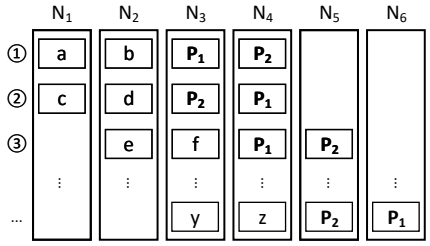
$$\begin{bmatrix}P_1\\P_2\end{bmatrix}=\begin{bmatrix}1&1\\1&2\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix},\quad \begin{bmatrix}P_1'\\P_2'\end{bmatrix}=\begin{bmatrix}1&1\\1&2\end{bmatrix}\begin{bmatrix}c\\d\end{bmatrix}, \tag{2}$$

and

$$\begin{bmatrix}Q_1\\Q_2\end{bmatrix}=\begin{bmatrix}1&1&1^2&1^3\\1&2&2^2&2^3\end{bmatrix}\begin{bmatrix}a\\b\\c\\d\end{bmatrix}. \tag{3}$$



(a) Two stripes with the same chunk placement



(b) Complete chunk placement set

Fig. 2: Examples of the same chunk placement (figure (a)) and a complete chunk placement set (figure (b)) in wide-stripe generation.

Based on Equations (2) and (3), we have

$$\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} + \begin{bmatrix} 1^2 \cdot P_1' \\ 2^2 \cdot P_2' \end{bmatrix}. \tag{4}$$

In general, if the m parity chunks of two (k,m) narrow stripes have identical encoding coefficients, we can use them directly as input to compute m new parity chunks of a $(2k,m)$ wide stripe, according to Equation (4). Also, if the m parity chunks of the two narrow stripes reside in the same m nodes, we can locally compute the m parity chunks of the wide stripe without any data transfer.

Challenges: While perfect merging can effectively generate a single wide stripe without any data transfer, how to systematically apply perfect merging to generate multiple wide stripes for large-scale storage systems remains non-trivial. In particular, the selection of the pairs of narrow stripes that satisfy perfect merging depends on how all narrow stripes are currently placed in the underlying storage system. The search of all narrow stripes for large-scale storage systems can be a time-consuming procedure.

III. Analysis

We first formulate our wide-stripe generation problem as a bipartite graph model (Section III-A). We show that there always exists an optimal scheme that can generate all wide stripes without any wide-stripe generation bandwidth given a sufficiently large number of narrow stripes (Section III-B).

A. Bipartite Graph Model

We first formulate the wide-stripe generation problem for a general large-scale storage system as follows. Consider a large-scale storage system with N nodes that store a sufficiently large number of (k,m) narrow stripes. Our goal is to select all pairs of narrow stripes that satisfy perfect merging and merge each of the pairs into a $(2k,m)$ wide stripe, such that there is no wide-stripe generation bandwidth across the N nodes.

Each (k,m) narrow stripe is randomly placed in any $k+m$ out of N nodes. If we treat the data and parity chunks differently in each chunk placement, then there are a total of $\frac{N!}{(N-k-m)!}$ possible chunk placements of all stripes. In this calculation, different orders of data chunks and parity chunks in a stripe implies different chunk placements. However, in our wide-stripe generation problem, the order of data chunks does not matter when we select narrow stripes for merging into a wide stripe; instead, only the order of parity chunks matters for the computation of new parity chunks in the wide stripe. For example, as shown in Figure 2(a), the two (2,2) stripes are considered to have the same chunk placement, since their data chunks appear in the same set of nodes (despite in different orders) and the parity chunks P_1 and P_2 are in the exactly same nodes. Thus, we only consider the chunk placement of a narrow stripe across N nodes subject to the order of parity chunks, and there are a total of $\frac{N!}{(N-k-m)!k!}$ possible chunk placements of all stripes, which we call a *complete chunk placement set*, as shown in Figure 2(b).

We model the wide-stripe generation problem via a bipartite graph $\mathcal{G}$ with two disjoint sets of vertices, denoted by $\mathcal{X}$ and $\mathcal{Y}$, as well as a set of edges, denoted by $\mathcal{E}$, that connect the vertices in $\mathcal{X}$ and $\mathcal{Y}$. Both $\mathcal{X}$ and $\mathcal{Y}$ are identical (i.e., $\mathcal{X}=\mathcal{Y}$), and each vertex in $\mathcal{X}$ and $\mathcal{Y}$ corresponds to one of all $\frac{N!}{(N-k-m)!k!}$ possible chunk placements of all stripes (i.e., $|\mathcal{X}|=|\mathcal{Y}|=\frac{N!}{(N-k-m)!k!}$). We add an edge in $\mathcal{E}$ between a vertex $x\in\mathcal{X}$ and a vertex $y\in\mathcal{Y}$ if any pair of stripes in the chunk placements x and y satisfies perfect merging.

B. Existence

Consider a large-scale storage system that store a sufficiently large number of stripes, such that in each of the $\frac{N!}{(N-k-m)!k!}$ possible chunk placements, there always exist (k,m) narrow stripes. We now show that we can always pair two different chunk placements, such that we can merge the (k,m) narrow stripes in the paired chunk placements to form $(2k,m)$ wide stripes without any wide-stripe generation bandwidth.

In the following, we first show the properties of the bipartite graph $\mathcal{G}$ modeled from our wide-stripe generation problem (Lemmas 1 and 2). We then show the feasibility of incurring zero bandwidth in wide-stripe generation.

Lemma 1. *$\mathcal{G}$ is a $\binom{N-m-k}{k}$-regular bipartite graph (i.e., each vertex has a degree $\binom{N-m-k}{k}$).*

Proof: For each vertex $x\in\mathcal{X}$, any (k,m) narrow stripe (denoted by s_1) in the chunk placement x occupies $k+m$ out of N nodes. It can pair with another (k,m) narrow stripe (denoted by s_2) to satisfy perfect merging if (i) the k data chunks of s_2 reside in any k of the remaining non-occupied $N-k-m$ nodes, and (ii) the m parity chunks of s_2 reside in the same nodes (in the same order) as those of s_1. There are a total of $\binom{N-m-k}{k}$ chunk placements in which s_2 can reside for perfect merging to hold. Thus, we can add edges from x to a total of $\binom{N-m-k}{k}$ vertices in $\mathcal{Y}$ (which correspond to the satisfying chunk placements). Similarly, we can add edges from each vertex in $\mathcal{Y}$ to a total of $\binom{N-m-k}{k}$ vertices in $\mathcal{Y}$ given that $\mathcal{X}=\mathcal{Y}$. The lemma follows. □

Lemma 2. *$\mathcal{G}$ has a perfect matching.*

Proof: By Lemma 1, for any subset $\mathcal{S}_x \in \mathcal{X}$, there are a total of $K|\mathcal{S}_x|$ edges from $\mathcal{S}_x$ to some subset $\mathcal{S}_y \in \mathcal{Y}$, where $K = \binom{N}{N-m-k}$. Since $\mathcal{G}$ is regular, $\mathcal{S}_y$ has a total of $K|\mathcal{S}_y|$ edges. This implies that $K|\mathcal{S}_x| = K|\mathcal{S}_y|$, and hence $|\mathcal{S}_x| = |\mathcal{S}_y|$. By Hall's theorem [29], there exists a matching of size $|\mathcal{X}|$ in G. Since $|\mathcal{X}| = |\mathcal{Y}|$, $\mathcal{G}$ has a perfect matching. □

Theorem 1. *We can always pair and merge the (k,m) narrow stripes in two chunk placements to form $(2k,m)$ wide stripes, without any wide-stripe generation bandwidth.*

Proof: By modeling the chunk placements as a bipartite graph $\mathcal{G}$ as shown above, each chunk placement can always be paired with another chunk placement by Lemma 2, such that the stripes in the paired chunk placements satisfy perfect merging. Merging those stripes into wide stripes incurs no wide-stripe generation bandwidth. The theorem follows. □

IV. StripeMerge

StripeMerge is a wide-stripe generation mechanism that selects and merges narrow stripes, with a primary objective of minimizing the wide-stripe generation bandwidth. While we can design the optimal scheme for StripeMerge without any wide-stripe generation bandwidth based on the analysis in Section III-B, we show that the optimal scheme incurs a high algorithmic complexity (Section IV-A). To this end, we present two variants of StripeMerge that trade the wide-stripe generation bandwidth for algorithmic efficiency: the greedy heuristic StripeMerge-G (Section IV-B), and the parity-aligned heuristic StripeMerge-P (Section IV-C).

A. Limitations of Optimal Scheme

From Theorem 1 (Section III-B), we show the existence of finding narrow stripes that satisfy perfect merging for wide-stripe generation via bipartite graph modeling, so as to eliminate all wide-stripe generation bandwidth. Thus, we can design the optimal scheme based on this property by solving a classical maximum matching problem on a bipartite graph. One subtlety is that there may be more than one (k,m) narrow stripe in a chunk placement at the beginning. We address this issue by iteratively applying the maximum matching problem.

The optimal scheme works as follows. Consider a storage system that stores (k,m) narrow stripes across N nodes. Suppose that each chunk placement has the same number of narrow stripes (we address this assumption later at the end). We partition the narrow stripes into multiple complete chunk placement sets, such that in each set, each chunk placement has one narrow stripe. For each complete chunk placement set, we form a bipartite graph $\mathcal{G}$ as in Section III-A. We then find the maximum matching of $\mathcal{G}$, which is also a perfect matching (Lemma 2), using a maximum matching algorithm (e.g., the Hopcroft-Karp algorithm [15]). Finally, we merge every matched pair of narrow stripes into a wide stripe, without any wide-stripe generation bandwidth (Theorem 1). We repeat the above processing for all complete chunk placement sets.

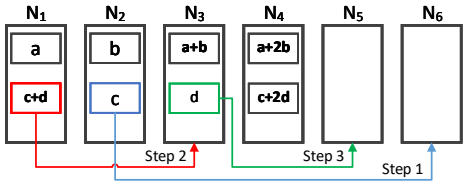


Fig. 3: Two stripes $\{a,b,a+b,a+2b\}$ and $\{c,d,c+d,c+2d\}$ satisfy perfect merging, after three chunks are moved (merging cost = 3).

The optimal scheme poses two practical concerns. First, its algorithmic complexity is prohibitively high in large-scale storage systems. The time complexity of the maximum matching algorithm, say the Hopcroft-Karp algorithm, is $O(n^{2.5})$ [15], where n is the number of vertices in a bipartite graph. Even though the algorithm has a polynomial time complexity, its running time overhead can be substantial. For example, consider a scenario where $(8,4)$ narrow stripes are distributed over 40 nodes. Then there are a total of $\frac{N!}{(N-k-m)!k!} \approx 6.6 \times 10^{13}$ possible chunk placements. Forming a bipartite graph for such a large number of chunk placements is expensive both in space and time. Second, in practice, each chunk placement may have a varying number of narrow stripes. It is thus difficult to guarantee that the complete chunk placement set can be formed, making the optimal scheme infeasible.

B. Greedy Heuristic: StripeMerge-G

The optimal scheme requires *all* wide stripes to be generated via perfect merging, which leads to a prohibitive algorithmic complexity and the infeasibility to address incomplete chunk placement sets. A natural remedy is to apply the perfect merging of existing narrow stripes to generate as many wide stripes as possible, while for the remaining narrow stripes, we transfer a number of chunks to make them satisfy perfect merging to generate the remaining wide stripes. We call the number of transferred chunks as the *merging cost*. Figure 3 shows how to transfer chunks for two narrow stripes to become perfect merging. First, we check the placement of data chunks (in $O(k)$ time), and find that data chunks b and c are both stored in N_2, so we choose to transfer chunk c to N_6 (i.e., Step 1 in Figure 3). Second, we ensure that all parity chunks with same encoding coefficients are in same nodes (in $O(m)$ time). We gather parity chunks $a+b$ and $c+d$ to the same node, so as to generate the new parity chunk $a+b+c+d$ (i.e., Step 2 in Figure 3). Third, we move the data chunk d from N_3 to an available node N_5 (i.e., Step 3 in Figure 3). Thus, the merging cost in Figure 3 is 3. Note that the computation of the merging cost can be done in $O(k+m)$ time, while $k+m$ is a relatively small constant compared to the total number of stripes.

We design a greedy heuristic, called StripeMerge-G, to merge the narrow stripes in ascending order of the merging costs. Its goal is to first merge the narrow stripe that satisfy perfect merging (i.e., with zero merging costs) if they exist, followed by merging the remaining narrow stripes that incur the immediate and minimal merging costs. Algorithm 1 shows the algorithmic details of StripeMerge-G. Given a set of narrow stripes that

Algorithm 1 StripeMerge-G

Input: $\mathcal{T}$, a set of narrow stripes: $\{s_1, s_2, \dots\}$
Output: $\mathcal{T}'$, a set of pairs of narrow stripes to be merged

1: $\mathcal{C} = \emptyset$
2: **for** $i = 1$ to $|\mathcal{T}| - 1$ **do**
3: **for** $j = i+1$ to $|\mathcal{T}|$ **do**
4: $c =$ Merging cost of s_i and s_j
5: $\mathcal{C} = \mathcal{C} + (c, s_i, s_j)$
6: **end for**
7: **end for**
8: Sort all tuples of $\mathcal{C}$ based on c in ascending order
9: **while** $\mathcal{C} \neq \emptyset$ **do**
10: Select the first tuple (c, s_{i*}, s_{j*}) of $\mathcal{C}$
11: $\mathcal{T}' = \mathcal{T}' + (s_{i*}, s_{j*})$
12: Remove all the tuples that include s_{i*} and s_{j*} in $\mathcal{C}$
13: **end while**
14: **return** $\mathcal{T}'$

Algorithm 2 StripeMerge-P

Input: $\mathcal{T}$, a set of (k,m) narrow stripes: $\{s_1, s_2, \dots\}$, and m sets of i-partial parity-aligned pairs: $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_m$.
Output: $\mathcal{T}'$, a set of pairs of narrow stripes to be merged

1: **for** $i = m$ to 1 **do**
2: **for** $j = 1$ to $|\mathcal{T}|$ **do**
3: $\mathcal{T}_{temp} =$ a set of pairs in $\mathcal{P}_i$ that contains s_j
4: Find the pair in $\mathcal{T}_{temp}$ that has minimum merging cost c
5: **if** $c \leq m$ **then**
6: $\mathcal{T}' = \mathcal{T}' +$ this pair
7: $\mathcal{T} = \mathcal{T} -$ the stripes of this pair
8: **end if**
9: **end for**
10: **end for**
11: $\mathcal{T}' = \mathcal{T}' +$ Algorithm 1 $(\mathcal{T})$
12: **return** $\mathcal{T}'$

are currently stored, StripeMerge-G first computes the merging costs of any pair of narrow stripes and constructs a set that contains all pairs of narrow stripes and their merging costs (lines 1-7). It sorts all the pairs of narrow stripes with respect to the merging costs in ascending order (line 8); note that each merging cost must be an integer ranging from 0 to $k+m$, so we can use counting sort to sort all merging costs in $O(n^2)$ time, where n is the number of narrow stripes and $n(n-1)/2$ is the number of all pairs of narrow stripes. It then selects the first pair of narrow stripes with the currently smallest merging cost, such that the pair of narrow stripes will be merged to form a wide stripe. It also removes all the elements that include any of the two narrow stripes that have just been merged (lines 9-13).

We analyze the running time performance of StripeMerge-G. Algorithm 1 needs to compute the merging costs of all pairs of narrow stripes. Thus, its time complexity is $O((k+m)n^2)$, where $O(k+m)$ is the computation complexity of the merging cost of a pair of narrow stripes (see above). The algorithm does not significantly decrease the complexity of the optimal scheme (i.e., from $O(n^{2.5})$), so it would still be time-consuming for a large number of stripes of the large-scale storage systems. Nevertheless, we leverage Algorithm 1 as a building block to design the parity-aligned heuristic (Section IV-C).

C. Parity-aligned Heuristic: StripeMerge-P

To reduce the computation complexity of Algorithm 1, our idea is based on the following observation: for a pair of narrow stripes that satisfy perfect merging, a necessary condition is that all their parity chunks have identical encoding coefficients and reside in identical nodes; we call this property *fully parity-aligned*. Based on the observation, we aim to identify the fully parity-aligned pairs of narrow stripes, so as to quickly obtain the pairs of narrow stripes that satisfy perfect merging. We argue that this approach significantly reduces the number of stripes as input in Algorithm 1.

Furthermore, we identify the *partially parity-aligned* pairs of narrow stripes in which some, but not all, parity chunks are aligned. Such pairs of narrow stripes satisfy perfect merging after we transfer a small number of parity chunks. To specify the partially parity-aligned pairs, we define a i-partial parity-aligned pair, where $1 \leq i \leq m$, as the pair of narrows stripes have i parity chunks that are aligned. We denote a set of i-partial parity-aligned pairs by $\mathcal{P}_i$.

To construct $\mathcal{P}_i$, StripeMerge-P stores the metadata of parity chunk placements in a hash table when generating parity chunks. The hash table stores key-value items, where each key refers to a certain placement of any i parity chunks ($1 \leq i \leq m$) across N nodes, and its value is a list of indices of the stripes that have the corresponding parity chunk placement. In this way, we can use the hash table to find the stripes that have i identical parity chunk placement to form $\mathcal{P}_i$ in $O(1)$ time. Note that for any (k,m) stripe with its parity chunks placed in m nodes, it can generate $2^m - 1$ different keys for all its i parity chunk placements ($1 \leq i \leq m$). For example, suppose that we have two stripes as in Figure 3: the first stripe (with index ℓ) has two parity chunks in nodes N_3 and N_4 and generates three key-value items as $[|3|4|, \ell]$, $[|3|*|, \ell]$, and $[|*|4|, \ell]$, while the second stripe (with index ℓ') has two parity chunks in nodes N_1 and N_4 and generates three key-value items as $[|1|4|, \ell']$, $[|1|*|, \ell']$, and $[|*|4|, \ell']$. Note that the second stripe has one item $[|*|4|, \ell']$ that has the same key as the item $[|*|4|, \ell]$ of the first stripe, so they are stored as one combined item in the hash table as $[|*|4|, \{\ell, \ell'\}]$. Thus, $\mathcal{P}_1 = \{(\ell, \ell')\}$ and $\mathcal{P}_2 = \emptyset$, meaning that the two stripes in Figure 3 is 1-partial-parity-aligned as they have one parity chunk in N_4. Note that the hash table incurs additional memory overhead, yet we show that the memory overhead is limited (Section V-A).

We now design StripeMerge-P, using StripeMerge-G (Section IV-B) as a building block. Algorithm 2 shows the algorithmic details of StripeMerge-P. Given a set of narrow stripes and the sets of i-partial parity-aligned pairs, the algorithm first selects each i-partial parity-aligned set (line 1), and examines each narrow stripe (line 2). Note that we start from $i = m$ to check the fully parity-aligned pairs first. With the hash table, we can quickly find the stripes that are i-partial parity-aligned with the examined stripe (line 3). We then find a pair that has the minimum merging cost in the set (line 4). If this pair is close to perfect merging (i.e., having low merging costs no larger than m (line 5)), then we merge them first (line 6) as

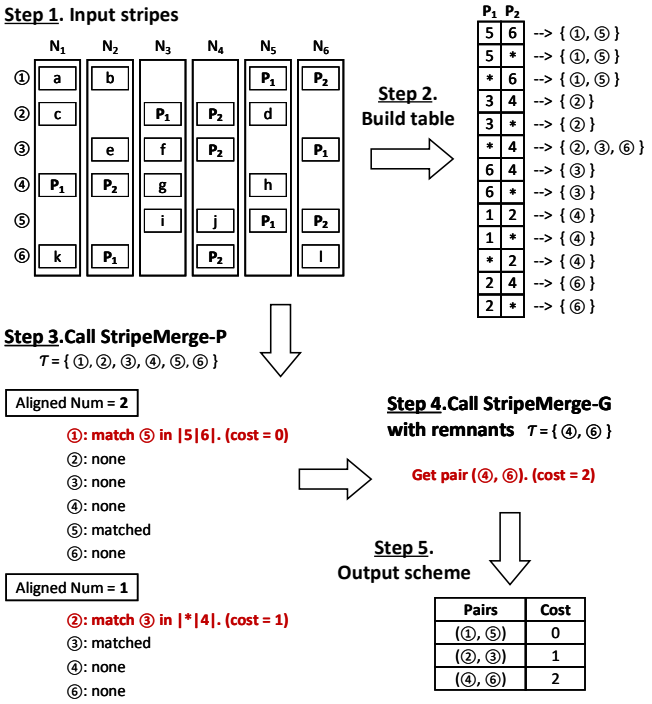


Fig. 4: Example of the operations in StripeMerge-P.

they can reduce a lot of stripes as input in Algorithm 1 (line 7). Finally, we call Algorithm 1 with a reduced input (line 11).

Note that the complexity of lines 1-10 in Algorithm 2 is $O((k+m)mn)$, where n is the number of narrow stripes, m is the number of parity chunks of a narrow stripe, and $O(k+m)$ is the complexity of the merging cost computation (Section IV-B). Note that m is generally a very small number compared to n. Thus, if line 11 can reduce a large number of stripes that will be the input of Algorithm 1, then the overall complexity can be lowered. We show via simulations that Algorithm 2 can reduce the running time significantly (Section V).

Figure 4 depicts an example on how Algorithm 2 works. In this example, there are six (2,2) stripes across six nodes as input (Step 1 in Figure 4). Suppose that before the algorithm is called, we pre-process the stripes and build the hash table when the narrow stripes are first created (Step 2 in Figure 4). Then we obtain a hash table with 13 keys, nine of which are mapped to a single stripe. We call Algorithm 2 to search for the pairs from the fully parity-aligned ones to the partially parity-aligned ones (Step 3 in Figure 4). The search will be skipped if no stripe is i-partial parity-aligned with the current stripe or it has been matched previously. We see that Stripe ① and Stripe ⑤ are fully parity-aligned, while Stripe ② and Stripe ③ are partially parity-aligned with a merging cost equal to one. After the pairs $(①, ⑤)$ and $(②, ③)$ are selected, we call Algorithm 1 to handle remaining stripes to obtain a complete scheme (Step 4 in Figure 4), and select the pair $(④, ⑥)$ with a merging cost equal to two. Finally, we obtain a scheme with a total merging cost three.

Discussion: The main idea of StripeMerge-P is to find parity-aligned narrow stripes, so as to reduce the cost of merging the narrow stripes. However, even though two narrow stripes are fully or partially parity-aligned, their data chunks may happen to reside in the same nodes. Such "bad" data chunk layouts lead to the relocation of data chunks into different nodes in wide-stripe generation, so the resulting merging cost may increase. We study this impact via simulations (Section V-A).

V. Performance Evaluation

We evaluate StripeMerge using both simulations and Amazon EC2 experiments. We compare StripeMerge with NCScale [49], a state-of-the-art storage scaling scheme built on network coding (Section II-C). We address two questions: (i) How much wide-stripe generation bandwidth can StripeMerge save via perfect merging compared to compared to NCScale? (ii) How much running time can StripeMerge-P reduce via parity alignment compared to StripeMerge-G?

A. Simulations

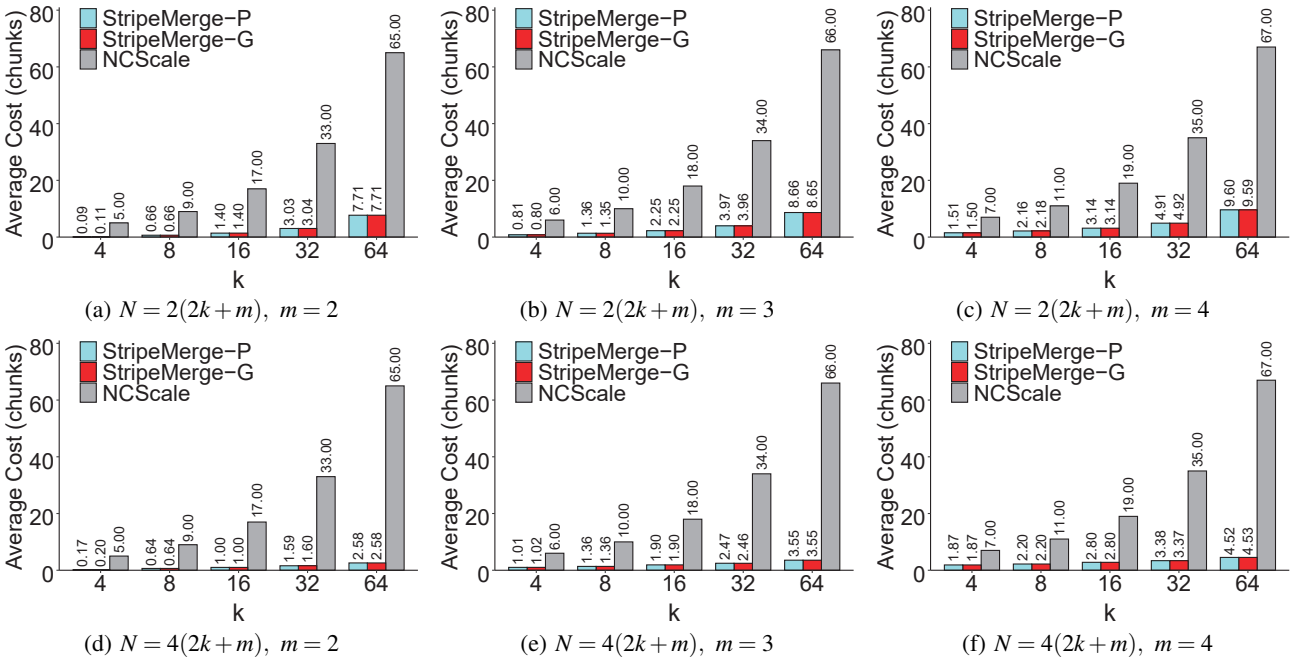
We implement a simulator for StripeMerge in C++ with about 950 SLoC. We deploy the simulator on a server equipped with an Intel Xeon Silver 4110 2.10 GHz CPU, 256 GiB RAM, and a Seagate ST1000DM003 7200 RPM 1 TiB SATA hard disk. The server runs on Ubuntu 16.04. We also implement NCScale in the simulator under the same settings for fair comparisons. Note that the simulator does not implement coding operations and data transfers, which are left to be done in Amazon EC2 experiments (Section V-B).

We configure our simulator with different parameter choices, including (k,m) (i.e., the coding parameters for narrow stripes) and N (i.e., the total number of nodes). Here, we set N as a multiple of $2k+m$, since the case of $N=2k+m$ is the minimum number of nodes for a pair of (k,m) narrow stripes to satisfy perfect merging (i.e., the data chunks are stored in $2k$ different nodes, while the parity chunks are stored in the same m nodes). We report the average results over five runs.

Experiment A.1 (Wide-stripe generation bandwidth): We measure the average wide-stripe generation bandwidth, in terms of the number of transferred chunks in the generation of a single wide stripe. Here, we randomly distribute the 10,000 stripes across N nodes. We evaluate different combinations of N and (k,m).

Figure 5 shows the wide-stripe generation bandwidth (per wide stripe) for StripeMerge-P, StripeMerge-G, and NCScale for $4 \le k \le 64$, $2 \le m \le 4$, as well as $N=2(2k+m)$ and $N=4(2k+m)$. Overall, StripeMerge significantly reduces the wide-stripe generation bandwidth of NCScale in all cases. For example, for $k=64, m=2$ and $N=4(2k+m)$, the reduction of wide-stripe generation bandwidth can be up to 96%. In addition, StripeMerge's wide-stripe generation bandwidth increases with k and m. The reason is that a larger k or m makes perfect merging more difficult to satisfy, as perfect merging requires two narrow stripes to have data chunks in different nodes and parity chunks in the same nodes.

Furthermore, StripeMerge's wide-stripe generation bandwidth mostly increases with N when k is large, while sometimes decreases with N when k is small. For example, Figures 5(a) and



(a) $N = 2(2k+m),\ m = 2$ (b) $N = 2(2k+m),\ m = 3$ (c) $N = 2(2k+m),\ m = 4$

(d) $N = 4(2k+m),\ m = 2$ (e) $N = 4(2k+m),\ m = 3$ (f) $N = 4(2k+m),\ m = 4$

Fig. 5: Experiment A.1: Average wide-stripe generation bandwidth cost per wide stripe (in number of chunks) for different N, k, and m.

5(d) show that when N increases from $2(2k+m)$ to $4(2k+m)$, the wide-stripe generation bandwidth per wide stripe only increases from 0.1 to 0.2 for $k = 4$, while it mostly decreases for $k > 4$. The reason is that if N increases, it will be more likely for the data chunks of a pair of narrow stripes to reside in different nodes, which helps perfect merging (a positive effect); on the other hand, it will be less likely for parity chunks to reside in identical nodes, which hinders perfect merging (a negative effect). Thus, when k is not large, the negative effect dominates as the number of parity chunks (i.e., m) is comparable to k, so the wide-stripe generation bandwidth increases with N. On the other hand, when k becomes much larger than m, the positive effect dominates and the wide-stripe generation bandwidth decreases with N. In summary, StripeMerge can benefit more from stripes of a wider size (i.e., a larger k) and a system of a larger scale (i.e., a larger N).

Finally, StripeMerge-P has nearly the same performance as StripeMerge-G. This implies that the parity-aligned property of StripeMerge-P does not influence the effectiveness in reducing the wide-stripe generation bandwidth.

Experiment A.2 (Running time versus (k,m)): We measure the running times of StripeMerge-G and StripeMerge-P (i.e., Algorithms 1 and 2, respectively) for different combinations of (k,m). As in Experiment A.1, we randomly distribute 10,000 stripes across N nodes, where we fix $N = 2(2k+m)$.

Figure 6 compares the running times per wide stripe of StripeMerge-G and StripeMerge-P for different (k,m). StripeMerge-P is always faster than StripeMerge-G for the same (k,m), implying that the parity-aligned property of StripeMerge-P effectively speeds up the algorithm. In particular, for small values of k (e.g., 4, 8, and 16), StripeMerge-P has almost zero running time per wide stripe.

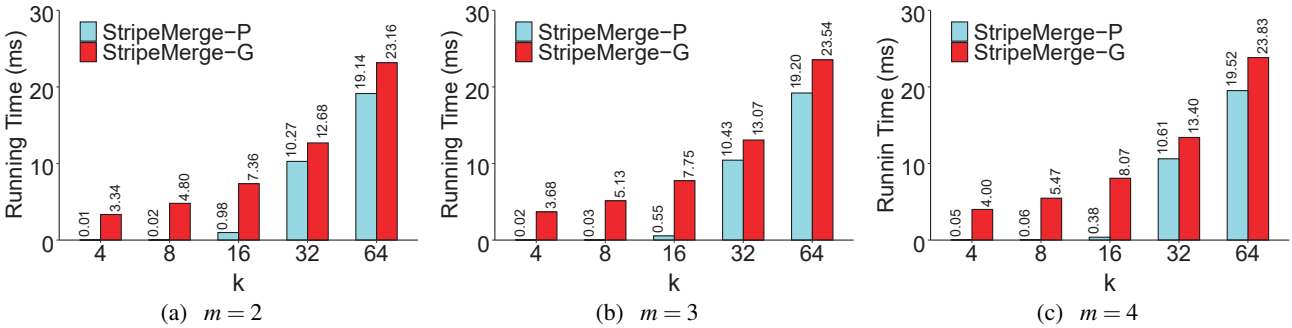
The relative reduction of the running time of StripeMerge-P over StripeMerge-G is lower for a larger k. For example, Figure 6(a) shows that for $k = 16$ the reduction is 86.5% , while for $k = 32$ the reduction is 18.9%. The reason is that more pairs of narrow stripes now have bad data chunk layouts for a larger k (Section IV-C), so StripeMerge-P cannot exploit the parity-aligned property to speed up the algorithm.

Experiment A.3 (Running time versus the number of narrow stripes): We measure the running times of StripeMerge-G and StripeMerge-P versus the number of narrow stripes (denoted by n). We fix $(k,m) = (16,4)$ and $N = 2(2k+m) = 72$.

Figure 7 compares the total running times of processing all stripes of StripeMerge-G and StripeMerge-P, where the number of narrow stripes varies from 1,000 to 100,000. The running time of StripeMerge-G increases dramatically with the number of narrow stripes, while that of StripeMerge-P increases linearly. The results are consistent with their time complexities $O((k+m)n^2)$ and $O((k+m)mn)$, respectively (Section IV). Thus, StripeMerge-P is more practical than StripeMerge-G for large-scale storage systems with numerous narrow stripes.

Experiment A.4 (Memory consumption of StripeMerge-P): We measure the memory consumption of StripeMerge-P, including the total memory usage and the memory usage of the hash table that stores parity-aligned metadata (Section IV-C) for different n. Here, we fix $(k,m) = (16,4)$ and $N = 2(2k+m) = 72$.

Figure 8 shows that the memory overhead of the hash table is limited compared to the total memory overhead. For example, processing 100,000 narrow stripes incurs 4.85 GiB of memory, while the hash table itself only incurs 72.5 MiB of memory. Thus, the additional memory overhead incurred by StripeMerge-P for tracking parity-aligned metadata is acceptable.



(a) $m = 2$ (b) $m = 3$ (c) $m = 4$

Fig. 6: Experiment A.2: Running times of StripeMerge-G and StripeMerge-P per wide stripe versus (k,m) (in units of milliseconds) for different k and m.

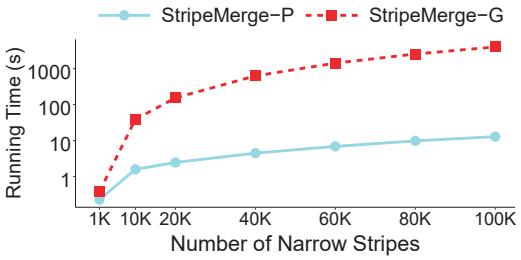


Fig. 7: Experiment A.3: Total running times of StripeMerge-G and StripeMerge-P of processing all stripes (in seconds) versus n.

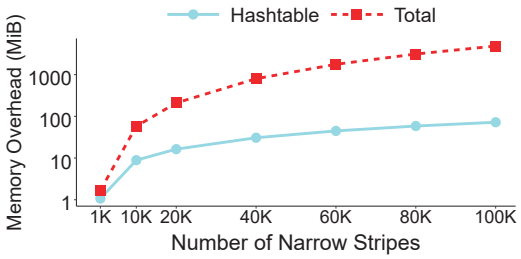


Fig. 8: Experiment A.4: Memory consumption of StripeMerge-P (in MiB) versus n.

B. Amazon EC2 Experiments

We further implement a StripeMerge prototype by extending our simulator (Section V-A) with the coding and data transfer functionalities (in about 1,000 SLoC), so that StripeMerge actually performs coding operations on the data chunks and transfers coded chunks among the nodes in wide-stripe generation. We deploy the StripeMerge prototype on Amazon EC2 instances in the US East (North Virginia) region. We implement the coding operations of Reed-Solomon codes based on two Intel ISA-L APIs [3]: `ec_init_tables`, which specifies coding coefficients, and `ec_encode_data`, which specifies encoding/decoding operations. We configure our prototype with different coding parameters (k,m) for narrow stripes. We allocate $N = 2(2k+m)$ `m5.xlarge` instances that serve as storage nodes. To evaluate the impact of network bandwidth, we also configure a dedicated instance that acts as a gateway, such that any transferred chunk originating from an instance must traverse the gateway before reaching another instance. We use the Linux traffic control command `tc` to control the outgoing bandwidth of the gateway.

In our experiments, we vary the gateway bandwidth from 1 Gb/s to 8 Gb/s. We also consider different chunk sizes. Before running each experiment, we randomly distribute 10,000 narrow stripes across the nodes. We report the average results of each experiment over five runs. We also provide the generation time variance due to fluctuating cloud network bandwidth.

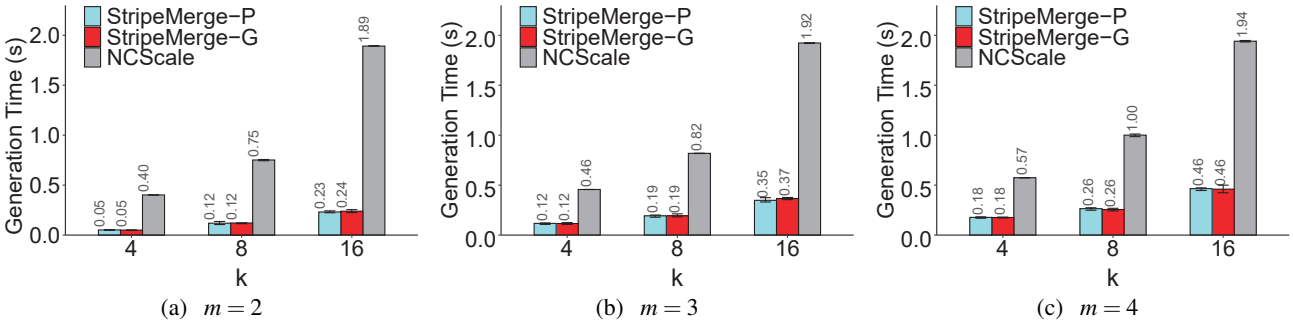
Experiment B.1 (Time breakdown): We measure the breakdown of the wide-stripe generation time and identify the bottleneck step. We decompose the generation into three steps: (i) the running time of the algorithm, which refers to the selection of pairs of narrow stripes to be merged, (ii) the transfer time, which refers to the transfers of chunks for merging, and (iii) the compute time, which refers to the local computation of merging parity chunks of narrow stripes into new parity chunks of wide stripes.

Method	Running Time	Transfer Time	Compute Time
StripeMerge-G	38.97	2,313.75	4.94
StripeMerge-P	1.58	2,323.35	4.95

TABLE I: Experiment B.1: Total time breakdown of StripeMerge-G and StripeMerge-P for processing 10,000 narrow stripes (in seconds).

Table I shows the breakdown of the total times of StripeMerge-G and StripeMerge-P for processing 10,000 narrow stripes, with $(k,m) = (16,4)$, $N = 2(2k+m) = 72$, a fixed chunk size of 64 MiB, and a fixed gateway bandwidth of 8 Gb/s. We see that the transfer time dominates, thereby justifying our goal of minimizing the wide-stripe generation bandwidth to improve the overall performance. Also, while StripeMerge-G's running time incurs 1.65% of the overall time for 10,000 stripes, this percentage will increase dramatically with the number of stripes (Experiment A.3). Thus, for large-scale storage systems that have numerous stripes, StripeMerge-G's running time will degrade the overall performance. In contrast, StripeMerge-P's running time incurs only 0.068% of the overall time for 10,000 stripes, while this percentage only increases linearly with the number of stripes (Experiment A.3).

Experiment B.2 (Wide-stripe generation time versus (k,m)): We measure the wide-stripe generation time per wide stripe for different (k,m). Figure 9 compares the wide-stripe generation times (per wide stripe) for StripeMerge-



(a) $m = 2$ (b) $m = 3$ (c) $m = 4$

Fig. 9: Experiment B.2: The wide-stripe generation time (per wide stripe), in seconds, under different k and m.

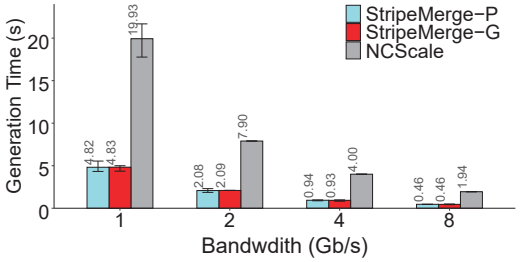


Fig. 10: Experiment B.3: Wide-stripe generation time (per wide stripe) (in seconds) versus gateway bandwidth.

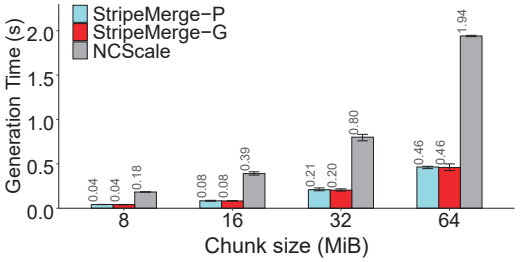


Fig. 11: Experiment B.4: The wide-stripe generation time (per wide stripe), in seconds, versus chunk size.

G, StripeMerge-P, and NCScale, where we fix the gateway bandwidth as 8 Gb/s and the chunk size as 64 MiB [12]. Similar to our simulation results (see Figures 5(a)-(c)), our empirical results also show that StripeMerge significantly reduces the overall wide-stripe generation time of NCScale under the same parameters of (k,m). For example, for $(k,m) = (16,2)$, the reduction is up to 87.8%. Note that StripeMerge's wide-stripe generation time also increases with both k and m, similar to our simulations for the same reasons.

Experiment B.3 (Impact of gateway bandwidth): We measure the wide-stripe generation time per wide stripe for different settings of gateway bandwidth, from 1 Gb/s to 8 Gb/s. Figure 10 shows the results for $(k,m) = (16,4)$, $N = 2(2k+m) = 72$, and a fixed chunk size of 64 MiB. The wide-stripe generation times of StripeMerge and NCScale decrease linearly with the gateway bandwidth, and StripeMerge maintains its performance gains over NCScale.

Experiment B.4 (Impact of chunk size): We measure the wide-stripe generation time per wide stripe for different chunk sizes, from 8 MiB to 64 MiB. Figure 11 shows the results for $(k,m) = (16,4)$, $N = 2(2k+m) = 72$, and a gateway bandwidth of 8 Gb/s. The wide-stripe generation times of StripeMerge and NCScale increase linearly with the chunk size, and again StripeMerge maintains its performance gains over NCScale.

VI. Related Work

There has been a rich body of studies on erasure-coded storage in the literature, including how to minimize the bandwidth in both repair and scaling problems. For the repair problem, some studies propose new constructions of regenerating codes [9], [17], [27], [37] that minimize the repair bandwidth, and locally repairable codes [18], [28], [35] that mitigate repair I/Os with extra storage. Some studies propose new repair-efficient techniques, such as lazy recovery that reduces the repair traffic by carefully delaying immediate repair operations [36], parallelizing and pipelining repair operations that reduce the repair time [22], [25], and adapting to dynamic workload changes by scheduling repair tasks in free timeslots [38]. For the scaling problem, some studies focus on designing scaling approaches that minimize the bandwidth under RAID-0 (i.e., no fault tolerance) [46], [50], RAID-5 (i.e., single fault tolerance) [14], [41], [47], [48], or RS codes [19], [43], [44], [49]. Maturana and Rashmi [24] study a scaling-related problem called code conversion, and propose convertible code constructions that minimize the I/Os in code conversion. Unlike the repair and scaling problems, StripeMerge studies how to minimize the bandwidth and mitigate the computational overhead in the wide-stripe generation problem.

Some studies also address the wide-stripe problem from different perspectives. VAST [4] employs locally decodable codes to improve repair performance of wide stripes. Haddock et al. [13] use general-purpose GPUs to improve decoding efficiency for wide stripes. A recent work ECWide [16] presents combined locality, the first mechanism that systematically addresses the wide-stripe repair problem and proposes efficient encoding and update schemes. In contrast, StripeMerge focuses on how to generate wide stripes efficiently and examines an inherently different problem from [16].

VII. Conclusions

We propose a novel mechanism, called StripeMerge, that merges narrow stripes to efficiently generate wide stripes for large-scale erasure-coded storage. We prove, via bipartite graph modeling, the existence of an optimal scheme for wide-stripe

generation. We further build StripeMerge with two heuristics, which realize efficient wide-stripe generation with only limited wide-stripe generation bandwidth overhead. Both simulations and Amazon EC2 experiments demonstrate the wide-stripe generation efficiency of StripeMerge over state-of-the-arts.

Acknowledgment: This work was supported by National Natural Science Foundation of China (61872414), Key Laboratory of Information Storage System Ministry of Education of China, and the Research Grants Council of Hong Kong (AoE/P-404/18). The corresponding author is Yuchong Hu.

REFERENCES

[1] Amazon S3 Glacier & S3 Glacier Deep Archive. https://aws.amazon.com/glacier/.
[2] corrupted fragment on decode #10. https://github.com/intel/isa-l/issues/10.
[3] Intel ISA-L. https://github.com/intel/isa-l.
[4] VastData. https://vastdata.com/providing-resilience-efficiently-part-ii/.
[5] M. K. Aguilera. Geo-distributed storage in data centers. Technical report, 2013.
[6] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of USENIX OSDI*, 2014.
[7] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC*, pages 539–551, 2017.
[8] L. Cheng, Y. Hu, and P. P. Lee. Coupling decentralized key-value stores with erasure coding. In *In Proc. of ACM SoCC*, pages 377–389, 2019.
[9] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory (TIT)*, 56(9):4539–4551, Sep 2010.
[10] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A Survey on Network Codes for Distributed Storage. In *Proc. of IEEE*, volume 99, pages 476–489, Mar 2011.
[11] 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.
[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. 2003.
[13] W. Haddock, P. V. Bangalore, M. L. Curry, and A. Skjellum. High performance erasure coding for very large stripe sizes. In *In Proc. of IEEE SpringSim*, pages 1–12, 2019.
[14] S. R. Hetzler. Data Storage Array Scaling Method and System with Minimal Data Movement, Aug. 7 2012. US Patent 8,239,622.
[15] J. E. Hopcroft and R. M. Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.
[16] 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. of USENIX FAST*, 2021.
[17] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage (TOS)*, 13(4):33, 2017.
[18] 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.
[19] 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 (TPDS)*, 26(6):1704–1717, 2015.
[20] IDC. Data age 2025. https://www.seagate.com/our-story/data-age-2025/.
[21] J. Lacan and J. Fimes. Systematic MDS Erasure Codes Based on Vandermonde Matrices. *IEEE Communications Letters*, 8(9):570–572, Sep 2004.
[22] R. Li, X. Li, P. P. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, pages 567–579, 2017.
[23] P. Luse and K. Greenan. Swift object storage: Adding erasure code. *SNIA Education September*, 2014.
[24] F. Maturana and K. V. Rashmi. Convertible codes: New class of codes for efficient conversion of coded data in distributed storage. In *Proc. of ITCS*, 2020.
[25] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proc. of ACM Eurosys*, page 30. ACM, 2016.
[26] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, pages 383–398, 2014.
[27] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*, pages 81–94, 2016.
[28] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Trans. on Information Theory (TIT)*, 2014.
[29] H. Philip. On representatives of subsets. *J. London Math. Soc*, 10(1):26–30, 1935.
[30] 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.
[31] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience*, 35(2):189–194, Feb 2005.
[32] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at USENIX FAST 2013, Feb 2013.
[33] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.
[34] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.
[35] 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 *Proc. of ACM VLDB Endowment*, pages 325–336, 2013.
[36] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, pages 1–7. ACM, 2014.
[37] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay codes: moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, page 139, 2018.
[38] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu. Dayu: fast and low-interference data recovery in very-large storage systems. In *Proc. of USENIX ATC*, pages 993–1008, 2019.
[39] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of Springer IPDPS*, Mar 2002.
[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, pages 307–320. USENIX Association, 2006.
[41] C. Wu and X. He. GSR: A Global Stripe-based Redistribution Approach to Accelerate RAID-5 Scaling. In *Proc. of IEEE ICPP*, 2012.
[42] S. Wu, Z. Shen, and P. P. Lee. On the optimal repair-scaling trade-off in Locally Repairable Codes. In *Proc. of IEEE INFOCOM*, 2020.
[43] 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. of IEEE SRDS*, 2020.
[44] 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 (TPDS)*, 27(9):2639–2652, Sep 2016.
[45] 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 (TC)*, 64(1):32–44, 2015.
[46] G. Zhang, J. Shu, W. Xue, and W. Zheng. SLAS: An Efficient Approach to Scaling Round-robin Striped Volumes. *ACM Trans. on Storage (TOS)*, 3(1):3, 2007.
[47] G. Zhang, W. Zheng, and K. Li. Rethinking RAID-5 Data Layout for Better Scalability. *IEEE Trans. on Computers (TC)*, 63(11):2816–2828, 2014.
[48] G. Zhang, W. Zheng, and J. Shu. ALV: A New Data Redistribution Approach to RAID-5 Scaling. *IEEE Trans. on Computers (TC)*, 59(3):345–357, 2010.
[49] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou. Toward optimal storage scaling via network coding: From theory to practice. In *Proc. of IEEE INFOCOM*, pages 1808–1816, 2018.
[50] W. Zheng and G. Zhang. FastScale: Accelerate RAID Scaling by Minimizing Data Migration. In *Proc. of USENIX FAST*, 2011.