Network-Aware Feasible Repairs for Erasure-Coded Storage

Márton Sipos, Josh Gahm, Narayan Venkat, and Dave Oran

***Abstract*— A significant amount of research on using erasure coding for distributed storage has focused on reducing the amount of data that needs to be transferred to replace failed nodes. This continues to be an active topic as the introduction of faster storage devices looks to put an even greater strain on the network. However, with a few notable exceptions, most published work assumes a flat, static network topology between the nodes of the system. We propose a general framework to find the lowest cost feasible repairs in a more realistic, heterogeneous and dynamic network, and examine how the number of repair strategies to consider can be reduced for three distinct erasure codes. We devote a significant part of the paper to determining the set of feasible repairs for random linear network coding (RLNC) and describe a system of efficient checks using techniques from the arsenal of dynamic programming. Our solution involves decomposing the problem into smaller steps, memorizing, and then reusing intermediate results. All computationally intensive operations are performed prior to the failure of a node to ensure that the repair can start with minimal delay, based on up-to-date network information. We show that all three codes benefit from being network aware and find that the extra computations required for RLNC can be reduced to a viable level for a wide range of parameter values.**

***Index Terms*— Distributed storage, matrix rank checks, network coding.**

I. Introduction and Related Work

A. Erasure Coding for Distributed Storage

IN RECENT years, distributed storage systems (DSS) have seen a trend towards erasure coding as a means to control the costs of storing and ensuring the resilience of large volumes of data. Even though traditional distributed storage systems employ replication to ensure data resilience, erasure coding provides equivalent or better resilience while using a fraction of the raw storage capacity required for replication. Both new technologies and increasing storage volume requirements suggest that erasure coding will continue to increase in importance as a factor in data center design. Offloading of encode and decode operations to GPUs, FPGAs and/or use of modern software libraries such as ISA-L, jerasure and Kodo promises to lower the computation costs of these operations, potentially expanding the set of cost-effective use cases for erasure coded storage. Additionally, the increased IOP density and IO bandwidth of next–generation storage devices, such as NVMe (Non-Volatile Memory Express), as compared with rotating media or earlier SSD devices, promises to lower the effective IO costs associated with coded storage, further expanding the set of use cases.

Manuscript received January 25, 2017; revised March 6, 2018; accepted April 3, 2018; approved by IEEE/ACM Transactions on Networking Editor M. Chen. Date of publication June 1, 2018; date of current version June 14, 2018. This work was supported by the ÚNKP-17-3-III New National Excellence Program of the Ministry of Human Capacities. The work of M. Sipos was supported by the Csibi Sándor Scholarship. *(Corresponding author: Márton Sipos.)*

M. Sipos was with Cisco Systems, Cambridge, MA 02142 USA. He is now with the Budapest University of Technology and Economics, 1117 Budapest, Hungary, and also with Aalborg University, 9100 Aalborg, Denmark (e-mail: siposm@aut.bme.hu).

J. Gahm was with Cisco Systems, Cambridge, MA 02142 USA. He is now with Facebook, Cambridge, MA 02142 USA (e-mail: gahm@fb.com).

N. Venkat was with Cisco Systems, Cambridge, MA 02142 USA. He is now with Amazon, Boston, MA 02210 USA (e-mail: narayan.venkat@gmail.com).

D. Oran was with Cisco Systems, Cambridge, MA 02142 USA. He is now with Network Systems Research and Design, Cambridge, MA 02138 USA (e-mail: daveoran@orandom.net).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/TNET.2018.2830800

Network interfaces have arguably seen a less dramatic increase in throughput than either storage or compute. A large bulk of the research in the area of erasure coding for distributed storage has focused on ameliorating the increased strain on the network during repair of lost erasure-coded data by reducing the amount of data transferred, commonly referred to as repair bandwidth. Unlike replicated storage where data can be recovered by simply copying the lost pieces from surviving nodes, repairing erasure coded pieces involves retrieving significantly more data. For example, Reed-Solomon (RS) is widely employed due to its optimal storage efficiency for a given level of reliability (more precisely, it is Maximum Distance Separable (MDS)). However, repairing lost pieces requires as many coded pieces as are required to recover the original data. To address this, several codes which perform repair more efficiently have emerged employing techniques such as functional repair [1], interference alignment [2] and piggybacking [3]. Dimakis *et al.* [1] characterizethe inherent trade-off between storage efficiency and repair bandwidth and introduce regenerating codes that achieve various points on the resulting trade-off curve. The two extremal points on the curve are of particular interest. Minimum Storage Regenerating (MSR) codes store the least amount of data to achieve a given level of redundancy. Minimum Bandwidth Regenerating (MBR) codes on the other hand store more data on each node, but require the least amount of data to be transferred during a repair, equal to the amount stored on the failed node.

Shah *et al.* [4] introduced the concept of flexible regeneration, where storage nodes contribute different amounts of data to repairs and introduced a lower bound on the total repair bandwidth. We introduced a slightly simpler lower bound for MSR codes in our previous paper on the topic [5] that forms the basis of our theoretical results for making functional repair network-aware. We have decided against the use of a cap on the amount of data transferred from any single node,

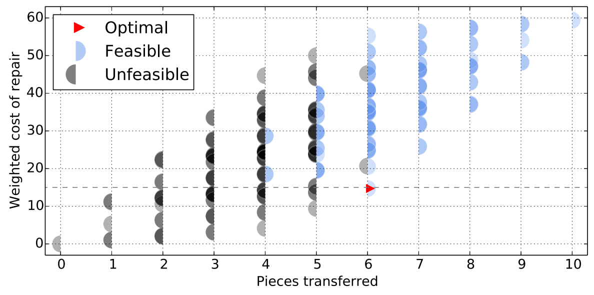

Fig. 1. An example of the network-aware repair space of an erasure code for which the optimal repair w.r.t. the weighted cost of repairs is not the one that transfers the fewest pieces.

as argued for in [4], as we felt it unduly excludes some repair strategies. The same research team has provided [6] a code for the MBR point that uses a form of exact repair (i.e. lost pieces are replaced with identical copies) termed 'repair-by-transfer'. This makes it interesting in severely bandwidth and computationally-limited systems. We refer to this code as RBT-MBR and include it in our evaluation. In contrast to exact repair, some codes employ functional repair, where lost pieces are replaced with functionally equivalent ones instead of exact copies. The basic idea of functional repair emerged from network coding [7], a highly effective technique to disseminate information through a network that allows intermediary nodes to transmit functions of the data they received as opposed to simply forwarding it appropriately. Ho *et al.* [8] proposed using functions based on linear coefficients randomly selected from a finite field and called the technique Random Linear Network Coding (RLNC). It was quickly applied to storage [9], [10]. The code's construction makes it well-suited for use in dynamic, heterogenous systems as it provides significant flexibility in the selection of pieces and nodes to retrieve and repair data from. Furthermore, it can be tailored to achieve any point on the trade-off curve provided a finite field of large enough size is used, even after the initial data distribution. However, it suffers from significant computational overhead and crucially, due to the random selection of coefficients, requires a relatively large finite field to retain data integrity with high probability. This makes it less suitable in its basic form for use in environment such as data centers, where even a minute probability of data loss is unacceptable. We dedicate a significant portion of the paper to dealing with this aspect, extending the usability of RLNC to these scenarios.

B. Network-Aware Repairs

Network topology and current traffic conditions play a crucial role in repair performance. To reflect these attributes, costs can be assigned to the transfer of pieces between nodes. Under these conditions, a solution that aims to minimize the number of transferred pieces would give a suboptimal solution. Figure 1 shows an example for a repair using RLNC in a two–rack setup where the cost of inter–rack transfers is 10 times higher than that of intra–rack transfers. The horizontal axis shows the number of pieces transferred by each repair. Traditionally, this has been considered as the de facto network cost of a repair and is sometimes denoted with γ [1] and referred to as repair bandwidth. The vertical axis shows the number of transferred pieces weighted with a linear cost function that reflects the current state of the network. Blue semicircles denote feasible repairs, grey semicircles unfeasible ones. We define a repair as feasible if the resulting system maintains its ability to reconstruct the data from any subset of nodes of a predefined size. Thus, unfeasible repairs don't necessarily result in immediate data loss, but rather decrease the reliability of the system over the course of several rounds of failures and repairs. A red triangle denotes the repair with the lowest weighted cost. A naive approach that is not network-aware would select one of the feasible repairs on the $x = 4$ column, a suboptimal choice for this particular cost function. This invokes two questions that this paper seeks to answer: how much do different types of codes benefit from being network-aware and where can the lowest cost feasible repairs be found in the repair space, independent of the cost function used. An answer to the latter would provide a solution to reduce the size of the space that should be considered.

Several codes have been proposed that take network topology into account while doing repairs. However, most do this when distributing the data initially, thus limiting their ability to adapt to dynamic network conditions. Li *et al.* [11] proposed a tree-based topology-aware repair scheme based on Prim's algorithm that doesn't have this limitation. The paper described a heuristic-based approach to manage the large repair space and deal with the scenario when a second storage node is lost during repair. Unfortunately, the practical applicability of this approach is slightly diminished by the delay such a selection process introduces. Other research has focused on looking at specific network topologies. Akhlaghi *et al.* [12] grouped nodes into two sets, a "cheap" and an "expensive" set, based on the cost of access. They introduced generalized regenerating codes and showed that by downloading more pieces from "cheap" nodes, the weighted cost of repairing failed nodes can be reduced. Gastón *et al.* [13] presented a similar model for a 2 rack system employing regenerating codes. It considered the different cost of accessing inter-rack and intra-rack data as well as the location of the newcomer node to define a threshold function which minimizes the amount of stored data per node and the bandwidth needed to repair a failed node. More recently, Hu *et al.* [14] introduced double regenerating codes, specifically tailored to minimizing inter-rack traffic in a multi-rack data center. They proposed a two-stage regeneration procedure that recombines data stored inside racks before using it in the repair. They show gains of up to 45.5% compared to a regenerating code at the MSR point, assuming that intra-rack traffic is free. Thus, a preliminary answer to the first question is that at least some erasure codes benefit from being network aware in specific cases.

C. Checking the Feasibility of Repairs

For codes employing exact repair, the code construction determines which repairs are feasible. For codes employing functional repair, an information flow graph is typically used to determine what lower bounds must be met on the amount of data transferred during repairs to ensure the maximum flow has a given value. Since for RLNC the coefficients used to create the replacement pieces are drawn uniformly randomly from a finite field, there is a non-zero probability that they introduce unwanted linear dependencies not represented on the information flow graph. A significant amount of work has been done to study this probability [15], [16]. Hu *et al.* [17] proposed a heuristic two-phase checking mechanism on the coefficient matrices. They noted that as long as a file is split into a small number of fragments and a single repair is checked for each potential failure, the number of matrices for which the rank should be checked stays manageable. This number scales linearly when multiple repairs need to be checked, making it inefficient for our proposed network-aware solution.

We propose a novel mechanism to reduce the number of computations based on Gaussian elimination. This is critical in enabling the checks to be performed on data distributions with a large number of storage nodes and files split into many fragments. Beyond reducing the computational load on the system, it plays an important role in our proposed framework by decreasing the time necessary to find a set of feasible repairs. Thus, it limits the probability that a potentially unchecked repair strategy is used or alternatively that the repair is delayed to wait for the checks to finish. We base our solution on the realization that the matrices that need to be checked share many of their rows. Therefore, we propose reusing submatrices after they have been reduced to an upper triangular form and memoized, a common technique of dynamic programming. Larger matrices can then be built by merging smaller matrices prepared in advance. Furthermore, many of these smaller matrices can also be reused in subsequent generations of failure and repair, further decreasing computational costs over the lifetime of the system. While our work is motivated by the challenges posed by making RLNC usable in environments such as data centers, it is applicable for other erasure codes that employ functional repair. It can also be used to select feasible repairs if the exact lower bounds on the amount of data that needs to be transferred is not known.

D. Structure and Overview of the Contributions of the Paper

Our first contribution is to make the repair of erasure-coded data network-aware by introducing a general framework that computes the feasibility of different possible repairs in advance. When a storage node fails, a repair is selected based on some cost function that reflects the current state of network connectivity among the storage nodes. By performing the potentially computationally-intensive feasibility checks in advance, the system is able to react to a node loss quickly and can base the repair selection on up-to-date network traffic data. The paper investigates the gains for different types of erasure codes. The practical applicability of the proposed framework is also considered by presenting techniques to reduce the number of repairs to consider independent of the cost function in use. This aspect is especially important for RLNC, where the set of feasible repairs of potentially lowest cost is of exponential size when using a naive approach. Our second contribution is a technique to make checking the feasibility of a large number of repairs less computationally demanding. We present two methods to decompose the problem into smaller parts and formulate some of the general properties of decompositions. We also propose a technique to apply a decomposition as a schema for the actual checks as part of our proposed framework. We characterize the effectiveness of our solutions using both analytic and simulation-based tools.

Section II defines the concepts and models formally and includes an algorithmic definition of our proposed network-aware repair framework and decomposition method. Section III examines different erasure codes and defines functions that determine the location of relevant, potentially minimal cost feasible repairs for each. Section IV looks at the cost of performing the feasibility checks and describes two algorithms to find good decompositions. Section V provides experimental evidence showing the benefits of network-awareness and decompositions. Finally, Section VI summarizes our findings.

II. System Model and Network-Aware Repair Framework

A. System Model

A file to be stored in the DSS is broken up into k pieces of identical size. Then, it is encoded using an erasure code to produce n ($n \geq k$) coded pieces. These are then distributed to the N ($n \geq N$) nodes: $\Omega_N = \left(node_1 \; node_2 \; \cdots \; node_N\right)$, with each storing exactly α ($\alpha = \frac{n}{N}$). When $node_f$ fails, all pieces it stored are considered lost and must be repaired onto a replacement node. We designate the replacement node with the same name and consider repairs, where the surviving nodes can transfer different numbers β_i of pieces to $node_f$: $\xi_j = \left(\beta_1 \; \beta_2 \cdots \beta_N\right)$, including 0 ($j$ is the index of the repair in an arbitrary ordering of all repairs). We call all possible repairs of a code where $node_f$ was lost its repair space: $\Xi_f = \{\xi_j \mid 0 \leq \beta_i \leq \alpha, \; \beta_f = 0\}$ and use the term generation to denote a round of loss and repair. We require the system to maintain its properties over an arbitrarily large number of generations. We only consider single node losses as they are most common in systems with well-separated failure domains [18]. Performing concurrent repairs allows for techniques that can further reduce network usage [19], but these are outside the scope of this paper. We expect our proposed framework to be useful in the case of multiple concurrent failures in reducing network costs. We assume that storage nodes are able to perform some basic operations on the data during repairs, mainly additions and multiplications. We consider storage systems and codes with parameters that are $N, n, k, \alpha \in \mathbb{N}^+$, $\beta_i \in \mathbb{N}$. A summary of notations is presented in Table I.

We consider a repair feasible if the resulting system state maintains data recoverability after sustaining subsequent L concurrent node losses. Each code, based on its parameters, therefore has a maximum number of L nodes it can lose concurrently while maintaining data recoverability, usually chosen to be $L = \lfloor \frac{n-k}{\alpha} \rfloor$.

Definition 1 (Feasible Repair): A repair is feasible if following its execution, data is recoverable from any $N - L$ nodes.

For codes employing exact repair, such as Reed-Solomon and RBT-MBR, the set of feasible repairs $\tilde{\Xi_f}$ as well as L is

TABLE I
TABLE OF NOTATIONS

N	$\triangleq$	the number of nodes
L	$\triangleq$	the number of unavailable nodes the system must support while maintaining data availability
α	$\triangleq$	the number of encoded symbols stored on a node, corresponding to rows in a coefficient matrix
r	$\triangleq$	the number of repairs to check for a possible node failure
$\mathbf{M}$	$\triangleq$	the set of matrices that need to be checked
$\mathbf{D}$	$\triangleq$	the set of matrices that form a decomposition
Ψ	$\triangleq$	a mapping that defines how each matrix in $\mathbf{D}$ is broken down into two smaller matrices
S_i	$\triangleq$	the set of matrices of size $i\alpha \times k$ in upper triangular form (UTF)
S_i^{sel}	$\triangleq$	the set of matrices of size $i\alpha \times k$ selected in decomposition D
s_i	$\triangleq$	an arbitrarily selected matrix from S_i or S_i^{sel}
$s_i^{(k)}$	$\triangleq$	the k-th matrix from S_i or S_i^{sel} given some arbitrarily defined order
$lz(i)$	$\triangleq$	the number of leading zeros in $s_i \in S_i$
$d(a,b)$	$\triangleq$	the number of divisions needed to get $s_i = s_a \oplus s_b$ into UTF if both s_a and s_b are in UTF
$m(a,b)$	$\triangleq$	the number of multiplications (same as the number of additions) needed to get $s_i = s_a \oplus s_b$ into UTF if both s_a and s_b are in UTF
$d_{Gauss}(k)$	$\triangleq$	the number of divisions needed to get s_k into UTF using Gaussian elimination
$m_{Gauss}(k)$	$\triangleq$	the number of multiplications/ additions needed to get s_k into UTF using Gaussian elimination

defined by the structure of the code. For regenerating codes employing functional repair, the set of feasible repairs is constrained by both the min-cut of an information flow graph [7] and the coefficient selection method. On the information flow graph, a flow to a data collector with a value of at least k must be maintained with any L vertices from the final level of topological sorting removed from the graph. For codes that generate coefficients randomly such as RLNC, further checks are necessary to ensure that the selection of coefficients does not introduce linear dependence not portrayed on the information flow graph. In this sense, on an information flow graph with edges of capacity 1, i edge-disjoint paths must correspond to i linearly independent pieces retrieved by the data collector. To ensure this condition is met when using randomly generated coefficients, Gaussian elimination can be used to check the rank of several coefficient matrices that correspond to potentially retrieved data.

We denote the set of matrices to check with $\mathbf{M}$ and define a mechanism that performs these checks in a computationally-efficient manner. Each matrix in $\mathbf{M}$ is obtained by concatenating the encoding coefficient vectors of the packets stored on $N - L - 1$ surviving and one replacement node, the data on each node represented by α rows of coefficients. Matrices that don't contain repaired rows need not be checked as they have been checked before the repair. Similarly, if the same node fails in two successive generations, the checks can be skipped entirely, since no new coefficients are introduced into the system. To denote matrices of size $i\alpha \times k$, we use an index set based on which nodes the matrix contains rows of encoding coefficients from: $s_i = \{index_1, index_2, \cdots, index_i\}$, where $|s_i| = i$. For example, $s_3 = \{3, 4, 5\}$ contains rows from $node_3$, $node_4$ and $node_5$.

B. Representing the Cost of Network Transfers

We define the cost functions using matrix $\mathbf{C}$, where $c_{i,j}$ denotes the cost to transfer a single piece from $node_i$ to $node_j$ and column $\mathbf{C}[j]$ the costs associated with transfers to $node_j$. We introduce two restrictions on $\mathbf{C}$. The diagonal elements must be $c_{i,i} = 0$, while all others $i \neq j$, $c_{i,j} \geq 0$.

$$\mathbf{C} = \begin{pmatrix} 0 & c_{1,2} & \cdots & c_{1,N} \\ c_{2,1} & 0 & \cdots & c_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,1} & c_{N,2} & \cdots & 0 \end{pmatrix} \tag{1}$$

We use this general way of modeling costs to make it applicable to different network topologies and traffic patterns. It can be based on any number of measured parameters such as available bandwidth, latencies, number of dropped packets, queueing delays, etc. It can be used, but is not limited, to minimize the total time required for repairing lost data. We make the assumption that the cost of transferring a single piece from $node_i$ to $node_j$ is not dependent on the total number of pieces sent between them in the period in which the cost is regarded as accurate. This assumption is valid if the examined period is short or the repair traffic is a negligible fraction of the traffic flowing on the same links.

We evaluate the network-aware cost-weighted repair space of the code as shown in Figure 1, where the weighted cost for repairing data on $node_f$ using repair ξ_j is $cost(\xi_j) = \xi_j \mathbf{C}[f]$, where $\mathbf{C}[f]$ is used to denote column f.

C. A Network-Aware Repair Framework

Our proposed framework selects the lowest cost repair and is independent of erasure code and network topology, as illustrated in Algorithm 1. Whenever there is a change in the layout of the data (the initial distribution of data and any subsequent repairs), the set of feasible repairs $\tilde{\Xi}_f$ is computed for each possible directly subsequent node failure using $precompute\_feasibility$, defined in Algorithm 2. Thus, the potentially computation–heavy search for feasible repairs takes place before an actual node failure. When a failure does occur, the system can react quickly, calculating the cost for each feasible repair based on a cost function reflecting up-to-date network conditions. The lowest cost repair can be performed with minimal delay. Then, the set of feasible repairs must be precomputed again to prepare the system for the following node failure and so on.

The implementation of the $is\_feasible()$ function from Algorithm 2 is determined by the erasure code in question and the definition of feasibility, which is left to the designer and operator of the system. For our purposes, we will define a repair as feasible if the system maintains the same level of reliability after it is performed, i.e. data remains recoverable from any $N - L$ nodes.

The practical applicability of our proposed framework is determined by the complexity of the $is\_feasible()$ function and the size of Ξ_f and $\tilde{\Xi}_f$. For codes that select encoding coefficients at random, computational complexity is also determined by the values of parameters N, n, k, α as the rank of a potentially large number of matrices in $\mathbf{M}$ must be checked.

Algorithm 1 Network-Aware Repair Framework

```
1: ↯ Initial data distribution ↯
2: $precompute\_feasibility$
3: repeat
4:     ↯ wait for $node_f$ failure ↯
5:     $min\_cost := \infty$
6:     for $\xi_j \in \tilde{\Xi}_f$ do
7:         if $cost(\xi_i) = \xi_i \mathbf{C}[f] < min\_cost$ then
8:             $min\_cost := cost(\xi_i)$
9:             $\xi\_sel := \xi_i$
10:        end if
11:    end for
12:    execute $\xi\_sel$
13:    $precompute\_feasibility$
14: until false
```

Algorithm 2 Precompute Feasibility

```
1: procedure $precompute\_feasibility$
2:     $\tilde{\Xi}_i := \emptyset$
3:     for $node_i \in \Omega_N$ do
4:         for $\xi_j \in \Xi_i$ do
5:             if $is\_feasible(\xi_j)$ then
6:                 $\tilde{\Xi}_i := \tilde{\Xi}_i \cup \xi_j$
7:             end if
8:         end for
9:     end for
10: end procedure
```

D. Decomposing Matrix Rank Checks Into Reusable Parts

To perform the rank checks efficiently, we propose decomposing the Gaussian elimination performed for the coefficient matrices in $\mathbf{M}$ into smaller steps that can be shared between different checks and across subsequent failure and recovery generations. Once a set of steps and order is identified, it can be used as a schema as long as N, L and α don't change.

Definition 2: A *decomposition*$(\mathbf{D}, \Psi)$ is a set of matrices $\mathbf{D} = \{s_1^{(1)}, s_1^{(2)}, \dots, s_1^{(k_1)}, \dots, s_j^{(k_j)}\}$, where s_i is a matrix constructed by concatenating row-by-row the encoding vectors of the packets stored on i nodes, $j \leq N - L$, $k_j \leq \binom{N}{j}$, $\mathbf{M} \subset \mathbf{D}$ and a mapping Ψ to match each matrix in $\mathbf{D}$ except those containing rows from a single node (s_1) to a pair of matrices from $\mathbf{D}$.

A decomposition can also be represented with a directed acyclic graph that defines the dependencies between matrices as shown in Figure 3. The matrices in $\mathbf{D}$ are the vertices of the graph and a root element is added that connects to all nodes representing matrices in $\mathbf{M} \subset \mathbf{D}$. Edges go between nodes according to Ψ to denote which pair of matrices should be transformed to upper triangular form (UTF) before tackling a given matrix. The checks are then a traversal of the graph, and each visited vertex (matrix) is memoized in upper triangular form to reduce the number of operations in subsequent visits. The fewer matrices that need to be visited and the lower the cost of each visit, the fewer computations are necessary.

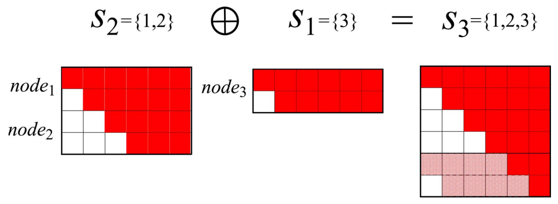

Fig. 2. An example of merging s_2 and s_1, $\alpha = 2$.

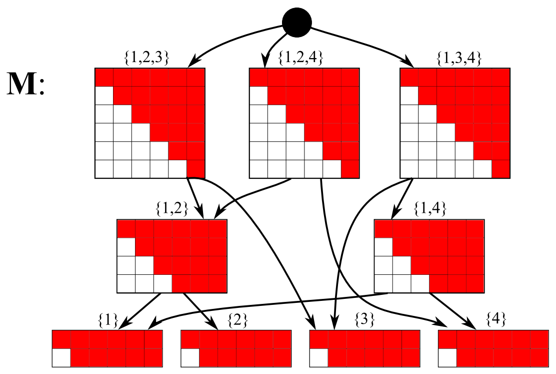

Fig. 3. An example of a valid decomposition.

We group matrices of identical size to simplify the notation and use S_i to denote the set of matrices of size $i\alpha \times k$ that includes all possible combinations of selecting all rows from i nodes. $S_i^{sel} = \{s_i | s_i \in S_i \cap \mathbf{D}\}$ is the set of matrices that are selected to be part of $\mathbf{D}$ from S_i. This grouping determines the levels of the topological sorting of $\mathbf{D}$. In order to traverse the graph, we define the merge operation $s_a \oplus s_b = s_i$, where $a + b = i$ and $s_a \cap s_b = \emptyset$ as follows. First, the rows of s_b are appended to the end of s_a. Second, the rows of s_a are used to create leading zeros in the appended rows to get the resulting s_i into UTF. An overview is shown in Figure 2 with 0 elements shown as white boxes, non-zero elements shown with solid red and elements that are going to be transformed to 0 shown with a wavy red fill. The mapping Ψ associates a pair of matrices s_a and s_b with each matrix $s_i \in S_i^{sel}, i > 1$.

A valid decomposition $\mathbf{D}$ is one that can be used to recreate all matrices in $\mathbf{M}$ in upper triangular form using the merge operation. Furthermore, it must also provide a means to build all matrices in $\mathbf{D}$ except for matrices containing rows from a single node. For example, $\mathbf{D} = \{\{1\}; \{2\}; \{3\}; \{4\}; \{1,2\}; \{1,4\}; \mathbf{M}\}$, shown in Figure 3, is a valid decomposition for $\mathbf{M} = \{\{1,2,3\}, \{1,2,4\}, \{1,3,4\}\}$, but $\mathbf{D}' = \{\{1\}; \{3\}; \{4\}; \{1,2\}; \{2,3\}; \mathbf{M}\}$ (not shown) is not, as neither $\{1,2\}$, $\{2,3\} \in \mathbf{D}'$ nor $\{1,3,4\} \in \mathbf{M}$ can be built from merging elements of $\mathbf{D}'$. This example highlights that valid decompositions must contain all matrices that contain rows associated with packets on a single node as well as intermediate matrices.

III. Locating Potentially Lowest-Cost Feasible Repairs

In this section we define specific cost functions for the different erasure codes in order to reduce the number of repairs to consider and to be able to characterize the repair space of each code in terms of where the lowest cost feasible repairs are. The codes were chosen to cover both exact and functional repair and both MSR and MBR points on the storage – repair bandwidth trade-off curve.

Let us look at finding the minimum cost feasible repair ξ_{min} and its associated cost: $\kappa = cost(\xi_{min}) = \sum_{i=1,i\neq f}^{N} \beta_i c_{i,f}$ after losing the data stored on $node_f$.

A. Reed-Solomon

Here we examine decoding-based repair for Reed-Solomon(RS) as this can be applied to any linear MDS code. We restrict our evaluation to the $\alpha = 1$ case to be in line with how RS is generally used for storage. Let $c^{(1)}, c^{(2)}, \cdots, c^{(N-1)} : c^{(i)} \in set(\mathbf{C}[f]) \setminus c_{f,f}$ be a permutation of costs in ascending order and $\beta^{(1)}, \beta^{(2)}, \cdots \beta^{(N-1)}$ the corresponding number of transferred pieces. Thus, the cost of the minimal cost repair is shown on Equation (2) and the number of feasible repairs to consider given no knowledge of $\mathbf{C}$ is $\left|\tilde{\Xi_f}\right| = \binom{n}{k}$.

$$\kappa_{\text{RS}} = \sum_{i=1}^{k} c^{(i)} \tag{2}$$

B. RBT-MBR

There are two distinct repair strategies to consider. Ideally, each surviving node transfers a single encoded piece ($\beta_i = 1, i \neq f$) as defined in [6]. Alternatively, if at least k distinct pieces are transferred, the decoding of the embedded MDS code can take place and missing code words can be re-encoded. Whilst this second repair strategy involves additional bandwidth and computation, it can result in lower transfer costs for some $\mathbf{C}$. Let $c^{(i)}$ and $\beta^{(i)}$ be defined as previously. The cost of the optimal repair $\kappa_{\text{RBT-MBR}}$ is specified in Equation (3) based on the two repair strategies.

$$\kappa_{\text{RBT-MBR}} = \min\left(\sum_{i=1}^{N-1} c^{(i)}, \sum_{i=1}^{N-L} (\alpha - i + 1) c^{(i)}\right) \tag{3}$$

The first term is the cost of transferring a single piece from each surviving node. The second term expresses retrieving as many pieces from the lower cost nodes as possible without getting duplicates. $\sum_{i=1}^{N-L}(\alpha - i + 1) = k$ because the embedded code is MDS and the way RBT-MBR is constructed [6]. With no knowledge of $\mathbf{C}$, the number of repairs that are potentially lowest cost is reduced to $\left|\tilde{\Xi_f}\right| = 1 + (N-L)!\binom{N-1}{N-L}$.

C. Network Coding

Unlike the previous codes, network coding does not have a fixed repair strategy, thus we resort to analyzing the information flow graph (IFG) to limit the search for $\tilde{\Xi_f}$. RLNC has been shown to be able to store and recover as many individual encoded pieces as the max-flow of the IFG as long as some constraints are met [20]. We use the system of checks described in Section II-D to select coding coefficients that enable the cut-set bounds to be be achieved. $\Omega_N^{(g)} \setminus node_f^{(g)}$ is the set of surviving storage nodes in generation g of loss and repair, and let $[S^{(g)}]^l$ denote the set of its l-subsets, i.e. $[S^{(g)}]^l := \{X \subset (\Omega_N^{(g)} \setminus node_f^{(g)}) \big| |X| = l\}$.

Theorem 3: Consider a DSS that uses an erasure code that achieves the cut-set bounds for the information flow graph and initially has the property of being able to recover the data from any $(N-L)$ nodes, where $(N-L)\alpha = k$. It maintains this property through an arbitrarily large number of single node failure and repair generations for any failure pattern if and only if the condition in Equation (4) is met.

$$\forall g \geq 1, \quad \forall s \in [S^{(g)}]^L : \sum_{node_i^{(g)} \in s} \beta_i^{(g)} \geq \alpha \tag{4}$$

Less formally, during a repair in generation g, any L sized selection of nodes must transfer at least α pieces for the system to be able to sustain the loss of L nodes following the repair. This constraint is sufficient to ensure that the number of edge-disjoint paths on the IFG between the data source and a data collector does not decrease to below k if L nodes are subsequently lost in the following generation. It is also necessary for MSR codes, where $(N-L)\alpha = k$.

Corollary 4: Theorem 3 also applies if $(N-L)\alpha > k$ with the change that the condition in Equation (4) is only sufficient, not necessary.

We have included a proof for both in the appendix (see in Supplementary Material).

Let $\beta^{(1)}, \beta^{(2)}, \cdots, \beta^{(N-1)}$ be a permutation of the number of pieces transferred from remaining nodes of ascending order and $c^{(1)}, c^{(2)}, \cdots, c^{(N-1)}$ the respective costs from $set(\mathbf{C}[f]) \setminus c_{f,f}$. Taking Equation (4) into consideration, we can define a more specific cost function for the optimal repair in Equation (5), by only considering repairs $\sum_{i=1}^{N-1} \beta_i \leq k$.

$$\begin{aligned} \kappa_{\text{RLNC-MSR}} &= \sum_{i=1}^{L} c^{(i)}\beta^{(i)} + \sum_{i=L+1}^{N-1} c^{(i)}\beta^{(L)} \\ &= \sum_{i=1}^{L-1} c^{(i)}\beta^{(i)} + \beta^{(L)} \sum_{i=L}^{N-1} c^{(i)} \end{aligned} \tag{5}$$

The first term expresses the cost for the L lowest values of $\beta^{(i)}$, the second term the cost for the rest of the nodes. Each of these must transfer at least $\beta^{(L)}$ to satisfy Equation (4). $\kappa_{\text{RLNC-MSR}}$ is minimized if the $c^{(i)}$ are in descending order, i.e. transferring more from cheaper nodes and less from expensive ones. The free variables are thus reduced to $\beta^{(1)}, \beta^{(2)}, \cdots, \beta^{(L)}$. Given that Equation (4) should be satisfied with equality for ξ_{min}, this leads to a significant reduction in the number of potential repairs to consider shown in Equation (6). Furthermore, it determines the positions of the lowest cost feasible repairs in Ξ_f and once $\mathbf{C}$ is known, the optimal repair can quickly be selected.

$$\left|\tilde{\Xi_f}\right| = \left|\left\{\xi : \sum_{i=1}^{L} \beta^{(i)} = \alpha\right\}\right| \tag{6}$$

This is an integer partitioning problem on α that is constrained by limiting solutions to those with L additive parts. The number of non-constrained partitions is given by a recurrence formula based on Euler's pentagonal number theorem. The first elements can also be found in the OEIS as sequence

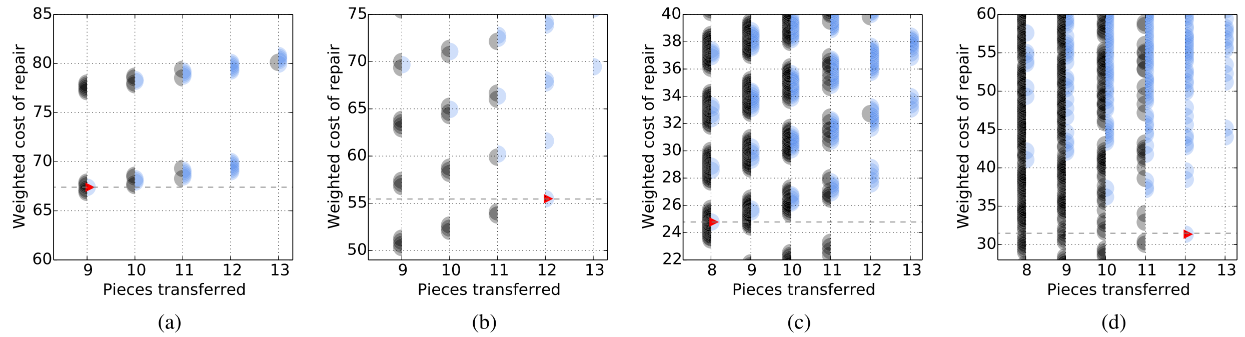

Fig. 4. Case study for RLNC(k, α, N). (a) RLNC(12, 6, 4) $c_1 < c_2+c_3$. (b) RLNC(12, 6, 4) $c_1 > c_2+c_3$. (c) RLNC(12, 4, 6) $c_1+c_2 < 2(c_3+c_4+c_5)$. (d) RLNC(12, 4, 6) $c_1 + c_2 > 2(c_3 + c_4 + c_5)$.

A000041 [21]. The number of solutions with L parts is equal to the number of partitions in which the largest part is of size L [22]. A similar recurrence formula exists for this constrained version of the problem [23].

As the bound in Equation (4) is sufficient to ensure data survival for all parameters, we may apply the previous results for non-MSR codes as well. However, it is possible to define tighter bounds for these codes, shown on Equation (7) [4].

$$\sum_{i=1}^{N-1} \beta^{(i)} \geq \gamma(\beta_{\max})$$

where

$$\gamma(\beta_{\max}) = \max(\alpha - \beta_{\max}, k \bmod \alpha) + \lfloor k/\alpha \rfloor \tag{7}$$

Shah *et al.* [4] introduce a cap on the amount of pieces any single node transfers, $1 \leq \beta_{\max} \leq \alpha$. They argue against full flexibility ($\beta_{\max} = \alpha$), as it involves transferring at least k pieces. However, for some $\mathbf{C}$, transferring k pieces is actually the lowest cost repair strategy. An example can be seen in Figure 4d. Based on Equation (7), we define the costs of optimal repairs for a given $\beta_{\max}$ on Equation (8).

$$\kappa_{\text{RLNC}}(\beta_{\max}) = \sum_{i=1}^{\lfloor \gamma(\beta_{\max})/\beta_{\max} \rfloor} c^{(i)} \beta_{\max} + c^{(\lfloor \gamma(\beta_{\max})/\beta_{\max} \rfloor + 1)} (\gamma(\beta_{\max}) \bmod \beta_{\max}) \tag{8}$$

To enumerate the set of feasible repairs based on these bounds, the procedure must be repeated for all possible integer values of $\beta_{\max}$, merging results. The number of feasible repairs of potentially optimal cost is shown on Equation (9), giving α similar constrained integer partitioning problems as for the MSR point. In this case the constraint in each is that the largest part of a partition must be at most $\beta_{\max}$. Due to the equivalence described in [22], it is the same problem as for the MSR point. Unfortunately, it is not clear whether this approach returns all possible lowest cost feasible repairs.

$$\left|\tilde{\Xi}_f\right| \geq \sum_{\beta_{\max}=1}^{\alpha} \left| \left\{ \xi : \sum_{i=1}^{N-1} \beta^{(i)} = \gamma(\beta_{\max}) \right\} \right| \tag{9}$$

1) Case Study: Let us look at two sets of parameters at the MSR point for which RLNC behaves differently depending on $\mathbf{C}$. We have selected these particular sets because of their low number of potentially minimum cost repairs. Let us assume with no loss in generality that $node_N$ has failed and $c_i = c_{i,N}$ are in ascending order. Figure 4 shows zoomed-in views of the part of Ξ_f containing ξ_{min}.

First, we look at $k = 12$, $\alpha = 6$, $N = 4$ and require that $L = 2$ failures be supported. Considering Equation (5) and assuming repairs do not introduce linear dependence, only 4 of them need to be compared to find ξ_{min}.

$$\xi_1 = (3\ 3\ 3\ 0), \quad \xi_2 = (2\ 4\ 4\ 0),$$
$$\xi_3 = (1\ 5\ 5\ 0), \quad \xi_4 = (0\ 6\ 6\ 0)$$

For $c_1 = c_2 + c_3$ all four repairs have the same cost. For $c_1 < c_2 + c_3$, ξ_1, which is the most balanced repair with the least amount of pieces transferred has the lowest cost as shown in Figure 4a. On the other hand, for $c_1 > c_2 + c_3$, $cost(\xi_1) > cost(\xi_2) > cost(\xi_3) > cost(\xi_4)$, i.e. the repair transferring the most amount of pieces has the lowest cost as shown in Figure 4. Thus, in these cases a traditional mechanism that only tries to minimize the amount of transferred data will sub-optimally pick ξ_1, giving an error of $cost(\xi_1) - cost(\xi_4) = c_1 - c_2 - c_3$. More importantly, ξ_2 and ξ_3 will not be the lowest cost repairs regardless of $\mathbf{C}$, thus the number of relevant repairs whose feasibility must be checked is further reduced to just those transferring 9 and 12 pieces, ξ_1 and ξ_4 in this case.

Second, we look at $k = 12$, $\alpha = 4$, $N = 6$ and require that $L = 3$ node failures be supported. In this case the lowest cost feasible repairs based on Equation (5) are:

$$\xi_1 = (1\ 1\ 2\ 2\ 2\ 0), \quad \xi_2 = (0\ 2\ 2\ 2\ 2\ 0),$$
$$\xi_3 = (0\ 1\ 3\ 3\ 3\ 0), \quad \xi_4 = (0\ 0\ 4\ 4\ 4\ 0)$$

The cut-off point between ξ_1 and ξ_4 is $c_1 + c_2 = 2(c_3 + c_4 + c_5)$. Due to the limited number of ways the number 4 can be reduced to additive components, there are no minimal cost feasible repairs with a total of 9 or 11 transferred linear combinations. Thus, there might not be a clear decreasing or increasing order of costs like in the previous example as shown in Figure 4c and 4d. Therefore, more repairs must be checked.

IV. Efficient Method to Check the Feasibility of Repairs for RLNC

A. Enumerating the Encoding Coefficient Matrices to Check

Having looked at finding the set of feasible, potentially lowest cost repairs, we now turn our attention to the selection of encoding coefficients for RLNC. So far, we have determined which repairs are feasible by examining cut-sets on an information flow graph. This section complements this by describing a system of checks that ensures all encoding matrices that may be defined by the graph have a rank of k, i.e. contain at least one invertible $k \times k$ submatrix. This is necessary since RLNC uses randomly generated coefficients.

Let us look at how these matrices can be enumerated through an example: $k = 4$, $\alpha = 2$, $N = 4$ and $L = 2$. With no knowledge of which node will fail, we must consider all feasible repairs that have the potential to be of lowest cost for each. In this case, by solving the integer partitioning problem for RLNC-MSR described in Section III-C, we find there is a single feasible repair for each potential failure that may have minimal cost: $\xi_1 = \begin{pmatrix}0 & 1 & 1 & 1\end{pmatrix}$, $\xi_2 = \begin{pmatrix}1 & 0 & 1 & 1\end{pmatrix}$, $\xi_3 = \begin{pmatrix}1 & 1 & 0 & 1\end{pmatrix}$ and $\xi_4 = \begin{pmatrix}1 & 1 & 1 & 0\end{pmatrix}$, i.e. $\beta_i = 1$. Each will generate $\alpha = 2$ rows of coefficients, which we will denote with $\{1'\}$, $\{2'\}$, $\{3'\}$ and $\{4'\}$ to reflect the node with which they will be associated. To ensure that data is recoverable even if L nodes are unavailable, we must check coefficient rows associated with all combinations of sets of nodes of size $N-L$. However, only matrices that also contain rows from repaired nodes must be checked, since others must have been previously checked. If checks fail, a new set of coefficients must be generated for that repair. Thus,

$$\begin{aligned}\mathbf{M} = \{&\{1',2\};\{1',3\};\{1',4\};\\ &\{1,2'\};\{2',3\};\{2',4\};\\ &\{1,3'\};\{2,3'\};\{3',4\};\\ &\{1,4'\};\{2,4'\};\{3,4'\}\}.\end{aligned} \tag{10}$$

Since a naive approach involving Gaussian elimination that checks the rank of individual matrices one by one is computationally expensive, we propose decomposing the process into reusable steps as described in Section II-D.

B. Characterizing the Costs Associated With the Checks

We would like to use decompositions that result in a small number of computations and matrices to memoize. Unfortunately, the number of valid decompositions is large and with increasing N, we quickly reach a combinatorial explosion. This section focuses on deriving the computational and storage costs associated with a decomposition and motivates the choice of algorithms for selecting effective decompositions.

1) The Cost of Reaching Upper Triangular Form Using Basic Gaussian Elimination: we examine the number of divisions and pairs of multiplications and additions needed to transform a $k \times k$ matrix into UTF as a baseline. An example is shown in Figure 5 for $k = 6$. For $\alpha = 2$, this corresponds to checking the coefficients associated with data stored on 3 nodes. Several simplifications can be made to the general Gauss elimination algorithm to save on computational cost: the back substitution step can be skipped and it is not necessary to reduce pivot elements to 1 (reduced row echelon form). Furthermore, all operations can be performed solely on the coefficient matrices.

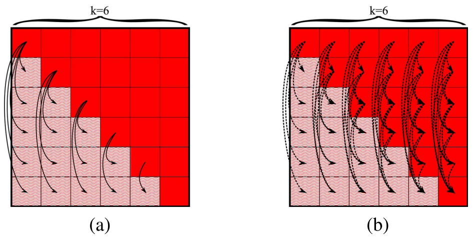

Fig. 5. Required operations for reducing a matrix ($k = 6$) to upper triangular form using Gaussian elimination. (a) Divisions. (b) Multiplications and additions.

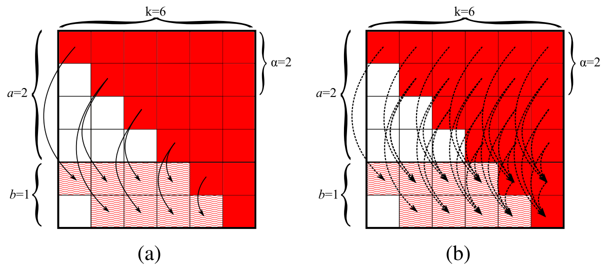

Fig. 6. Required operations for reducing a matrix ($k = 6$) to upper triangular form using merging $a = 2, b = 1, \alpha = 2$. (a) Divisions. (b) Multiplications and additions.

The number of divisions is the number of elements below the diagonal for square matrices:

$$d_{\text{Gauss}}(k) = \sum_{j=1}^{k-1} j = \frac{k(k-1)}{2} \tag{11}$$

The number of multiplications is the same as the number of additions. It can be calculated for square matrices by counting for all elements on or above the diagonal the number of elements below them:

$$m_{\text{Gauss}}(k) = \sum_{j=1}^{k-1} (k-j)(k-j+1) = \frac{k(k^2-1)}{3} \tag{12}$$

2) The Cost of Merging $s_a \oplus s_b = s_i$: the computational cost of getting the result of merging UTF matrices of size $a\alpha \times k$ and $b\alpha \times k$ into a UTF, where $a + b = i$ and $i \geq 2$. This includes the practically important case where $a = b = i/2$.

The number of leading 0 elements in an UTF matrix of size $i\alpha \times k$ (where $i\alpha \leq k$) gives a good indication on the number of operations that can be skipped when performing a merge:

$$lz(s_i) = lz(S_i) = lz(i) = \sum_{j=0}^{i\alpha-1} j = \frac{(i\alpha-1)i\alpha}{2} \tag{13}$$

The number of divisions can be calculated by subtracting from the total number of elements that need to be reduced to 0 those that have already been reduced in the two submatrices:

$$\begin{aligned}d(a,b) &= lz(i) - lz(a) - lz(b)\\ &= \sum_{j=1}^{b\alpha} a\alpha = ab\alpha^2.\end{aligned} \tag{14}$$

Equation (15) gives the number of multiplications and additions. We have included how we derived it in the appendix (see in Supplementary Material).

$$m(a,b) = ab\alpha^2 \left(k - \frac{(a+b)\alpha}{2} + 1\right) \tag{15}$$

Figure 5a shows an example for $a = 2, b = 1, \alpha = 2, k = 6$. The arrows start from the elements that need to be divided to create a leading 0 at the location they point to. The number of multiplications is the same as the number of additions required for eliminating elements and is derived in Equation (15). Figure 5b shows an example for $a = 2, b = 1, \alpha = 2, k = 6$. The arrows start from the elements that are multiplied with a constant and then added to the location they point to.

By comparing Figures 5 and 6 we can observe that intuitively, not having to eliminate the same elements multiple times (white boxes) is important for the effectiveness of our proposed technique compared to basic Gauss elimination.

3) The Computational Cost of Decompositions: Having looked at the cost of individual merges, we now turn to the total computational cost of a decomposition that checks a single repair for each node failure: the number of divisions in (16) and the number of multiplications/additions in (17).

$$\begin{aligned} DIVS(\mathbf{D}) &= \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{sel}} d(a,b) + \sum_{s_1 \in S_1} lz(1) \\ &= \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{sel}} d(a,b) + N\frac{\alpha^2-\alpha}{2} \end{aligned} \tag{16}$$

$$MA(\mathbf{D}) = \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{sel}} m(a,b) + \sum_{s_1 \in S_1} \sum_{j=1}^{\alpha} (k-j+1) \tag{17}$$

The second term in both equations is the cost of transforming matrices $s_1 \in S_1$, including the r sets of repaired rows for each of the possible node failures into upper triangular form. If matrices from previous generations are stored, then the summation can simply skip these. If a decomposition avoids memoizing matrices containing repaired rows, as per our goal, the value of r only influences the number of computations by determining the size of S_1 and S_{N-L}.

$$\begin{aligned} |S_1| &= N(r+1) \\ |S_{N-L}| &= N\binom{N-1}{N-L-1}r \end{aligned} \tag{18}$$

The number of reusable matrices depends non-trivially on the decomposition. We leave investigating this aspect as future work and present experimental results in Section V.

4) The Memory Requirements of Decompositions: Our proposed approach requires memory to store memoized matrices. The number of matrix elements that need to be stored for decomposition $\mathbf{D}$ is given in Equation (19).

$$MEMO_{\text{naive}}(\mathbf{D}) = \sum_{i=1}^{N-L} |S_i^{sel}| i\alpha k \tag{19}$$

A simple improvement can be achieved by only storing non-zero elements. This decreases storage costs to those given in Equation (20).

$$MEMO_{\text{reduced}}(\mathbf{D}) = \sum_{i=1}^{N-L} |S_i^{sel}|(i\alpha k - lz(i)) \tag{20}$$

C. Finding Efficient Decompositions

1) Observations on the Relationships Between Costs: Based on Equations (14) and (15), we can make some observations that will help in determining good decomposition strategies. Let $a, b, a', b', c, d \in \mathbb{N}^+$, $s_b \in S_b$, $s_c \in S_c$, $s_d \in S_d$ and $a + b = a' + b' = i$ and $c + d = b$.

1) $m(a,b) = m(b,a)$ and $d(a,b) = d(b,a)$ (because a and b are interchangeable in Equations (14), (15))
2) $d(a,b) > d(a',b') \Leftrightarrow |a-b| < |a'-b'|$ (consequence of Equation (14) and because more imbalanced sets have a larger number of already reduced elements; the larger the matrix, the more reduced elements it has)
3) $m(a,b) > m(a',b') \Leftrightarrow |a-b| < |a'-b'|$ (consequence of Equation (15) in which the part in brackets is the same for a, b and a', b')
4) $m(a,b) > m(a',b') \Leftrightarrow d(a,b) > d(a',b')$ (same argument as previous observation)
5) $lz(s_b) \geq lz(s_c) + lz(s_d)$ (consequence of Equation (13) and $i^2 > a^2 + b^2$)
6) $d(a,b) \leq d(a,c) + d(a+c,d)$ (consequence of previous observation and Equation (14))

Observation 4) is important because a decomposition that minimizes the number of divisions will also minimize the number of multiplications and additions. Observation 2) and 3) have the consequence that a decomposition that decreases i by one (i.e. selecting $a = 1, b = i-1$ or $b = 1, a = i-1$) has the lowest computational cost in that decomposition step. We refer to this method as *decrease and conquer*. Conversely, $a = \lfloor \frac{i}{2} \rfloor$ or $b = \lfloor \frac{i}{2} \rfloor$ has the highest number of computations for any given i. On the other hand, it also reduces the size of the matrix by the greatest degree. Thus matrices between $S_{\lfloor i/2 \rfloor}$ and S_i can be skipped and less memory is needed. We refer to this approach as *divide and conquer* and propose the following method to deal with odd levels: if i is even, divide the problem into $a = b = \frac{i}{2}$. If i is odd, fall back to the previous approach and decrease the problem to $a = a - 1$, $b = 1$. An alternative decrease and conquer decomposition would be to select $a = \lceil \frac{i}{2} \rceil$ and $b = \lfloor \frac{i}{2} \rfloor$, then do a second decomposition if $a \neq b$ to cover S_b using S_a and S_1.

There is therefore a trade-off between minimizing the number of levels (and reducing memory requirements in the process) in a decomposition and the cost of moving between levels using merging. However, it is not immediately apparent how the number of matrices in each level ($|S_i^{sel}|$) changes for different points on the trade-off curve. This metric also plays a key role in determining the number of computations. In the following subsections, we propose two heuristic-based algorithms to find decompositions for both the decrease and conquer and divide and conquer methods.

2) The First Step of a Decomposition: As described in Section IV-A, matrices containing rows from $N-L-1$ existing nodes and one hypothetically replacement node must be checked. If we consider that r hypothetical repairs are checked for each possible node failure, $|M| = (L+1)r\binom{N}{N-L-1}$. Instead of starting from these matrices, the first step of a decomposition should be treated differently to ensure that no repaired rows are present in matrices in S_{N-L-1}. Since at the time of the checks it is not known which node

Algorithm 3 Greedy Algorithm for Decrease and Conquer

```
1: build $G(i,1)$
2: while $X \neq \emptyset$ do
3:     find $v_y$, where $deg(v_y) \geq deg(v_j), \forall v_j \in G$
4:     $S_{i-1}^{sel} = S_{i-1}^{sel} \cup v_y$
5:     $Y = Y \setminus v_y$
6:     for all $v_x$, where $(v_x, v_y) \in E(G)$ do
7:         $X = X \setminus v_x$
8:         $E(G) = E(G) \setminus (v_x, v_*)$    ▷ all edges involving $v_x$
9:     end for
10: end while
11: return $S_{i-1}^{sel}$
```

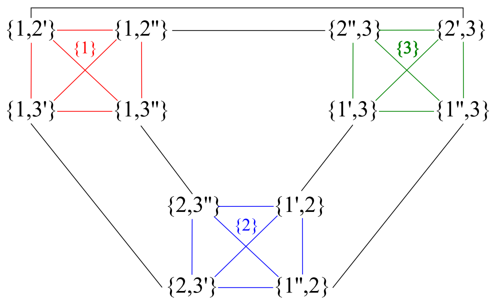

Fig. 7. The shared rows of matrices in $\mathbf{M}$ that should be checked for $N = 3$, $L = 1$, $r = 2$. Single and double apostrophes denote rows with coefficients from potential repairs. For example, $2'$ and $2''$ are rows resulting from the two different potential repairs of $node_2$.

will fail, hypothetical repaired rows limit the reusability of matrices.

To account for this, we propose a simple schema to determine the first step of the decomposition, shown in Figure 7. The matrices in S_{N-L} can be built in the following way: by selecting $S_{N-L-1}^{sel} = S_{N-L-1}$ (i.e. taking all possible matrices that contain $N-L-1$ non-repaired rows) and adding every possible repaired row to each of them, we arrive at S_{N-L}. This allows the decomposition to start from $\mathbf{M} = S_{N-L-1}$ instead of S_{N-L} and only include coefficients from existing rows to maximize matrix reuse. We will show in Section IV-C3 that this first step is optimal in selecting the minimal number of matrices from S_{N-L-1}.

3) Greedy Algorithm for Decrease and Conquer: We wish to select S_{i-1} in such a way that all elements of S_i^{sel} can be generated by adding an element of S_1 to an element of S_{i-1}. Let $G = (V, E)$ be an undirected bipartite graph with vertices divided into sets $V = X \cup Y$, where $X = \{s_i | s_i \in S_i^{sel}\}$ and $Y = \{s_{i-1} | s_{i-1} \in S_{i-1}^{all\,combos}\}$. There is an edge between a vertex $v_x \in X$ and $v_y \in Y$ if and only if for the corresponding s_i and s_{i-1}, $s_{i-1} \subset s_i$. We wish to cover all vertices $v_x \in X$ using as few vertices $v_y \in Y$ as possible. In each iteration of Algorithm 3, the vertex v_y with the highest degree is selected and removed from the graph. All vertices v_x it is connected to are also removed along with any edges containing v_x. The algorithm terminates when there are no more vertices in X.

This greedy algorithm is well known and is analogous to the approach of selecting a covering set in such a way that at every choice, the set that covers the most uncovered elements is selected. This is a $H(\mathbf{n})$-approximation algorithm and it has been proven [24] that no polynomial-time algorithm with

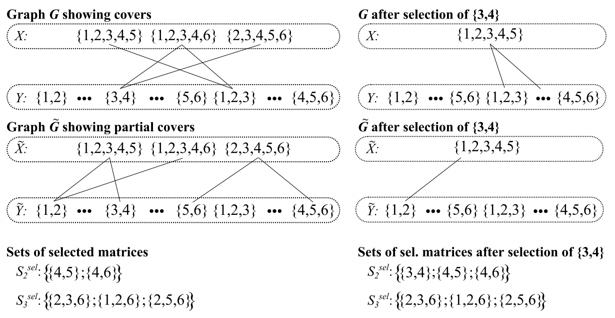
Graph G showing covers
X: {1,2,3,4,5} {1,2,3,4,6} {2,3,4,5,6}
Y: {1,2} ••• {3,4} ••• {5,6} {1,2,3} ••• {4,5,6}
G after selection of {3,4}
X: {1,2,3,4,5}
Y: {1,2} ••• {5,6} {1,2,3} ••• {4,5,6}
Graph $\tilde{G}$ showing partial covers
$\tilde{X}$: {1,2,3,4,5} {1,2,3,4,6} {2,3,4,5,6}
$\tilde{Y}$: {1,2} ••• {3,4} ••• {5,6} {1,2,3} ••• {4,5,6}
$\tilde{G}$ after selection of {3,4}
$\tilde{X}$: {1,2,3,4,5}
$\tilde{Y}$: {1,2} ••• {5,6} {1,2,3} ••• {4,5,6}
Sets of selected matrices
S_2^{sel}: {{4,5};{4,6}}
S_3^{sel}: {{2,3,6};{1,2,6};{2,5,6}}
Sets of sel. matrices after selection of {3,4}
S_2^{sel}: {{3,4};{4,5};{4,6}}
S_3^{sel}: {{2,3,6};{1,2,6};{2,5,6}}

Fig. 8. Example for a state change of the divide and conquer algorithm showing both graphs and the sets of already selected matrices. Some elements of Y and $\tilde{Y}$ have been omitted $(\ldots)$ due to space constraints. The next step (result shown on the right) selects and adds {3,4} to S_2^{sel} as it is tied with {1,2,3} in covering the most matrices in G and partially covers more matrices in $\tilde{G}$. The algorithm then removes {3,4}, {1,2,3,4,6} and {2,3,4,5,6} from G and $\tilde{G}$ along with all of their edges. It then adds an edge between {1,2,3,4,5} and {1,2,5} (not shown) in G to reflect that {1,2,5} now fully covers {1,2,3,4,5}.

a better approximation factor exists for this NP-hard problem. Fortunately, $\mathbf{n} = \max|s_i| = i$, as all matrices from the set used for the cover have exactly i elements. Thus, even though $|S_i| = \binom{k}{i}$ increases computational costs quickly, the approximation factor increases slowly $\left(\frac{\log i}{\log(i-1)}\right)$ with i and remains acceptable even for large values of i.

We could apply this algorithm for the first step of finding a decomposition and it would select a covering set that is the combination of $N-L-1$ out of N rows.

Proposition 5: If r repairs are checked for each possible node failure, then $|S_{N-L}| = |S_{N-L-1}|(L+1)r$, which is the same as our previously proposed first step and is in fact the best we can hope to achieve (we have included a proof in the appendix see in Supplementary Material).

We have decided to follow the technique proposed in Subsection IV-C2 instead of this algorithm for the first step of a decomposition to ensure that if multiple minimum cost set covers exist (this is the case for $r = 1$), the one that maximizes matrix reuse through memoization is selected.

4) Greedy Algorithm for Divide and Conquer: We propose extending the previous algorithm to deal with the more general case when a matrix of size $i \times k$ is decomposed into two submatrices of size $a \times k$ and $b \times k$, where $a + b = i$. Figure 8 shows an example for a state change of the algorithm. We will keep the graph G to have a pairing that denotes which submatrices cover which matrices. The set X remains the matrices that need to be covered and Y contains all possible submatrices of size $a \times k$ and $b \times k$ that can be built from elements of S_1. There is an edge between a vertex $v_x \in X$ and $v_y \in Y$ if and only if $v'_y \notin Y$, where $v'_y := v_x \setminus v_y$. In other words, v'_y is the relative complement of v_y with respect to v_x, the rows of v_x left uncovered by v_y. We refer to it as its pair. Unless $a = 1$ or $b = 1$, G has no edges in the beginning because $\forall v_y \in Y \rightarrow v'_y \in Y$. Put differently, in the beginning no single element in Y can cover an element in X. In the special case of $a = 1$ or $b = 1$, G is initialized as described in the previous subsection.

We introduce a second graph $\tilde{G}$ ($\tilde{X} = X$, $\tilde{Y} = Y$), to have a pairing that denotes which submatrices cover which matrices partially. $v_y \in \tilde{Y}$ partially covers $v_x \in \tilde{X}$ if its pair $v'_y \in \tilde{Y}$. These are submatrices, whose pairs have not yet been selected and thus can only provide partial cover for v_x. $\tilde{G}$ must be initialized to have all partially covering edges, thus in the initial state of the algorithm $E(\tilde{G}) = \{(v_x, v_y) | v_x \setminus v_y \in \tilde{Y}, v_x \in \tilde{X}, v_y \in \tilde{Y}\}$. In the special case where $a = 1$ or $b = 1$, $\tilde{G}$ will have no edges and may be disregarded as the algorithm falls back to the decrease and conquer algorithm.

Algorithm 4 selects submatrices from Y until all matrices in X have been covered. It selects the submatrix v_y that has the highest degree in G, i.e. covers most matrices. Tie–breaks are handled by selecting the submatrix that has the higher degree in $\tilde{G}$, i.e. partially covers more matrices. When a submatrix is selected, it is removed from both G and $\tilde{G}$. Furthermore, all matrices it covers are removed from both G and $\tilde{G}$ along with any edges they are part of. Any matrix $\tilde{v_x}$ it partially covered is updated in G: an edge is added between v_x and v'_y to reflect that v'_y can cover v_x following the selection.

Algorithm 4 Greedy Algorithm for Divide and Conquer

```
1: Build $G(a,b), \tilde{G}(a,b)$
2: while $X \neq \emptyset$ do
3:     $MAX = \{v_y | v_y \in G, deg(v_y) \geq deg(v_j), \forall v_j \in G\}$
4:     find $\tilde{v_y} \in MAX$, where $deg(\tilde{v_y}) \geq deg(\tilde{v_j}), \forall v_j \in \tilde{G}$
5:     if $|v_y| = a$ then
6:         $S_a^{sel} = S_a^{sel} \cup v_y$
7:     else
8:         $S_b^{sel} = S_b^{sel} \cup v_y$
9:     end if
10:    $Y = Y \setminus v_y$
11:    $\tilde{Y} = \tilde{Y} \setminus \tilde{v_y}$
12:    $E(G) = E(G) \setminus (v_*, v_y)$          ▷ all edges involving $v_y$
13:    $E(\tilde{G}) = E(\tilde{G}) \setminus (\tilde{v_*}, \tilde{v_y})$          ▷ all edges involving $\tilde{v_y}$
14:    for all $v_x$ covered by $v_y$ do
15:        $X = X \setminus v_x$
16:        $\tilde{X} = \tilde{X} \setminus \tilde{v_x}$
17:        $E(G) = E(G) \setminus (v_x, v_*)$      ▷ all edges involving $v_x$
18:        $E(\tilde{G}) = E(\tilde{G}) \setminus (\tilde{v_x}, \tilde{v_*})$      ▷ all edges involving $\tilde{v_x}$
19:    end for
20:    for all $\tilde{v_x}$ partially covered by $\tilde{v_y}$ do
21:        $E(\tilde{G}) = E(\tilde{G}) \cup (\tilde{v_y}', \tilde{v_x})$, where $\tilde{v_y} \cup \tilde{v_y}' = v_x$
22:    end for
23: end while
24: return $S_a^{sel}, S_b^{sel}$
```

Proposition 6: For every partially covered v_x by the selected v_y, there will always be exactly one v'_y that will cover v_x (we have included a proof in the appendix see in Supplementary Material).

D. Performing Invertability Checks Given a Decomposition

A decomposition is only dependent on N, L, α, parameters that typically don't change during the lifetime of a system. Therefore, several decompositions can be computed in advance to cover the likely parameter set values before the system comes online. Thus, even if finding a good valid decomposition is computationally expensive, it does not negatively influence the general repair performance of the system. This subsection examines how a decomposition can be applied. If a check fails, new coefficients can be generated and operations associated with the failed check should be redone.

1) The Benefits of Performing Checks in a Top-Down Manner: Given a valid decomposition $\mathbf{D}$, either a bottom-up or a top-down approach can be used to do the checks. Bottom-up checks start with matrices from S_1^{sel} and then reduce each matrix $s_j \in S_j^{sel}$ level-by-level to an upper triangular form and memoize it after merging two smaller matrices based on Ψ. This approach has the benefit of avoiding recursive calls, but will only provide relevant information on the rank of a matrix $s_m \in \mathbf{M}$ after all smaller matrices $s_j \in S_j$, $j < m$ have been reduced to an upper triangular form. Conversely, a top-down approach starts with matrices $s_m \in \mathbf{M}$ and attempts to merge $s_m = s_a \oplus s_b$, where the choice of s_a and s_b is defined by Ψ. If either s_a and/or s_b are not yet in UTF, the algorithm is called recursively on s_a and/or s_b and so on. Once a matrix is reduced into UTF, it is memoized so it can be reused for other matrices from $\mathbf{M}$. Thus, the invertability of some s_m will be known earlier than using a bottom-up approach. This can be used to provide probabilistic statements on the overall result of the checks before all matrices are checked.

This is most important in situations where the checks are time constrained. Either there aren't enough free computational resources in the system and the checks must be postponed or two node failures occur in quick succession and a repair must be started before the checks for the second failure have time to complete. A more informed decision may still be made on what repair to select based on the matrices that have been computed so far. By using a top-down approach and ordering the matrices in M in a certain way, the amount of useful information at any given point in time can be increased. One possibility is to take one repair for each node failure first. Once at least one feasible repair is found for every failure, one more repair for each failure can be checked and so on. This minimizes the time until at least one feasible repair is found for every possible failure scenario. Alternatively, if it is acceptable for the system to temporarily go below the predetermined L number of concurrent node failures it must sustain, the matrices may be ordered differently. In this case, checks may also start by looking at a single repair for each node failure first. However, instead of completing all checks for that repair, only one matrix is checked at each step, moving on to check matrices associated with other node failures. With each pass, the degree of confidence with which it can be stated that a repair is feasible for any L node losses increases.

2) Memoizing Matrices Across Generations: A further reduction in computations can be achieved if matrices from a decomposition are stored and reused across multiple generations. Any matrix that contains rows from the recently lost node should be discarded, but all others can be reused in the subsequent generation. The proposal to start a decomposition from $\mathbf{M} = S_{N-L-1}$ described in subsection IV-C2 also encourages matrix reuse across generations as it ensures that only matrices containing rows from actual nodes (as opposed to rows from hypothetical repairs) are memoized. We leave the characterization of the choice of decomposition on the expected number of matrices that can be reused as future work and present simulation-based results to show the effectiveness of this enhancement. Checks for the initial data distribution cannot reuse matrices from previous generations and no matrices are skipped as there were no previous failures.

V. Discussion

A. Evaluation Method for Network Awareness

In this section we evaluate how much each erasure code benefits from being network aware and perform a thorough analysis on the number of computations required to perform the feasibility checks. We compare our proposed framework that guarantees finding the least cost repairs to a naive approach that selects one of the repairs with the lowest traffic but has no knowledge of transfer costs. We perform our analysis using sets of code parameters CODE(N, α, k) that meet the following constraints: $2 \leq N \leq 20$, $1 \leq \alpha \leq 10$, $5 \leq k \leq 32$. We also required that each code must be able to sustain $L \geq 2$ node losses while maintaining data recoverability following each repair. We have selected such a wide range of values to cover most parameter sets of interest. We have strived to take into consideration other practical aspects as well. We have chosen to have an upper bound on k as it is the main factor that determines the decoding overhead during read operations for non-systematic codes such as RLNC and RBT-MBR. We require that codes have a storage efficiency of $\frac{N\alpha}{k} \leq 2.5$. We decided to use 2.5 as a cut-off point to include more cases for RBT-MBR, despite being a relatively high value for a practical system. For Reed-Solomon we only consider $\alpha = 1$ as this maximizes its ability to lose nodes. Recently, Guruswami and Wootters [25] proposed sub–packetization for RS raising the possibility that repair for RS may benefit from other values of α (or non-integer values of β_i). However, we are unaware of any practical implementation or the exact implications of this construction and have therefore decided to consider RS as used in current systems. For RLNC and RBT-MBR, we restrict our evaluation to sets which have a repair space size for a given failed node of at most 2^{16} and 2^{24} respectively. While one of the key benefits of our proposed approach is that it can handle large repair spaces by only considering a small fraction of repairs, we would have been unable to compare our solution to the baseline approach without this constraint. Fortunately, most practically interesting cases for N, k and α are included. 50 sets of parameters meet these constraints for RS, 8 for RBT-MBR and 214 for RLNC.

Each run for each code, costs and set of code parameters consisted of 100 iterations of node loss and recovery. Operations were performed over GF(2^8). Two types of cost matrices $\mathbf{C}$ were considered. First, **I**: one that is based on a static network topology where nodes are grouped evenly in 2 racks. Costs have two types: inter-rack$(10\times)$ and intra-rack $(1\times)$. We used this model to evaluate the benefits of network awareness assuming a simple, static topology such as that defined in [13]. Second, we used a cost matrix that also portrays current network traffic conditions. The same $\mathbf{C}$ is multiplied entrywise in each round with a different matrix containing values drawn randomly from the following uniform distributions: **II**: $U(0.75, 1.25)$, **III**: $U(0.5, 1.5)$, **IV**: $U(0.25, 1.75)$, **V**: $U(0, 2)$. This allows us to portray the relationship between the variance in network activity and the potential benefit for an erasure code to be network-aware.

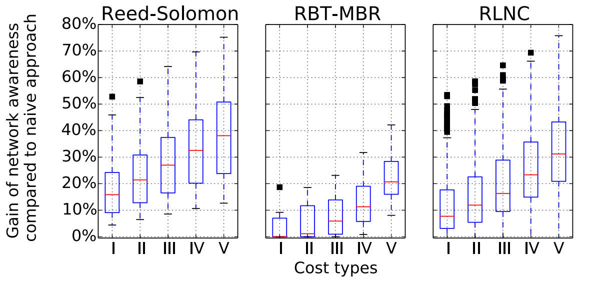

Fig. 9. Gains of network awareness.

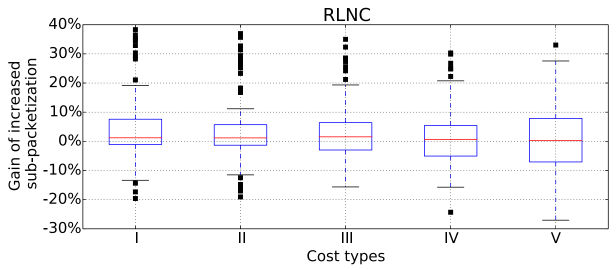

Fig. 10. Comparing different levels of sub–packetization for RLNC. Results show the differences between pairs of codes.

B. The Benefits of Network Awareness

Figure 9 shows how much each erasure code benefits from knowledge of network costs. Because of the different parameter sets, the codes should not be compared against each other directly. However, a general trend can be observed: the larger the variance in the costs, the larger the gain is compared to the naive approach. Thus, a distributed storage system with more dynamic traffic patterns will potentially see a larger benefit from performing network-aware repairs. For Reed-Solomon and RBT-MBR which use exact repair, most cost types show gains in being network aware across all parameter sets. In the case of RLNC, there is a significant gain overall. RBT-MBR does not benefit at all from being network-aware in a balanced static two–rack scenario regardless of the actual values of the cost. This is due to decoding-based repairs having always to access a certain amount of data from outside the rack in a balanced scenario. If the difference between the costs of going to the different inter-rack nodes and intra-rack nodes respectively is the same, a decoding-based repair will entail a cost that is greater than repairing by simply transferring data. However, if there is variance between the cost of accessing nodes in the same cost category, repair by transfer is not always optimal.

C. The Benefits of Sub–Packetization for RLNC

For RLNC, several codes with the same repair and storage efficiency can be created for a given N number of nodes by varying the level of sub–packetization, i.e. by changing the number of k packets a file is divided into. Intuitively, a higher level of sub–packetization yields more feasible repairs and may give more flexibility in choosing a repair with lower

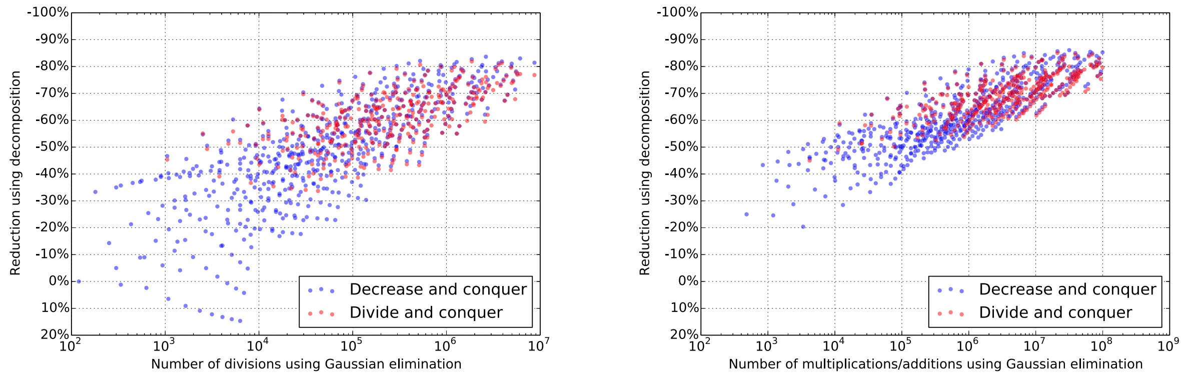

Fig. 11. Reduction in the number of operations compared to Gaussian elimination.

cost. We examined whether this is the case for the two-rack scenario. We selected all 80 pairs of codes from our measurements for which $N_1 = N_2$ and $\frac{k_1}{\alpha_1} = \frac{k_2}{\alpha_2}$. For chains of codes that meet these criteria, we looked at subsequent pairs. For example, from RLNC$_1$($N_1 = 7, k_1 = 5, \alpha_1 = 1$), RLNC$_2$($N_2 = 7, k_2 = 10, \alpha_2 = 2$) and RLNC$_3$($N_3 = 7, k_3 = 15, \alpha_3 = 3$) we have compared RLNC$_1$ to RLNC$_2$, and RLNC$_2$ to RLNC$_3$ respectively. Figure 10 shows on average a slight decrease in cost. Thus, at least for relatively static environments (cost levels **I**, **II**, **III**), higher levels of sub–packetization increase the advantages of network awareness.

D. Evaluation Method for Decompositions

To test the effectiveness of the proposed decomposition techniques, we have implemented both divide and conquer and decrease and conquer algorithms in Python. We removed the constraint on the size of the repair space to include a total of 629 RLNC codes. We can apply the decrease and conquer approach to all of these. The divide and conquer approach falls back to decrease and conquer in all but 348 cases. The figures have had the 281 duplicate cases removed for clarity.

As a reference for how operations translate to time, the checks for RLNC$_4$($N = 10$, $k = 14$, $\alpha = 2$) using Gaussian elimination involve 76440 division and 764400 multiplications and additions for $r = 1$. On an Intel i7-3770@4.2GHz running MATLAB R2016a, it takes roughly 50 seconds for $r = 1$ and 502 seconds for $r = 10$ repairs to be checked for each node.

E. The Benefits of Doing Decompositions

Figure 11 shows the benefits of using our proposed technique for $r = 1$. By decomposing the rank checks and reusing intermediary results, a reduction of up to 87% is achieved in the number of multiplication/addition operations compared to doing Gaussian eliminations on the matrices in $\mathbf{M}$. The gains grow in value and significance with the number of operations for both types of decomposition, making checks feasible for a wider range of parameters. Divisions show a similar trend, though in a few cases we actually see an increase in the number of computations and results show more variance.

Looking at different values for r in Figure 12, we can see how the difference between the computational cost difference increases as r grows. This is as expected based on Equation (18). All other figures show simulations for $r = 1$.

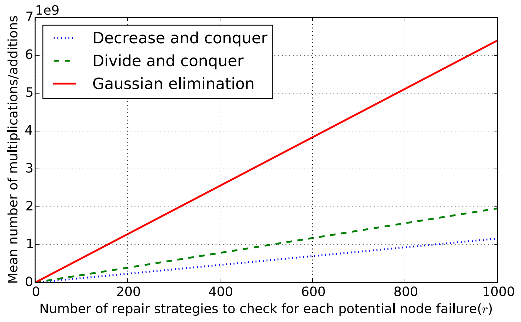

Fig. 12. Checking the feasibility of multiple repair strategies for each node failure.

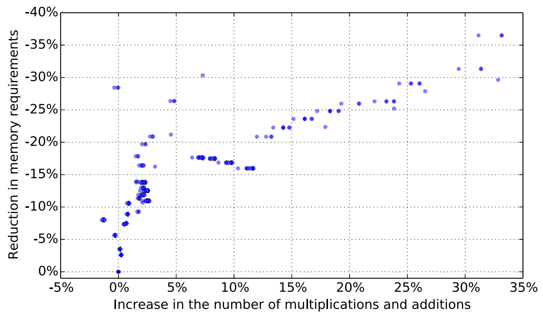

Fig. 13. Comparing divide and conquer to decrease and conquer on memory and computational costs.

F. Comparing Divide and Conquer to Decrease and Conquer

Section IV-C briefly described the trade-off between the number of memoized matrices and the number of operations. Figure 13 illustrates a direct comparison between the two proposed methods on this aspect. Divide and conquer requires up to 37% less memory than decrease and conquer. The downside is an increase of up to 33% in the number of multiplications/additions. The memoized matrices for the examined codes typically require around 200kB - 300kB of memory, with a maximum of 6.2MB. This is negligible compared to the data associated with the coefficients. Thus, most practical systems should employ decrease and conquer to reduce the amount of computations as much as possible. Division operations show a similar trend.

G. Memoizing Across Generations

Finally, we look at the benefit of reusing matrices across generations of node failure and recovery. We expected to

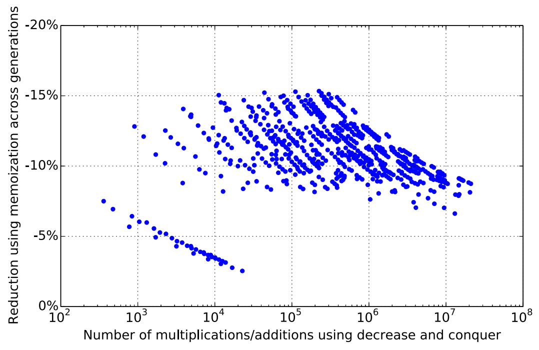

Fig. 14. Benefits of reusing matrices across generations.

show further large gains in computation cost by extending our solution with this simple technique because most matrices remain valid through at least two subsequent generations. However, results in Figure 14 only show relatively modest gains of between 5%-15%. This can be explained by looking more closely into which matrices can be reused. While around $\frac{N-1}{N}$ of the smallest matrices contain rows are present in two subsequent generations, the ratio is smaller for larger matrices. Furthermore, most of the computations can be associated with the larger matrices. Nevertheless, the modest additional memory cost makes reusing matrices across generations a good choice to further reduce computational costs.

VI. Conclusions and Future Work

First, we have answered one of the key questions from the introduction: yes, knowledge of network conditions benefits a wide range of erasure codes. We have proposed a framework to check the feasibility of repairs in advance and provided analytic results on reducing the space of relevant feasible repairs. We hope to make further advances for specific network topologies by looking at specific types of cost matrices.

Second, we have presented a set of techniques to increase the practical value of our work for erasure codes employing randomly selected coefficients, such as RLNC. Our proposed decomposition of matrix rank checks shows significant reductions in the number of computations needed and has negligible memory costs. We presented two algorithms to perform the decomposition and found that in practice a decrease and conquer type solution works better. We wish to continue this work by formalizing the problem of ordering the matrices that are to be checked in such a way that useful information on their rank is gained as soon as possible.

References

[1] 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.

[2] Y. Wu and A. G. Dimakis, “Reducing repair traffic for erasure coding-based storage via interference alignment,” in *Proc. IEEE Int. Symp. Inf. Theory*, Jun./Jul. 2009, pp. 2276–2280.

[3] K. V. Rashmi, N. B. Shah, and K. Ramchandran, “A piggybacking design framework for read- and download-efficient distributed storage codes,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2013, pp. 331–335.

[4] N. B. Shah, K. V. Rashmi, and P. V. Kumar, “A flexible class of regenerating codes for distributed storage,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2010, pp. 1943–1947.

[5] M. Sipos, J. Gahm, N. Venkat, and D. Oran, “Erasure coded storage on a changing network: The untold story,” in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[6] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, “Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff,” *IEEE Trans. Inf. Theory*, vol. 58, no. 3, pp. 1837–1852, Mar. 2012.

[7] 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.

[8] T. Ho *et al.*, “A random linear network coding approach to multicast,” *IEEE Trans. Inf. Theory*, vol. 52, no. 10, pp. 4413–4430, Oct. 2006.

[9] S. Acedańsk, S. Deb, M. Médard, and R. Koetter, “How good is random linear coding based distributed networked storage?” in *Proc. 1st Workshop Netw. Coding, Theory Appl.*, 2005, pp. 1–6.

[10] S. Deb, C. Choutte, M. Médard, and R. Koetter, “Data harvesting: A random coding approach to rapid dissemination and efficient storage of data,” in *Proc. IEEE ICC*, Mar. 2004, pp. 1–12.

[11] J. Li, S. Yang, X. Wang, and B. Li, “Tree-structured data regeneration in distributed storage systems with regenerating codes,” in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[12] S. Akhlaghi, A. Kiani, and M. R. Ghanavati, “A fundamental trade-off between the download cost and repair bandwidth in distributed storage system,” in *Proc. IEEE Int. Symp. Netw. Coding (NetCod)*, Jun. 2010, pp. 1–6.

[13] B. Gastón, J. Pujol, and M. Villanueva. (Jan. 2013). “A realistic distributed storage system that minimizes data storage and repair bandwidth.” [Online]. Available: https://arxiv.org/abs/1301.1549

[14] Y. Hu, P. P. C. Lee, and X. Zhang, “Double regenerating codes for hierarchical data centers,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2016, pp. 245–249.

[15] A. Khan and I. Chatzigeorgiou, “Improved bounds on the decoding failure probability of network coding over multi-source multi-relay networks,” *IEEE Commun. Lett.*, vol. 20, no. 10, pp. 2035–2038, Oct. 2016.

[16] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and M. Medard, “On code parameters and coding vector representation for practical RLNC,” in *Proc. IEEE Int. Conf. Commun.*, Jun. 2011, pp. 1–5.

[17] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang, “NCCloud: Applying network coding for the storage repair in a cloud-of-clouds,” in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–8.

[18] K. V. Rashmi *et al.*, “A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster,” in *Proc. 5th USENIX Conf. Hot Topics Storage File Syst.*, 2013, pp. 1–67.

[19] 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. Int. Conf. Syst. Storage*, New York, NY, USA, 2014, pp. 1–7.

[20] S. Jaggi, Y. Cassuto, and M. Effros, “Low complexity encoding for network codes,” in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2006, pp. 40–44.

[21] N. J. A. Sloane. (1991) *The On-Line Encyclopedia of Integer Sequences*. [Online]. Available: http://http://oeis.org/A000041

[22] H. Alder, “Partition identities—From Euler to the present,” *Amer. Math. Monthly*, vol. 76, no. 7, pp. 733–746, 1969.

[23] R. P. Stanley, *Enumerative Combinatorics*, vol. 1, 2nd ed. New York, NY, USA: Cambridge Univ. Press, 2011.

[24] I. Dinur and D. Steurer, “Analytical approach to parallel repetition,” in *Proc. 46th Annu. ACM Symp. Theory Comput.*, 2014, pp. 624–633.

[25] V. Guruswami and M. Wootters. (2015). “Repairing Reed–Solomon codes.” [Online]. Available: https://arxiv.org/abs/1509.04764

Authors’ photograph and biography not available at the time of publication.