RJ10415 (A0709-002) September 5, 2007
Computer Science

IBM Research Report

Space Efficient Matrix Methods for Lost Data Reconstruction in Erasure Codes

Bruce Cassidy, James Lee Hafner
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

SPACE EFFICIENT MATRIX METHODS FOR LOST DATA RECONSTRUCTION IN ERASURE CODES

Bruce Cassidy

James Lee Hafner

IBM Research
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

ABSTRACT:
In previous work, we showed how simple matrix manipulation algorithms can completely solve the problem of data reconstruction in a uniform way across all XOR-based erasure codes. In those algorithms, a workspace matrix is constructed and manipulated. At the end of the algorithm, XOR formulas for reconstructing lost data can be read directly from the workspace. Empty formulas implied irretrivable data loss. The dimension of the (square) workspace matrix was equal to the total number of "bits in the codeword", i.e., the number of data and parity elements. In this paper, we show how the workspace can be reduced significantly. Sample (self-explanatory) Mathematica code is provided as well.

Contents

1. Introduction

In [2, 3], we showed how data reconstruction formulas for lost data sectors in XOR-based erasure code could be derived by simple matrix manipulation algorithms. The algorithms use the generator/parity check matrix for the erasure code as input. In this way, the algorithms are completely generic; they do not rely on any specific geometric or combinatorial view of the erasure code construction. The methods start with a binary workspace matrix and perform row/column operations on that matrix. At the end of the algorithms, certain columns of the workspace are interpreted as XOR formulas for reconstruction of specific lost data elements. The bit positions in the column indicate whether a good (readable) data element or parity element is required in the reconstruction formula. Other features of the algorithms include reversibility – once a data element is reconstructed, another algorithm can be applied to feed that fact into the reconstruction formulas for other elements.

One issue with these algorithms is space and computational costs. The workspace matrix is of size $(n + q) \times (n + q)$ where n is the total number of data elements and q is the total number of parity elements. (In coding theory terms, n is the bit-size of the data word and $n + q$ is the bit-size of the codeword; i.e., the code is $(n, n + q)$. Worst case computational cost is $O(f(n + q)^2)$ where f is the number of lost data elements. For example, for the EVENODD [1], say with parameters $p = 17$ on 16 disks, $n = 14 * 16 = 224$, $q = 32$ and the matrix dimension is then 256×256, which, with a bit level implementation, is still 8KB.

In this paper, we show how the algorithms can be improved significantly both in space and time (time comes for free when space is reduced). The number of column operations in the algorithms is still the same but the bit level complexity improves with space savings directly. In the previous work, we suggested that one space savings method reduces the workspace to size $(n+q) \times (f+q)$, a significant savings. However, we show here how this can be reduced further to $q \times (f + q)$ – a savings of $n(f + q)$. There is also a version that can be implemented in space $q \times q$. For the EVENODD example above, with two disks lost (so 32 data elements), this is now a minimum of 256B to represent the workspace, or 2KB if each bit is represented by a byte (for ease of implementation).

The key idea is to compute only the dependence in the reconstruction formulas on the parity elements. The data element dependencies come for free for the following reason. A parity element contains some lost data elements and some good data elements. When used to reconstruct some lost data elements, it will always depend on the *same* set of good elements. So, revised parity values can be precomputed by removing (by XOR) from each parity value all the good data elements on which it depends. This leaves only the relationships between lost data elements and these revised parity values. A rigorous proof of this is given in the Section 5.

The paper is organized as follows. We first describe the basic algorithm from the previous work. We then describe the space-saving version. We show how the reduced workspace algorithm formulas are used to reconstruct lost data. We also discuss, without too much detail, how these algorithms can be more space efficient by enabling dynamic space allocation (in practice, this is probably not the most reasonable approach – we mention it only for theoretical value). We also work out again the example in our previous paper(s) and the same example using the new algorithm. Lastly we prove that the revised version of the algorithm produces equivalent results. In the appendix, we include Mathematica code that implements the algorithm (in a slightly revised form).

2. Column-Incremental Construction

In this section, we present again the main algorithm of [2, 3]. The workspace W is an $(n+q) \times (n+q)$ binary matrix in the form $W = (R \,|\, B)$, where R is $(n+q) \times n$ and B is $(n + q) \times q$. During the algorithm, R holds the incomplete reconstruction formulas for the data elements (they are incomplete because they may depend on lost data elements that have not yet been processed by the algorithm). The submatrix B

contains vectors in the null space of the (implicit) reduced generator matrix where the processed lost data elements have been removed. The rows of R and B are indexed by the data and parity elements. The columns of R are indexed by the data elements. The columns of B have no particular labeling. Without loss of generality, we can rearrange the rows and columns of R and the rows of B so that the initial section corresponds to data elements and the last section corresponds to parity elements (in some order).

The algorithm uses three basic matrix operations:

- Swap: exchange two columns (or rows)
- Sum and Replace: add column c to column d (modulo 2) and replace column d with the sum (similarly for rows).
- Zero: zero all the entries in a column (or row).

The input to the algorithm is a set F of indices of lost elements (data or parity) and the parity check matrix H. The algorithm then proceeds as follows (with notation changed slightly):

Algorithm: Column-Incremental Construction

1. Construct a square workspace matrix W of size $(n+q)$. In the first n columns and rows, place an identity matrix. In the last q columns, place the parity check matrix H. Let R represent the first n columns and B represent the last q columns of W, so $W = (R \,|\, B)$, where initially, $R = \begin{pmatrix} I_n \\ 0 \end{pmatrix}$ and $B = H$.

2. For each lost element in the list F, let r indicate the row corresponding to the lost element; perform the following operation:

   (a) Find any column position b in B that has a one in row r. If none exists, Zero any column in R that has a one in row r and continue to the next lost element. (Note that by zero-ing these columns we have zero-ed the entire row r in W.)

   (b) For each one in row r of W (both R and B portions), say in column c, if $c \neq b$, Sum and Replace column b of B into column c.

   (c) Zero column b in B. (This is equivalent to adding column b to itself.) Continue to the next lost element, until the list has been processed.

3. (Optional) Use the columns of B to improve the weight of non-trivial columns of R (corresponding to lost data elements processed so far). See [2, 3].

4. **Output** R (the first n columns of W) and the non-zero columns of B (from the last q columns of W).

In fact, we do not need the results from B, as the formulas we care about are only in R. The columns in R corresponding to good data elements will be "identity" columns (columns from an identity matrix) – that is, they are not changed in the algorithm. This means we could have started with the submatrix of R consisting only of the columns associated to lost data elements. So the workspace can be size $(n + q) \times (f + q)$. A column corresponding to lost data elements will be non-trivial and represent a reconstruction formula (or it will be all zero and indicate a data loss event).

Note that even though the final formulas have an *a priori* dependence on the good data elements (derivable from the good data to parity relationships in parity check matrix), we still need to know what is happenning in the data portion of R and B in order to determine what column we select in the Step 2a. This is resolved in the next section.

3. Space-Efficient Column-Incremental Construction

In this section, we present the revised algorithm. On the surface, it looks like the above algorithm, but the workspace is much smaller. The input is again the list of lost data elements F and the parity vectors B_0 of the check matrix H – B_0 is the first n rows of H. This is also the last q columns of the generator matrix (with the appropriate labeling of the elements).

In the above algorithm, the data and parity elements are treated equally. That is because we maintain workspace rows for each. In the revised algorithm below, we suppress the portion of the workspace corresponding to the data elements. Consequently, we have to treat them somewhat differently.

Algorithm: Space-Efficient Column-Incremental

1. Construct a workspace matrix $W = (C \,|\, D)$ of size $q \times (f + q)$ where C is the all zero matrix of size $q \times f$ and D is the identity matrix of size q. So, initially, $W = (0 \,|\, I_q)$. Order the list F and label the columns of C according to this order.

2. Let $i = 0$. While $i < f$, do the following:

    (a) Let r be the ith element of F.

(b) If r is a data element label, let $\vec{a}$ be the rth row of B_0 and set $\vec{u} = \vec{a}.W$. Set $\vec{u}[i] = 1$. Otherwise, (r is a parity element label), let $\vec{u}$ be the row of W corresponding to this parity element.

(c) Find b such that $f \leq b < q$ and $\vec{u}[b] = 1$. If none exists, continue to the next i.

(d) For each column position c where $\vec{u}[c] = 1$, Sum and Replace column b of D into column c (including column $c = b$).

(e) Increment i.

3. (Optional) Use the columns of D (via Sum and Replace) to improve the weight of the non-trivial columns of C.

4. **Output** C, and (optionally) the non-zero columns of D.

Before we continue by describing variations on the algorithm and the proof that this does accomplish the same result, we make the following remarks.

- If the parity elements are labeled appropriately, the initial workspace matrix is exactly the last q rows of the workspace matrix from the original algorithm.

- The revised Step 2d is identical to the corresponding step in the original algorithm. It includes the former Step 2c per the comments.

- The key new Step 2b represents a more complicated but equivalent formulation of extracting the now implicit row r from the original workspace W. For a parity element, our new workspace holds the same parity rows as before, so the step is the same – extract the row. For a data element, this step computes what row r would be in W at this step of the algorithm (assuming we selected the same column b at every step).

- In Step 2d we no longer have to zero any special columns of C. Such columns will automatically be zero. In addition, a more mathematical formulation of this step is
$$W = W + \vec{d}.\vec{u},$$
where $d = W[*, b]$, the bth column of W. The vector product here is an outer product: $\vec{d}$ (a column) has dimension $q \times 1$ and $\vec{u}$ (a row) has dimension $1 \times (f + q)$ so the outer product has dimension $q \times (f + q)$, as does W.

- The number of steps of the algorithm and the number of column operations remains the same as the previous algorithm. However, the column vector operations are now only q bits each, not $n + q$. The bit complexity is bounded now by $O(fq(f + q))$: process f lost elements, each involves at most $(f + q)$ column operations on vectors of size q. The gain is by a multiplicative factor of $q/(n + q)$ which is approximately the storage overhead of the code.

- In [2, 3], we discussed the Reverse Incremental Construction algorithm that can be used to update the workspace as data elements are reconstructed. This can lead to more efficient formulas for data elements since dependencies on reconstructed elements can shorten some formulas. A similar algorithm can be applied here, except that is it much simpler. Namely, move the column in C corresponding to a reconstructed element into an all-zero column of D. (Either completely remove the column from C or relabel it as "unmarked" so as not to interpret an all-zero column as a data loss.) This doesn't immediately improve the weights of any other formulas. However, the optional first "optimization" step in Algorithm Reverse Incremental (of [2, 3]) can then be applied. Briefly, if the non-zero positions of some column forms a superset of those of the reconstructed column, then these positions can be zero'ed in the column. This has the effect of replacing combinations of revised parity elements with the single reconstructed data element.

3.1. Reconstruction

With the output C from the algorithm, we reconstruct lost data elements as follows. Precompute a set of revised parity values by XORing out of each parity value the good data elements that were used to compute the parity. (Or, zero the rows in H corresponding to the lost data elements and compute the product $S.H$ where S is the vector of data and parity values). Now, if a column in C is all zero, then the corresponding data element is lost. If not, then the column represents a formula for reconstructing the data element as an XOR of the indicated revised parity values.

3.2. Other space efficiencies

There are more space efficient versions of this algorithm, but they trade that space for dynamic space allocation at each step of the algorithm. In effect, they take advantage of the fact that C is initially all zero and D is the identity matrix. This means that we know *a priori* what the values in parts of these matrices are. For example, in C, it turns out that the columns of C go non-zero (unless there is a data loss event) from left to right as we process the lost data elements in order.

So we only need space in C when the column goes non-zero. Similarly, the columns of D are initially the identity columns. With an appropriate ordering of the parity elements (and so the labels on the rows and columns of D), we can dynamically grow D knowing what portion is acted on and what is implicitly zero and the identity.

In addition, we observe that at each step, D zeros some column and C grows a column. This means there is an implementation that uses a workspace of size $q \times q$, and logically and dynamically partitions it into C and D portions. It also shows that the worst case running time is actually $O(fq^2)$. The bookkeeping of such an algorithm would be more difficult and perhaps not justified by the extra savings.

4. An Example

In this section, we recreate an example from [2, 3] and then show how the space efficient version improves it. For more details, see the cited papers.

Consider the EVENODD(3,5) code [1] with prime $p = 3$, 5 total disks, 3 data disks and 2 parity disks. The data and parity layout in the strips and stripe for one instance is given in the following diagram:

S_0	S_1	S_2	P	Q
$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	P_0	Q_0
$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	P_1	Q_1

The columns labeled S_0, S_1, S_2 are the data strips in the stripe (one per disk); the columns labeled P and Q are the P-parity and Q-parity strips, respectively. We order the data elements first by strip and then, within the strip, down the columns (this is the same view as the ordering of host logical blocks within the stripe). In this example, $n = 6$ and $q = 4$.

The parity check matrix H is:

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The parity check matrix is row blocked to indicate the strip boundaries and it contains in the lower portion (last four rows) an embedded identity matrix.

4.1. The Example – Column-Incremental

Suppose we lose strip S_0 and only data element $d_{0,2}$ of S_2 in the EVENODD(3,5) code above, so elements row-indexed in the matrix H (and in our workspace matrix W) by the indices $\{0, 1, 4\}$.

The initial workspace of the original Column-Incremental Algorithm is

$$W = (R \,|\, B) = \left(\begin{array}{cccccc|cccc}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right).$$

For row $r = 0$, we find some column in B that has a one in this row. There are two choices, $b = 6$ or $b = 8$. We choose $b = 6$ because its weight is less. We add this to columns $c = 0$ and $c = 8$ (where there is a one in row 0), then zero column $b = 6$. The result is

$$W = \left(\begin{array}{cccccc|cccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right).$$

For $r = 1$, select column $b = 7$ (again, this has the minimum weight), then add this to columns $c = 1, 9$, then zero column $b = 7$. This gives:

$$W = \left(\begin{array}{cccccc|cccc}
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\ \hline
1&0&1&0&0&0&0&0&1&1\\
0&1&0&1&0&0&0&0&1&0\\ \hline
1&0&0&0&1&0&0&0&0&1\\
0&1&0&0&0&1&0&0&1&1\\ \hline
1&0&0&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&0&0&1\\ \hline
0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&1
\end{array}\right).$$

Similarly, for $r = 4$ (using $b = 9$), the result is

$$W = \left(\begin{array}{cccccc|cccc}
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\ \hline
0&0&1&0&1&0&0&0&1&0\\
0&1&0&1&0&0&0&0&1&0\\ \hline
0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&1&1&0&0&1&0\\ \hline
1&0&0&0&0&0&0&0&1&0\\
1&1&0&0&1&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0&1&0\\
1&0&0&0&1&0&0&0&0&0
\end{array}\right) \qquad (4.1)$$

Observe that any column corresponding to a data element that is *not* lost has remained unchanged as an identity column.

4.2. The Example – Space-Efficient Column-Incremental

We now revisit the same example, but apply our Space-Efficient Algorithm. As before, we assume that elements numbered $\{0, 1, 4\}$ are lost. So, $f = 3$ and $q = 4$. Our new workspace matrix is

$$W = (C \mid D) = \left(\begin{array}{ccc|cccc}
0&0&0&1&0&0&0\\
0&0&0&0&1&0&0\\
0&0&0&0&0&1&0\\
0&0&0&0&0&0&1
\end{array}\right).$$

(We do not show the horizontal bar to separate the strip mapping in this form – it is not relevant.)

We now proceed to apply the algorithm. For $i = 0$, $r = 0$ so we have a data element. The vector $\vec{a} = \langle 1, 0, 1, 0 \rangle$ (the zeroth row of H). The product $\vec{u} = \vec{a}.W =$

$\langle 0,0,0 \,|\, 1,0,1,0\rangle$. We set $\vec{u}[0]=1$ so now $\vec{u}=\langle 1,0,0 \,|\, 1,0,1,0\rangle$. We next find $b=3$, the position of the first non-zero entry in the second portion of $\vec{u}$. Column $b=3$ of W equals $\langle 1,0,0,0\rangle$, so we add this to the zeroth, third and fifth columns of W to produce the updated workspace matrix:

$$W=(C \,|\, D)=\left(\begin{array}{ccc|cccc}1&0&0&0&0&1&0\\0&0&0&0&1&0&0\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\end{array}\right).$$

For $i=1$, $r=1$ and this is again a data element. The vector $\vec{a}=\langle 0,1,0,1\rangle$, row 1 of H. We compute $\vec{a}.W$ and add one in position 1 to get $\vec{u}=\langle 0,1,0 \,|\, 0,1,0,1\rangle$. We scan for $b=4$, and select column $\langle 0,1,0,0\rangle$ from W and add it to columns 1,4,6 to get

$$W=(C \,|\, D)=\left(\begin{array}{ccc|cccc}1&0&0&0&0&1&0\\0&1&0&0&0&0&1\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\end{array}\right).$$

For $i=2$, $r=4$; the vector $\vec{a}=\langle 1,0,1,1\rangle$; we compute $\vec{u}=\langle 1,0,1 \,|\, 0,0,0,1\rangle$; find $b=6$, column 6 of W equal $\langle 0,1,0,1\rangle$ and updating W we get

$$W=(C \,|\, D)=\left(\begin{array}{ccc|cccc}1&0&0&0&0&1&0\\1&1&1&0&0&0&0\\0&0&0&0&0&1&0\\1&0&1&0&0&0&0\end{array}\right).$$

The first three columns, labeled $\{0,1,4\}$ match exactly the identically labeled columns and last $q=4$ rows of (4.1).

5. Proofs

To prove that the above algorithm works as claimed, we have to show how the workspace in the original algorithm changes. We will then see that the effect on the lower portion of the workspace is captured by the revised algorithm.

The original algorithm has a workspace divided into two sections, $W=(R \,|\, B)$, where initially, $B=H$ and $R=\begin{pmatrix}I_n\\0\end{pmatrix}$. We subscript the notation on the workspace to show the changes that it goes through during the iteration steps in the algorithm. In addition, we change the notation by dividing the workspace into smaller blocks. Our initial condition is then easily represented by:

$$W_0=\left(\begin{array}{c|c}R_0&B_0\\\hline C_0&D_0\end{array}\right)$$

where $R_0 = I_n$, B_0 is the parity vectors from the parity check matrix, $C_0 = 0$, the all zero matrix (of size $q \times n$) and $D_0 = I_q$. The matrix $\begin{pmatrix} B_0 \\ D_0 \end{pmatrix} = H$, the parity check matrix. Suggestively, we have labeled the lower portion of W_0 in the same way as we labeled the workspace in the space-efficient algorithm.

For any matrix X, we let $X[*, b]$ (resp., $X[r, *]$) represent the bth column (resp., row) of X.

The original algorithm creates a sequence W_i, $0 \leq i \leq f$, where $f = |F|$, the number of lost elements. The transformation from W_i to W_{i+1} is governed by the main step. This is described mathematically as follows. Let $r = F_i$, the ith element in F. Write $W_i[r, *] = \langle \vec{u}_r \,|\, \vec{v}_r \rangle$, where $\vec{u}_r$ has length n and $\vec{v}_r$ has length q. By induction, the column $W_i[*, r]$ is all zero except for a one in row r, so that $\vec{u}_r[r] = 1$. The algorithm finds b with $0 \leq b < q$ so that $\vec{v}_r[b] = 1$.

Construct the matrix S_i by adding the row $W_i[r, *]$ into an I_{n+q} at row $n + b$:

$$S_i = \left(\begin{array}{c|c} I_n & 0 \\ \hline K_i & I_q + L_i \end{array}\right).$$

where K_i (resp., L_i) is all-zero except $K_i[b, *] = \vec{u}_r$ (resp., $L_i[b, *] = \vec{v}_r$). Note that $L_i[b, b] = 1$. The original algorithm updates the workspace W_i by the rule:

$$W_{i+1} = W_i.S_i.$$

This implies the following recursions (writing I for I_q):

$$\begin{aligned} R_{i+1} &= R_i + B_i.K_i \\ B_{i+1} &= B_i.(I + L_i) \\ C_{i+1} &= C_i + D_i.K_i && (5.1) \\ D_{i+1} &= D_i.(I + L_i) = D_i + D_i.L_i. && (5.2) \end{aligned}$$

By induction and the fact that $D_0 = I$, we see that

$$D_i = \prod_{j=0}^{i-1} (I + L_j).$$

From this we see that

$$B_i = B_0.D_i \tag{5.3}$$

and since $C_0 = 0$,

$$C_i = \sum_{j=0}^{i-1} D_j.K_j.$$

Furthermore, since $R_0 = I_n$,

$$\begin{aligned} R_i &= R_0 + B_0.K_0 + B_1.K_1 + \cdots + B_{i-1}.K_{i-1} \\ &= I_n + B_0.K_0 + B_0.D_1.K_1 + \cdots + B_0.D_{i-1}.K_{i-1} \\ &= I_n + B_0.\sum_{j=0}^{i-1} D_j.K_j \\ &= I_n + B_0.C_i. \end{aligned} \tag{5.4}$$

These equations show that the changes to R_i and B_i are induced only from the initial matrix B_0 and the changes occurring only in the lower portion of W_i. Similarly, the changes to the lower portion are completely determined by the lower portion. This clearly suggests that using only the lower workspace should be sufficient.

The next step is to see how C_i and D_i are changed according to the steps in the space-efficient algorithm. Let $d_b = D_i[*, b]$. Since $(K_i \mid L_i)$ is all zero except for the vector $\langle \vec{u}_r \mid \vec{v}_r \rangle$ in row b, we see that the product $D_i.K_i = d_b.\vec{u}_r$ and $D_i.L_i = d_b.\vec{v}_r$ (both of these vector products are "outer" products). Consequently, by (5.1-5.2), C_i and D_i are updated by adding column d_b to each column of C_i (resp., D_i) for which the corresponding position of $\vec{u}_r$ (resp., $\vec{v}_r$) is non-zero.

There are two things left to show. First, we need to show that the vector $\langle \vec{u}_r \mid \vec{v}_r \rangle$ selected in the original algorithm is equal to the vector constructed in the space-efficient algorithm. Second, we need to show that the formulas for reconstruction using only these equations is correct.

For the first issue, we proceed as follows. In the above description, $\langle \vec{u}_r \mid \vec{v}_r \rangle$ is the rth row of W_i; thus $\vec{u}_r$ is the rth row of R_i and $\vec{v}_r$ is the rth row of B_i. This is exactly what we do in the space-efficient algorithm when the lost element is a parity element. In the case that the lost element is a data element, the space-efficient algorithm computes $\vec{u} = \vec{e}_r + \vec{a}.\,(C_i \mid D_i)$ where $\vec{e}_r$ is all-zero except for a one in position r and $\vec{a}$ is the rth row of B_0. By (5.4) the left portion of $\vec{u}$ is exactly the rth row of R_i, that is $\vec{u} = \vec{u}_r$. Similarly, by (5.3), $\vec{v}_r$ is the rth row of B_i, as required.

To complete the proof, we first observe that we only need to consider what happens to C_i since that is where the reconstruction formulas are embedded. Second, we only need consider the case where all the lost elements are data elements. The effect of losing a parity element at step i in the algorithm is to zero the i column of C_i and this results in "no formula"; that is, since every column of C_i starts out as all-zero, there is no detectable change to C_i. Now, without loss of generality, we can relabel the lost (data only) elements and assume that the set $F = \{0, 1, \ldots, f-1\}$.

This means that the resulting workspace after the last step has the form

$$W = \left(\begin{array}{cc|c} \widehat{R} & 0 & \widehat{B} \\ U & I_{n-f} & V \\ \hline \widehat{C} & 0 & D \end{array}\right)$$

where we have written $R = \begin{pmatrix} \widehat{R} & 0 \\ U & I \end{pmatrix}$, $B = \begin{pmatrix} \widehat{B} \\ V \end{pmatrix}$ and $C = \begin{pmatrix} \widehat{C} & 0 \end{pmatrix}$. But by construction, $\widehat{R} = 0$ because this matrix has no dependence on the first f rows.

Let $\vec{d} = \langle \vec{d}_L, \vec{d}_G \rangle$ be data value vector organized with the lost elements first and $\vec{q}$ is the parity value vector, then by the theorems of [2, 3], we have

$$\begin{aligned} \langle \vec{d}_L, \vec{d}_G \rangle &= \langle \vec{d}_L, \vec{d}_G \,|\, \vec{q} \rangle \cdot \left(\begin{array}{cc} 0 & 0 \\ U & I_{n-f} \\ \hline \widehat{C} & 0 \end{array}\right) \\ &= \langle \vec{d}_G . U + \vec{q} . \widehat{C}, \vec{d}_G \rangle . \end{aligned}$$

Examining the lower $n - f$ rows of $R_f = R$ in (5.4) we see that $U = 0 + V_0 . \widehat{C}$ where V_0 is the lower portion of B_0. Thus,

$$\vec{d}_L = \vec{d}_G . V_0 . \widehat{C} + \vec{q} . \widehat{C} = (\vec{d}_G . V_0 + \vec{q}) . \widehat{C}.$$

The term in parentheses in this last expression is exactly the revised parity values where all the good data elements have been removed according the parity formulas. Thus, the reconstruction of the lost data is computed from these revised parity values according the formulas in $\widehat{C}$, the output of the space-efficient algorithm. This completes the proof.

6. Additional Observations

In many cases, the resulting reconstruction formulas can be rearranged so as to provide for an efficient reconstruction procedure. In effect, this provides for a systematic approach to the Generic Hybrid Reconstruction in [2, 3].

The idea is partition the reconstruction (column) vectors in the output matrix C into sets and then sort the sets so that the following hold:

- In each partition, the vectors are ordered so that each vector contains the previous vector, that is, is one at every position where the previous vector has a one.

- The partitions are the largest possible with the first property.

The following simple algorithm provides a reasonable sorting procedure.

Algorithm: Partition Sort

1. Sort the columns by weight.
2. For each column i in order, do the following:
    (a) Find a subsequent column $j > i$ which contains the current column. If none exists, continue.
    (b) Move column j to position $i + 1$, pushing all other columns $i < k < j$ to $k + 1$.
3. **Output** the reordered columns with their original data element labels.

In the example, there is a single partition where we reorder the vectors as:

$$C == \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

In so doing, we reorder the data elements to $\{1, 4, 0\}$. Reconstructing in this order is then the most efficient. Note that the first formula says that element $d_{1,0}$ is equal to the revised parity element P'_1. Then $d_{0,2} = d_{1,0} \oplus Q'_1$. Finally, $d_{0,0} = d_{0,2} \oplus P'_0$ (and this computation is more efficient than $P'_0 \oplus P'_1 \oplus Q'_1$).

7. Summary

In summary, we presented a revised version of the Column-Incremental Algorithm in [2, 3] that can be used to generate reconstruction formulas for any XOR-based erasure code. The revised algorithm is significantly more space and time efficient than the original. Reconstruction formulas depend implicitly on good data elements: the good data elements are pre-XORed into their respective parity elements. The output of the new algorithm is a set of reconstruction formulas that use only these revised parity values. The algorithm is very simple; sample Mathematica code is provided as well.

A. Sample Mathematica Code

The following Mathematica code snipet implements the main algorithm. It decomposes the workspace W into the two separate components C and D (called `wsc` and `wsd` respectively). The input is the list of `lost` data or parity elements and `qvec`, the parity vectors in the parity check matrix. The function `AddColumn` adds a column vector to a specified column of a matrix; this is given below. The function `ZeroMatrix` is self-explanatory. All other functions are built-in to Mathematica.

The main algorithm outputs a conjunction of the two workspace matrices, with a column of "-" separating them for visual ease. The columns of `wsc` are labeled by the elements in `lost`.

```
Reconstruct[lost_,qvec_]:=Block[{tmp,wsc,wsd,n,q,l,u,v,r,b,d,x},
   l = Length[lost];
   n = Dimensions[qvec][[1]];
   q = Dimensions[qvec][[2]];
   wsc = ZeroMatrix[q,l];
   wsd = IdentityMatrix[q];
   x = Table["-",{i,1,q}]; (* used for output pretty print *)

   Do[
     r = lost[[i]];

     If[ r<=n,
     (* data case *)
     (* u and v are what would be in top part if we tracked it *)
       u = Mod[qvec[[r]].wsc, 2];
       u[[i]] = 1;  (* from the implied identity initialization *)
       v = Mod[qvec[[r]].wsd, 2],
     (* parity case *)
     (* we track what happens down here explicity *)
       u = wsc[[r-n]];
       v = wsd[[r-n]]
     ];

     b = 0;
     Do[ If[v[[j]]==1, b=j;Break[]], {j,1,q}];
     If[ b==0, Continue[]];  (* data loss event, nothing to do *)

     d = Transpose[wsd][[b]]; (* the b-th column of wsd *)
```

```
      (* compute these efficiently
         wsc = wsc + Outer[Times,d,u];
         wsd = wsd + Outer[Times,d,v];
      *)
      Do[
        If[ u[[j]] == 1, wsc = AddColumn[wsc,j,d]];
      ,{j,1,l}];

      Do[
        If[ v[[j]] == 1, wsd = AddColumn[wsd,j,d]];
      Null, {j,1,q}];

    ,{i,1,l}];  (* end of loop over lost elements *)

    (* Only wsc is what we care about but we show both *)
    (* The columns of wsd have no particular order.
       Rows of wsc and wsd correspond to parity elements "in order".
     *)
    Return[Transpose[Join[Append[Transpose[wsc],x],Transpose[wsd]]]];
];
```

Here is the function `AddColumn`. The inputs are the matrix `mat` to modify (we output the revised matrix and do not change the matrix in place), the column index `idx` indicating what column to change and the vector `vec` to add in:

```
AddColumn[mat_,idx_,vec_]:=Block[{tmp},
    tmp = Transpose[mat];
    tmp[[idx]] = Mod[tmp[[idx]]+vec,2];
    Return[Transpose[tmp]];
];
```

Clearly, there are more efficient implementations of these methods (in particular, transposition into a row-based view would remove a lot of the `Transpose` calls), but this version is more in the spirit of the original algorithm.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44:192–202, 1995.

[2] V. Deenadhayalan, J. L. Hafner, KK Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. Technical Report RJ 10354, IBM Research, San Jose, CA, 2005.

[3] J. L. Hafner, V. Deenadhayalan, KK Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, pages 183–196, San Francisco, CA USA, December 2005.