Heuristics for Optimizing Matrix-Based Erasure Codes for Fault-Tolerant Storage Systems

James S. Plank
Catherine D. Schuman
B. Devin Robison

Technical Report UT-CS-10-664
EECS Department
University of Tennessee
Knoxville, TN 37996

December 14, 2010

http://www.cs.utk.edu/~plank/plank/papers/CS-10-664.html

This paper has been submitted for publication. Please see the link above for publication status of the paper.

Heuristics for Optimizing Matrix-Based Erasure Codes for Fault-Tolerant Storage Systems

James S. Plank, Catherine D. Schuman
EECS Department
University of Tennessee
Knoxville, TN USA
Email: plank@cs.utk.edu, cschuman@eecs.utk.edu

B. Devin Robison
Department of Chemical Engineering
University of Utah
Salt Lake City, UT USA
Email: devin083@gmail.com

Contact Author: James S. Plank. *plank@cs.utk.edu*. 865-256-2397. Fax: 865-974-4404. This material has been cleared through the authors' institutions. Approximate word count: 8,300.

***Abstract*—Large scale, archival and wide-area storage systems use erasure codes to protect users from losing data due to the inevitable failures that occur. All but the most basic erasure codes employ bit-matrices to perform encoding and decoding. These bit-matrices are massaged so that encoding and decoding become described by lists of exclusive-or (XOR) operations.**

When converting matrices to lists of XOR operations, there are CPU savings that can result from strategically scheduling the XOR operations and leveraging intermediate results so that fewer XOR's are performed. It is an open problem to derive a schedule from a bit-matrix that minimizes the number of XOR operations.

We attack this open problem, deriving two new heuristics called *Uber-CHRS* and *X-Sets* to schedule encoding and decoding bit-matrices with reduced XOR operations. We evaluate these heuristics in a variety of realistic erasure coding settings and demonstrate that they are a significant improvement over previously published heuristics. In particular, a hybrid of the two heuristics, which we call *Uber-XSet*, provides consistently good schedules across all of our tests. We provide an open-source implementation of these heuristics so that practitioners may leverage our work.

***Keywords*-Erasure codes; Fault-tolerant storage; RAID; Disk failures;**

I. Introduction

As storage systems have grown in size, scale and scope, the failure protection offered by the standard RAID levels (1-6, 01 and 10) is in many cases no longer sufficient. These systems protect to a maximum of two simultaneous disk or node failures. However, proliferation of components, the requirement of wide-area networking and enriched failure modes have led storage designers to tolerate larger numbers of failures [1], [11]. For example, companies like Google, IBM and Cleversafe, and academic storage projects like Oceanstore [16], DiskReduce [8], HAIL [5], and others [26] all employ storage systems that tolerate at least three failures. As systems grow further and are deployed over wider networks, their fault-tolerance requirements will increase even further.

Systems that tolerate failures beyond RAID employ Reed-Solomon codes for fault-tolerance [19], [21], [24]. With Reed-Solomon codes, k data disks are encoded onto m coding disks in such a way that the system of $n = k + m$ disks may tolerate the failure of any m disks without data loss. Reed-Solomon codes employ finite field arithmetic over w-bit words, which, when implemented in software, is computationally expensive. A technique called *Cauchy Reed-Solomon (CRS)* coding [4] converts the finite field arithmetic into a sequence of bitwise exclusive-or (XOR) operations and improves performance. Currently, CRS codes represent the best performing general purpose erasure codes for storage systems [22].

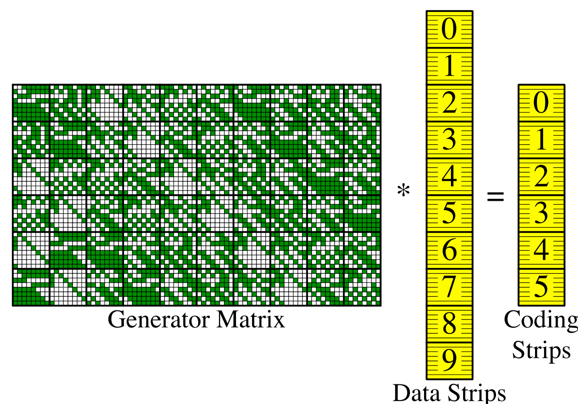

Figure 1. Example 16-disk CRS system that can tolerate six disk failures: $k = 10$, $m = 6$, $w = 8$.

To help motivate our work, Figure 1 shows a 16-disk system that employs CRS to tolerate any six disk failures. This is the exact configuration and erasure code used by Cleversafe, Inc, (http://www.cleversafe.com) in the first release of its commercial dispersed storage system. The parameters of the code are $k = 10$, $m = 6$ and $w = 8$. Thus there are 16 disks, ten of which store *data strips* and six of which store *coding strips*. The last parameter (w) means that each

strip is partitioned into $w = 8$ *packets*. The packets are then encoded using bitwise exclusive-or operations (XOR), as if they were bits in a binary matrix-vector product. The actual encoding is defined by a (48×80) *Generator* matrix, which is derived from finite field arithmetic over eight bit words [4]. Decoding is defined similarly.

Figure 1 exemplifies a matrix-vector product, $AX = B$, that is central to nearly all erasure coded systems. All elements of A, X and B are bits. However, they are used to encode and decode packets and strips of data and coding information that are much larger than a bit, using the XOR operation. The reason that packets are large is that XOR's of 32 and 64 bit words are supported by most machine architectures, and many vector co-processors XOR larger regions very efficiently. Thus, although conceptually the equations are binary, in reality they are used to XOR large regions efficiently.

The number of XOR operations required by an erasure code has a direct relationship to the performance of encoding or decoding. While there are other factors that impact performance, especially cache behavior and device latency, reducing XOR's is a reliable and effective way to improve the performance of a code. As such, nearly all special-purpose erasure codes, from RAID-6 codes such as EVENODD [2], RDP [6], X [27] and P [15] codes, to codes for larger systems such as STAR [14], T [17] and WEAVER [10] codes, have minimizing XOR operations at their core.

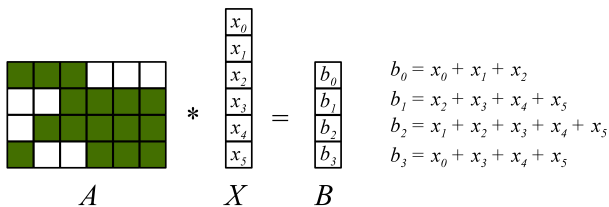

Figure 2. Example of a bit matrix to describe encoding or decoding operations.

Given an encoding or decoding scenario embodied by a matrix-vector product $AX = B$, such as the one depicted in Figure 2, one way to perform the the encoding or decoding is to calculate each product bit b_i by XOR-ing every bit x_j such that $A_{i,j}$ equals one. This is equivalent to evaluating each of the equations on the right side of Figure 2 independently.

However, there are opportunities for calculating B with fewer XOR's, based on the structure of the matrix. For example, in Figure 2, bits b_1, b_2 and b_3 each contain the sum $(x_3 + x_4 + x_5)$. If one calculates that intermediate sum first and uses it to calculate b_1, b_2 and b_3, then one saves four XOR operations over calculating those three products independently. As a second observation, there are times when one may calculate a product bit from other product bits or intermediate sums, and this calculation is cheaper than calculating the product bit from bits of X. For example, in Figure 2, bit b_0 is equal to $b_2 + b_3$. Thus, it is cheaper to calculate those two product bits and XOR them together than it is to calculate b_0 from x_0, x_1 and x_2.

This paper explores techniques to calculate a binary matrix-vector product with the fewest number of XOR operations. We call this "XOR-Scheduling." We can only discover lower bounds on the number of XOR operations using enumeration, which we do for small matrices. For larger matrices, we must resort to heuristics. Two such heuristics have been presented in the literature [12], [13]. In this work, we develop two new techniques called *Uber-CSHR* and *X-Sets*. We describe them in detail and then compare the performance of all heuristics in two settings — first with respect to small matrices for which we have been able to determine optimal schedules by enumeration, and second with respect to larger matrices, such as Figure 1, that occur in real erasure coding applications. The result is a significant improvement in the performance of both encoding and decoding using bit matrices. In particular, a hybrid of the two heuristics, called *Uber-XSet* shows excellent performance across all of our tests.

II. Previous Work

There have been two previous research projects that address this problem. The first proposes *Code Specific Hybrid Reconstruction (CSHR)* [12], which is based on the idea that it is often advantageous to calculate a product bit using a previously generated product bit as a starting point. CSHR is very effective in improving the performance of some erasure codes. For example, the Minimum Density codes for RAID-6 [3], [22] have encoding matrices with a provably minimum number of ones. However, their decoding matrices are quite dense, and without optimization, their performance would be unacceptably slow. CSHR works very well on these matrices, enabling the them to decode with performance on par with the other RAID-6 codes [22]. CSHR has an open-source implementation in the **Jerasure** erasure coding library [23].

The second project attacks the issue of identifying and exploiting common sums in the XOR equations [13]. The authors conjecture that deriving an optimal schedule based on common sums is NP-Complete, and then give a heuristic based on Edmonds' maximum matching algorithm [7]. They demonstrate how the heuristic can be employed to encode Reed-Solomon coding variants with similar performance to RDP [6], EVENODD [2] and STAR codes [14]. The authors do not give their heuristic a name, so we call it *Subex*, since it attempts to minimize the number of XOR's by identiying common subexpressions. There is no implementation of Subex[1].

[1]Cheng Huang, Microsoft Research, personal communication.

III. The Contribution of This Paper

In this paper, we make significant inroads into the problem of deriving optimal and good XOR schedules from binary matrices. In particular, we do the following:

1) We develop a methodology for deriving the optimal schedule for small matrices. We use this to evaluate the effectiveness of the previous heuristics for scheduling Cauchy bit matrices for $w \leq 8$.
2) We extend CSHR by identifying two places where it may be refined by parameterization. We call the resulting heuristic *Uber-CSHR*.
3) We derive new scheduling heuristics based on the concept of an *XOR set* (X-Set). The Subex heuristic may be expressed in terms of X-Sets. We parameterize the X-Set heuristics in three ways, resulting in a suite of enriched heuristics. One of these parameterizations is a hybrid of Uber-CSHR and X-Sets, which we call *Uber-XSet*.
4) We evaluate all heuristics on a variety of encoding and decoding scenarios that can best leverage them.
5) We provide an open-source implementation of these heuristics so that other researchers and practitioners may leverage our work.

IV. Formal Specification

We are given a $(r \times c)$ bit matrix A, that we apply to a c-element bit vector X to calculate a r-element bit vector B. We call the bits of X *data* bits and the bits of B *target* bits. We perform this calculation with a sequence of XOR operations. The sequence of XOR operations is a *schedule*, whose size we desire to minimize. We assume that we may use temporary storage in the calculation. Ideally, we would not use more storage than the r bits in B, but we do not make this a requirement.

While we may represent a schedule by specifying XOR operations, we find it useful to employ an alternate representation. We define an *element* of the system to be a c-bit word, which may be represented by a bit string of size c. Each data bit x_i corresponds to an element of the system which is composed of all zero bits, plus a one bit in place i. Thus, in Figure 2, $x_0 = 100000$, $x_1 = 010000$, etc. When we add two elements, we simply XOR their bits. Thus, for example, $x_0 + x_1 = 110000$.

Each product bit b_i is also an element of the system, which may be represented by row a_i of the matrix. For example, b_0 in Figure 2 is equal to 111000. We call the product bits *target elements*, and since the order in which we calculate them is immaterial, we also refer to B as the *target set* of elements $\{b_0, b_1, \ldots b_{r-1}\}$. Since B is ultimately defined by the matrix A, we equivalently refer to A as the target set as well. A *schedule*, S is an ordered set of elements, $\{s_0, s_1, \ldots\}$ that has the following properties:

- The first c elements are the elements of X: $s_i = x_i$ for $i < c$.
- Any s_k that is not an element of X must equal $s_i + s_j$, where $i < j < k$.

Given a matrix A, we desire to create a schedule S_A^{opt} such that S_A^{opt} contains all the target elements, and $|S_A^{opt}|$ is minimized. For example, Figure 3 shows an optimal schedule for the matrix in Figure 2. The figure uses two different ways to represent the schedule. In Figure 3(a), the elements of the schedule are simply listed in order. Figure 3(b) uses a pictoral representation where the first c elements of the schedule are omitted, since they are simply the elements of X. The target elements are denoted on the right side of the picture.

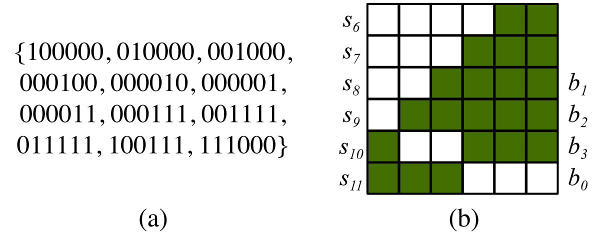

Figure 3. An optimal schedule for the matrix in Figure 2: (a) A listing of the elements of the schedule. (b) A pictoral representation which omits the data elements.

While schedules defined in this way do not specify exactly how to perform the XOR's to generate each element, each schedule S guarantees that all the elements it contains may be calculated with exactly $|S| - |X|$ XOR's. One may generate the XOR's from a schedule with a simple doubly-nested loop.

The following theorem, while simple and intuitive, is quite helpful in finding good schedules:

Theorem 4.1: Let s_i, s_j and $s_k \in S$ such that $i < j < k-1$ and $s_k = s_i + s_j$. Then there exists a legal schedule S' defined as follows:

$$S' = \left\{ s'_x | s'_x = \left\{ \begin{array}{ll} s_x & \text{when } x \leq j \\ s_k & \text{when } x = j+1 \\ s_{x-1} & \text{when } j+1 < x \leq k \\ s_x & \text{when } x > k \end{array} \right\} \right\}$$

Put in English, if we are constructing a schedule that includes s_i and s_j, and the schedule will include $s_k = s_i + s_j$, then we may place s_k into the schedule immediately after s_j. For example, in Figure 3, since element b_3 is equal to $x_0 + s_7$, it can be moved directly after s_7 without changing the legality of the schedule.

Proof: The only element whose order has changed relative to the other elements is s'_{j+1}. Therefore the only element that can violate the property of being the sum of two previous elements is s'_{j+1}. Since it is the sum of s'_i and s'_j and $i < j$, s'_{j+1} is also a legal element, and S' is a valid schedule. ■

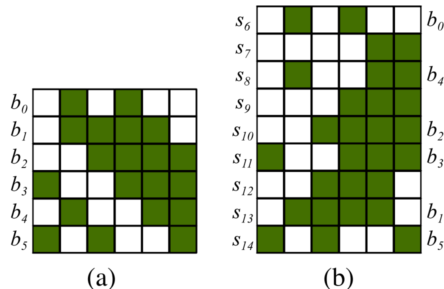

Figure 4. (a) C40: The Cauchy bit matrix for the value 40 in $GF(2^6)$. (b) An optimal schedule for C40.

V. Optimal Schedules

Given a target set A of elements, one may use a breadth-first search (BFS) to determine an optimal schedule S_A^{opt}. Let G be an unweighted graph such every possible schedule S of c-bit elements corresponds to a node N_S in G. Suppose $S' = S + \{e\}$, and both S and S' are legal schedules. Then there is an edge in G from N_S to $N_{S'}$. Let S_X be the schedule containing only the data elements. Then an optimal schedule will be represented by any node $N_{S_A^{opt}}$ such that S_A^{opt} contains the target elements and the distance from N_{S_X} to $N_{S_A^{opt}}$ is minimized.

This is a standard unweighted shortest path problem, which may be solved with BFS. Unfortunately, the performance of BFS is proportional to the number of nodes whose distance from N_{S_X} is less than $N_{S_A^{opt}}$'s distance, and this number is exponential in c and $|S_A^{opt}|$. We don't give an exact formula, because it is unimportant – we can only perform this search for small values of c and $|S_A^{opt}|$.

Theorem 4.1 allows us to prune the search by coalescing nodes. If there is an edge from N_S to $N_{S'}$ and $S' = S + \{b_i\}$, then Theorem 4.1 specifies that any optimal schedule that starts with S is equivalent to one that starts with S'. Thus, we may coalesce nodes S and S', thereby reducing our search significantly.

To get an intuitive feel for the structure of these optimal schedules, we determined optimal schedules for all elemental CRS coding matrices whose size is ≤ 8. To review, classic Reed-Solomon coding [19] operates on $w-$bit words, employing Galois Field arithmetic over $GF(2^w)$. It utilizes a matrix-vector product $AX = B$, where X corresponds to the data and B corresponds to the *codeword* (data plus coding). Each element of the equation is a $w-$bit word.

CRS coding [4] converts each word in X and B into a w-element bit vector, and each word in A to a $(w \times w)$ bit matrix as in Figure 1. Thus, each w-bit word corresponds to a $(w \times w)$ bit matrix. Since these matrices form the building blocks of CRS coding, we evaluate their optimal schedules.

We present an example in Figure 4, which examines the Cauchy bit matrix for the value 40 when $w = 6$. We call this matrix *C40,* and it is pictured in Figure 4(a). The optimal schedule for C40, generated by our pruned BFS, is depicted in Figure 4(b). It requires 9 XOR's. The path length to this schedule, however, is just three edges, because Theorem 4.1 allows us to coalesce six nodes in the path. For clarity, the path to this schedule is

$$(S_X + \{b_0\}) \rightarrow (S_X + \{b_0, s_7, b_4\}) \rightarrow$$
$$(S_X + \{b_0, s_7, b_4, s_9, b_2, b_3)\} \rightarrow S_{C40}^{opt}.$$

This schedule is interesting because it is an improvement over both CSHR and Subex, which each generate schedules with 11 XOR's. We will examine these later, but a key observation is that neither technique identifies the importance of s_{12} as an intermediate sum, since $b_1 = s_{12} + x_1$, and $b_5 = s_{12} + b_3$.

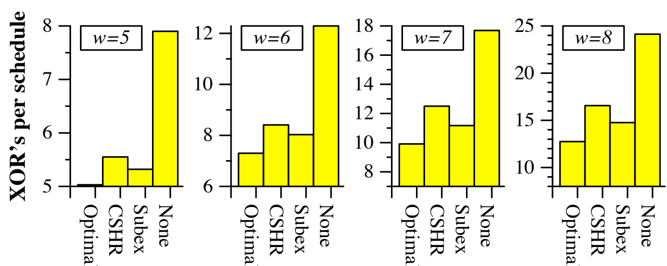

Figure 5. Comparison of schedules on Cauchy matrices for $5 \leq w \leq 8$.

Figure 5 compares the performance of optimal schedules to CSHR, Subex, and no scheduling for the Cauchy matrices for $5 \leq w \leq 8$. To generate these graphs, we considered the $(w \times w)$ Cauchy matrix for each value of i from 1 to $2^w - 1$. We scheduled each of these matrices with each of the four techniques and plot the average number of XOR's per schedule. The performance of the heuristics relative to optimal gets worse as w increases. For $w = 8$, CSHR and Subex perform 30% and 16% worse than optimal. This is significant because eight is a natural value for implementations, since each strip is partitioned into eight packets, thereby allowing strip sizes to be powers of two.

A simple way to employ these results is to store the optimal schedules and then use them for encoding and decoding CRS coding applications for these values of w. This obviates the need for storing and manipulating bit matrices, simplifying implementation. For example, consider the CRS matrix from Figure 1. This matrix has 1968 ones, and encoding it with no scheduling requires 1920 XOR's for the 48 rows, an average of exactly 40 XOR's per row. Using the optimal schedules generated above reduces this number to 1210 XOR's, or 25.20 XOR's per row. This is a significant reduction. The results below improve on this still further.

VI. Heuristics for Larger Matrices

When c grows, it becomes impractical, if not impossible to generate optimal schedules. For example, generating the

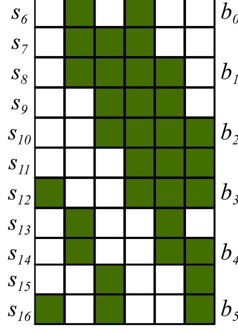

Figure 6. The schedule generated by CSHR for C40.

schedules for the (8×8) Cauchy matrices took multiple days on ten workstations. The CRS coding matrix from Figure 1 has 48 rows and 80 columns. Generating an optimal schedule for a matrix of this size would take too much computation and memory. Thus, we need to rely on heuristics. We explore two types of heuristics: those based on CSHR, and those based on the concept of an *XOR set (X-Set)*. The Subex heuristic may be expressed in terms of the latter.

A. CSHR and Uber-CSHR

CSHR is defined as follows. Let $X(b_i)$ be the number of XOR's required to generate b_i from the data elements. At each step of the algorithm, one selects a target element b_j to generate whose value of $X(b_j)$ is minimal. That element is generated, and then for every remaining target element b_i, $X(b_i)$ may be improved by building b_i from b_j and the data, rather than solely from the data. If this is the case, $X(b_i)$ is updated to reflect the improvement.

We present an example using C40. Target b_0 is first generated with one XOR operation. Following that, targets b_i, for $1 \leq i \leq 3$, can each be generated from b_{i-1} with two XOR operations. Finally, b_4 and b_5 are generated from the data elements with two XOR's each. The resulting schedule, which employs 11 XOR's, is shown in Figure 6. As mentioned in Section V, even though element b_5 can be calculated from the sum of s_9 and b_3, CSHR does not discover this fact, because it only considers building the target elements from data elements and previously built target elements.

CSHR also does not consider the order in which target elements are constructed from the data elements. For example, s_7 can equal 011100 as depicted in Figure 6, or 010110. The choice is arbitrary.

We extend CSHR in two straightforward ways to yield the *Uber-CSHR* heuristic. First, instead of considering only previously generated target elements as new potential starting points for subsequent target elements, we can consider every intermediate sum. Second, instead of considering only the newest target (or intermediate sum) as a potential starting point for other targets, we can consider every combination of two generated targets (or intermediate sums), or three and so on.

Thus, our Uber-CSHR heuristic has two parameters, which we label Uber-CSHR$_L^{T|I}$:

1) $T|I$: Whether it uses just target elements (T) or all intermediate sums (I) as starting elements.
2) L: We consider all combinations of L previously generated targets or intermediate sums as potential starting points.

CSHR as originally specified is equal to Uber-CSHR$_1^T$. If we apply Uber-CSHR$_2^I$ to C40, we can achieve an optimal schedule of 9 XOR's. The schedule is identical to the one pictured in Figure 6, except elements s_{13} and s_{15} are deleted. This is because the algorithm identifies the fact that b_4 and b_5 may be constructed as the sum of two previous intermediate sums/targets: $b_4 = s_7 + b_2$, and $b_5 = s_9 + b_3$.

The running time complexity of Uber-CSHR$_L^T$ is $O(cr^{L+1}\log(r))$, and of Uber-CSHR$_L^I$ is $O(c|S|^{L+1}\log(r))$, where S is the schedule generated. We will evaluate and discuss running time more thoroughly in Section VII below.

B. X-Sets and Their Derivatives

The second set of new heuristics is based on a data structure called the *XOR Set (X-Set)*. Like CSHR, these heuristics are greedy, taking a current schedule and iteratively adding an element to it until the schedule contains all the target elements. At each iteration, there is a set of target elements $B_{\notin}$ that are not in the current schedule. Each element $b \in B_{\notin}$ will carry with it a list of X-Sets. Each X-Set is composed of elements $\{s_{x0}, s_{x1}, \ldots\}$ such that each s_{xi} is an element of the schedule so far, and the sum of all s_{xi} equals b.

At the first iteration, the schedule is initialized to contain all the data elements, and each target has one X-Set composed of the data elements that sum to it. As an example, we show the initial X-Sets of C40 in Figure 7(a).

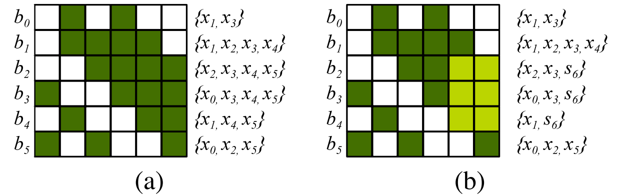

Figure 7. (a) Initial X-Sets for C40. (b) X-Sets after adding $s_6 = x_4+x_5$.

At each iteration, we choose a new element to add to the schedule. We restrict this element to being the sum of two elements from the same X-Set for some element $b \in B_{\notin}$. For example, in Figure 7(a), we can choose $x_1 + x_2$ and $x_1 + x_3$ (and many other sums), but we can't choose $x_0 + x_1$ because those two elements do not appear in the same X-Set. Suppose we choose $x_i + x_j$. Then we modify the X-Sets so that every X-Set that contains both x_i and x_j replaces

those two elements with their sum. For example, suppose we choose $s_6 = x_4 + x_5$ as our first non-data element in the schedule. Then Figure 7(b) shows the modification of the X-Sets.

Clearly, it is advantageous to select sums that reduce the size of as many X-Sets as possible. This is how the Subex heuristic works [13]. Specifically, the X-Sets are converted to a graph G where each current element of the schedule is a node, and there is an edge between s_i and s_j if the two elements appear in the same X-Set. Each edge is weighted by the number of X-Sets in which the edge appears. G is then pruned to contain only edges of maximal weight, and a maximum matching of this pruned graph is determined using Edmonds' general matching algorithm [7]. All of these edges are added to the schedule, updating the X-Sets as described above. The intuition behind this algorithm is to choose edges that maximize the number of X-Sets whose sizes are reduced. The matching algorithm ensures that a maximum number of these edges is chosen at each step.

We plot an example in Figure 8. This is the graph created from the X-Sets of Figure 7(b). When this graph is pruned, we are only left with the three edges of weight two. There is no matching that contains two of these edges, so any edge may be selected next.

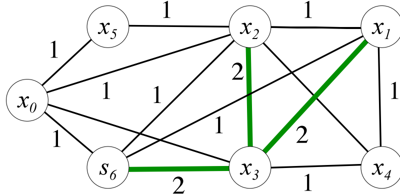

Figure 8. The Subex graph generated from the X-Sets of Figure 7(b).

We view Subex as merely one way to exploit X-Sets. We augment this exploitation in four additional ways, which we detail in the following four subsections.

1) Theorem 4.1: In relation to X-Sets, Theorem 4.1 states that if an X-Set contains two elements, then its target should be generated immediately. For example, in Figure 7(a), target b_0 should be generated first. Thus, any algorithm that employs X-Sets should always search first for X-Sets with two elements. All the algorithms described below work in this manner.

2) **Threshold***: Considering Additional X-Sets:* To this point, we have only considered one X-Set per target. This helps us find common sums, but it misses an important source of XOR reduction. Recall that in the optimal schedule pictured in Figure 3, $b_0 = b_2 + b_3$. This is a sum that will never be discovered by the X-Sets described above, because b_2 and b_3 both contain $(x_3 + x_4 + x_5)$, and adding the elements together cancels these bits. As described, the X-Sets do not consider elements that cancel bits.

To address this deficiency, whenever we add a new element to a schedule, we attempt to add additional X-Sets to a target's list. Consider a new element s and a target $b \in B_{\notin}$. For all X-Sets XS in b's list, if there are two elements s_x and s_y in XS such that $s_x + s_y = s$, we simply replace s_x and s_y with s. This is no different from the description above. If we have modified any of b's X-Sets in this manner, then we delete any other of B's X-Sets whose size is more than its minimum X-Set size plus a *threshold*.

If we have not modified any of b's X-Sets, then we create a new X-Set composed of s and data elements such that the sum of all the elements in this new X-Set equals b. If the size of this new X-Set is within *threshold* of the smallest X-Set for b, then we add the new X-Set to b's list.

While this adds to the complexity, running time and memory utilization of the heuristic, it also adds flexibility and the ability to discover better schedules, since the X-Sets may now include elements that cancel bits rather than always adding them. We present an example starting from the initial state of C40, depicted in Figure 7(a). Figure 9 shows the X-Sets after the element $b_0 = x_1 + x_3$ is added to the schedule, and *threshold* equals one.

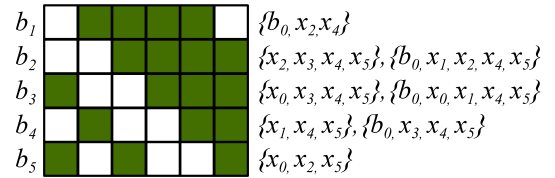

Figure 9. New X-Sets when $b_0 = x_1 + x_3$ is added to the schedule, and *threshold* is one.

Element b_1's X-Set contains x_1 and x_3, so those two elements are replaced by b_0. The remaining targets do not contain x_1 and x_3 in their X-Sets, so we consider creating new X-Sets for them. First, consider b_2. If we use only b_0 and the data elements, we create a new X-Set $\{b_0, x_1, x_2, x_4, x_5\}$. This X-Set's size is five, which is one more than the minimum X-Set for b_2, so we add this new X-Set to b_2's list. We add similar sets for b_3 and b_4. With b_5, the new X-Set would be $\{b_0, x_0, x_1, x_2, x_3, x_5\}$, whose size is three bigger than b_5's other X-Set. Since that size is greater than the threshold, the X-Set is not added to b_5's list.

3) **L***: Considering More Starting Elements:* The next parameter is identical to Uber-CSHR — it considers additional combinations of intermediate sums to generate new X-Sets. We illustrate with an example. Suppose we are scheduling C40 with a *threshold* of zero and have added elements 010100, 000011, 010011 and 000111 to the schedule. The state of the schedule is depicted in the top box of Figure 10. There are four targets remaining and their X-Sets are as depicted. From Theorem 4.1, we add elements b_2 and b_3 next. In the middle box of Figure 10, we show the state after adding b_2. A second X-Set — $\{b_2, x_2, x_5\}$ — is added

to b_1's list of X-Sets. However, there is a third X-Set, $\{b_2, b_4, x_4\}$, that we can add if we consider combinations of two intermediate sums, rather than one. In the bottom box, we show the state after adding b_3. Now there is a second X-Set — $\{b_2, b_3, x_5\}$ — that we can add to b_5's list if we consider two intermediate sums. Without this last X-Set, there are no elements that we can add to improve both b_1 and b_5's X-Sets. However, this last X-Set enables us to add $b_2 + x_5 = 001110$ to the schedule, shortening the X-Sets of both targets. This yields the optimal schedule shown in Figure 4(b). Without these additional X-sets, we cannot generate an optimal schedule.

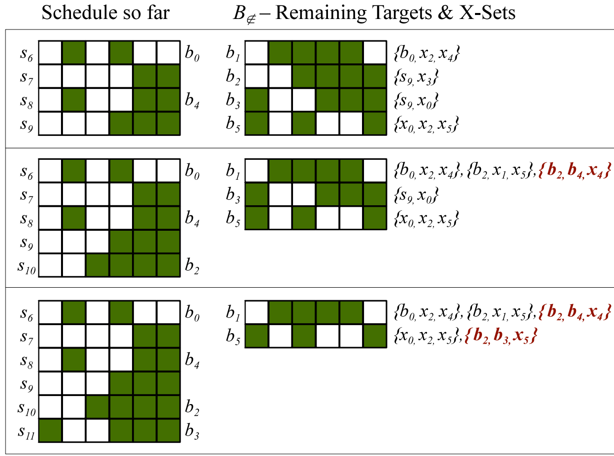

Figure 10. An example of scheduling C40 to motivate the *nstart* parameter.

The L parameter specifies that when a new element s is generated, we look at all combinations of L non-data elements in the schedule that include s, to see if any of them can generate new X-Sets that fit the *threshold*. Thus, setting L to zero will generate no additional X-Sets. Setting it to one will add X-Sets as described in Section VI-B2 above. Setting it to two will generate the X-Sets in Figure 10.

As in Uber-CSHR, increasing L affects the running time of the heuristic significantly. Specifically, if the schedule size is $|S|$, there are $\binom{|S|-c}{L}$ combinations of elements to test when adding a new element to the schedule.

4) **Technique:** *Selecting the Next Element:* The final parameter of the X-Set heuristic is probably the most important: how we select the next element in a schedule. While Subex uses Edmonds' matching algorithm, there are other techniques of varying complexity that we can employ. We derived and tested about ten alternative techniques, and after some exploratory work, have narrowed them to the five described below. The first four leverage edge weights as in Subex. The last is a hybrid of X-Sets and Uber-CSHR.

Maximum-Weight (MW): Given a current schedule S and remaining targets $B_{\notin}$ with their X-Sets, we define the *weight* of an element e to be the number of targets $b \in B_{\notin}$ whose minimum X-Set size will shrink if e is added to the schedule. For example, in the bottom box of Figure 10, the element $(b_2 + x_5)$ has a weight of two, since it will decrease both b_1 and b_5's minimum X-Set size. Other elements, like $(b_0 + x_2)$ and $(x_0 + x_2)$, have a weight of one, since they only shrink one target's X-Set. The **MW** technique selects any element with maximum weight.

MW-Smallest-Set (MW-SS): Of all the elements with maximum weight, this technique selects one that decreases the size of the smallest X-Set. This technique may be implemented to run nearly as fast as **MW**, and refines it so that it generates targets more quickly.

MW-Matching: As with Subex, we can view all elements with maximum weight as edges in a graph, and run Edmonds' algorithm on it to find a maximum matching. While Subex chooses all edges in the matching to go into the schedule, **MW-Matching** simply selects one edge in the matching and adds it to the schedule. It only adds one, because the X-Sets may change significantly after adding this one element to the schedule.

MW^2: This is the most expensive technique. Let $|$**MW**$|$ be the number of elements with maximum weight. For each of these elements, we simulate adding it to the schedule, and then calculate $|$**MW**$|$ of the resulting schedule. **MW^2** selects the element that maximizes this value. This is expensive since it involves generating $|$**MW**$|$ schedules at each step, and then throwing away all but one.

Uber-XSet: This technique is a hybrid of Uber-CSHR^I_L and X-Sets. It starts by only considering targets in $B_{\notin}$ whose minimum X-Set sizes are the smallest. Uber-CSHR^I_L would simply pick any sum from one of these X-Sets. The **Uber-XSet** technique instead chooses the sum that has the maximum overall weight. For example, in Figure 9 **Uber-XSet** only considers targets b_1, b_4 and b_5, since their minimum X-Sets sizes are three. Of all elements that will shrink one of these targets, the element $000011 = (x_4 + x_5)$ is chosen because it shrinks one of these X-Sets (b_4's), and has a weight of three. The intent with Uber-XSet is for it be a good general-purpose heuristic, as it blends properties from both Uber-CSHR and X-Sets.

VII. Evaluation

We have implemented all the heuristics described in the previous sections. To evaluate them, we have chosen two sets of tests: one for small matrices and one for large matrices. The smaller test allows us to compare the heuristics with provably optimal schedules. The larger test allows us to compare the heuristics on matrices that are used in practice.

In the tests, we focus primarily on the effectiveness of the generated schedules. We focus to a lesser degree on the running times of the heuristics. Some, like Uber-CSHR$^{T|I}_1$ run very fast, and others, like the X-Set heuristics when L grows beyond one, run very slowly. In our tests, we limit

the heuristics to those that take roughly an hour or less to run on a commodity microprocessor.

The reason we view running time as secondary to the effectiveness of the schedules produced is that real implementations will very likely precompute and store the schedules rather than compute them on the fly. Since the first c elements of a schedule are identity elements, a schedule of size $|S|$ may be represented by a list of $(|S|-c)$ c-bit numbers. Consider the largest example in this paper: the decoding matrices from Figure 1. The best decoding schedules reported below average 812 elements, which means that they can be represented with 732 80-bit words. There are $\binom{10}{6} = 210$ combinations of 6-disk failures that must be scheduled, resulting in (210)(732)(10) = 1.47 MB that is required to store all possible decoding schedules. This small amount of storage easily justifies devoting significant running time to generating good schedules.

A. Small Matrices

We limit our presentation here to the 255 Cauchy matrices for $w = 8$. This is the largest value of w for which we know the optimal schedules and is also a value that is used in some commercial installations. We also limit our focus solely to the size of the schedules and not the running times of the heuristics.

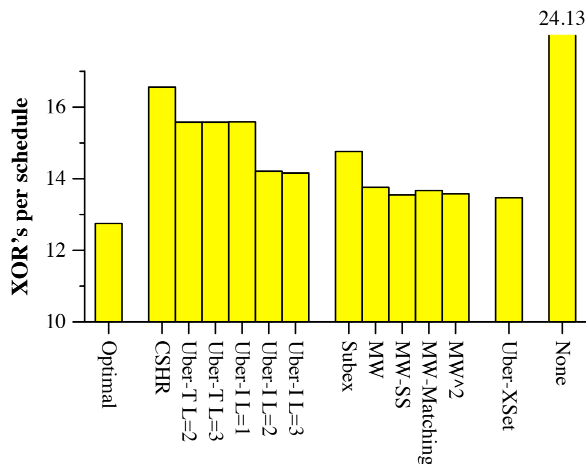

Figure 11. The best schedules for each heuristic on the 255 Cauchy matrices for $w = 8$.

We plot the average XOR's per schedule in Figure 11. Focusing first on the heuristics based on CSHR, we see that Uber-CSHR$_2^I$ and Uber-CSHR$_3^I$ generate significantly better schedules than the others. As anticipated, considering intermediates rather than targets yields better schedules, as does considering larger combinations of targets/intermediates. However, as L increases from two to three, the improvement is marginal (the results for $L > 3$ were the same as $L = 3$).

For the heuristics based on X-Sets, we tested all 49 combinations of *threshold* and L from 0 to 6. We plot the best schedules produced, which occurred for each technique when $L \geq 3$ and *threshold* was any value. All of them improve on Subex by 7 to 9 percent in terms of overall XOR's. The best overall technique is the hybrid, *Uber-XSet*.

B. Large Matrices

We performed two sets of tests on larger matrices. The first test evaluates decoding matrices for Blaum-Roth RAID-6 codes [3], which are based on bit matrices that can be quite dense when decoding. The second test evaluates both encoding and decoding with Cauchy Reed-Solomon codes in 16-disk systems where 8, 10 and 12 disks are devoted to data, and the remaining 8, 6, and 4 are devoted to coding.

1) Blaum-Roth RAID-6 Codes: With Blaum-Roth RAID-6 codes, there are k disks of data and $m = 2$ disks of coding. The encoding is defined by a $(2w \times kw)$ bit matrix where $w \geq k$ and $(w + 1)$ must be a prime number. The codes are called "Minimal Density" because the matrices achieve a lower bound on the number of non-zero entries [3]. As such, they encode with very high performance without any scheduling. Blaum-Roth codes are implemented in the **Jerasure** erasure coding library [23] and are employed in the Hadoop DiskReduce project [8].

When two data disks fail, a new $(2w \times kw)$ matrix is generated for decoding. Unlike the encoding matrices, these decoding matrices can be quite dense, but CSHR has been shown to improve the performance of decoding significantly [22]. We use these decoding scenarios (when two data disks fail) as our first scheduling test. We test n-disk systems for $8 \leq n \leq 18$. In each system we test $w = 16$, because that is a legal value for all systems, and it has the additional benefit that it allows strips to be powers of two in size. We did test smaller values of w on Blaum-Roth and the other Minimal Density RAID-6 codes [22], but omit the results since they are very similar to the results for $w = 16$.

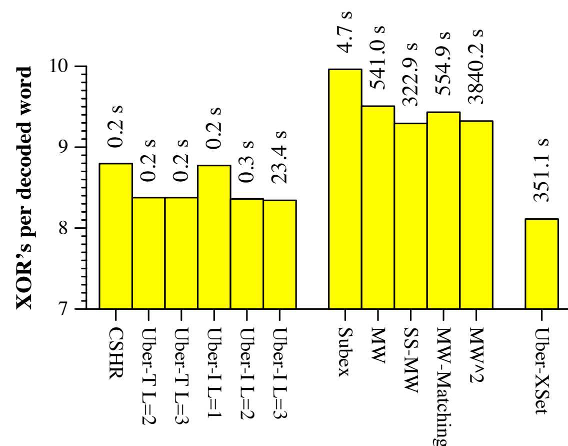

Figure 12. Performance of the heuristics decoding the Blaum-Roth RAID-6 code for $k = 8$, $w = 16$.

We focus first on $k = 8$, which corresponds to a 10-disk system. There are $\binom{8}{2} = 28$ combinations of two data disks that may fail. The decoding bit matrices have 32 rows and 128 columns, and average 1897.21 ones. Without scheduling, they would take 58.29 XOR's per decoded word, which is prohibitively expensive (encoding takes 7.22 XOR's per word). Figure 12 shows the effectiveness and running times of the various heuristics on the 28 combinations of failures. The bars show the XOR's per decoded word generated by schedule, and the numbers above each bar show the average execution time of each heuristic. For each X-Set heuristic, numbers are displayed for the best combination of *threshold* ≤ 6 and $L \leq 3$. The heuristics were implemented in C++ and executed on an Intel Core2 Quad CPU Q9300's running at 2.50 GHz. The implementations are single-threaded, and each test utilizes one core. We don't view the exact performance numbers as particularly important – their relative performance and ballpark numbers instead give a feel for how the various heuristics and parameters perform in general.

The results differ from Figure 11 in several ways. First, as a whole, the Uber-CSHR heuristics outperform the X-Set heuristics, both in terms of running time and schedules produced. While Uber-CSHR$_3^I$ produces slightly better schedules than Uber-CSHR$_2^T$, the latter's running time is so fast that it would preclude the need to store schedules. If running time is not a concern, the best schedules are produced by *Uber-XSet* – 8.11 XOR's per decoded word as opposed to 8.34 for Uber-CSHR$_3^I$. Since *Uber-XSet* is a hybrid of X-Sets and Uber-CSHR, it is not surprising that its performance, unlike the other X-Set heuristics, is on par with the Uber-CSHR heuristics.

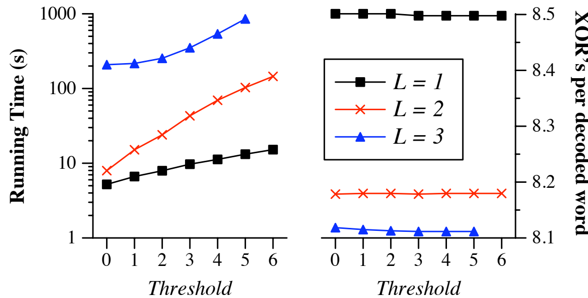

Figure 13. Running times and schedules produced for the *Uber-XSet* heuristic on the 28 decoding matrices for the Blaum-Roth RAID-6 code, $k = 8$, $w = 16$.

Examining **Uber-XSet** further, we plot the running times and XOR's per decoded word for all values of *threshold* and L in Figure 13. We limit the Y-axis to 1000 seconds. Although the XOR values decrease slightly as *threshold* increases, *threshold* has much less impact on the XOR's than L. L is also the major influence in the running time, although the running times do increase as *threshold* increases.

To compare the running times of the various X-Set heuristics, we plot them for $L = 2$ in Figure 14. The right side of Figure 14 shows the number of XOR's of the schedules produced by the best instance of each technique. **MW**, **MW-SS** and **MW-Matching** all have very similar running times. As anticipated **MW**^{2} runs the slowest since it performs one step of lookahead for each potential new element, and then discards the lookahead information after selecting the element. As noted above, *Uber-XSet* produces the best schedules of the three, and its running time is slightly higher than **MW**, **MW-SS** and **MW-Matching**.

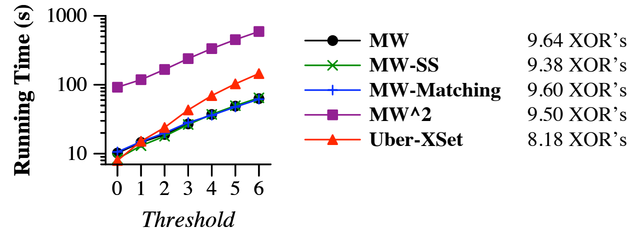

Figure 14. Performance of the X-Set heuristics on the 28 decoding matrices for the Blaum-Roth RAID-6 code, $k = 8$, $w = 16$, $L = 2$.

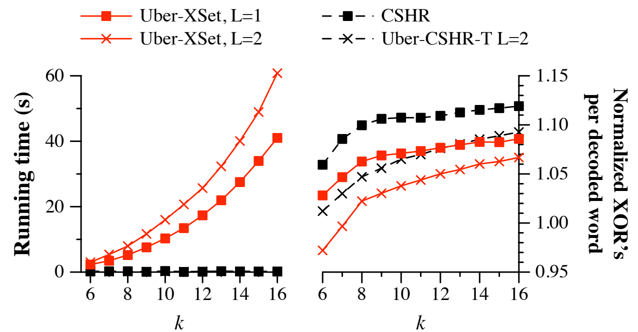

Figure 15. Running times and schedules produced for CSHR, Uber-CSHR$_2^T$ and *Uber-XSet* on all $\binom{k}{2}$ decoding matrices for Blaum-Roth codes, $w = 16$, *threshold* $= 0$.

In Figure 15, we compare the running times and schedules produced by *Uber-XSet*, Uber-CSHR$_2^T$ and CSHR for all RAID-6 installations from $k = 6$ to 16. In all tests, $w = 16$, and for the *Uber-XSet* tests, *threshold* is zero. To compare the differing values of k, we normalize the number of XOR's per decoded word by dividing by k.

As the figure shows, adding the X-Set information to Uber-CSHR imposes quite a penalty on the running time, even with a *threshold* of zero, and $L = 1$. No instance of *Uber-XSet* runs faster than two seconds. CSHR and Uber-CSHR$_2^T$ both run well faster than a second in all cases. *Uber-XSet* produces the best schedules when $L = 2$. When $L = 1$, the schedules produced are no better than Uber-CSHR$_2^T$. Thus, if the application does not store schedules, the latter is a much better scheduling technique to employ. Although not shown on the graph, the Subex heuristic runs with speeds on

par with *Uber-XSet*, $L = 1$, but the schedules produced are much worse, having normalized XOR's between 1.21 and 1.24.

As a bottom line, on these codes, the Uber-CSHR$_2^T$ has a very nice blend of speed and schedules produced. If running time is not a concern then *Uber-XSet* with a high value of L produces the best schedules.

2) Cauchy Reed-Solomon Codes: Our second large matrix test explores CRS codes on 16-disk systems that are fault-tolerant to 4, 6 and 8 failures. Thus, the $[k, m]$ pairs are [12,4], [10,6] and [8,8]. We choose $w = 8$, since that is a natural value in these systems. We focus first on the [10,6] code from Figure 1, whose bit matrix has 1968 non-zero entries.

In Figure 16, we show the best encoding schedules for each heuristic on this matrix. We tested all combinations of *threshold* ≤ 6 and $L \leq 2$. The "Optimal-Sub" heuristic breaks the bit matrix into 60 (8×8) sub-matrices and uses the results of the enumeration in Section V to schedule the sub-matrices.

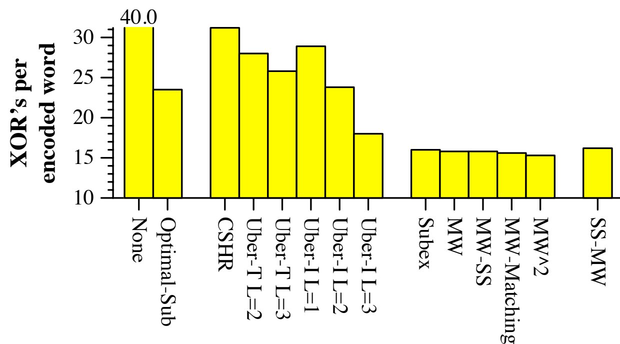

Figure 16. The best schedules for each heuristic on the CRS encoding matrix for $k = 10$, $m = 6$, $w = 8$.

For this matrix, the Uber-CSHR schedules are poorer than the X-Set schedules. Only the Uber-CSHR$_3^I$ schedule comes close to the worst X-Set schedule. Within the X-Set schedules, the best is generated by the **MW**^{2} heuristic when *threshold* ≥ 3 and $L = 2$. This is markedly different from the RAID-6 tests above, where the *Uber-XSet* hybrid performs much better than the others.

We explore the differences between the various X-Set heuristics with respect to decoding in Figure 17. There are $\binom{10}{6} = 210$ combinations of six data disks that can fail when $k = 10$ and $m = 6$. For each combination, we measured the running time and schedule size of all heuristics. For the X-Set heuristics, we used *thresh* = 2 and $L \in \{1, 2\}$. In the figure, we plot the XOR's per decoded word of the schedule and the running time.

As in Figure 16, **MW**^{2} generates the best schedule. However, the average running times are very large. **MW-Matching** generates the next best schedules and takes far less time to run, although six seconds may still be too large

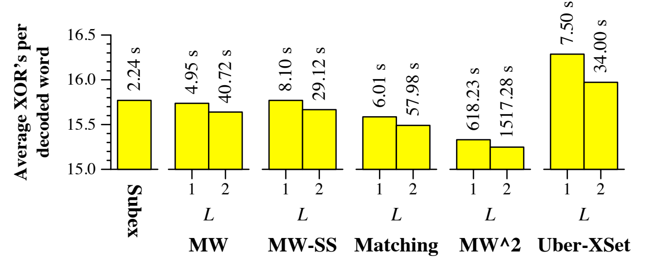

Figure 17. Average running time and decoding schedules for the $\binom{10}{6}$ CRS decoding matrices for six failures, $k = 10$, $m = 6$, $w = 8$, *threshhold* = 2.

to employ without precomputation. Of the six techniques, *Uber-XSet* performs the worst. Uber-CSHR$_3^T$, which is not pictured on the graph, produces the best schedules among the heuristics which run under a second; however, the schedules it produces are much worse, averaging 25.9 XOR's per decoded word.

Finally, Figure 18 shows the normalized performance of encoding and decoding with CRS codes on 16-disk disk systems where $k \in \{8, 10, 12\}$ and $w = 8$. The figures evaluate no scheduling plus five heuristics: CSHR, Uber-CSHR$_3^T$, Subex, **MW**^{2} and *Uber-XSet*. They show the performance of encoding, plus decoding from all combinations of four and six data disk failures. In both figures we show the "original" Cauchy matrices as defined by Blomer [4] and the "good" Cauchy matrices generated by **Jerasure** [23], which are sparser than the original matrices. The X-Set heuristics generate much better schedules than the others, but run slower (running times are on par with those in Figure 17). Both graphs show the effectiveness of scheduling — the reduction of XOR operations is drastic in all cases, improving the performance over no scheduling by over 60 percent.

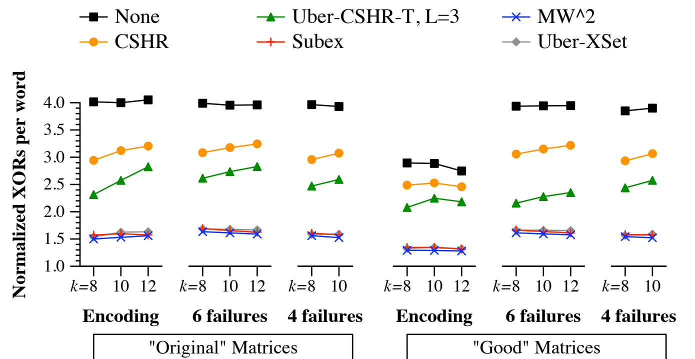

Figure 18. Performance of heuristics with CRS coding, $m = (16 - k)$, $w = 8$.

With encoding, the matrix impacts performance regardless of the scheduling algorithm – The "good" matrices require much fewer XOR operations than the "original" matrices.

This discrepancy goes away upon decoding — sparse encoding matrices do not lead to sparse decoding matrices, and the performance of the X-Set heuristics is independent of the encoding matrices.

One conclusion that one may draw from Figures 16-18 is that for CRS coding, none of the new heuristics improve significantly over the original Subex heuristic. The schedules produced perform nearly as well as the best X-Set heuristics, and the running times are faster, although probably too slow to allow an application to not store schedules. Unfortunately, the Blaum-Roth tests show that Subex does not produce good enough schedules to qualify it as a universally applicable heuristic. The hybrid, *Uber-XSet*, although slightly worse than Subex in the CRS tests, shows the best performance on all of the tests in this paper.

VIII. Implementation

Our implementations of all the heuristics are available as open source C++ programs released under the New BSD License [20]. The Uber-CSHR program is small (under 500 lines) and consumes a small amount of memory, regardless of the value of L. The X-Set program is larger (roughly 1800 lines), much of which is composed of Edmonds' matching algorithm. As *thresh* and L grow, the X-Set program consumes a significant amount of memory, which becomes the limiting factor in running the heuristics, rather than time. Both implementations are single-threaded and run without any external libraries.

It is our intent for storage practitioners to leverage these programs as they implement their erasure codes, and for researchers to use them to spur further work. The implementation is notable also as the first implementation of Subex. CSHR was implemented as part of the **Jerasure** erasure coding library [23].

IX. Conclusion

Erasure codes that are based on bit matrices are already in heavy use, and their use will increase in the future. We have described heuristics whose intent is to schedule the XOR operations of these erasure codes so that the number of operations is minimized. These heuristics expand upon on previous work in the area [12], [13]. We have implemented the heuristics and provide them to the community as open source C++ programs [20].

We have evaluated the heuristics in relation to optimal schedules that were generated via a pruned enumeration. The *Uber-XSet* heuristic comes closest to optimal on the 255 (8×8) Cauchy bit matrices, requiring 5.6 percent more XOR's than the optimal schedules. The improvement over previous heuristics is significant. CSHR requires 30 percent more XOR's and Subex requires 16 percent. No scheduling results in 89 percent more XOR's.

We performed a second evaluation of the heuristics on two sets of encoding and decoding scenarios that may be used in fault-tolerant storage installations: RAID-6 based on Blaum-Roth erasure codes, and Cauchy Reed-Solomon codes to tolerate larger numbers of failures. For the RAID-6 codes, the performance of decoding requires the use of scheduling, and two new heuristics improve the state of the art. Uber-CSHR$_2^T$ yields improved schedules and runs very quickly. *Uber-XSet* yields the best schedules but takes longer to run.

In the Cauchy Reed-Solomon examples, the X-Set heuristics all yield schedules that are a significant improvement over what is currently implemented (namely, no scheduling and CSHR). The **MW**^{2} heuristic generates the best schedules, albeit very slowly. The Subex heuristic generates schedules that are nearly as good, but much more quickly. When all tests are considered, the best heuristic is the *Uber-XSet* heuristic, which generates the best schedules in the enumeration and the Blaum-Roth codes, and very good schedules for Cauchy Reed-Solomon coding.

One limitation of this work is that it attacks an exponential search space with heuristics, but doesn't glean fundamental insights that may lead to better solutions to the problem. For that reason, the result is somewhat more pragmatic than theoretical. It is our hope that this work helps spur more theoretical work in the field. For example, while there are no known lower bounds for the number of non-zero entries in certain generator matrices [3], there are not lower bounds on the resulting number XOR operations.

Finally, the we readily acknowledge that reducing XOR's is not the only way to improve the performance of an erasure code. Careful attention to cache behavior can improve performance by over 30 percent without changing the number of XOR's [18]. Hand-tuning the code with respect to multiple cores and SSE extensions will also yield significant performance gains. Other code properties, like the amount of data required for recovery, may limit performance more than the CPU overhead [9], [25]. We look forward to addressing these challenges in future work.

X. Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0615221 and CSR-1016636, and a research award from Google.

References

[1] Bairavasundaram, L. N., Goodson, G., Schroeder, B., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. An analysis of data corruption in the storage stack. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008).

[2] Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing 44*, 2 (February 1995), 192– 202.

[3] Blaum, M., and Roth, R. M. On lowest density MDS codes. *IEEE Transactions on Information Theory 45*, 1 (January 1999), 46–59.

[4] BLOMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.

[5] BOWERS, K., JUELS, A., AND OPREA, A. Hail: A high-availability and integrity layer for cloud storage. In *16th ACM Conference on Computer and Communications Security* (2009).

[6] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row diagonal parity for double disk failure correction. In *3rd Usenix Conference on File and Storage Technologies* (San Francisco, CA, March 2004).

[7] EDMONDS, J. Paths, trees and flowers. *Canadian Journal of Mathematics 17* (1965), 449–467.

[8] FAN, B., TANTISIRIROJ, W., XIAO, L., AND GIBSON, G. DiskReduce: RAID for data-intensive scalable computing. In *4th Petascale Data Storage Workshop, Supercomputing '09* (Portland, OR, November 2009), ACM.

[9] GREENAN, K. M., LI, X., AND WYLIE, J. J. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery and tradeoffs. In *26th IEEE Symposium on Massive Storage Systems and Technologies (MSST2010)* (Nevada, May 2010).

[10] HAFNER, J. L. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 211–224.

[11] HAFNER, J. L., DEENADHAYALAN, V., BELLUOMINI, W., AND RAO, K. Undetected disk errors in RAID arrays. *IBM Journal of Research & Development 52*, 4/5 (July/September 2008), 418–425.

[12] HAFNER, J. L., DEENADHAYALAN, V., RAO, K. K., AND TOMLIN, A. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 183–196.

[13] HUANG, C., LI, J., AND CHEN, M. On optimizing XOR-based codes for fault-tolerant storage applications. In *ITW'07, Information Theory Workshop* (Tahoe City, CA, September 2007), IEEE, pp. 218–223.

[14] HUANG, C., AND XU, L. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers 57*, 7 (July 2008), 889–901.

[15] JIN, C., JIANG, H., FENG, D., AND TIAN, L. P-Code: A new RAID-6 code with optimal properties. In *23rd International Conference on Supercomputing* (New York, June 2009).

[16] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS* (Cambridge, MA, November 2000), ACM, pp. 190–201.

[17] LIN, S., WANG, G., STONES, D. S., LIU, X., AND LIU, J. T-Code: 3-erasure longest lowest-density MDS codes. *IEEE Journal on Selected Areas in Communications 28*, 2 (February 2010).

[18] LUO, J., XU, L., AND PLANK, J. S. An efficient XOR-Scheduling algorithm for erasure codes encoding. In *DSN-2009: The International Conference on Dependable Systems and Networks* (Lisbon, Portugal, June 2009), IEEE.

[19] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[20] PLANK, J. S. Uber-CSHR and X-Sets: C++ programs for optimizing matrix-based erasure codes for fault-tolerant storage systems. Tech. Rep. CS-10-665, University of Tennessee, December 2010.

[21] PLANK, J. S., AND DING, Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience 35*, 2 (February 2005), 189–194.

[22] PLANK, J. S., LUO, J., SCHUMAN, C. D., XU, L., AND WILCOX-O'HEARN, Z. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies* (February 2009), pp. 253–265.

[23] PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.

[24] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8* (1960), 300–304.

[25] XIANG, L., XU, Y., LUI, J. C. S., AND CHANG, Q. Optimal recovery of single disk failure in RDP code storage systems. In *ACM SIGMETRICS* (June 2010).

[26] XU, G., WANG, G., ZHANG, H., AND LI, J. Redundant data composition of peers in P2P streaming systems using cauchy reed-solomon codes. In *Sixth International Conference on Fuzzy Systems and Knowledge Discovery* (Tianjin, China, 2009), IEEE.

[27] XU, L., AND BRUCK, J. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory 45*, 1 (January 1999), 272–276.