On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications

Cheng Huang, Jin Li, and Minghua Chen
Microsoft Research, Redmond, WA 98052

***Abstract*— For fault-tolerant storage applications, computation complexity is the key concern in choosing XOR-based codes. We observe that there is great benefit in computing common operations first (COF). Based on the COF rule, we describe a generic problem of optimizing XOR-based codes and make a conjecture about its NP-completeness. Two effective greedy algorithms are proposed. Against long odds, we show that XOR-based Reed-Solomon codes with such optimization can in fact be as efficient and sometimes even more efficient than the best known specifically designed XOR-based codes.**

I. Introduction

Erasure correcting codes are often adopted by storage applications to provide fault tolerance [5]. For such applications, encoding and decoding complexity is the key concern in determining which codes to use. XOR-based codes use pure XOR operation during coding computation, which makes implementation most efficient in both hardware and software. Hence, such codes are highly desirable.

XOR-based codes can be implemented by transforming from any existing codes, which originally could be defined in finite fields [3]. Among numerous available codes, Reed-Solomon codes [14], as probably the most widely used codes, are both flexible in coding parameters and capable of recovering the maximum number of failures (the MDS property [11]). Hence, they appear as natural choices. However, it has long been assumed that XOR-based Reed-Solomon codes are inefficient (see the seminal paper [1] and all its followers) and thus inappropriate for storage applications. This might be one of the most prominent reasons that motivated decades of efforts in designing specific XOR-based codes.

However, the biggest problem of specifically designed codes is that they are in general not flexible. While codes providing 2 or 3-fault-tolerance (recoverable from 2 or 3 storage node failures) are well studied [1], [2], [6], [7], [9]. Efficient codes offering more redundancy still appear out of reach, even though there *do* exist a few schemes [2], [8]. In this paper, we reexamine XOR-based Reed-Solomon codes and argue for their suitableness in storage applications. Since the complexity of XOR-based codes is solely determined by the total number of XOR operations in encoding or decoding, we make a simple yet key observation that common XOR operations should be computed first (the COF rule). Based on the COF rule, we can optimize arbitrary XOR-based codes (not just Reed-Solomon codes). We describe the optimization problem as finding a computation path, which computes all required outputs and minimizes the total number of XOR operations at the same time. We relate the problem of optimizing XOR-based codes (*OXC* in short) to a known NP-complete problem and make a *conjecture* that the current problem is also NP-complete.

Two greedy approaches are proposed to find approximate solutions to the OXC problem. When optimization is applied to XOR-based Reed-Solomon codes, we show that these codes can in fact be as efficient and sometimes even more efficient than the best known specifically designed XOR-based codes, which is in contrary to long time odds. In particular, in 2-fault-tolerant case, XOR-based Reed-Solomon codes are more efficient in encoding than EVENODD codes [1] and as efficient as the RDP scheme [6]. They are less efficient in decoding though. In 3-fault-tolerant case, XOR-based Reed-Solomon codes are more efficient in encoding than both the generalized EVENODD codes [2] and the STAR scheme [9]. They are also more efficient in decoding in most cases. As the adoption of erasure correcting codes in large scale production storage systems is looming on the horizon, it is conceivable that redundancy beyond 3 will become necessary and XOR-based Reed-Solomon codes with optimization are likely to live up to such requirements.

The rest of the paper is organized as follows. Section II revisits EVENODD codes as an example of specifically designed XOR-based codes. Section III briefly describes the transformation of Reed-Solomon codes into XOR-based codes. Section IV presents the OXC problem, the complexity conjecture and two greedy approaches. The performance of OXC is evaluated in Section V and we conclude in Section VI.

II. Revisiting EVENODD codes

A. EVENODD: an example

EVENODD codes [1] are probably the most widely referred XOR-based codes in fault-tolerant storage applications. Many other schemes adopt a similar concept, where data blocks are arranged in a two dimensional array and XOR is the only required operation. Schemes as such are often referred as *array codes*. Low complexity is the key advantage of array codes, which is especially desirable for storage applications. Below we give a simple example of EVENODD codes.

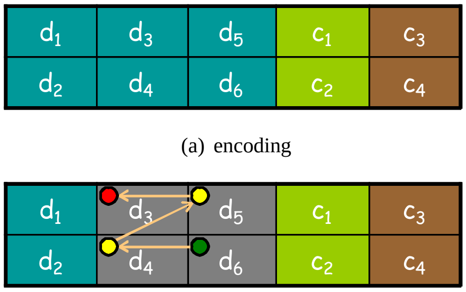
1) EVENODD encoding: We examine a $(5, 3)$ EVENODD code. There are 3 data blocks ($k = 3$) and 2 redundant blocks ($r = 2$). An EVENODD code is in the form of a $(p-1) \times (p+2)$ two dimensional array, where p is a prime number. Hence, each block is segmented into $(p-1)$ *cells*. Figure 1 shows this particular EVENODD code, where $p = 3$ and each block (corresponding to one column in the Figure) is segmented into 2 cells. The encoding is straightforward. The first redundant block is simply the XOR of all the data blocks. In terms of cells, they can be represented as (use + as a simple notation for XOR)

$$c_1 = d_1 + d_3 + d_5,$$
$$c_2 = d_2 + d_4 + d_6,$$

which can be regarded as computing *horizontal* parities. The second redundant block can be computed as

$$S = d_4 + d_5,$$
$$c_3 = d_1 + d_6 + S,$$
$$c_4 = d_2 + d_3 + S,$$

which can be regarded as computing diagonal parities (S is called *adjustor*). It is easy to count that the total number of XORs is 9.



(a) encoding

(b) decoding

Fig. 1. An EVENODD code example.

2) EVENODD decoding: EVENODD codes guarantee recoverability when there are no more than two block failures (i.e., two columns completely wiped out). For instance, we examine a particular failure pattern, when the second and the third data blocks are unavailable. The decoding turns out to be straightforward as well. Using all the remaining parity blocks, the adjustor can first be computed as

$$S = c_1 + c_2 + c_3 + c_4.$$

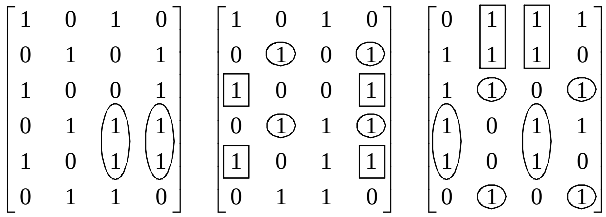
Once S is known, d_6 can be computed as $d_6 = c_3 + d_1 + S$. Then, d_4 can be computed as $d_4 = c_2 + d_2 + d_6$. Next, d_5 can be computed as $d_5 = d_4 + S$. And finally, $d_3 = d_1 + d_5 + c_1$. The decoding process is completed and all failed blocks are recovered, as shown in Figure 1(b). The total number of XORs is 10.

B. EVENODD: a matrix perspective

The encoding and decoding of linear block codes can be represented in a matrix form. Here, we use the same EVENODD code example to illustrate.

1) encoding with COF: Denote data cells as $\mathbf{D} = [d_1\ d_2\ d_3\ d_4\ d_5\ d_6]$ and parity cells as $\mathbf{C} = [c_1\ c_2\ c_3\ c_4]$. Then, the encoding can be represented as $\mathbf{C} = \mathbf{D} \times \mathbf{M}_e$, where the *encoding matrix* $\mathbf{M}_e$ is in the following form:

$$\mathbf{M}_e = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \tag{1}$$



(a) EO encoding (b) encoding with COF (c) decoding with COF

Fig. 2. A matrix perspective of EVENODD code (EO for EVENODD).

Note that $\mathbf{M}_e$ represents a portion of the code's generator matrix. For systematic codes, it is convenient to ignore the rest systematic part.

Given the encoding matrix, a *naive* approach to compute the redundant blocks is to XOR all data cells whenever the encoding matrix has non-zero entries. For example, $c_1 = d_1 + d_3 + d_5$, $c_3 = d_1 + d_4 + d_5 + d_6$, and so on. In this way, counting the total number of non-zeros entries yields the encoding complexity. Hence, we might conclude that 10 XORs are required (note that three 1's in one column counts for 2 XORs). However, if we are slightly more careful, we will observe that some XORs are computed more than once. Indeed, if the EVENODD encoding is mapped onto the matrix representation, it is equivalent to computing $d_4 + d_5$ only once (the calculation of the adjustor), which saves 1 XOR and exactly accounts for the difference between the matrix-based naive approach and the original EVENODD encoding. Figure 2(a) illustrates this.

Now, an interesting question to ask is: can we find more *shared* XORs, which can be computed once and in turn further reduce the total number of operations? Indeed, we observe that $d_2 + d_4$ (denoted as $d_{2,4}$) and $d_3 + d_5$ (denoted as $d_{3,5}$) are shared XORs, as shown in Figure 2(b). If we adopt a simple rule to compute such *common operations first* (COF), $d_{2,4}$ and $d_{3,5}$ will be computed. Then, $c_1 = d_1 + d_{3,5}$, $c_2 = d_{2,4} + d_6$, $c_3 = d_1 + d_4 + d_5 + d_6$ (as usual), and $c_4 = d_{2,4} + d_{3,5}$. The total number of XORs is 8, less than the original EVENODD encoding.

2) decoding with COF: We consider the same failure pattern, where the second and third data blocks are unavailable (i.e., cells d_3, d_4, d_5 and d_6 are erasures). It is straightforward to derive decoding equations from the encoding matrix $\mathbf{M}_e$ (essentially performing matrix inversion) and obtain $\mathbf{D}' = \mathbf{C}' \times \mathbf{M}_d$, where $\mathbf{D}' = [d_3\ d_4\ d_5\ d_6]$, $\mathbf{C}' = [d_1\ d_2\ c_1\ c_2\ c_3\ c_4]$, and the decoding matrix $\mathbf{M}_d$ is

$$\mathbf{M}_d = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}. \tag{2}$$

Again, the naive approach requires 12 XORs. But, applying the COF rule, and compute shared XORs first (e.g., $d_1 + d_2$, $c_1 + c_4$ and $c_2 + c_3$ in this case, also shown in Figure 2(c)), the total number of required XORs is 9. This is also less than the original EVENODD decoding (10 XORs).

III. A 2-fault-tolerant Reed-Solomon code

In this section, we construct a $(5, 3)$ Reed-Solomon code and apply the COF rule to both encoding and decoding.

A. Primer on isomorphism

Reed-Solomon codes are constructed in finite fields, where the addition operation is simply XOR, but the multiplication operation is handled specially. Elements of finite fields can be represented using polynomials, which help to understand the addition and multiplication operations.

Consider a simple finite field with only 4 elements, which can be constructed taking polynomials modulo $x^2 + x + 1$. Since addition in this finite field is XOR, $+$ and $-$ are the same, hence, we can compute $x^2 = x+1$ (modulo x^2+x+1) and $x^3 = xx^2 = x(x + 1) = x^2 + x = 1$ (modulo $x^2 + x + 1$). It is easy to imagine all polynomials can be represented using 4 basic *elements*, being 0, 1, x and $x + 1$. Given these elements, the addition and multiplication between any two pair can be easily computed and stored in look-up tables. With the addition and multiplication tables, Reed-Solomon codes can be implemented using table-lookups, which is how they are often realized. For rigorous representations, please see [3].

From the polynomial perspective, however, there is another way to represent the multiplication operation. Assuming we would like to compute $x(x+1)$. Instead of directly computing $x(x+1) = x^2+x = 1$ (modulo x^2+x+1), we can consider a more general case by writing the term into $(ax+b)(x+1)$. Of course, $a = 1$ and $b = 0$ here. Hence, $(ax + b)(x + 1) = a(x^2 + x) + b(x + 1) = a + b(x + 1)$ (modulo $x^2 + x + 1$). Therefore, $(ax + b)(x + 1)$ can be represented as

$$(ax+b)(x+1) = \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}. \tag{3}$$

Let $a = 1$ and $b = 0$, we can get $x(x + 1) = 1$. Another example sets $a = 1$ and $b = 1$, and we can get $(x + 1)(x + 1) = x$. Both can be easily verified using direct polynomial multiplications. Hence, the multiplication in finite fields can be transformed into pure XOR operations. This mechanism is called *isomorphism*. The significance is that a and b *no longer* need to be a simply bit. They can be a byte, a word or 64 bits, even 128 bits (with SSE/SSE2), the maximal length a single XOR instruction can operate on. Through isomorphism, arbitrary codes defined on finite fields (not just Reed-Solomon codes) can be implemented using pure XOR operations. For more details, please see [4], [13].

B. A 2-fault-tolerant Reed-Solomon code

[11] describes a convenient way to construct Reed-Solomon codes when the redundant blocks are no more than 3. To offer 2-fault-tolerance for 3 data blocks, we can use the above finite field of size 4 and the following encoding matrix:

$$\begin{bmatrix} c_a & c_b \end{bmatrix} = \begin{bmatrix} d_a & d_b & d_c \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & x \\ 1 & x+1 \end{bmatrix}, \tag{4}$$

where c_a, c_b are redundant blocks and d_a, d_b, d_c data blocks, representing elements in the finite field. Let $c_a = c_1x + c_2$, $d_a = d_1x + d_2$, etc. Now, c_1, c_2, d_1 and d_2 are elements in binary, so we get the following through isomorphism:

$$\begin{bmatrix} c_1 & c_2 & c_3 & c_4 \end{bmatrix} = \begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}. \tag{5}$$

Applying the COF rule, we observe that $d_1 + d_3$ and $d_4 + d_6$ are shared XORs and should be computed first. In the end, the total number of XORs is 8, less than the EVENODD encoding. Similarly, it is straightforward to verify that decoding the second and third data blocks requires 9 XORs, also less than the EVENODD decoding. Careful readers might observe that, in this particular example, the encoding and decoding matrices of the Reed-Solomon code happen to be the same as the EVENODD code. However, we note that this is *not* true in general.

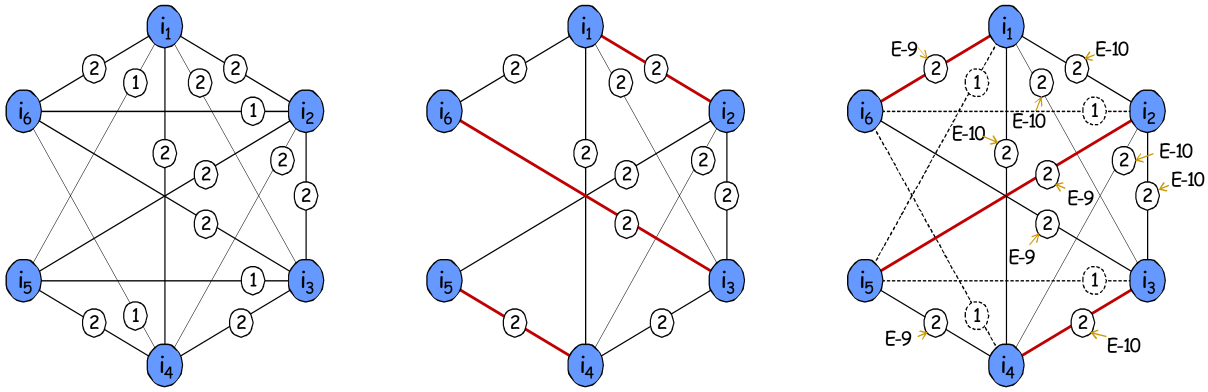
IV. Optimizing XOR-base codes

Conceivably, the COF rule is applicable to arbitrary XOR-based codes, no matter whether they are specially designed XOR-based codes, or simply isomorphism of regular Reed-Solomon or other types of codes. However, when the encoding matrix or decoding matrix is large, it becomes nontrivial to determine which shared XORs should be computed first and used as intermediate results for other. In this section, we define the problem of optimizing XOR-based codes formally (*OXC* in short), present a conjecture of the NP-completeness of the problem, and propose two effective greedy algorithms.

A. Problem formulation

The OXC problem is stated as follows. Given a set of inputs (denoted as i_1, i_2, $\cdots$, $i_{|I|}$) and a coding matrix $\mathbf{M}$ (either encoding or decoding), a set of outputs (denoted as o_1, o_2, $\cdots$, $o_{|O|}$) are computed from the inputs and the coding matrix, where XOR is the only computation operation. Define a *computation path* as the order of XOR operations involving the inputs and/or the intermediate results from previous XORs. A computation path is *valid*, if it yields all required outputs after all XORs along the path are computed. The *length* of a computation path is simply the total number of XORs contained in the path. *Given the inputs and the coding matrix, the OXC problem is to find a valid computation path with the minimum length.*

Next, we relate the OXC problem to a known NP-complete problem and make a conjecture about its complexity. As illustrated by various shapes (rectangle, circle, and eclipse) in Figure 2(c), we can use *covers* to represent shared XORs. Here, we define a general conceptual *rectangle cover* (denoted as $\mathbf{RC}$) for shared XORs. A rectangle cover may span multiple rows (*rectangle height*, $h_{\mathbf{RC}}$) and columns (*rectangle width*, $w_{\mathbf{RC}}$) of the coding matrix. It does *not* need to be contiguous



(a) graph with edge counters

(b) approach I: cardinality matching

(c) approach II: weighted matching

Fig. 3. Illustration of greedy approaches.

in either rows or columns. Intuitively, a rectangle cover has to be *rectangle*, so it contains the same number of entries among all rows (or columns). A rectangle cover can only contain 1's and *no* 0's at all. All columns of a rectangle cover share same XORs. Hence, computing any single column is sufficient and the number of XORs required is $(h_{\mathbf{RC}} - 1)$. Now, we define the *cost* of a rectangle cover (denoted as $c_{\mathbf{RC}}$) as $c_{\mathbf{RC}} = (h_{\mathbf{RC}} - 1) + w_{\mathbf{RC}}$, where $(h_{\mathbf{RC}} - 1)$ accounts for the number XORs to be computed within the rectangle, and $w_{\mathbf{RC}}$ one potential XOR with outside inputs per column of the rectangle. Finally, we define a set of *none-overlapping complete rectangle covers* (denoted as $\mathbf{RC}_i$'s), which do *not* overlap with each other and cover all 1's of the coding matrix. We have the following corollary.

Corollary 1: A computation path is equivalent to a set of none-overlapping complete rectangle covers. Moreover, the length of the computation path equals to the total cost of all rectangle covers minus the number of outputs, i.e., $\sum c_{\mathbf{RC_i}} - |O|$. (We leave the proof to interested readers and simply mention that minus $|O|$ is because each column overcounts exactly by 1.)

To this end, the OXC problem is equivalent to finding a set of none-overlapping complete rectangle covers of the coding matrix with the minimum total cost ($|O|$ is constant and thus can be ignored). To get rid of the none-overlapping requirement, we can apply a simple technique and modify the cost function of rectangle covers. For each entry in a rectangle cover, we add a large constant L to its cost. Then, for none-overlapping rectangle covers, the number of times that L is counted in total cost equals to the number of 1's in the coding matrix. On the other hand, once two rectangle covers overlap, L will be counted more times. Hence, as long as L is large enough (e.g., more than the total entries in the coding matrix, $|I| \times |O|$), overlapping rectangle covers will *never* yield the minimum cost. With this cost function modification, we only need to find a set of complete rectangle covers with minimum total cost. In [15], the *minimum weighted rectangle covering* (MWRC) problems is shown to be NP-complete. Note that the general MWRC problem includes arbitrary cost functions for each rectangle. It is *not* clear that the problem is still NP-complete with the current cost function. Hence, we only make a conjecture here that the OXC problem is also NP-complete. In the rest of this section, we describe two greedy algorithms to derive approximate solutions.

B. Greedy approach I: cardinality matching

Let's use the coding matrix in Figure 2(c) to describe the algorithm. The inputs are i_1, i_2, i_3, i_4, i_5 and i_6. The outputs are o_1, o_2, o_3, and o_4. Based on the coding matrix, in order to compute cell o_1, we need to XOR 4 inputs, i.e., $o_1 = i_2 + i_3 + i_4 + i_5$. There are many ways to compute o_1. For instance, we can first compute $i_2 + i_3$, $i_4 + i_5$ and then sum them up. Or, we can compute $i_2 + i_3$, and then add i_4 and i_5 one by one. To list all possibilities, we draw all inputs as *nodes* in a graph and connect two nodes with an edge whenever there is a potential XOR computation. Clearly, between any two-node pair among i_2, i_3, i_4 and i_5, there exists an edge. Hence, the graph contains a 4-clique. This is for one output. Similarly, for other outputs, the graph will contain different cliques. Putting all cliques into the same graph, some edges belong only to one clique, while others might belong to multiple. We keep a counter on each edge. Intuitively, the counter represents how many times one particular XOR is shared during the computation of different outputs. To reduce the total number of operations, it's natural to compute the mostly shared XORs first. In the graph notation, it is to compute the edges with the highest counter value.

For instance, Figure 3(a) shows the complete graph and edge counters corresponding to the coding matrix. The highest counter is 2. To compute such edges first, we remove all edges with less counter values and obtain the subgraph in Figure 3(b). The next step is to find the maximum number of *disjoint* edges (no two edges share the same node) and compute them first. The intuition is that disjoint edges represent XORs on completely different nodes and computing them at the same time do *not* affect each other. Computing the maximum number of disjoint edges can get the maximum reduction of XORs at once. It turns out that finding the maximum number of disjoint edges is a well-studied graph theory problem, called *maximum cardinality matching*. A matching is a set of edges in a graph, where there are no two edges share the same node. A maximum matching is a matching with the maximum number of edges. Given a graph, there are many polynomial time algorithms to find a maximum matching. When a graph contains multiple maximal matchings, our algorithm proceeds with any of them. The XORs corresponding to the matching are computed first. Given the matching shown in Figure 3(b), we will first compute $i_1 + i_2$, $i_3 + i_6$ and $i_4 + i_5$.

Once these XORs are computed, we examine the remaining XORs. We can still use a matrix to represent all the XORs. The matrix will be modified from the original coding matrix, where entries corresponding to XORs, which have already been computed, need to be removed. Also, we need to add new entries for the intermediate results from the above computations. To this end, we add three new imaginary inputs $i_{1,2}$, $i_{3,6}$ and $i_{4,5}$ to represent the intermediate results. Now the decoding matrix becomes

$$\mathbf{M}' = \left[\begin{array}{cccc} 0 & 1 \to 0 & 1 \to 0 & 1 \\ 1 & 1 \to 0 & 1 \to 0 & 0 \\ 1 & 1 \to 0 & 0 & 1 \to 0 \\ 1 \to 0 & 0 & 1 \to 0 & 1 \\ 1 \to 0 & 0 & 1 \to 0 & 0 \\ 0 & 1 \to 0 & 0 & 1 \to 0 \\ \hline\hline 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{array}\right], \qquad (6)$$

where the three bottom rows are newly added. Taking the second column as an example, it has 2 non-zero entries. This corresponds to $o_2 = i_{1,2} + i_{3,6}$ and is indeed the same as the original computation of $o_2 = i_1 + i_2 + i_3 + i_6$.[1]

Given the new coding matrix, it's possible to find more shared XORs operations and again compute them only once. Apparently, we can apply the same procedure to find the maximum number of shard XORs. Indeed, the procedure is repeated until there are no more shared XORs. It is easy to show that the algorithm terminates after finite rounds, and within each round, both preparing the graph and finding maximum matching take polynomial time. Hence, the overall complexity is still in polynomial time. We will elaborate more when discussing practicality issues later.

C. Greedy approach II: weighted matching

As mentioned already, there might exist multiple maximum matchings in a graph. For instance, in Figure 2(b), matching i_1 with i_6, i_2 with i_5 and i_3 with i_4 is also a maximum matching. In the above greedy approach, we proceed with any maximum matching. In this part, we consider a variation of the above approach. We still like to find a maximum matching (i.e., the maximum number of disjoint pairs), but we like the matching to cover as fewer *dense* nodes as possible. The density of a node is defined by its degree. The intuition is that if all nodes covered by the maximum matching are removed, as well as all the edges connected to these nodes, the remaining graph should be as dense as possible such that it's likely to contains more matchings for the next round.

Now we describe the second approach, which differs from the first greedy approach in the way of finding a maximum matching. Starting from the original graph with counter values, we assign weights to all edges. For an edge with the maximum counter value, its weight is set to be a large constant (say E) minus the degrees of its both end nodes. For instance, the edge between i_1 and i_6 has weight $E-9$ (rest shown in Figure 3(c)). For an edge with a smaller counter value, it's excluded. Once we go through all edges and obtain a subgraph, we find a *maximum weighted matching*, which is known to be solvable in polynomial time. Note that a vanilla maximum weighted matching does *not* guarantee finding the maximum number of matching pairs. Hence, the constant E is added to compensate that. As long as we make E large enough (e.g., the sum of all nodes' degrees), a maximum weighted matching will always contain the maximum number of matching pairs (i.e., also a maximum cardinality matching). Maximum weighted matching has comparable complexity as maximum cardinality matching, so the complexity of the second greedy approach is also comparable to the first approach. Finally, we note that our empirical experience shows that neither approach is superior, so we simply run both approaches and take a better result in practice.

V. Performance evaluation

In this section, we apply the both greedy approaches to optimize XOR-based Reed-Solomon codes. We compare encoding and decoding complexities to the naive approach, as well as to the best known specifically designed XOR-based codes.

A. Limited exploration of available Reed-Solomon codes

We shows 2-fault-tolerant and 3-fault-tolerant cases, which are the focus of a large number of specifically designed XOR-based codes. Even with limited redundancy, there are still numerous ways to construct a Reed-Solomon code. Here, we use the Reed-Solomon codes presented in [11] (Ch. 11. Theorem 11). For a given finite field GF($Q = 2^q$), the parity check matrix is given as

$$\mathbf{H} = \begin{bmatrix} 1 & \cdots & 1 & 1 & 0 & 0 \\ \alpha_1 & \cdots & \alpha_{Q-1} & 0 & 1 & 0 \\ \alpha_1^2 & \cdots & \alpha_{Q-1}^2 & 0 & 0 & 1 \end{bmatrix}. \qquad (7)$$

To construct a (n, k) systematic Reed-Solomon code ($r = n - k \leq 3$), we can choose any r out of 3 rows and k out of the first $(Q - 1)$ columns from $\mathbf{H}$. It's easy to verify that this gives us a $r \times n$ parity check matrix, which corresponds to a (n, k) systematic Reed-Solomon code. Still, the number of available codes (i.e., $\binom{Q-1}{k}$ column combinations) are quite large. Hence, we further limit our exploration to include only columns that are contiguous in $\mathbf{H}$ (cyclic is fine). In short, given (n, k), we only consider $(Q - 1)$ codes when $r = 3$ and $3(Q - 1)$ codes when $r = 2$ (3 times more due to row combinations). For each (n, k), we choose a Reed-Solomon code that incurs the minimum number of XORs in *encoding*

[1] The coding matrix is modified in a way similar to the *rectangle replacement* method in [16], which, however, only deals with one pair of match at a time and hence is different from our approach.

(after optimization) as the desirable code and compute its corresponding average decoding complexity over all failure patterns.

B. Comparison

	encoding complexity				decoding complexity			
k	EVENODD	RS (naive)	RS (**OXC**)	RDP	EVENODD	RS (naive)	RS (**OXC**)	RDP
3	4.5	5	4	4	5	6.67	4.83	4
5	8.75	11	8	8	9.5	16.33	9.97	8
7	12.83	17	12	12	13.67	23.43	14.25	12
11	20.9	29.75	20	20	21.8	46.66	24.67	20
13	24.92	36.75	24	24	25.83	55.43	29	24

(a) 2-fault-tolerant case

	encoding complexity			decoding complexity			
k	gen. EO / STAR	RS (naive)	RS (**OXC**)	gen. EO	STAR	RS (naive)	RS (**OXC**)
5	13.5	18	11.33	28.8	13.6	22.27	12.57
7	19.67	28	17.33	36	21.06	32.29	17.58
11	31.8	50.75	27	49.1	34.2	67.08	32.02
13	37.83	63	32.25	55.44	41.04	79.75	36.96
17	49.86	95	44.6	67.84	54.4	128.78	55.17
19	55.89	107.8	50.4	73.98	61.74	144.12	60.50

(b) 3-fault-tolerant case

Fig. 4. Comparison with best known specifically designed XOR-based codes.

In 2-fault-tolerant case, we compare XOR-based Reed-Solomon codes to the EVENODD codes [1] and the RDP scheme [6]. From Figure 4(a), we observe that, with optimization, the encoding of Reed-Solomon codes can be as efficient as EVENODD/RDP. The decoding of Reed-Solomon codes are *less* efficient though. In 3-fault-tolerant case, we compare with the generalized EVENODD codes [2] (as *gen. EO*) and the STAR scheme [9]. The encoding complexity of both specifically designed schemes are the same. The decoding of the STAR scheme is more efficient than the generalized EVENODD codes. We observe that the Reed-Solomon codes appear more efficient than both schemes in encoding over all k's, and more efficient in decoding over most k's. This is very interesting and suggests that designing more efficient 3-tolerant XOR-based codes might be possible. Moreover, in all cases, we observe that OXC shows great improvement over the naive approach, where the complexity literally counts the number of 1's in coding matrices. We believe the significant gap between OXC and the naive approach contributes to the long time misconception that Reed-Solomon codes are inappropriate as XOR-based codes.

C. Practicality discussion

In order for OXC to be practically useful, computation paths should be computed offline and stored physically. For encoding, the additional storage overhead is *not* an issue at all, since there is only one computation path to store. For decoding, the number of paths to be stored can be potentially large (literally, one path per erasure pattern). To alleviate the overhead, we consider two scenarios: 1) when the redundancy is limited (e.g., 2 or 3-fault-tolerant), the total number of path might *not* be large and thus all paths can be stored; and 2) when there are more redundancy, computation paths to recover limited failures can be stored. In storage applications, these are more likely to be the most common failures and the most performance gain will be achieved when the common cases are optimized. During rare cases when more failures occur, the decoding falls back into the naive approach. Less efficient decoding in those cases should *not* have great impact on overall system performance. Moreover, OXC might be particularly suitable for codes with inherent hierarchy (e.g., Pyramid Codes [10]), where most decodings happen within small groups with limited redundancy.

VI. Summary

We make a simple and yet important observation that common XOR operations should be computed first in XOR-based coding. We describe the OXC problem and make a conjecture about its complexity. Two greedy approaches are proposed, which effectively show that XOR-based Reed-Solomon codes with optimization can be as efficient and sometimes even more efficient than the best known specifically designed XOR-based codes. Moreover, XOR-based Reed-Solomon codes with optimization are likely suitable for large scale production storage systems with higher redundancy requirements.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.

[2] M. Blaum, J. Bruck, and A. Vardy, "MDS Array Codes with Independent Parity Symbols," *IEEE Trans. Information Theory*, 42(2), 529-542, Mar. 1996.

[3] R. E. Blahut, "Algebraic Codes for Data Transmission," Cambridge Univ. Press, Cambridge, U.K. 2002.

[4] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," *Technical Report No. TR-95-048*, ICSI, Berkeley, California, Aug. 1995.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID – High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, 26(2), 145-185, 1994.

[6] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-Diagonal Parity for Double Disk Failure Correction", *the 4^{th} USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[7] G.-L. Feng, R. H. Deng, F. Bao, and J.-C. Shen, "New Efficient MDS Array Codes for RAID Part I: Reed-Solomon-Like Codes for Tolerating Three Disk Failures", *IEEE Trans. on Computers*, 54(9), Sep. 2005.

[8] G.-L. Feng, R. H. Deng, F. Bao, and J.-C. Shen, "New Efficient MDS Array Codes for RAID Part II: Rabin-Like Codes for Tolerating Multiple (≥ 4) Disk Failures", *IEEE Trans. on Computers*, 54(12), Dec. 2005.

[9] C. Huang, and L. Xu, "STAR: an Efficient Coding Scheme for Correcting Triple Storage Node Failures", *the 4^{th} USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[10] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems", *IEEE International Symposium on Network Computing and Applications (NCA 2007)*, Cambridge, MA, Jul. 2007.

[11] F. J. MacWilliams, and N. J. A. Sloane, "The Theory of Error Correcting Codes, Amsterdam: North-Holland", 1977.

[12] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems", *Software – Practice & Experience*, 27(9), 995-1012, Sep. 1997.

[13] J. S. Plank, and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," *IEEE International Symposium on Network Computing and Applications (NCA 2006)*, Cambridge, MA, Jul., 2006.

[14] I. S. Reed, and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. Soc. Indust. Appl. Math.*, 8(10), 300-304, 1960.

[15] R. Rudell, "Logic Synthesis for VLSI Design", Ph.D. thesis, UC Berkeley, 1989.

[16] S. Soe, and K. Karplus, "Logic Minimization Using Two-Column Rectangle Replacement", ACM/IEEE Design Automation Conference, Jun. 1991.