EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures

Mario Blaum, *Senior Member, IEEE,* Jim Brady, *Fellow, IEEE,* Jehoshua Bruck, *Senior Member, IEEE,* and Jai Menon

Abstract— **We present a novel method, that we call EVENODD, for tolerating up to two disk failures in RAID architectures. EVENODD employs the addition of only two redundant disks and consists of simple exclusive-OR computations. This redundant storage is optimal, in the sense that two failed disks cannot be retrieved with less than two redundant disks. A major advantage of EVENODD is that it only requires parity hardware, which is typically present in standard RAID-5 controllers. Hence, EVENODD can be implemented on standard RAID-5 controllers without any hardware changes. The most commonly used scheme that employes optimal redundant storage (i.e., two extra disks) is based on Reed–Solomon (RS) error-correcting codes. This scheme requires computation over finite fields and results in a more complex implementation. For example, we show that the complexity of implementing EVENODD in a disk array with 15 disks is about 50% of the one required when using the RS scheme.**

The new scheme is not limited to RAID architectures: it can be used in any system requiring large symbols and relatively short codes, for instance, in multitrack magnetic recording. To this end, we also present a decoding algorithm for one column (track) in error.

Index Terms— **RAID architectures, erasure-correcting codes, Reed–Solomon codes, disk arrays.**

I. Introduction

DISK arrays [16], in particular RAID-3 and RAID-5 disk arrays, have become an accepted way for designing highly available and reliable disk subsystems. In such arrays, the exclusive-OR of data from some number of disks is maintained on a redundant disk. When a disk fails, the data on it can be reconstructed by exclusive-ORing the data on the surviving disks, and writing this into a spare disk. The mean time to data loss (MTTDL) of such a system is proportional to the square of the disk mean time between failures (MTBF) and inversely proportional to the square of the number of disks and the mean time to reconstruct (MTTR) the failed disk [16]. Data are lost if a second disk fails before the reconstruction is complete. Such arrays have acceptable MTTDL when the number of disks in the subsystem is small. However, the average number of disks in an installation is growing because of two reasons. First, disk form factors are becoming smaller, so each disk holds less data. Second, installation requirements for data are increasing, caused by normal growth and by the increase in new forms of data like audio, video and fax. As these trends accelerate, it was shown that traditional arrays which can protect from the simultaneous loss of no more than one disk will prove to be inadequate by the year 2000 [7]. Also, [7] explores whether improving disk MTBF or decreasing MTTR can adequately compensate for the increase in the number of disks per installation, and concludes that it will not.

As a result, a lot of interest has arisen in Large Disk Arrays and in attempting to design systems that will not lose data even when multiple disks fail simultaneously [2], [5], [6], [9], [13]. For this, the use of erasure-correcting codes [9] with higher correcting capability than simple parity is suggested (in coding theory terminology, an erasure is an error whose location is known).

Theoretically, in order to retrieve the information lost in two failed (erased) disks, we need at least two redundant disks (in coding theory, this is known as the Singleton bound [12]). A natural scheme, then, for recovering the information lost in two disks, is using the so called Reed–Solomon codes [12]. However, Reed–Solomon codes involve operations over finite fields. It would be desirable to have codes doing exclusive-OR operations only, as in the case of simple parity. This was achieved in [17], although this code has the following drawback: when the error correcting capability of the code is broken, there is an infinite error propagation. Moreover, since the code is of convolutional type, there is an overhead redundancy at the end of the data. For higher correcting capability, the codes in [8], [14], [15] have the same disadvantages. Therefore, the problem still is finding codes based on exclusive-OR operations and of block type. The solution was achieved in [1], [2], [5], [10], [11] and later generalized in [6] for multiple erasures. However, those solutions, although very simple, still involve a recursion at the encoding process and during small write operations. There are applications in which the size of each individual symbol can be as big as a whole sector: during updates operations, we will want to update a minimal number of redundant symbols when we update a single information symbol. The schemes in the papers above force the updating of most of the redundant symbols each time an information symbol is updated.

In this paper, we present an efficient encoding procedure that is based on exclusive-OR operations and independent parities, therefore there is no recursion. We also present a simple decoding procedure for two erasures and also for a single error. As a result of the simple encoding procedure the small write operation is greatly simplified, since any modified information

Manuscript received November 29, 1993; revised April 11, 1994. This paper was presented in part at the International Symposium in Computer Architecture (ISCA), Chicago, IL, April 1994.

M. Blaum and J. Menon are with the IBM Research Division, Almaden Research Center, San Jose, CA 95120 USA.

J. Bruck was with the IBM Research Division, Almaden Research Center, San Jose, CA 95120. He is now with the California Institute of Technology, Pasadena, CA 91125 USA.

J. Brady is with the IBM SSD, San Jose, CA 95120 USA.

IEEE Log Number 9407129.

symbol affects only two symbols in the redundancy most of the time. This implies that when a disk sector is modified, only two other disk sectors will need to be modified at the same time. We note here that EVENODD corresponds to a new 2-erasure correcting code which is optimal in terms of the redundancy and has very efficient encoding and decoding algorithms. Hence, it can be used in other applications where there is a need of correcting two erased symbols with low complexity, for example, in multitrack magnetic recording [1], [14], [15], [17]. As we stated above, we will show how to adapt the decoding algorithm to correct one error in such applications.

The paper is organized as follows: in the next section we make some simple reliability calculations that show why single parity arrays may not be reliable enough for some applications and justify the need to consider building arrays which can survive two simultaneous disk failures. Then, in Section III, we describe the encoding procedure used by our new EVENODD scheme. In Section IV we present the corresponding decoding procedure which will be used after the failure of one or two disks, and we prove that it can, in effect, retrieve the contents of up to two disks. We also show how to correct one error. In Section V we give an algebraic description of the code. In Section VI we address the implementation of small write operations. In Section VII we address the complexity of implementation of EVENODD by comparing it to that of traditional Reed–Solomon codes. In Section VIII, we present some concluding remarks. For a discussion of performance issues, the reader is referred to [3].

II. Reliability Calculations

Under assumptions of independent disk failures, [16] derives an equation for mean time to data loss (MTTDL) for an N disk system organized into groups of size G as

$$\text{MTTDL} = \frac{(\text{MTBF})^2}{N(G-1)(\text{MTTR})}. \tag{1}$$

In this equation, MTBF is mean-time-to-failure of a single disk and MTTR is the mean-time-to-repair of a single disk. Assuming $N = 96$, $G = 16$, MTBF = 200 000 hours and MTTR = 1 hour, the mean time to data loss of the system is 3000 years. This seems adequate, and seems to imply that single parity is sufficient. However, there are two reasons why the above calculation in (1) is too optimistic.

First, (1) does not take into account uncorrectable error rates of disk devices. Uncorrectable error rates after error-correcting codes are 1 error in 10^{13} bits read for current state-of-the-art disks. Consider that a disk in a $15 + P$ (an array with 15 disk and a single parity disk) array fails. Assume that each disk has a capacity of 3 GB, so it has 6 million 512 byte sectors. To reconstruct the failed disk, 90 million sectors (6 million from each of the 15 surviving disks) must be successfully read. There is a data loss if even one of these sectors cannot be read successfully. The probability of reading all 90 million sectors successfully is 0.96. This means that 4% of all disk failures will result in data loss due to uncorrectable errors. This may be unacceptable for many applications.

Another reason for having a second parity disk is the fact that during the reconstruction process after a failure, the system has no backup: a second failure during reconstruction will translate in data loss. This is an unacceptable risk for applications in which data integrity is essential.

The discussion above implies that single parity arrays may not be sufficiently reliable for some applications. In this paper, we focus on how to efficiently design arrays which can withstand two simultaneous failures.

III. Encoding

We will assume that there are $m + 2$ disks with the information stored in the first m disks while the redundant data are stored in the last two disks. It is possible, however, to distribute the redundancy among all disks in order to avoid bottleneck effects when repeated write operations are performed. That is, we shall describe a scheme which is an extension of RAID-4 (where parity is dedicated), but it can be easily made an extension of RAID-5 (where parity is distributed).

We assume that m, the number of information disks, is a prime number. This requirement is important, since without this assumption the scheme would fail. It will become clear when we prove our main result, i.e., the correction capability of the code. However, the primality of m is not a very hard constraining requirement. If we want to store an arbitrary number of disks, not necessarily prime, we can take the next prime following this arbitrary number and assume that there are disks with no information (all the information bits are 0).

In order to simplify the presentation, we assume that each of the m disks has only $m - 1$ symbols of information on it. Our procedure works for disks with arbitrary capacity by treating each block of $m - 1$ symbols separately. For simplicity, in some of our examples, we will assume that each symbol is a bit. In some applications, a symbol may be as big as a 512 byte disk sector. It is not necessary to assume that the symbols are binary. (in fact, our scheme works even when the symbols are elements in an arbitrary Abelian group).

Based on the assumptions above, the problem of tolerating two disk failures can be described as follows:

Problem Definition: Consider an $(m-1) \times (m+2)$ array, m a prime number, such that symbol a_{ij}, $0 \leq i \leq m-2$, $0 \leq j \leq m+1$, is the ith symbol in the jth disk. Again, in some applications, a column of the array may be thought as a disk and a symbol as a disk sector. The last two disks (m and $m+1$) are the disks with the redundant information. The question is how to compute the content of the redundant part based on the information part such that the information contained in any two disks can be reconstructed from the other m disks.

Our encoding scheme solves the foregoing problem and requires only exclusive-OR operations for computing the redundancy.

Before formally describing the encoding procedure, we consider the following notation: $\langle n \rangle_m = j$ if and only if $j \equiv n \pmod m$ and $0 \leq j \leq m-1$. For instance, $\langle 7 \rangle_5 = 2$ and $\langle -2 \rangle_5 = 3$. We also assume through this paper that there

is an imaginary 0-row after the last row, i.e., $a_{m-1,j} = 0$, $0 \le j \le m-1$ (with this convention, the array is now an $m \times (m+2)$ array). This assumption is not necessary, but it is useful for notational purposes as we will see in the description of the code.

A. The Encoding Procedure

Let

$$S = \bigoplus_{t=1}^{m-1} a_{m-1-t,t}. \tag{2}$$

Then, for each $l, 0 \le l \le m-2$, the redundant symbols are obtained as follows:

$$a_{l,m} = \bigoplus_{t=0}^{m-1} a_{l,t} \tag{3}$$

$$a_{l,m+1} = S \oplus \left(\bigoplus_{t=0}^{m-1} a_{\langle l-t \rangle_m, t} \right). \tag{4}$$

Equations (3) and (4) define the encoding. We have two types of redundancy: horizontal redundancy and diagonal redundancy. Disk m is simply the exclusive-OR of disks $0, 1, \cdots, m-1$. Its contents are exactly the same as the parity contents of the parity disk in an equivalent RAID-4 array with one less disk. Disk $(m+1)$ carries the diagonal redundancy according to (4). Let us look closely at this equation, and assume that the symbols are bits. We see that there are two possibilities for the diagonal redundancy: the parity may be even or odd. This even or odd parity is determined by bit S in (2), which gives the parity of diagonal $(m-2, 1)$, $(m-3, 2), \cdots, (0, m-1)$. If this diagonal has an EVEN number of 1's, then we have even parity in the rest of the diagonals. Otherwise, we have ODD parity. This is the reason we call this scheme the EVENODD scheme.

The $(m-1) \times (m+2)$ array defined above can recover the information lost in any two columns. In other words, the minimum distance of the code is 3, in the sense that any nonzero array in the code has at least 3 columns that are nonzero. The proof relies on the fact that m is a prime number and it is based on ideas similar to those in [1], [6], [10], [11]. Here, we prove it in the next section, by showing that the decoding algorithm to be given there can retrieve any pair of erased columns. This implies that the minimum distance of the code is exactly 3, since, if we encode an array with only one nonzero information column, the resulting encoded array will have (column) weight exactly 3. The condition that a special diagonal carries either even or odd parity is not arbitrary. We will see in the examples that without this assumption, the resulting code does not have minimum distance 3, therefore it cannot retrieve any two columns that are erased.

As we can see, the encoding is very simple and circuits implementing (3) and (4) are straightforward. More generally, we would implement (3) and (4) in software in the RAID controller, using exclusive-OR hardware. The next example illustrates the encoding for $m = 5$.

Example 3.1: Let $m = 5$, and let the symbols be denoted by a_{ij}, $0 \le i \le 3$, $0 \le j \le 6$. The redundant symbols are in columns 5 and 6. A practical implementation of this example is to consider 7 disks numbered 0 through 6, each disk has 4 disk sectors, the data sectors are on disks numbered 0, 1, 2, 3, and 4, and the redundant disk sectors are on disks numbered 5 and 6. Equation (2) gives

$$S = a_{3,1} \oplus a_{2,2} \oplus a_{1,3} \oplus a_{0,4}.$$

According to (3) and (4) the redundant symbols are obtained as follows:

$$a_{l,5} = a_{l,0} \oplus a_{l,1} \oplus a_{l,2} \oplus a_{l,3} \oplus a_{l,4}, \quad 0 \le l \le 3$$

$$a_{0,6} = S \oplus a_{0,0} \oplus a_{3,2} \oplus a_{2,3} \oplus a_{1,4}$$

$$a_{1,6} = S \oplus a_{1,0} \oplus a_{0,1} \oplus a_{3,3} \oplus a_{2,4}$$

$$a_{2,6} = S \oplus a_{2,0} \oplus a_{1,1} \oplus a_{0,2} \oplus a_{3,4}$$

$$a_{3,6} = S \oplus a_{3,0} \oplus a_{2,1} \oplus a_{1,2} \oplus a_{0,3}.$$

For instance, assume that we want to encode the 5 columns

1	0	1	1	0		
0	1	1	0	0		
1	1	0	0	0		
0	1	0	1	1		

We have to fill up the last two columns with the encoded symbols. Notice that $S = a_{3,1} \oplus a_{2,2} \oplus a_{1,3} \oplus a_{0,4} = 1$. Therefore, the diagonals will have odd parity. The encoding gives the following array:

1	0	1	1	0	1	0
0	1	1	0	0	0	0
1	1	0	0	0	0	1
0	1	0	1	1	1	0

Notice that the sets of symbols associated with horizontal parity are illustrated as follows:

♢	♢	♢	♢	♢	♢	
♣	♣	♣	♣	♣	♣	
♡	♡	♡	♡	♡	♡	
♠	♠	♠	♠	♠	♠	

Similarly, the sets of symbols associated with diagonal parity are illustrated as follows (note that ∞ is associated with the special diagonal that determines whether the diagonal parity is EVEN or ODD):

♢	♣	♡	♠	∞		♢
♣	♡	♠	∞	♢		♣
♡	♠	∞	♢	♣		♡
♠	∞	♢	♣	♡		♠

Notice that we are assuming that in an $(m-1) \times (m+2)$ array, the parity is stored in columns m and $m+1$. However, the next array may carry the parity in columns $m+1$ and 0, the next in columns 0 and 1, and so on. That way, the parity gets equally distributed among all disks.

We also want to point out that if we do not make the assumption that the diagonals carry either even or odd parity, the code has not minimum distance 3 (in coding theory terminology, the code is not maximum distance separable or MDS [12]). In effect, assume that all the diagonals (except, perhaps, diagonal $(m-2,1)$, $(m-3,2)$, $\cdots$, $(0,m-1)$) carry even parity. In other words, assume that the encoding is given only by (3) and (4) but in (4), the parameter S is ignored. Then, the following is a codeword of weight 2:

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	0	0	0	1	0

If columns 1 and 5 are erased in the array above, it is not possible to retrieve them, since the all-zero array is also in the code. This counter-example shows the importance of the EVENODD assumption: it is the key to the MDS property of the code.

IV. Decoding

An essential part of EVENODD is the decoding algorithm for two erasures. This algorithm, to be described next, can be implemented either in software or in hardware, depending on the application. It will be executed when a disk fails, or when two disks fail simultaneously. Then we prove that the algorithm in effect corrects two erasures.

We also give a decoding algorithm that corrects one error, i.e., only one column has failed, but its location is unknown. This is not the model in RAID architectures, where disk failures are catastrophic events in which an external pointer identifies the failed disks. However, in other applications, like in multitrack magnetic recording, track errors are common [15].

Before giving the actual algorithm for correction of two erasures, we give an example that illustrates the idea behind it.

Example 4.1: We again assume that $m = 5$, as in Example 3.1. Assume that we have the following array, in which columns (disks) 0 and 2 have been erased (lost):

?	0	?	1	0	1	1
?	1	?	0	0	0	1
?	1	?	0	0	1	1
?	1	?	1	1	0	0

This is the main case for the algorithm: two information columns have been erased. The cases in which at least one of the two parity columns has been erased are special cases that are easy to handle, as we will see below.

The first step is finding the parity of the diagonals: it is not difficult to see (and we will prove it in Theorem 4.1) that this parity is given by the exclusive-OR of the bits of the two parity columns. If this exclusive-OR is 0, then the diagonals have even parity, otherwise they have odd parity. In the array above, we can see that the exclusive-OR of the bits in the two redundant columns is 1, therefore the diagonals have odd parity.

Next, the algorithm starts a recursion to retrieve the missing bits $a_{l,0}$ and $a_{l,2}$, $0 \le l \le 3$. We first need an entry where we can start. For instance, diagonal (3, 1), (2, 2), (1, 3), (0, 4) intersects column 2 in entry (2, 2) only: this is the special diagonal, which has odd parity. Since the only bit missing in this diagonal is bit (2, 2), by retrieving it using the other bits, we conclude that $a_{2,2} = 0$. Next we retrieve bit (2, 0) using the horizontal parity, which is always even. We will obtain $a_{2,0} = 0$. Next, we consider the diagonal going through entry (2, 0), which consists of the entries (2, 0), (1, 1), (0, 2), (3, 4), (2, 6). The only bit missing is in entry (0, 2), and we conclude that $a_{0,2} = 0$. Again using the horizontal parity, we conclude that $a_{0,0} = 0$. Now using the diagonal through (0, 0), we obtain that $a_{3,2} = 0$, which implies, using the horizontal parity, that $a_{3,0} = 1$. Using the diagonal through (3, 0), we obtain that $a_{1,2} = 0$, which finally implies that $a_{1,0} = 1$. The final reconstructed array is

0	0	0	1	0	1	1
1	1	0	0	0	0	1
0	1	0	0	0	1	1
1	1	0	1	1	0	0

The decoding algorithm to be given next formalizes the idea behind this example.

Algorithm 4.1 (Two Erasure Decoding Algorithm): Consider the $(m-1) \times (m+2)$ array of symbols a_{ij}, such that the last two columns are redundant according to (2), (3), and (4). If one column (disk) has failed, say column (disk) $i, i \neq m+1$, then it can be retrieved using the exclusive-OR of columns (disks) l, $0 \le l \le m$, $l \neq i$. If column $(m+1)$ fails, then the symbols can be retrieved using (2) and (4).

Next, assume that columns (disks) i and j have failed, where $0 \le i < j \le m+1$. We have four cases:

$\boldsymbol{i = m}$ **and** $\boldsymbol{j = m+1}$, i.e., both the redundant disks have failed. We can reconstruct disk m using (3) and disk $(m+1)$ using (2) and (4). In other words, the reconstruction is equivalent to the encoding.

$\boldsymbol{i < m}$ **and** $\boldsymbol{j = m}$, namely, one redundant disk and one data disk have failed. We can reconstruct disk i as follows: let

$$S = a_{\langle i-1\rangle_m, m+1} \oplus \left(\bigoplus_{l=0}^{m-1} a_{\langle i-l-1\rangle_m, l} \right), \tag{5}$$

where we assume that $a_{m-1,l} = 0$ for $0 \le l \le m+1$. Then,

$$a_{k,i} = S \oplus a_{\langle i-1\rangle_m, m+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \neq i}}^{m-1} a_{\langle k+i-l\rangle_m, l} \right), \quad 0 \le k \le m-2 \tag{6}$$

and $a_{k,m}$, $0 \le k \le m-2$, is obtained using (3) once disk i is reconstructed.

$\boldsymbol{i < m}$ **and** $\boldsymbol{j = m+1}$, namely, one redundant disk and one data disk have failed. We can reconstruct disk i using (3) and disk $m+1$ using (2) and (4) once disk i is reconstructed.

$\boldsymbol{i < m}$ **and** $\boldsymbol{j < m}$. This is the main case. Both failed disks carry information and we cannot retrieve them using the parities separately, as in the previous three cases. We analyze this case in detail.

Assume that $a_{m-1,l} = 0$ for $0 \le l \le m-1$ and compute the diagonal parity S as follows:

$$S = \left(\bigoplus_{l=0}^{m-2} a_{l,m} \right) \oplus \left(\bigoplus_{l=0}^{m-2} a_{l,m+1} \right) \tag{7}$$

(i.e., S is the exclusive-OR of the symbols in columns m and $m+1$). Find the horizontal syndromes $S^{(0)} = S_0^{(0)}, S_1^{(0)}, \cdots, S_{m-1}^{(0)}$ and the diagonal syndromes $S^{(1)} = S_0^{(1)}, S_1^{(1)}, \cdots, S_{m-1}^{(1)}$ as follows:

$$S_u^{(0)} = \bigoplus_{\substack{l=0 \\ l \neq i,j}}^{m} a_{u,l} \tag{8}$$

$$S_u^{(1)} = S \oplus a_{u,m+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \neq i,j}}^{m-1} a_{\langle u-l\rangle_m, l} \right) \tag{9}$$

where $0 \le u \le m-1$. Next, we retrieve the symbols in columns i and j as follows:

1) Set $s \leftarrow \langle -(j-i)-1\rangle_m$ and $a_{m-1,l} \leftarrow 0$ for $0 \le l \le m-1$.
2) Let $a_{s,j} \leftarrow S^{(1)}_{\langle j+s\rangle_m} \oplus a_{\langle s+(j-i)\rangle_m, i}$ and $a_{s,i} \leftarrow S_s^{(0)} \oplus a_{s,j}$.
3) Set $s \leftarrow \langle s-(j-i)\rangle_m$. If $s = m-1$ then stop, else go to step 2.

Algorithm 4.1 is recursive and very simple to implement in software. We can also develop the recursion and obtain a closed formula for each entry as a function of the syndromes. This approach is useful if we want a hardware implementation.

Before proving that Algorithm 4.1 allows us to retrieve up to two erased columns (which shows that the minimum distance of the code is 3), we illustrate it with an example.

Example 4.2: Consider the same array as in Example 4.1. We will reconstruct the missing columns (disks) 0 and 2 using Algorithm 4.1 now.

The first step is finding the parameter S, which is the exclusive-OR of the last two columns. We have seen that $S = 1$, meaning that the diagonals have odd parity. From the array and (8) and (9), we find the syndromes. We obtain

$$S^{(0)} = 0\,1\,0\,1\,0 \quad \text{and} \quad S^{(1)} = 0\,1\,0\,1\,0.$$

Now, we start the recursion to retrieve the missing bits $a_{l,0}$ and $a_{l,2}$, $0 \le l \le 3$. We set $s \leftarrow \langle -(j-i)-1\rangle_m = \langle -3\rangle_5 = 2$, then,

$a_{2,2} \leftarrow S_4^{(1)} = 0$	$a_{2,0} \leftarrow S_2^{(0)} \oplus a_{2,2} = 0$	$s \leftarrow 0$
$a_{0,2} \leftarrow S_2^{(1)} \oplus a_{2,0} = 0$	$a_{0,0} \leftarrow S_0^{(0)} \oplus a_{0,2} = 0$	$s \leftarrow 3$
$a_{3,2} \leftarrow S_0^{(1)} \oplus a_{0,0} = 0$	$a_{3,0} \leftarrow S_3^{(0)} \oplus a_{3,2} = 1$	$s \leftarrow 1$
$a_{1,2} \leftarrow S_3^{(1)} \oplus a_{3,0} = 0$	$a_{1,0} \leftarrow S_1^{(0)} \oplus a_{1,2} = 1$	$s \leftarrow 4$

The algorithm stops, since $s = 4 = m-1$. The reconstructed array is the same as in Example 4.1.

We can see in this example the intuition behind the requirement that m is a prime number: if m was not prime, there would be cases in which the main recursion will stop before all the entries in the two erased columns are received. We will strongly use the primality of m in the proof to be given next. For instance, if $m = 4$ (not a prime!), the following array is in the code and it has weight 2:

1	0	1	0	0	0
0	0	0	0	0	0
1	0	1	0	0	0

It is easy to show that in general, if m is not prime, the code has minimum distance 2.

Next we prove that Algorithm 4.1 allows to retrieve up to two erased columns, automatically showing that the minimum distance of the code is 3.

Theorem 4.1: Algorithm 4.1 can correct up to any two erased columns.

Proof: If only one column is erased, the reconstruction is done by using the parity, so assume that columns i and j have been erased, $0 \le i < j \le m+1$. We consider the four cases of Algorithm 4.1.

$\boldsymbol{i = m, j = m+1}$: This case is equivalent to the encoding.

$\boldsymbol{i < m, j = m}$: Notice that S as given by (5) gives the parity of diagonal $(\langle i-1\rangle_m, 0), (\langle i-2\rangle_m, 1), \cdots, (i, m-1)$. This diagonal is the only one that does not intersect column i, which is unavailable. Since all the diagonals have the same parity S, by solving for $a_{k,i}$ in (4), we obtain (6) which gives the erased entries in column i. Once column i is obtained, the algorithm finds column m using (3).

$\boldsymbol{i < m, j = m+1}$: This case is again similar to the encoding.

$\boldsymbol{i < j < m}$: First of all, we show that S as given by (7) gives the diagonal parity, i.e., (2) and (7) are equivalent (of course, we cannot use (2) to compute S since columns i and j are unavailable). In effect, notice that, according to (3) and (4),

$$\begin{aligned}
&\left(\bigoplus_{l=0}^{m-2} a_{l,m}\right) \oplus \left(\bigoplus_{l=0}^{m-2} a_{l,m+1}\right) \\
&\quad = \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus \left(\bigoplus_{l=0}^{m-2}\left(S \oplus \bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right) \\
&\quad = \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus (m-1)S \\
&\qquad \oplus \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right) \\
&\quad = \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right)
\end{aligned} \tag{10}$$

since $m-1$ is even, therefore $(m-1)S \equiv 0 \bmod 2$.

Also, since we are assuming that $a_{m-1,t} = 0$ (imaginary row of zeros), we have

$$\begin{aligned}
0 &= \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus \left(\bigoplus_{l=0}^{m-1}\left(\bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right) \\
&= \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right) \\
&\quad \oplus \left(\bigoplus_{t=1}^{m-1} a_{m-1-t,t}\right)
\end{aligned}$$

Thus, we obtain

$$\begin{aligned}
&\left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{l,t}\right)\right) \oplus \left(\bigoplus_{l=0}^{m-2}\left(\bigoplus_{t=0}^{m-1} a_{\langle l-t\rangle_m,t}\right)\right) \\
&\qquad = \bigoplus_{t=1}^{m-1} a_{m-1-t,t}
\end{aligned}$$

which replaced in (10), gives

$$\left(\bigoplus_{l=0}^{m-2} a_{l,m}\right) \oplus \left(\bigoplus_{l=0}^{m-2} a_{l,m+1}\right) = \bigoplus_{t=1}^{m-1} a_{m-1-t,t}. \tag{11}$$

Equation (11) proves that (2) and (7) both give the parameter S that determines the diagonal parity, as claimed.

Now we have to prove that columns i and j are uniquely retrieved by the algorithm. Assume that the syndromes $S^{(0)}$ and $S^{(1)}$ have been found according to (8) and (9). Having set the parameter s as $\langle -(j-i)-1\rangle_m$ in the first step of the algorithm, in step 2 we set

$$a_{\langle -(j-i)-1\rangle_m,j} \leftarrow S^{(1)}_{\langle i-1\rangle_m}. \tag{12}$$

Assignment (12) is correct in view of (4) and (9). Once we have found $a_{\langle -(j-i)-1\rangle_m,j}$ we obtain $a_{\langle -(j-i)-1\rangle_m,i}$ by setting

$$a_{\langle -(j-i)-1\rangle_m,i} \leftarrow S^{(0)}_{\langle -(j-i)-1\rangle_m} \oplus a_{\langle -(j-i)-1\rangle_m,j}. \tag{13}$$

Assignment (13) is correct in view of (3) and (8).

Next, in step 3, the parameter s is reset as $\langle -2(j-i)-1\rangle_m$. Going back to step 2, the algorithm sets

$$a_{\langle -2(j-i)-1\rangle_m,j} \leftarrow S^{(1)}_{\langle -2(j-i)+j-1\rangle_m} \oplus a_{\langle -(j-i)-1\rangle_m,i} \tag{14}$$

$$a_{\langle -2(j-i)-1\rangle_m,j} \leftarrow S^{(0)}_{\langle -2(j-i)-1\rangle_m} \oplus a_{\langle -2(j-i)-1\rangle_m,j}. \tag{15}$$

In general, having found $a_{\langle -(l-1)(j-i)-1\rangle_m,i}$, $1 \le l-1 \le m-2$, the algorithm finds $a_{\langle -l(j-i)-1\rangle_m,j}$ and $a_{\langle -l(j-i)-1\rangle_m,i}$ by setting

$$a_{\langle -l(j-i)-1\rangle_m,j} \leftarrow S^{(1)}_{\langle -l(j-i)+j-1\rangle_m} \oplus a_{\langle -(l-1)(j-i)-1\rangle_m,i} \tag{16}$$

$$a_{\langle -l(j-i)-1\rangle_m,i} \leftarrow S^{(0)}_{\langle -l(j-i)-1\rangle_m} \oplus a_{\langle -l(j-i)-1\rangle_m,j}. \tag{17}$$

Again, Assignment (16) is determined by (4) and (9), while Assignment (17) is determined by (3) and (8).

Now, since m is a prime number, in particular, the numbers m and $j-i$ are relatively prime. Therefore, the $m-1$ values $s = \langle -l(j-i)-1\rangle_m$ obtained in step 3 of the algorithm are all distinct. If we take $l = m$, we obtain $s = m-1$, and the algorithm stops. Thus, the $m-1$ erased entries in both columns i and j are obtained after $m-1$ iterations of step 2. When the mth value of s is determined, in step 3, the algorithm stops, since $s = m-1$.

This completes the proof. □

Next, we give the decoding algorithm in case one column is in error.

Algorithm 4.2 (One Error Decoding Algorithm): Consider the $(m-1) \times (m+2)$ array of symbols $A = (a_{ij})$, such that the last two columns are redundant according to (2), (3), and (4). Let $B = (b_{ij})$ be a possibly corrupted version of A. Assume that at most one column is in error, i.e., A and B coincide except perhaps in a single column. Further assume that $b_{p-1,j} = 0$ for all $j = 0, 1, \cdots m+1$ (this is again the imaginary row of zeros).

The decoding procedure works as follows. First, we compute the horizontal syndrome $S^{(0)} = S^{(0)}_0, S^{(0)}_1 \cdots, S^{(0)}_{m-1}$ as

$$S^{(0)}_i = \bigoplus_{l=0}^{m} b_{i,l} \tag{18}$$

and the diagonal syndrome $S^{(1)} = S^{(1)}_0, S^{(1)}_1 \cdots, S^{(1)}_{m-1}$ as

$$S^{(1)}_i = b_{i,m+1} \oplus \left(\bigoplus_{l=0}^{m-1} b_{\langle i-l\rangle_m,l}\right) \tag{19}$$

for $i = 0, 1, \cdots m-1$.

In the sequel $\underline{0}$ and $\underline{1}$ stand for (0, 0, $\cdots$, 0) and (1, 1, $\cdots$, 1), respectively. Next, we distinguish between the following four cases:

Case 1: $S^{(0)} = \underline{0}, S^{(1)} \in \{\underline{0}, \underline{1}\}$. In this case the algorithm concludes that no errors have occurred and no further action is taken. Note that $S^{(1)} = \underline{0}$ corresponds to the case in which all the diagonals have even parity, while $S^{(1)} = \underline{1}$ corresponds to the case in which the diagonals have odd parity.

Case 2: $S^{(0)} \neq \underline{0}$, $S^{(1)} \in \{\underline{0}, \underline{1}\}$. In this case the error is in column m—the horizontal parity column. We can reconstruct this column by using (3).

Case 3: $S^{(0)} = \underline{0}$, $S^{(1)} \notin \{\underline{0}, \underline{1}\}$. In this case the error is in column $m+1$—the diagonal parity column. We can reconstruct this column by using (2) and (4).

Case 4: $S^{(0)} \neq \underline{0}$, $S^{(1)} \notin \{\underline{0}, \underline{1}\}$. This is the main case. The column in error must be one of the information columns. The error itself is given by the first $m-1$ bits of the horizontal syndrome $S^{(0)}$. Hence, the problem is to locate the information column in error. To this end we proceed as follows. For any vector $\underline{x} = (x_0, x_1, \cdots, x_{n-1})$ let $\rho(\underline{x}) = (x_{n-1}, x_0, \cdots, x_{n-2})$ be the cyclic rotation of $\underline{x}$ to the right, and let $\rho_j(\cdot)$ denote the result of applying $\rho(\cdot)$ successively j times (for example, $\rho_3(0,1,0,0) = (1,0,0,0)$). We then find the first index j with $0 \leq j \leq m-1$, such that $\rho_j(S^{(0)}) \in \{S^{(1)}, \underline{1} \oplus S^{(1)}\}$. This index j corresponds to the location of the column in error. If there is no such j, the algorithm declares an uncorrectable error pattern. The final step is to add modulo-2 the first $m-1$ bits of the syndrome $S^{(0)}$ to the jth column of $B = (b_{ij})$.

As we can see, Algorithm 4.2 involves cyclic shifts and exclusive-OR operations only, which makes it very easy to implement. Next we illustrate Algorithm 4.2 with an example.

Example 4.3: As in previous examples, we assume $m = 5$. Suppose we are given the following, possibly corrupted, array (to which we have appended the imaginary zero row):

1	0	0	1	0	1	1
0	1	1	0	0	1	0
1	1	0	0	0	0	1
1	1	0	1	1	1	0
0	0	0	0	0	0	0

Using (18) and (19), we find that the horizontal and diagonal syndromes are $S^{(0)}$ = (1, 1, 0, 1, 0) and $S^{(1)}$ = (0, 1, 0, 0, 1), respectively. Note that $\rho_2(\underline{s}_0) = \underline{1} \oplus \underline{s}_1$. Hence the column at location $j = 2$, that is the third column from the left in the array, is in error. Adding the first four bits of $\underline{s}_0$ to this column, we obtain the decoded array

1	0	1	1	0	1	1
0	1	0	0	0	1	0
1	1	0	0	0	0	1
1	1	1	1	1	1	0

V. Algebraic Description of EVENODD

The array codes described in [6] were shown to be equivalent to Reed–Solomon codes of length m, m prime, with operations taken modulo the polynomial $M_m(x) = (x^m - 1)/(x-1) = x^{m-1} + x^{m-2} + \cdots + x + 1$. Note that the polynomial $M_m(x)$ is not necessarily irreducible (in fact, it is irreducible if and only if 2 is primitive in $GF(m)$ [6]), and therefore these codes are not defined over a field, but rather over the ring of polynomials of degree $\leq m-2$ modulo $M_m(x)$.

In terms of a $(m-1) \times (m+2)$ array, we shall assume that each column in the array is a polynomial modulo $M_m(x)$. As we have seen in the previous sections, it is also convenient to assume that the array has an imaginary row of zeros, which makes it an $m \times (m+2)$ array. A cyclic shift of a column in such array can cause the bit corresponding to the last row to be nonzero. However, if that is the case, the arithmetic modulo $M_m(x)$ forces to take the complement of the shifted column, restoring the zero in the last position. As in [6], we will use the notation $a(\beta) = a_{m-2}\beta^{m-2} + \cdots + a_1\beta + a_0$ to denote a polynomial modulo $M_m(x)$. Thus $a(\beta)b(\beta)$ denotes polynomial multiplication modulo $M_p(x)$. The usual multiplication of polynomials is written as $a(x)b(x)$.

With this notation, an alternative definition of EVENODD is

$$\left\{ A = (a_0(\beta), a_1(\beta), \cdots, a_{m-1}(\beta), a_m(\beta), a_{m+1}(\beta)): \right.$$
$$\left. a_m(\beta) = \bigoplus_{i=0}^{m-1} a_i(\beta), a_{m+1}(\beta) = \bigoplus_{i=0}^{m-1} \beta^i a_i(\beta) \right\}. \tag{20}$$

Note that the parameter S, defined in (2) and taking part in (4), essentially renders (4) to be the sum of cyclic shifts modulo $M_m(x)$, rather than ordinary cyclic shifts.

The following is a parity-check matrix for EVENODD:

$$H = \begin{pmatrix} 1 & 1 & \cdots & 1 & 1 & 0 \\ 1 & \beta & \cdots & \beta^{m-1} & 0 & 1 \end{pmatrix} \tag{21}$$

Note that the parity symbols $a_m(\beta)$ and $a_{m+1}(\beta)$ depend on the information symbols but not on each other. This suggests a generalization of EVENODD based on the parity-check matrix given by (21), see [4] for more details. It is easy to see using the parity-check matrix that the minimum distance of EVENODD is 3, giving an alternative proof to the basic MDS property of the code.

VI. Small Write Operations

In systems involving many disks, we often encounter the situation in which many small write operations are needed. A small write operation is a write that updates a single data sector (one symbol). EVENODD offers great flexibility to do this, since the symbols involved can have an arbitrary size. Typically, we would implement a symbol as a disk sector.

Every time an information symbol is rewritten, and this information symbol is not in diagonal $(m-2, 1), (m-3, 2), \cdots, (0, m-1)$, then only two redundant symbols are affected, so we need only three read and three write operations. With a symbol as a disk sector, when a disk sector is updated, in most cases, we only need to read three disk sectors (the disk sector being updated and two redundant disk sectors containing parity) and write three disk sectors. Explicitly, if symbol a_{ij}, $0 \leq i \leq m-2$, $0 \leq j \leq m-1$, $\langle i+j \rangle_m \neq m-1$, is replaced by symbol r (i.e., $a_{ij} \leftarrow r$), we have to make the following

modifications in the redundant symbols:

$$a_{i,m} \leftarrow a_{i,m} \oplus a_{ij} \oplus r \tag{22}$$

$$a_{\langle i+j\rangle_m,m+1} \leftarrow a_{\langle i+j\rangle_m,m+1} \oplus a_{ij} \oplus r. \tag{23}$$

On the other hand, if the rewritten information symbol is in diagonal $(m-2,1)$, $(m-3,2)$, $\cdots$, $(0,m-1)$, then all the symbols in column $m+1$ are affected (and of course, the corresponding symbol in column m).

Explicitly, if symbol a_{ij}, $0 \le i \le m-2$, $0 \le j \le m-1$, $\langle i+j\rangle_m = m-1$, is replaced by symbol r (i.e., $a_{ij} \leftarrow r$), we have to make the following modifications in the redundant symbols:

$$a_{i,m} \leftarrow a_{i,m} \oplus a_{ij} \oplus r \tag{24}$$

$$a_{t,m+1} \leftarrow a_{t,m+1} \oplus a_{ij} \oplus r, \qquad 0 \le t \le m-2. \tag{25}$$

Again, we illustrate the small write operations with an example.

Example 6.1: Assume that the we have the following encoded array:

0	0	0	0	0	0	0
1	1	0	1	0	1	0
0	1	1	1	0	1	1
0	1	0	0	1	0	0

Say, we replace entry (0, 1) by a 1. Since it is not in diagonal (3, 1), (2, 2), (1, 3), (0, 4), according to (22) and (23), we have to modify symbols (0, 5) and (1, 6). The new array is

0	1	0	0	0	1	0
1	1	0	1	0	1	1
0	1	1	1	0	1	1
0	1	0	0	1	0	0

Thinking of columns as disks and symbols as disk sectors, we had to access 3 sectors, one from each of 3 disks. Finally, if we modify symbol (2, 2), since it is in diagonal (3, 1), (2, 2), (1, 3), (0, 4), according to (24) and (25), we have to modify symbols (2,5), (0,6), (1,6), (2,6), and (3,6). If columns represent disks and symbols represent disk sectors, we still only need to modify disk sectors on two disks in addition to modifying the disk sector containing the data to be modified. On one of the two redundant disks we need to change four consecutive sectors, on the other redundant disk we need to change a single sector. Changing four consecutive sectors takes almost the same time as changing a single sector (since seek and latency times are much larger than sector transfer times). The new array in our example is

0	1	0	0	0	1	1
1	1	0	1	0	1	0
0	1	0	1	0	0	0
0	1	0	0	1	0	1

So far, in practical applications, we have considered each symbol as a 512-byte sector. There are many other possibilities, since the size of a symbol offers great flexibility. For example, another possible solution is to let each symbol be an 8-bit byte, and $m = 257$ (a Fermat prime number). Therefore, we have an array of up to 259 disks, more than enough for present and future applications. Note that the array does not have to have 259 disks (this is just the maximum number); if it has fewer disks, simply treat the remaining columns as having zeros. Each column of the array consists of 256 bytes, i.e., half a sector. In this case, a small write operation consists of writing a whole column. Thus, the two redundant columns will be modified accordingly. Say, each symbol $a_{i,j}$, $0 \le i \le m-2$ in column j, $0 \le j \le m-1$, is replaced by r_i. Then, we have to do the following modifications in the redundant symbols:

$$a_{i,m} \leftarrow a_{i,m} \oplus a_{i,j} \oplus r \tag{26}$$

$$a_{i,m+1} \leftarrow a_{i,m+1} \oplus a_{\langle i-j\rangle_m,j} \oplus r_{\langle i-j\rangle_m} \oplus a_{m-1-j,j} \oplus r_{m-1-j}, \tag{27}$$

where $0 \le i \le m-2$. That is, when a sector is updated, the two corresponding redundant sectors are also updated according to (26) and (27).

VII. Complexity Comparison with Existing Schemes

In this section, we compare the complexity of EVENODD with the one of a traditional error-correcting code, a Reed–Solomon (RS) code [12]. Both EVENODD and a RS code require an optimal number of redundant disks, namely two. However, one major advantage of EVENODD is that it only requires parity hardware, which is typically present in standard RAID-5 controllers. Hence, EVENODD can be implemented on standard RAID-5 controllers without hardware changes. The scheme based on RS codes, on the other hand, requires special hardware to support finite field type of computations. Hence, it cannot be incorporated into standard RAID-5 controllers. We note here that the 2-D scheme of [9] has the same property as EVENODD, that is, it only needs standard parity hardware. However, if we assume that the m information disks are set in a square array of side $\sqrt{m}$, 2-D needs $2\sqrt{m}$ redundant disks while EVENODD needs only two redundant disks. So, our scheme is much more efficient.

Next we will make a detailed comparison between EVENODD and RS schemes. We will consider RS codes over 8-bit bytes, or $GF(2^8)$ in the language of finite fields. This is a standard in the industry, allowing for codes of length up to 257 bytes. More specifically, we will consider the finite field generated by the primitive polynomial $p(x) = 1 + x^2 + x^3 + x^4 + x^8$. Let α be a primitive element in $GF(2^8)$ such that $p(\alpha) = 0$, and let $m \le 255$. Then, a parity-check matrix for the RS code is the following:

$$H = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & 0 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{m-1} & 0 & 1 \end{pmatrix}. \tag{28}$$

At the encoding, if $b_0, b_1, \cdots, b_{m-1}$ is a string of information bytes, according to (28), the redundant bytes p and q are

obtained as follows:

$$p = \bigoplus_{i=0}^{m-1} b_i \tag{29}$$

$$q = \bigoplus_{i=0}^{m-1} b_i \alpha^i. \tag{30}$$

Now, if we compare with the encoding procedure of EVENODD given by (3) and (29), we can see that (3) and (29) are equivalent. Therefore, the difference in complexity at the encoding is a result of the difference in computing the second redundancy disk, namely, (4) and (30). We analyze the complexity of the encoding both for EVENODD and for the RS scheme by counting the number of exclusive-OR (XOR) operations for each of them.

We assume that each symbol is an 8-bit byte, and the information symbols constitute an $(m-1) \times m$ array, where m is prime. With this assumption, the number of XOR operations due to (3) or (29) at the bit level is $8(m-1)^2$.

Let us count next the number of XOR's in (4) of EVENODD. The first step is computing the symbol S, which is given by (2). This takes $(m-2)$ XOR operations at the byte level. At the bit level, this gives a total of $8(m-2)$ XOR operations. Now, for each l in (4), we have a total of m XOR operations at the byte level. At the bit level, this gives a total of $8m$ XOR operations for each l, and since l runs from 0 to $m-2$, (4) takes $8(m-1)m$ XOR operations. Adding to the number of XOR operations used in computing S, (4) takes a total of $8((m-2)+(m-1)m) = 8(m^2-2)$ XOR operations. We observe that this number is quadratic in m and slightly bigger than the number of operations from (3). The discrepancy is due to the calculation of S first, but we cannot do better than quadratic complexity. By adding the total from (3), we conclude that EVENODD needs a total of

$$8(2m^2 - 2m - 1)$$

XOR operations.

Let us look at the RS scheme now, specifically at (30). Each multiplication of a byte by α, is represented by the following companion matrix A:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{31}$$

Notice that multiplying the byte $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7)$ by α takes 3 XOR operations. In fact, the outcome of multiplying the byte above by the matrix A will produce the byte $(c_7, c_0, c_1 \oplus c_7, c_2 \oplus c_7, c_3 \oplus c_7, c_4, c_5, c_6)$. Therefore, multiplying by α^i will take $3i$ XOR operations. So, implementing (30) on the bytes $b_0, b_1, \cdots, b_{m-1}$ takes

$$8(m-1) + \sum_{i=1}^{m-1} 3i = \frac{3m^2 + 13m - 16}{2}$$

TABLE I
NUMBER OF XOR OPERATIONS NEEDED TO ENCODE $(m-1)$ BYTES PER DISK IN A DISK ARRAY WITH m INFORMATION DISKS

# of information disks	EVENODD	Reed-Solomon	improvement factor
5	312	376	1.21
7	664	954	1.44
11	1752	3250	1.86
13	2488	5112	2.05
17	4344	10624	2.45
23	8088	24442	3.02
29	12948	46648	3.59
31	14872	56250	3.78
41	26232	124000	4.73
43	28888	142002	4.92

XOR operations. Since we have $(m-1)$ bytes, this gives a total of

$$0.5(m-1)(3m^2 + 13m - 16) = 1.5m^3 + 5m^2 - 14.5m + 8$$

XOR operations. Adding the $8(m-1)^2$ XOR operations from (29), we conclude that the encoding of the RS scheme requires

$$1.5m^3 + 13m^2 - 30.5m + 16$$

XOR operations.

As we can see, the complexity of the encoding of EVENODD is quadratic in the number of information disks m, while the complexity of RS codes is cubic. Table I compares EVENODD to RS codes for different values of m, assuming that m is prime (as we have stated, this is not a hard constraint, since EVENODD codes can be shortened to cover cases in which m is not a prime). The last column of Table I contains the quotient between the number in column 3 (i.e., the number of operations needed in the RS code) and the number in column 2 (i.e., the number of operations needed in EVENODD). For instance, we can see that for $m = 43$ (last row), a RS code requires nearly 5 times as many operations as EVENODD at the encoding.

We can see in Table I that the number of XOR operations needed for encoding EVENODD decreases dramatically with respect to a RS code when the number of disks increases. Similar calculations show the advantage of EVENODD in small write operations and in the decoding.

An alternative implementation of the encoding of RS codes is implementing each matrix A^i in hardware. Thus, we will save XOR operations for larger values of m. However, the hardware for this implementation is more complicated, and the matrices A^i are not sparse anymore, therefore EVENODD still has the edge.

We also compared the complexity of EVENODD and the RS based schemes with that of a simple parity scheme. The number of operations required in implementing the parity scheme on an m disk array with $(m-1)$ bytes per disk is $8(m-1)^2$. Hence, EVENODD is asymptotically twice as complicated as simple parity. Notice that this is optimal since there are two redundancy disks in EVENODD. The complexity of the RS scheme is asymptotically about $0.1875m$ times more

TABLE II
COMPARISON OF THE NUMBER OF XOR OPERATIONS IN A SIMPLE PARITY SCHEME WITH EVENODD AND RS SCHEMES

# of information disks	EVENODD vs. Parity	RS vs. Parity
5	2.43	2.93
7	2.30	3.31
11	2.19	4.06
13	2.15	4.43
17	2.12	5.18
23	2.08	6.30
29	2.07	7.43
31	2.06	7.80
41	2.05	9.68
43	2.04	10.06

complex than the simple parity scheme. Table II presents the comparison for various values of m. As we can see, already in the case of $m = 23$ EVENODD is about twice more complex than the simple parity scheme (this is optimal), while the RS scheme requires more than 6 times XOR operations compared with the simple parity scheme.

VIII. CONCLUDING REMARKS

We have presented a novel method, called EVENODD, for tolerating double disk failure in RAID architectures. EVENODD has the following advantages over other methods proposed for recovery against two disk failures:

1) EVENODD employs the addition of only two redundant disks for tolerating two disk failures (this is optimal).
2) It consists of simple exclusive-OR computations and only requires parity hardware, which is typically present in standard RAID-5 controllers. Hence, EVENODD can be implemented in standard RAID-5 controllers without any hardware changes.
3) It can be incorporated to known RAID techniques. For example, parity can be distributed among all disks, avoiding bottleneck effects when repeated write operations are involved (RAID-5).
4) The symbols can have any size, from bits to multiple sectors. There are no constrains to bits or to bytes.
5) Most small write operations affect two redundant symbols only, i.e., for every write we need up to three read and three write operations. Only when the affected symbol is in diagonal $(m-2,1)$, $(m-3,2)$, $\cdots$, $(0,m-1)$ we have to modify all the symbols in column $m+1$ and one symbol in column m. In any case, the parities are independent.
6) The traditional known scheme that employs optimal redundant storage (i.e., two extra disks) is based on Reed–Solomon (RS) error-correcting codes, requires computation over finite fields and results in a more complex implementation. For example, we showed that the complexity of implementing EVENODD in a disk array with 15 disks is about 50% of the one required when using the RS scheme.
7) Other codes involving only exclusive-OR operations are of convolutional type. For the codes in [8], [17], an error in the decoding propagates indefinitely. Since our codes are of block type, they do not have this problem. Also, the redundancy of our codes is slightly smaller, since convolutional codes have an overhead redundancy.
8) There are also optimal block codes based on exclusive-OR operations. However, these codes still need a recursion at the encoding and during small write operations. EVENODD has independent parities, making the complexity even smaller.

An apparent constraint in our construction is that the number of information disks has to be a prime number. However, if the desired number of disks is not a prime number, one can simply assume that there are more disks which have all zeros without affecting the encoding and decoding procedures.

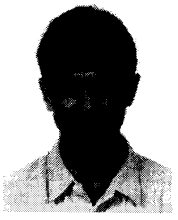
From the perspective of error-correcting codes, we have constructed a new code that is capable of correcting either two erasures or one error. The application described in this paper is in RAID type of architectures, but the code can be also used in magnetic recording and in other situations involving large symbols and short codewords.

ACKNOWLEDGMENT

We are grateful to the reviewers for their useful comments that helped in improving the presentation.

REFERENCES

[1] M. Blaum, "A class of byte-correcting array codes," IBM Research Report, RJ 5652 (57151), May 1987.
[2] ——, "A coding technique for recovery against double disk failures in disk arrays," in *Proc. IEEE Int. Conf. Commun.*, Chicago, IL, June 1992, pp. 1366–1368.
[3] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures" in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Chicago, IL, Apr. 1994.
[4] M. Blaum, J. Bruck, and A. Vardy, "Binary codes with large symbols," in *Proc. 1994 IEEE Int. Symp. Inform. Theory*, June 1994.
[5] M. Blaum, H. Hao, R. Mattson, and J. Menon, "A coding technique for double disk failures in disk Arrays," U.S. Patent 5 271 012, Dec. 1993.
[6] M. Blaum and R. Roth, "New array codes for multiple phased burst correction," *IEEE Trans. Inform. Theory*, pp. 66–77, Jan. 1993.
[7] W. Burkhard and J. Menon, "Disk array storage system reliability," in *Proc. 23rd Annu. Int. Symp. Fault-Tolerant Computing*, Toulouse, France, June 1993.
[8] T. Fuja, C. Heegard, and M. Blaum, "Cross parity check convolutional codes," *IEEE Trans. Inform. Theory*, July 1989, pp. 1264–1276.
[9] G. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson, "Coding techniques for handling failures in large disk arrays," Report No. UCB/CSD 88/477, Dec. 1988.
[10] R. Goodman and M. Sayano, "Size limits on phased burst error correcting array codes," *Electron. Lett.*, vol. 26, pp. 55–56, 1990.
[11] R. Goodman, R. J. McEliece, and M. Sayano, "Phased burst correcting array codes," *IEEE Trans. Inform. Theory*, pp. 684–693, Mar. 1993.
[12] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam, The Netherlands: North-Holland, 1977.
[13] S. W. Ng, "Some design issues of disk arrays," IBM Research Report, RJ 6590 (63550), Dec. 1988.
[14] A. M. Patel, "Multitrack error correction with cross-parity check coding," IBM Technical Report TR02.813, 1978.
[15] A. M. Patel, "Adaptive cross parity code for a high density magnetic tape subsystem," *IBM J. Res. Develop.*, vol. 29, pp. 546–562, 1985.
[16] D. A. Patterson, G. A. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks," in *Proc. SIGMOD Int. Conf. Data Management*, Chicago, IL, 1988, pp. 109–116.
[17] P. Prusinkiewicz and S. Budkowski, "A double track error-correction code for magnetic tape," *IEEE Trans. Comput.*, pp. 642–645, June 1976.

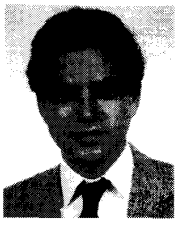


Mario Blaum (S'84–M'85–SM'92) was born in Buenos Aires, Argentina. He received the degree of Licenciado from the University of Buenos Aires in 1977, the M.Sc. degree from the Israel Institute of Technology in 1981 and the Ph. D. degree from the California Institute of Technology in 1984, all these degrees in mathematics.

From January to June, 1985, he was a Research Fellow at the Department of Electrical Engineering at Caltech. In August, 1985, he joined the IBM Research Division at the Almaden Research Center, where he is presently a Research Staff Member. From September 1990 to September 1991 he was a Consulting Professor at Stanford University, where he taught a course in Error-Correcting Codes. His research interests include error-correcting codes, storage technology, combinatorics and neural networks.

Jim Brady (M'83–SM'89–F'94) has substantial experience in designing large, complex systems. He has had major responsibility in the design of XRF, RSS, System/390 architecture, Expanded Storage, MVS SP 1.2, MVS SYSPLEX, and VM HPO 4.0. He developed one of the first software error models and received an IBM Outstanding Innovation Award for his work on system modeling. Both of these efforts produced efforts that moved analytical models from being relative predictors, to accurate estimators.

He joined IBM in 1961 in the Omaha Branch Office as a Systems Engineer-Scientific. He held numerous positions in the Branch covering most of the large system's customers in the Omaha area. In 1967 he was promoted to Advisory Advanced Systems Specialist working on the development of large systems marketing requirements. His next assignment involved working on the problems of software and system availability. In 1971 he became a Consulting Marketing Representative in Nashville. In 1975 he moved to Poughkeepsie into the systems technology area, where he has held various management positions, the last being Program Manager—Systems Technology. He moved to San Jose in 1983 where he was Product Manager—Storage Systems Strategy and Architecture, an organization concerned with the identification of growth opportunities for SSD and the integration of new technology into large systems products. In 1988 he started the Storage System Lab, a joint effort between the Storage Systems Division and IBM Research, which develops new systems technologies such as RAID and storage hierarchies. In 1991 he became the chief architect of a new storage controller. In 1993 he was appointed IBM Fellow. He is current President of the IBM Academy of Technology.



Jehoshua Bruck (S'86–M'89–SM'93) received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion, Israel Institute of Technology, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University in 1989.

He is an Associate Professor of Computation and Neural Systems and Electrical Engineering at the California Institute of Technology. His research interests include parallel and distributed computing, fault-tolerant computing, error-correcting codes and neural networks. He has an extensive industrial experience, including, serving as manager of the Foundations of Massively Parallel Computing Group at the IBM Almaden Research Center from 1990 to 1994, a research staff member at the IBM Almaden Research Center from 1989 to 1990 and a researcher at the IBM Haifa Science center from 1982 to 1985. Dr. Bruck is the recipient of a 1994 National Science Foundation Young Investigator Award, a 1992 IBM Outstanding Innovation Award for his work on "Harmonic Analysis of Neural Networks" and a 1994 IBM Outstanding Technical Achievement Award for his contributions to the design and implementation of the SP-1, the first IBM scalable parallel computer. He also received five IBM Plateau Invention Achievement Awards and he holds 15 patents.

Jai Menon received the B.Tech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1977, and the M.S. and Ph.D degrees in computer science from Ohio State University in 1978 and 1981, respectively. For his Ph.D, he did research in database machine architectures, and he is contributing author on two books on database machine architectures.

Since 1982, he has been with the IBM Almaden Research Center, San Jose, CA, where he has been working in the area of I/O and Storage Systems. Since 1987, he has been Manager of Storage Attachment Architecture in the Computer Science Department at the Almaden Research Center. He has received an Outstanding Technical Achievement Award and five Invention Achievement Awards from IBM. His group is one of the leading groups doing research in disk arrays. He has published 15 papers on disk arrays, and presented several disk array tutorials.