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

Mario Blaum* Jim Brady† Jehoshua Bruck* Jai Menon*

*IBM Almaden Research Center
San Jose, CA 95120
{blaum,bruck,menonjm}@almaden.ibm.com

†IBM SSD
San Jose, CA 95193

Abstract

We present a novel method, that we call EVENODD, for tolerating up to two disk failures in RAID architectures. EVENODD is the first known scheme for tolerating double disk failures that is optimal with regard to both storage and performance. EVENODD employs the addition of only two redundant disks and consists of simple exclusive-OR computations. 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 only previously known scheme that employes 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 show that the number of exclusive-OR operations involved in implementing EVENODD in a disk array with 15 disks is about 50% of the number required when using the RS scheme.

1 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. 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 [6]. Also, [6] 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 [4, 5, 8, 13]. For this, the use of erasure-correcting codes [8] 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 [11]). A natural scheme, then, for recovering the information lost in two disks, is

1063-6897/94 $03.00

using the so called Reed-Solomon codes [11]. 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 [7, 14, 15] have the same disadvantages. Therefore, the problem still is finding codes based on exclusive-OR operations and of block type. A solution was obtained in [1, 4, 9, 10] and later generalized in [5] for multiple erasures. However, those solutions, although very simple, still involve recursive computations (which are inefficient and hard to implement using existing exclusive-OR hardware) at the encoding process and during small write operations.

In this paper, we present a novel and efficient encoding procedure that is based on exclusive-OR operations and does not involve recursive computations. We also present a simple decoding procedure. We have calculated the complexity of implementation of EVENODD and compared it to that of the traditional single parity scheme as well as to the scheme based on Reed-Solomon codes. EVENODD requires about twice as many exclusive-OR operations compared to the simple parity scheme; this is optimal since we have two redundant disks. EVENODD is substantially more efficient when compared to Reed-Solomon codes; for example, the number of exclusive-OR operations involved in implementing EVENODD in a disk array with 15 disks is about 50% of the number required when using the RS scheme.

In addition to having an optimal number of operations for the encoding procedure, EVENODD has the advantages that the encoding procedure can be implemented using existing parity hardware and that small write operations are very efficient. In particular, 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, EVENODD can be used in other applications where there is a need of correcting two erased symbols with low complexity; for example, in multi-track magnetic recording [1, 7, 14, 15, 17]. Moreover, the decoding algorithm can be easily adapted to correct one random error in such applications.

The paper is organized as follows: in the next section we describe the encoding procedure used by our new EVENODD scheme. In Section 3 we present the corresponding decoding procedure which will be used after the failure of one or two disks. In Section 4 we address the implementation of small write operations. In Section 5 we address the complexity of implementation of EVENODD by comparing it to that of traditional Reed-Solomon codes. In Section 6 we discuss performance issues of our scheme. Finally, in Section 7, we present some concluding remarks.

2 Encoding

We will assume that there are $m + 2$ disks with the information stored in the first m disks while the redundancy is 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, our description is of a scheme which is an extension of RAID-4 (where parity is dedicated), but it is easy to imagine how it can be made an extension of RAID-5 (where parity is distributed). We assume that m is a prime number. This requirement has no effect on the optimality of EVENODD. EVENODD can handle an arbitrary number of disks simply by assuming 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. However, it is not necessary to assume that the symbols are binary (in fact, it can be shown that our scheme works even when the symbols are elements in an Abelian group). A practical implementation is to consider a symbol as an 8-bit byte, and to assume that $m - 1 = 256$ symbols (half a sector). Notice that 257 is a prime number.

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

Problem Definition: Consider the $(m-1) \times (m+2)$ array, m a prime number, such that symbol a_{ij}, $0 \leq i \leq m-2$, $0 \leq j \leq m+1$, is the i-th symbol in the j-th disk. Again, we can think of a column of the array as a disk and a symbol as a byte. 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. We assume throughout 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). We also assume that m is a prime, however, this is not a restriction because we can always treat the extra columns as imaginary columns of 0's.

We illustrate the problem definition in the following figure. Here we assume that there are 6 information disks, so the closest prime we can choose is $m = 7$. The 6 information disks are labeled with ♡ and the two redundancy disk are labeled with ♣. The column of 0's as well as the last row of 0's do not exist in practice and are included for better illustration of the encoding and decoding procedures.

♡	♡	♡	♡	♡	♡	0	♣	♣
♡	♡	♡	♡	♡	♡	0	♣	♣
♡	♡	♡	♡	♡	♡	0	♣	♣
♡	♡	♡	♡	♡	♡	0	♣	♣
♡	♡	♡	♡	♡	♡	0	♣	♣
♡	♡	♡	♡	♡	♡	0	♣	♣
0	0	0	0	0	0	0	0	0

Before formally describing the encoding procedure, we present the following notation: $\langle n \rangle_m = j$ if and only if $j \equiv n \pmod m$ and $0 \le j \le m-1$. For instance, $\langle 7 \rangle_5 = 2$ and $\langle -2 \rangle_5 = 3$.

The Encoding Procedure

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

$$a_{l,m} = \bigoplus_{t=0}^{m-1} a_{l,t} \quad (1)$$

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

where

$$S = \bigoplus_{t=1}^{m-1} a_{m-1-t,t}. \quad (3)$$

Notice that we have two types of redundancy: horizontal redundancy and diagonal redundancy. Disk m (the horizontal redundancy) is simply the exclusive-OR of disks $0, 1, \ldots, m-1$. Its contents are exactly the same as the parity contents of the parity disk in an equivalent RAID-4 array with one less disk. It is illustrated as follows (for $m = 7$) with 6 information columns:

♢	♢	♢	♢	♢	♢	0	♢	
♣	♣	♣	♣	♣	♣	0	♣	
♡	♡	♡	♡	♡	♡	0	♡	
♠	♠	♠	♠	♠	♠	0	♠	
□	□	□	□	□	□	0	□	
△	△	△	△	△	△	0	△	
0	0	0	0	0	0	0	0	0

Disk $(m+1)$ carries the diagonal redundancy according to Equation (2). It is illustrated as follows:

♢	♣	♡	♠	□	△	0		♢
♣	♡	♠	□	△	∞	0		♣
♡	♠	□	△	∞	♢	0		♡
♠	□	△	∞	♢	♣	0		♠
□	△	∞	♢	♣	♡	0		□
△	∞	♢	♣	♡	♠	0		△
0	0	0	0	0	0	0	0	0

Looking closely at Equation (2) (assuming 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 Equation (3), which gives the parity of diagonal $(m-2,1), (m-3,2), \ldots, (0, m-1)$. 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. Note that in the foregoing figure, ∞ is associated with the special diagonal that determines whether the diagonal parity is EVEN or ODD.

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 non-zero array in the code has at least 3 columns that are non-zero. The proof relies on the fact that m is a prime number and it is based on ideas similar to those in [1, 5, 9, 10]. A complete presentation of the proof can be found in the full version of the paper [2].

As we can see, the encoding is very simple and circuits implementing Equations (1) and (2) are straight-

forward. In particular, Equations (1) and (2) can be implemented in software in the RAID controller using the exclusive-OR hardware. The next example illustrates the encoding for $m = 5$.

Example 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. According to Equations (1) and (2) the redundant symbols are obtained as follows:

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

$$\begin{aligned} a_{0,6} &= S \oplus a_{0,0} \oplus a_{3,2} \oplus a_{2,3} \oplus a_{1,4} \\ a_{1,6} &= S \oplus a_{1,0} \oplus a_{0,1} \oplus a_{3,3} \oplus a_{2,4} \\ a_{2,6} &= S \oplus a_{2,0} \oplus a_{1,1} \oplus a_{0,2} \oplus a_{3,4} \\ a_{3,6} &= S \oplus a_{3,0} \oplus a_{2,1} \oplus a_{1,2} \oplus a_{0,3} \end{aligned}$$

where Equation (3) gives

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

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

3 Decoding

An essential part of EVENODD is the decoding algorithm. 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.

The Decoding Procedure

Consider the $(m-1) \times (m+2)$ array of symbols a_{ij}, such that the last two columns are redundant according to Equations (1), (2) and (3). 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 Equations (2) and (3).

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

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

i<m and j=m, namely, one redundant disk and one data disk have failed. We can reconstruct disk i as follows: let

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

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

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

for

$$0 \le k \le m-2,$$

and $a_{k,m}$, $0 \le k \le m-2$, is obtained using Equation (1) once disk i is reconstructed.

i<m and j=m+1, namely, one redundant disk and one data disk have failed. We can reconstruct disk i using Equation (1) and disk $m+1$ using Equations (2) and (3) once disk i is reconstructed.

i<m and 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:

$$S = \left(\bigoplus_{l=0}^{m-2} a_{l,m} \right) \oplus \left(\bigoplus_{l=0}^{m-2} a_{l,m+1} \right) \quad (6)$$

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

$$S_u^{(0)} = \bigoplus_{\substack{l=0 \\ l \neq i,j}}^{m} a_{u,l} \quad (7)$$

$$S_u^{(1)} = S \oplus a_{u,m+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \neq i,j}}^{m-1} a_{\langle u-l \rangle_m, l} \right), \quad (8)$$

where $0 \leq u \leq m-1$. 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 $0 \leq l \leq m-1$.
2. Let

$$a_{s,j} \leftarrow S^{(1)}_{\langle j+s \rangle_m} \oplus a_{\langle s+(j-i) \rangle_m, i}$$
$$a_{s,i} \leftarrow S_s^{(0)} \oplus a_{s,j}$$

3. Set $s \leftarrow \langle s-(j-i) \rangle_m$.
4. If $s = m-1$ then stop, else go to step 2.

The algorithm 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.

Next we illustrate the decoding algorithm with an example.

Example 2 We again assume that $m = 5$, as in Example 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

The first step is finding the parameter S, which is the exclusive-OR of the last two columns. We see that $S = 1$. This means that the diagonals have odd parity. Now, from the array above and Equations (7) and (8), we find the syndromes. We obtain,

$$S^{(0)} = 0\,1\,0\,1\,0$$

and

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

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

$$\begin{aligned}
a_{2,2} &\leftarrow S_4^{(1)} = 0 \\
a_{2,0} &\leftarrow S_2^{(0)} \oplus a_{2,2} = 0 \\
s &\leftarrow 0 \\
a_{0,2} &\leftarrow S_2^{(1)} \oplus a_{2,0} = 0 \\
a_{0,0} &\leftarrow S_0^{(0)} \oplus a_{0,2} = 0 \\
s &\leftarrow 3 \\
a_{3,2} &\leftarrow S_0^{(1)} \oplus a_{0,0} = 0 \\
a_{3,0} &\leftarrow S_3^{(0)} \oplus a_{3,2} = 1 \\
s &\leftarrow 1 \\
a_{1,2} &\leftarrow S_3^{(1)} \oplus a_{3,0} = 0 \\
a_{1,0} &\leftarrow S_1^{(0)} \oplus a_{1,2} = 1 \\
s &\leftarrow 4 \text{ STOP.}
\end{aligned}$$

The reconstructed array is then

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

4 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 symbol. EVENODD offers great flexibility to implement small writes since the symbols involved can have an arbitrary size. We note that large writes are also easy to implement using EVENODD.

We illustrate the small write operations with an example.

Example 3 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 the encoding procedure, 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

Namely, we had to access only 3 symbols, one from each of 3 disks. However, if we modify symbol (2,2), since it is in diagonal (3,1),(2,2),(1,3),(0,4), according to the encoding procedure, we have to modify symbols (2,5), (0,6), (1,6), (2,6) and (3,6):

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

A possible practical implementation of small write operations 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:

$$a_{i,m} \leftarrow a_{i,m} \oplus a_{i,j} \oplus r_i \quad (9)$$

$$a_{i,m+1} \leftarrow a_{i,m+1} \oplus a_{\langle i-j \rangle_m, j} \oplus r_{\langle i-j \rangle_m} \oplus \oplus a_{m-1-j,j} \oplus r_{m-1-j}\ , \quad (10)$$

where $0 \le i \le m-2$. That is, when a sector is updated, the two corresponding redundant sectors are also updated according to Equations (9) and (10). Namely, this implementation has the advantage that every small write involves updates in only two other disk sectors.

5 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 [11]. 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 2D scheme of [8] 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 $\sqrt{m}$, 2D 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:

$$H = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 & 1 & 0 \\ 1 & \alpha & \alpha^2 & \dots & \alpha^{m-1} & 0 & 1 \end{pmatrix} \quad (11)$$

At the encoding, if $b_0, b_1, \dots, b_{m-1}$ is a string of information bytes, according to (11), the redundant bytes p and q are obtained as follows:

$$p = \bigoplus_{i=0}^{m-1} b_i \quad (12)$$

$$q = \bigoplus_{i=0}^{m-1} b_i \alpha^i \quad (13)$$

Now, if we compare with the encoding procedure of EVENODD given by Equations (1) and (2), we can see that Equations (1) and (12) are equivalent. Therefore, the difference in complexity at the encoding is a result of the difference in computing the second redundancy disk, namely, Equations (2) and (13). 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 Equation (1) or Equation (12) at the bit level is $8(m-1)^2$.

Let us count next the number of XOR's in Equation (2) of EVENODD. The first step is computing the symbol S, which is given by Equation (3). 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 Equation (2), 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 $m-2$, Equation (2) takes $8(m-1)m$ XOR operations. Adding to the number of XOR operations used in computing S, Equation (2) 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 Equation (1). The discrepancy is due to the calculation of S first, but we cannot do better than quadratic complexity. By adding the total from Equation (1), 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 Equation (13). Each multiplication of a byte by α, is represented by the following companion matrix A:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \quad (14)$$

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 (13) on the bytes $b_0, b_1, \ldots, b_{m-1}$ takes

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

# 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

Table 1: Number of XOR operations needed to encode $(m-1)$ bytes per disk in a disk array with m information disks.

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 $4096(m-1)^2$ XOR operations from (12), 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 1 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 1 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 1 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

# 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

Table 2: Comparison of the number of XOR operations in a simple parity scheme with EVENODD and RS schemes.

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 complex than the simple parity scheme. Table 2 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.

6 Performance

For performance, we assume an implementation where the symbol is an 8-bit byte, m=257, and the array may have up to 259 disks (though it will typically have far fewer disks and the remaining columns will be treated as having zeroes). As we have discussed before, with such an implementation, a small write requires 3 read-modify-write (RMW) operations on 3 different disks; 1 RMW to the data sector on 1 disk, 1 RMW to the sector containing horizontal parity on a second disk, and 1 RMW to the sector containing diagonal parity on a third disk. As in RAID-5, we assume that the horizontal and diagonal parities are rotated across all disks. Consider a 5 disk system (assume zeroes for all the other 254 columns). Then, a typical layout would be as follows. The first sector on disks 0, 1 and 2 contain user data, the first sector of disk 3 contains horizontal parity, and the first sector of disk 4 contains diagonal parity. The second sector on disks 1, 2 and 3 contain user data, the second sector on disk 4 contains horizontal parity, and the second sector on disk 0 contains diagonal parity. This layout is continued (with the positions of data and parity being rotated every sector) until all the disk sectors have been accounted for. If there are t sectors in a disk, this 5 disk system can store $3t$ sectors of user data.

Consider a write to disk 0, sector 0. This requires us to read and write to three disk sectors on three disks; specifically, it requires us to read and write sector 0 on disks 0, 3 and 4. In disk terms, we need to do 3 read-modify-write operations, one on each of 3 disks. The total time for a read-modify-write operation is seek time plus 1/2 a revolution (to get to disk sector of interest) plus one revolution (to come back to sector of interest for writing) plus one sector write. For a typical disk today, seek time is 10 msecs and revolution time is 11 msecs. So, a read-modify-write takes 10+5.5+11 or 26.5 msecs, ignoring the 1 sector write time which is very small (0.1 msecs or so). A simple read (or write) operation takes 10+5.5 or 15.5 msecs, again ignoring the 1 sector read or write time. So, for our calculations, we assume a RMW takes 26.5/15.5 or 1.71 as long as a simple read or write operation.

Let us compare the performance of our scheme to that of a traditional RAID-5 system with single parity. In particular, we compare our 5 disk system to a 4 disk (3+P) RAID-5 system, since both systems can store 3 disks worth of user data. Of course, our system is more expensive, but this is justified by the extra reliability we provide for the user's data. There is no difference in performance if the workload only consists of large reads and large writes. Differences occur only if the workload has small reads and writes. Here we consider a database type of workload consisting only of small reads and small writes. In order to simplify our analysis, and since this is reasonably accurate, we treat each

read-modify-write operation as 1.71 disk operations. Then, a small read incurs 1 disk operation. A small write incurs 1.71 disk operations on each of 2 disks for RAID-5 and 1.71 disk operations on each of 3 disks for EVENODD. Let the fraction of reads in the workload be r (fraction of writes is $1-r$). Then, each disk in the 4 disk RAID-5 system handles $r/4+(2*1.71)(1-r)/4$ disk operations and each disk in the 5 disk EVENODD system handles $r/5 + (3 * 1.71)(1 - r)/5$ disk operations per IO request. A typical disk today can handle 50 disk operations per second, and r, the fraction of reads is 0.75 for typical workloads. Then, it is easy to see that both the RAID-5 and the EVENODD systems can support a maximum throughput of about 124 IOs/sec. A more realistic comparison is between a 16 disk RAID-5 and a 17 disk EVENODD scheme. In this case, the RAID-5 can support 498 IOs/sec and the EVENODD scheme can support 418 IOs/sec. It is worth pointing out, however, that if the read fraction is 1, EVENODD can support 850 IOs/sec which is better than the 800 IOs/sec which RAID-5 can support.

Throughput is one aspect of performance. Another is response time. In an EVENODD array, there is a potential for the small write response time to be quite high, because of the 3 different RMW operations that must be completed. In fact, as we know, the small write response time of a RAID-5 array is itself high because of the 2 different RMW operations needed. In [12], we show that a write cache built of Non-Volatile memory (or battery-backed DRAM) is a very effective way to improve write response times of a RAID-5 array. We believe that a write cache is also a very effective way to improve write response times of EVENODD arrays.

To summarize this section, in spite of the fact that the EVENODD approach provides higher reliability and requires more disk operations per write, a 5 disk EVENODD system can provide the same throughput as a 4 disk RAID-5 system on database type workloads. In general, however, RAID-5 systems will have better performance for larger arrays (for example 15+P versus 17 disk EVENODD) and when the small write content of the workload is greater than 0.25 while EVENODD will have better performance when the small write content of the workload is very small and there are fewer disks in the array. If response times are important, the EVENODD array should be constructed with a Non-Volatile write cache, just as we believe that a RAID-5 array should be constructed with a Non-Volatile write cache if we want excellent write response times.

7 Concluding Remarks

We have presented a novel method, called EVENODD, that is the first known scheme for tolerating double disk failure in RAID architectures that is optimal with regard to both storage and performance. 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 and is independent of data striping technique. 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 constraints 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 $(m-2,1),(m-3,2),\ldots,(0,m-1)$ 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 employes 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 number of exclusive-OR operations required for 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 convolutional codes. For the codes in [7, 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 involve recursive computation at the encoding and during small write operations. EVENODD has independent parities, making the complexity even smaller.

From the perspective of error-correcting codes, we have constructed a new code that is capable of correcting two erasures. Recently, we have generalized the EVENODD scheme to deal with more than two erasures. It turned out, that the natural generalization works for the case of 3 erasures as well as for 4 erasures (in most cases of m) [3]. 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.

References

[1] M. Blaum, "A Class of Byte-Correcting Array Codes," IBM Research Report, RJ 5652, May 1987.

[2] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures," IBM Research Report, RJ 9506, September 1993.

[3] M. Blaum, J. Bruck and A. Vardy, "Binary Codes with Large Symbols," to appear in the Proceedings of the 1994 IEEE International Symposium on Information Theory, June 1994.

[4] M. Blaum, J. Menon, R. Mattson and H. Hao, "Method and Means for Encoding and Rebuilding Data Contents of up to 2 Unavailable DASDs in an Array of DASDs", Patent 5,271,012 issued on Dec 1993.

[5] M. Blaum and R. Roth, "New Array Codes for Multiple Phased Burst Correction," IEEE Trans. on Information Theory, pp. 66-77, January 1993.

[6] W. Burkhard and J. Menon, "Disk Array Storage System Reliability," 23rd Annual International Symposium on Fault-Tolerant Computing, June 1993, Toulouse, France.

[7] T. Fuja, C. Heegard and M. Blaum, "Cross Parity Check Convolutional Codes," IEEE Trans. on Information Theory, pp. 1264-1276, July 1989.

[8] 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, December 1988.

[9] R. Goodman and M. Sayano, "Size Limits on Phased Burst Error Correcting Array Codes," Elec. Lett., 26 (1990), 55–56.

[10] R. Goodman, R. J. McEliece and M. Sayano, "Phased Burst Correcting Array Codes," IEEE Trans. on Information Theory, pp. 684-693, March 1993.

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

[12] J. Menon, "Performance of RAID5 Disk Arrays With Read and Write Caching," to appear in the Journal of Distributed and Parallel Databases, Volume 2, Number 3, July 1994; also appeared as IBM Research Report, RJ 9485, Aug 1993.

[13] S. W. Ng, "Some Design Issues of Disk Arrays," IBM Research Report, RJ 6590 (63550), December 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., 29 (1985), 546–562.

[16] D. A. Patterson, G. A. Gibson and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks," SIGMOD International Conference on Data Management, pp. 109-116, Chicago, 1988.

[17] P. Prusinkiewicz and S. Budkowski, "A Double Track Error-Correction Code for Magnetic Tape," IEEE Trans. on Computers, pp. 642-645, June 1976.