Decoding STAR Code for Tolerating Simultaneous Disk Failure and Silent Errors

Jianqiang Luo
Wayne State University
Detroit, MI 48202
jianqiang@wayne.edu

Cheng Huang
Microsoft Research
Redmond, WA 98052
cheng.huang@microsoft.com

Lihao Xu
Wayne State University
Detroit, MI 48202
lihao@cs.wayne.edu

Abstract

As storage systems grow in size and complexity, various hardware and software component failures inevitably occur, resulting in disk malfunction in failures*, as well as* silent errors*. Existing techniques and schemes overcome the failures and silent errors in a separate fashion. In this paper, we advocate using the STAR code as a unified and systematic mechanism to* simultaneously *tolerate failures on one disk and silent errors on another. By exploring the unique geometric structure of the STAR code, we propose a novel efficient decoding algorithm –* EEL*. Both theoretical and experimental performance evaluations show that EEL constantly outperforms a naive* Try-and-Test *approach by large factors in overall decoding throughput.*

1 Introduction

As storage systems grow in size and complexity, various hardware and software component failures inevitably occur [18, 23, 2, 13], which often result in various user data losses and errors [3, 1, 2]. From an application's point of view, a data loss can be either *failure* or *silent error*. Here a failure refers to a data loss with explicit error report from a disk drive to the whole system or application. The simplest and most common one is an entire disk drive failure in a *fail-and-stop* fashion, resulting in whole data loss on the disk. Another failure case is the *latent sector failure* within a disk drive [1, 2, 22], which can also be detected and reported during the scrubbing process using a disk driver's internal *error control codes*[1] [24]. A silent error, however, refers to a data *corruption* without any error indication from the disk drive itself to the system. This type of error includes corrupted data, torn writes, lost writes, misdirected writes and wrong reads, which usually results from various bugs in the firmware on disk controllers and various other related software in the storage stacks along a data I/O path [21, 2, 13, 11, 9].

[1]Error control code is a general term for erasure correcting code, error detecting code and error correcting code.

Failures have been known for a long time, and various techniques have been developed to cope with them, from data mirroring to simple replication to more advanced error control code based RAID-5 and RAID-6 type of data protection schemes [7, 5, 10, 4, 8]. In contrast, silent errors are less well-understood and not as sufficiently accounted for. By detecting and converting silent errors into failures, *checksum* at disk sector or block level is an effective and the most common technique to combat silent errors. Unfortunately, recent studies [2, 13, 28] show that checksum may not be sufficient by itself. For example, a silent error from a write to a wrong sector or block due to a firmware bug cannot be detected by the checksum on the very sector or block at all. As a result, various band-aids have been proposed as additional measures to deal with silent errors. These proposals include write verification, physical and logical identity, version mirroring, and implementing checksum in file system level [2, 13, 6].

From a different perspective, now that error control codes [4, 8] have been used in storage systems to tolerate disk failures, it's natural to ask: why not extend error control codes to deal with silent errors as well? Indeed, for storage systems which already employ error control codes for data reliability, we advocate using these codes to overcome both failures and silent errors simultaneously as a more unified and systematic mechanism. It has been recently shown that RAID-6 codes have limitations on data reliability as hard disk capacity grows dramatically, and it calls for using triple parity codes to address this problem [14]. The *STAR* code [12] is such a code and a very suitable candidate for such purpose. The STAR code was recently introduced to tolerate three simultaneous disk failures in a storage system. While initially designed to tolerate only disk failures, well known results from information theory dictate that it can also protect simultaneous failures on one disk and silent errors in another disk [17]. Note that this is a common property of any code which can tolerate three disk failures, not unique to the STAR code, as will be elaborated in Section 3.1. Nevertheless, the STAR code is unique in its geometric structure, which allows us to design a special decoding algorithm that can recover failures and errors very efficiently. Therefore, the very focus of this paper is to provide such a decoding algorithm so that the STAR code can be used to effectively and efficiently deal with failures and silent errors at the same time, and thus significantly improve the data reliability of storage systems.

The decoding algorithm can be used in two situations. One situation is during an inter-disk scrubbing process. Different from intra-disk scrubbing process which uses checksums at sector or block level to detect disk errors, an inter-disk scrubbing process collects data from multiple disks and check the data consistency according to certain constraints, such as the parity constraint imposed by the STAR code. In this situation, even if failures are present, the decoding algorithm can still detect silent errors and correct them. The second situation is in a data reconstruction process. When one disk completely fails, temporal correlations can result in the probability of silent errors in other similar disks used in the same system much higher than normal [2, 13]. Hence, when recovering the disk failure, using the decoding algorithm to perform error detection and correction at the same time can achieve much higher data reliability.

The main contributions of this paper include: 1) the design of an efficient decoding algorithm named *EEL* (Efficient Error Locating) for the STAR code to tolerate one failure and one silent error at the same time; 2) a rigorous correctness proof of the decoding algorithm; and 3) performance evaluation of the decoding algorithm with thorough comparisons to a naive try-and-test decoding algorithm for the same purpose.

2 Related Work

Directly related to this work is certainly the STAR code design [12]. The STAR code is designed to tolerate three disk failures, and it has shown much better encoding/decoding performance than other similar codes [12]. The decoding algorithm presented in [12], however, can only correct up to three disk failures. In coding theory, though, various decoding algorithms have been designed to correct erasures and errors at the same time for certain codes, such as the BCH code and the Reed-Solomon code [17, 25]. However, there is *no* general erasure-and-error decoding algorithm for any code, except the naive one which will be described and compared later in Section 5. It is worth noting that any RAID-6 code can be used to correct one silent disk error in a RAID type system. Indeed such decoding algorithms, namely, error correcting algorithms, have been designed and published for some of the existing RAID-6 codes, such as the EVENODD code [4], the X-Code [26] and the B-Code [27]. Such decoding algorithms can be designed for other RAID-6 codes, such as the RDP code [8] and the Liberation code [19], which the corresponding original papers did not provide. However, if there is a disk failure, RAID-6 codes can not correct any disk error, thus causing data lost.

The results presented in this paper complement various techniques [6, 21, 13] developed to cope with silent errors. All these techniques, such as employing checksum technique in file system, only use the redundancy within a disk to detect errors; however, our approach utilizes inter-disk redundancy of the STAR code. Hence, they are orthogonal and can be used together. In fact, use of the STAR code with our 1-erasure-and-1-error decoding algorithm will address the *parity pollution* issue as raised in [13], where a silent error in a data block spreads to other data blocks through various parity (checksum) calculations. With the STAR code, silent errors can be corrected within their data blocks without further spreading to other blocks.

3 Basics of Coding Theory and the STAR Code

In this section, we first list some related basic terms and results from information and coding theory as what can be achieved by using an error control code, and then give a brief description of the STAR code.

3.1 Related Coding Theory Terms and Results

An (n, k) error control code uses mathematical means to transform a user data of k symbols into a block of n (same size) symbols by adding $(n-k)$ *parity* symbols. The resulting n-symbol block is called a *codeword*. This computation process of obtaining a codeword from k data symbols is called an *encoding* operation. Each parity symbol is computed using a parity constraint (usually a linear equation) from a subset of the k data symbols and the parity symbol. In a storage system, when data is read from a disk, some of its symbols may get lost, and some other symbols may get corrupted. A lost symbol is called an *erasure*, and a corrupted symbol is called an *error* in coding theory. The difference between an erasure and an error is that the erasure's location in its codeword is *known* while the error's is *unknown*. Neither of their values is known. Hence when an error control code is used in a storage system, an erasure corresponds to a disk failure, and an error corresponds to a silent error. Obviously an erasure is easier to deal with than an error, since only its value needs to be *recovered*. To *correct* an error, though, its location needs to be first identified before its value is recovered.

When it is used to deal with errors, an error control code can be used to *detect* and/or *correct* certain number of errors within its designed error control capability using different *decoders* (or *decoding algorithms*). A code's error control capability is mainly measured by its *minimum distance* [17]. The larger distance a code has, the more error control capability it has. Also a code's minimum distance depends on the number of its parity symbols. Usually, though not always, the more parity symbols a code has, the larger minimum distance it can have.

The simplest decoder is a pure *error detection decoder*, which only decides whether there is error in a received codeword[2] or not, by analyzing at least $k+1$ symbols. However, it does *not* and cannot tell the number of errors in the codeword. Hence the output from an error detection decoder is only one bit: whether the received codeword is a valid codeword or not. Such a decoder usually can be realized very efficiently by simply checking if *all* the parity constraints of the code are satisfied. The parity constraints can be checked by computing *syndromes*. The syndrome associated with a parity symbol is a function, usually just the XOR sum, of the parity symbol and all the data symbols it is derived from. For example, if $p = a_1 \oplus a_2$, where a_1 and a_2 are two data symbols, and p is a parity symbol, then the corresponding syndrome s is $s = p \oplus a_1 \oplus a_2$, where $\oplus$ is the binary exclusive-XOR operation. A nonzero syndrome indicates there is error in the codeword.

Upon error detection, an *error correction decoder* can be further invoked to locate and correct error(s) in a received codeword. An error correction decoder corrects all the errors within its designed correction capability. When there are more errors in a received codeword, however, it can declare a *decoding failure* event, which simply reports that there are too many errors for the decoder to correct. This is a useful piece of information. It is then upon the upper layer to decide what to do next, e.g., re-read the whole codeword in a storage system, or simply discard the whole erroneous codeword and inform the user/application.

When there are more errors in a received codeword than the code can cope with, it is very often that a detection decoder is fooled into regarding the received codeword as a valid codeword, albeit one different from the original codeword. For example, for a single-parity code used in RAID-5, an error detection decoder has no way to tell a codeword with no error from a codeword with *even* number of errors. When this event happens, it is called a *decoding error*. A decoding error can similarly occur to an error correction decoder where the decoder "corrects" a received codeword into a valid but different codeword from the original (intended) one. The consequences of a decoding error event in a data system are usually severe, since the system then uses wrong data without knowing it, often resulting in system/application crash. Hence a *decoding error* should be avoided as much as possible.

[2]We borrow this term from communication systems to refer to a codeword read from a disk, even though in storage systems a data I/O unit is a block rather than a codeword, as a codeword can be easily extracted and assembled from a data block.

Depending on a system's needs, hybrid decoders can be designed to deal with error detection, error correction and erasure recovery *simultaneously*, as long as the following information theoretical bound is met [17, Ch.1.3]: $d + e + E \leq D - 1$, and $e \leq d$, where d is the number of *detectable* errors, e the number of *correctable* errors, E the number of *recoverable* erasures, and D the *minimum distance* [17, Ch.1.3] of the code. For example, for a code with its minimum distance of four (such as the STAR code), the code can 1) recover up to three erasures, or 2) recover one erasure *and* correct another one error *simultaneously* by using a *different* decoder. The very focus of this paper is on the design of an efficient decoding algorithm to handle the second case for the STAR code.

3.2 Notations

Table 1 lists all the notations to be used in the rest of the paper. A letter in lower case denotes a symbol or a value, such as $a_{i,j}$, and a letter in upper case denotes a column, such as C_j. All the symbols are within one codeword.

Notation	Definition
$\langle x \rangle_p$	x *mod* p
$a_{i,j}$	original symbol at row i and column j
$c_{i,j}$	symbol read at row i and column j
C_j	column read (of symbols) at index j
$e_{i,j}$	error symbol at row i and column j
E_j	column (of error symbols) at index j
S_j	column (of syndromes) at index j
$\bigoplus_i a_i$	XOR all symbol a_i's (return one symbol)
$\bigoplus_j C_j$	XOR all column C_j's (return one column)
$C \bigoplus a$	XOR each symbol in column C with symbol a (return one column)
$C_i \bigoplus C_j$	XOR column C_i and C_j (return one column)
$f_\downarrow(C, i)$	cyclic shifting column C downward by i positions

Table 1. Notations defined

Note that $a_{i,j}$ denotes an original symbol. It is the correct value in a codeword. $c_{i,j}$ denotes the symbol read from a disk. It may be unknown due to a disk failure or incorrect due to a silent error. $e_{i,j}$ denotes the error symbol causing $a_{i,j}$ flip to $c_{i,j}$. If $e_{i,j}$ is 0, then there is no error; otherwise, the read symbol is corrupted with $c_{i,j} = a_{i,j} \oplus e_{i,j}$.

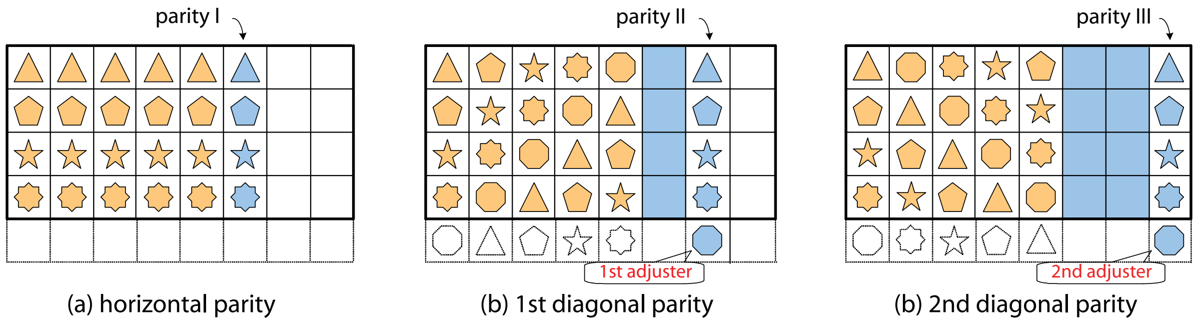


Figure 1. Construction of the STAR code

3.3 STAR code: A Brief Description

The STAR code can be described by a $(k-1) \times (k+3)$ 2-dimensional array. For best storage and computation performance, $k = p$, where p is a prime number, though a general STAR code for arbitrary k can be easily derived from its closest p [12, 20]. For simplicity, we only limit our discussion for $k = p$ throughout the paper.

A STAR codeword consists of p user data columns and 3 parity columns. The $1st$ parity column is a *horizontal* parity column, which is calculated by XORing all the data symbols in the same row. The $2nd$ parity column is a *diagonal* parity column. Its computation is as follows. First, an *adjuster* is computed from the data symbols along the main diagonal of slope 1. Second, the data symbols along other slope 1 diagonals are computed as diagonal parity symbols. Third, the adjuster is complemented to all the diagonal parity symbols. The $3rd$ parity column follows a similar construction as the $2nd$ parity column, except that it is computed along diagonals of slope -1.

Figure 1 shows the construction of the STAR code for $p = 5$. Together with the $(5-1) \times (5+3)$ two dimensional array, the figure also contains an *imaginary* $5th$ row, where all the data symbols are set to 0. It is shown only to help understand the adjuster and parity calculation. Without the $3rd$ parity column, the STAR code reduces to the $(p+2, p)$ EVENODD code [4]. The algebraic construction of the three parity columns is defined as follows ($0 \le i < p-1$):

$$a_{i,p} = \bigoplus_{j=0}^{p-1} a_{i,j};$$

$$a_{i,p+1} = t_1 \oplus \left(\bigoplus_{j=0}^{p-1} a_{<i-j>_p,j} \right), \text{where } t_1 = \bigoplus_{j=0}^{p-1} a_{<-1-j>_p,j};$$

$$a_{i,p+2} = t_2 \oplus \left(\bigoplus_{j=0}^{p-1} a_{<i+j>_p,j} \right), \text{where } t_2 = \bigoplus_{j=0}^{p-1} a_{<-1+j>_p,j}.$$

Here, t_1 and t_2 are the *adjusters* for the $1st$ and $2nd$ diagonal parity columns, respectively.

When a column is treated as a *super* symbol, the STAR code is a $(p+3, p)$ code, and its minimum (column) distance is four [12]. (When used in storage systems, a column usually is mapped to a disk drive.) An efficient decoding algorithm for recovering three erasures for the STAR code was presented in [12]. In this paper, however, our focus is on how to simultaneously correct one erasure and one error for the STAR code.

4 Error Detection for the STAR code

In general, correcting a codeword with errors involves two steps: *error detection* and *error correction*. The error detection step determines whether there is any error in a codeword, and, if so, the error correction step is invoked to correct the error. Depending on the positions of erasure and error, both the detection and correction algorithms vary slightly. Nevertheless, the essence stays the same. Hence, for illustration purpose, it's sufficient to focus on one single error type in the rest of the paper, namely, *both the erasure and error columns are among the data columns.*

Assume the erasure column is C_u. Let R_u^0 denote the column recovered from the horizontal parity column, R_u^1 from the $1st$ diagonal parity column and R_u^2 from the $2nd$ diagonal parity column. R_u^0, R_u^1 and R_u^2 can be calculated as follows:

$$R_u^0 = \bigoplus_{j=0, j\neq u}^{p} C_j;$$

$$R_u^1 = f_\downarrow(C_{p+1} \bigoplus \left(\bigoplus_{j=0, j\neq u}^{p-1} f_\downarrow(C_j, j) \right) \bigoplus r_u^1, -u),$$

$$\text{where } r_u^1 = \bigoplus_{j=0}^{p-1} c_{<-1-j+u>_p,j} \bigoplus c_{<-1+u>_p,p+1};$$

$$R_u^2 = f_\downarrow(C_{p+2} \bigoplus \left(\bigoplus_{j=0, j\neq u}^{p-1} f_\downarrow(C_j, -j) \right) \bigoplus r_u^2, u),$$

$$\text{where } r_u^2 = \bigoplus_{j=0}^{p-1} c_{<-1+j-u>_p,j} \bigoplus c_{<-1-u>_p,p+2}.$$

Although R_u^0, R_u^1 and R_u^2 are computed from different parity columns, they represent the same data column. Hence, we can simply compare them to detect whether there is an error in the codeword. If they are all equal, then there is no error. The erasure column can be simply recovered by setting it to R_u^0, and the decoding process completes. Otherwise, there must exist at least one error column.

5 A Naive Decoding Algorithm: Try-and-Test

As there is no published erasure-and-error decoding algorithm to compare with, we first describe a simple decoding algorithm. The idea is straightforward: simply use a *try-and-test* approach – whenever an error is detected, the algorithm tests each of the survival columns sequentially until the error column is found. For each column being tested, the algorithm first treats it as another erasure column and then checks *parity consistency* of the remaining columns.

The parity consistency is checked as follows. Assume both u and v are data columns. u denotes the original erasure column and v denotes the tested error column. Then, all the three parity columns are available. The Try-and-Test approach uses the first two parity columns to decode column u and v, following the erasure decoding algorithm of the STAR code [12]. It then re-encodes the $3rd$ parity column from all the data columns, and compares it with the original one. If they are equal, then column v is indeed the error column, and as a byproduct, both columns u and v are recovered during this process. Otherwise, it tests the next column similarly until the error column is found or all the columns are tested. If all the columns are tested, but none of them can be deemed as an error, then there are more than one error column, and the decoding algorithm declares a decoding failure event.

The above parity consistency check can be readily generalized to other cases, where both column u and v are parity columns, or one is a data column while the other is a parity column. In addition, interested readers can easily prove that the Try-and-Test approach can indeed correct one erasure column and one error column for the STAR code. we simply skip these simple but tedious discussions here due to space limit.

6 The *EEL* Algorithm

Now we propose a new decoding algorithm - the *EEL* (*Efficient Error Locating*) Algorithm. The *EEL* Algorithm leverages the unique intrinsic geometric structure of the STAR code and locates the error column without performing the Naive Algorithm's *try-and-test* operation in locating the error column, which in turn greatly improves the decoding efficiency. Although the *EEL* Algorithm presented here is for *1-erasure-and-1-error* case, the algorithm can be simplified for *1-error* case. The details are left to interested readers again.

The *EEL* Algorithm consists of three steps: computing syndrome, locating error column, and recovering erasure and error columns.

6.1 Syndrome Computation

Recall from Section 3.1, as it has three parity constraints, the STAR code has three syndromes, and they can be computed as follows, where syndrome S_0 represents the horizontal parity constraint, S_1 the $1st$ diagonal parity constraint, and S_2 the $2nd$ parity constraint.

$$S_0 = \bigoplus_{j=0}^{p} C_j;$$

$$S_1 = C_{p+1} \bigoplus \left(\bigoplus_{j=0}^{p-1} f_\downarrow(C_j, j) \right) \bigoplus t_1,$$

$$\text{where } t_1 = \bigoplus_{j=0}^{p-1} c_{<-1-j>_p,j};$$

$$S_2 = C_{p+2} \bigoplus \left(\bigoplus_{j=0}^{p-1} f_\downarrow(C_j, -j) \right) \bigoplus t_2,$$

$$\text{where } t_2 = \bigoplus_{j=0}^{p-1} c_{<-1+j>_p,j}.$$

We note that the syndrome computation is time consuming, as its complexity is in the order of p^2. Fortunately, it is possible to leverage the results produced in the error detection step and thus greatly speedup this computation. In the rest of this section, we focus on the most common and hardest erasure and error pattern, where both the erasure and error are data columns. Again, u denotes the index of the erasure column, and no data is available from column u. Hence, we set $C_u = 0$.

We obtain the following relationship between the syndromes S_j's and R_j's, the results from the error detection step:

$$S_0 = R_u^0;$$

$$S_1 = f_\downarrow(R_u^1 \bigoplus r_u^1 \bigoplus t_1, u);$$

$$S_2 = f_\downarrow(R_u^2 \bigoplus r_u^2 \bigoplus t_2, -u).$$

Here, t_1, t_2, r_u^1, and r_u^2 are calculated similarly as in the syndrome computation.

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	$v_3+u_3+v_2$	$u_1+v_2+u_0+v_1$
1		u_1	v_1			u_1+v_1	$u_0+u_3+v_2$	$u_2+v_3+u_0+v_1$
2		u_2	v_2			u_2+v_2	$u_1+v_0+u_3+v_2$	$u_3+u_0+v_1$
3		u_3	v_3			u_3+v_3	$u_2+v_1+u_3+v_2$	$v_0+u_0+v_1$
4								

(a) Step 1: compute syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	$v_3+u_3+v_2$	$u_1+v_2+u_0+v_1$
1		u_1	v_1			u_1+v_1	$u_0+u_3+v_2$	$u_2+v_3+u_0+v_1$
2		u_2	v_2			u_2+v_2	$u_1+v_0+u_3+v_2$	$u_3+u_0+v_1$
3		u_3	v_3			u_3+v_3	$u_2+v_1+u_3+v_2$	$v_0+u_0+v_1$
4							u_3+v_2	u_0+v_1

(b) Step 2: compute adjuster syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	v_3	u_1+v_2
1		u_1	v_1			u_1+v_1	u_0	u_2+v_3
2		u_2	v_2			u_2+v_2	u_1+v_0	u_3
3		u_3	v_3			u_3+v_3	u_2+v_1	v_0
4							u_3+v_2	u_0+v_1

(c) Step 3: cancel adjuster syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	v_0	v_0+v_1
1		u_1	v_1			u_1+v_1	v_0+v_1	v_1+v_2
2		u_2	v_2			u_2+v_2	v_1+v_2	v_2+v_3
3		u_3	v_3			u_3+v_3	v_2+v_3	v_3
4							v_3	v_0

(d) Step 4: cancel erasure symbols

Figure 2. Locating error column in the *EEL* Algorithm

6.2 Locating the Error Column

6.2.1 An Example

Locating the error column is the key step in our decoding algorithm. It is instructive to use an example to demonstrate how this step works. Again the erasure u and error v are both data columns. For the error column v, we know that each symbol equals the XOR sum of the corresponding original data symbol and error symbol; that is, $c_{i,v} = a_{i,v} \oplus e_{i,v}$. For simplicity, denote $e_{i,v}$ by v_i. For the erasure column u, we set $C_u = 0$, so $a_{i,u} = e_{i,u}$ since $a_{i,u} = e_{i,u} \oplus c_{i,u}$ and $c_{i,u} = 0$. Similarly, denote $a_{i,u}$ by u_i. In the example shown in Figure 2, $u = 1$ and $v = 2$; $+$ denotes the XOR operation. Note here u_i's, v_i's (the values of the erasure and error columns) and v (the error column location) are unknown, but $u = 1$ (the erasure column location) is known.

The error column can be located in the following five steps with explanations:

Step I – compute syndrome. We first show the relationship between symbols in the syndromes and the erasure/error columns. Using the first symbol of S_0 as an example (denoted as $S_{0,0}$). From Section 6.1, we know that $S_{0,0} = \oplus_{j=0}^{5} c_{0,j}$. For columns $j = 0, 3, 4, 5$ (neither erasure nor error), $c_{0,j} = a_{0,j}$. On the other hand, from the STAR code's encoding rule, $a_{0,5} = \oplus_{j=0}^{4} a_{0,j}$. Thus, by simple substitution, we get $S_{0,0} = u_0 \oplus v_0$. The rest of the syndrome symbols can be derived similarly, as shown in Figure 2(a).

Step II – compute adjuster. Now we simplify the syndromes for further calculation. In particular, the *adjuster* for the $1st$ diagonal parity column can be computed by XORing all the symbols in syndrome column S_0 and S_1, as shown in [4, 12]. We place the *adjuster* for S_1 at the last row in S_1. The adjuster for the $2nd$ diagonal parity column can be computed similarly. The results are shown in Figure 2(b).

Step III – complement adjuster. Then, we XOR the *adjuster* in S_1 with the rest of the symbols in S_1. This cancels the *adjuster* from the rest of the syndrome column S_1. The same operation is applied on S_2. Figure 2(c) shows the results.

Step IV – cancel erasure symbols. Now we cancel the symbols of the erasure column from S_1 and S_2, respectively. For S_1, this is achieved by shifting S_1 downwards by $-u$ (or 4) positions and XORed with S_0. For S_2, it is shifted downwards by $u = 1$ positions before XORed with S_0. All shifts are cyclic modular 5. The results are shown in Figure 2(d).

Step V – locate error column. The last step is to locate the error column. We observe that, if S_1 is cyclically shifted downwards by 4 positions, it matches S_2 exactly. In fact, the number of shifts (denoted as shift down position or h) is completely determined by the positions of the erasure and error column. It satisfies the following equation:

$h = -v$ (error position) $+ u$ (erasure position) (mod p)

In this example, $u = 1$, $h = 4$ and $p = 5$, so $v = (u - h) = (1 - 4)$ (mod 5) $= 2$ and the error column is finally located!

6.2.2 The Error Locating Algorithm

The above example illustrates how to locate the error column. The error locating algorithm follows the same steps as those given in the example. Thus, the algorithm is quite straightforward. Appendix A in [15] provides a formal description of the algorithm together with its correctness proof.

6.3 Recovering Erasure and Error Columns

After the error column is located, the next step is to correct both erasure and error columns. An intuitive approach is to treat the error column as another erasure column, then recover them using the STAR erasure decoding algorithm [12]. This approach, however, is not efficient as it does not utilize the intermediate results produced during the above error locating process. Now we present a much more efficient algorithm which can correct the erasure and error columns directly by fully utilizing those intermediate results. We first illustrate the algorithm by completing the previous example and then give a formal description of the algorithm itself.

6.3.1 An Example

We continue the example in Section 6.2.1 to demonstrate how the correcting algorithm works. In Figure 2(d), there are 5 rows in syndrome S_1. We can treat each row as an equation and each error symbol as a variable. Thus, there are totally 5 equations and 4 variables. In row 4, there is only one variable v_3, so we can compute v_3 from the equation represented by row 4. In row 3, there are two variables v_3 and v_2. After v_3 is solved, we can calculate v_2. Following the similar steps, we can solve v_1 from row 2 and v_0 from row 1. Now, all the error symbols in error column 2 are corrected.

The next step is to recover the erasure column 1. In syndrome S_0, there are four rows, and each row is the XOR sum of the symbols from erasure column 1 and error column 2. Each row is again treated as an equation. Since all the error symbols of column 2 are now known, we can compute u_0 from row 0 of S_0, u_1 from row 1, u_2 from row 2, and u_3 from row 4. All the erasure symbols of column 1 are thus recovered.

Finally, the error correction process completes by XORing the error symbols of column 2 with C_2 to recover the original data column 2.

6.3.2 The Erasure and Error Recovery Algorithm

Now we present a formal description of the erasure and error recovery algorithm. The pseudocode of the algorithm is described in Algorithm 1.

Algorithm 1 Recovering Erasure and Error columns

```
/*Step 1: Solve $E_v$, the error symbols*/
    $E_v \bigoplus f_\downarrow(E_v, v-u) = S_0 \bigoplus f_\downarrow(S_1, -u);$

/*Step 2: Recover $E_u$, the erasure column*/
    $E_u = S_0 \bigoplus E_v;$

/*Step 3: Recover the original data column $C_v$*/
    $C_v = C_v \bigoplus E_v.$
```

In Step 1, we have a group of p linear equations generated from the error locating algorithm, and we need to solve p variables v_i's ($0 \le i \le p-1$) from the equations. Observe that

1. each equation is the XOR sum of two variables, one from E_v and the other from $f_\downarrow(E_v, v-u)$;
2. each variable v_i appears exactly twice in the equations;
3. $v_{<-1>_p} = 0$, since it is in the imaginary row $p-1$.

Hence the above equations in turn can be efficiently solved in a zig-zag fashion just as in the erasure decoding for the EVENODD code [4] by the following 3 steps:

1. Step 1a: start from the last row of the equations, which contains two variables $v_{<-1>_p}$ from E_v and $v_{<-1-1*(v-u)>_p}$ from $f_\downarrow(E_v, v-u)$. Since $v_{<-1>_p} = 0$, then the only one unknown variable $v_{<-1-1*(v-u)>_p}$ is solved. And now we are at row $\langle -1-l*(v-u)\rangle_p$ (l=0). Go to Step 1b.
2. Step 1b: row $\langle -1-l*(v-u)\rangle_p$ of the equations, consists of two variables $v_{<-1-l*(v-u)>_p}$ from E_v and $v_{<-1-(l+1)*(v-u)>_p}$ from $f_\downarrow(E_v, v-u)$. Since variable $v_{<-1-l*(v-u)>_p}$ is known, the only one unknown $v_{<-1-(1+1)*(v-u)>_p}$ can be recovered.
3. Step 1c: in step 1b we are at row $v_{<-1-l*(v-u)>_p}$. If $l = p-2$, this process stops; otherwise, go to the next row $\langle -1-(l+1)*(v-u)\rangle_p$ and repeat step 1b.

□

After Step 1 is finished, Step 2 and 3 can be performed to recover the erasure and error columns.

7 Performance Evaluation

Now we evaluate the computation performance of our decoding algorithm by comparing it with the naive decoding algorithm described in Section 5. We first count the number of XOR needed for both algorithms, since XOR is

Erasure	Error	Error Detection	Correction (*EEL*)	Total (*EEL*)	Correction (Naive)	Total (Naive)
Data	No	$3p^2 - 3p$	0	$3p^2 - 3p$	0	$3p^2 - 3p$
Parity	No	$2p^2 - 2p$	$p^2 - p$	$3p^2 - 3p$	$p^2 - p$	$3p^2 - 3p$
Data	Data	$3p^2 - 3p$	$21p - 16$	$3p^2 + 18p - 16$	$6.5p^2 - 6.5p$	$9.5p^2 - 9.5p$
Data	Parity	$3p^2 - 3p$	$20p - 15$	$3p^2 + 17p - 15$	$13p^2 - 22p$	$16p^2 - 25p$
Parity	Data	$2p^2 - 2p$	$p^2 + 14p - 13$	$3p^2 + 12p - 13$	$4.5p^2 - p$	$6.5p^2 - 3p$
Parity	Parity	$2p^2 - 2p$	$p^2 + 4p - 5$	$3p^2 + 2p - 5$	$8p^2 - 8p$	$10p^2 - 10p$

Table 2. Decoding cost comparison (in XORs)

the most frequent operation in decoding and thus dominates decoding performance. Then we measure the wall time of the decoding algorithms through experiments.

7.1 XOR Numbers

There are six possible erasure and error pattern combinations, as listed in the first two columns in Table 2. For both decoding algorithms, the XOR number needed to correct different pattern is different. Therefore, we first compute the number of XORs needed for each pattern. Then, by assuming each column has the same probability to be an erasure or an error column, we calculate the *average* number of XORs as the cost for a decoding algorithm.

7.1.1 XOR number for the *EEL* Algorithm

Table 2 shows that there are four possible erasure and error patterns when there is an error. We stick to the most common one - an erasure and an error in two data columns - to demonstrate how to count the number of XORs needed for decoding. The XOR numbers for the other patterns can be counted similarly.

Recall that there are in total four steps in our EEL decoding algorithm: 1) error detection, 2) syndrome computation, 3) error locating and 4) erasure and error recovery.

1. As shown in Section 4, the *error detection* step computes R_u^0, R_u^1 and R_u^2, each costing $p * (p-1)$ XORs. Hence the total cost for this step is $3p^2 - 3p$. Note the symbols in the last imaginary row are not involved in computation.

2. As discussed in Section 6.1, the *syndrome computation* step computes syndromes S_0, S_1 and S_2 from R_u^0, R_u^1 and R_u^2 obtained in the error detection step. S_0 needs $p-1$ XORs, and S_1 and S_2 each need $4p-2$ XORs. Hence the total cost of this step is $9p - 5$.

3. The *error locating* step first simplifies syndromes S_1 and S_2; each computation needs $4p-3$ XORs. Then vector *equivalence test* is conducted using *shift* and *compare*. In the test, however, no XOR operation is needed. As a result, the total cost for this step is only $8p - 6$ XORs.

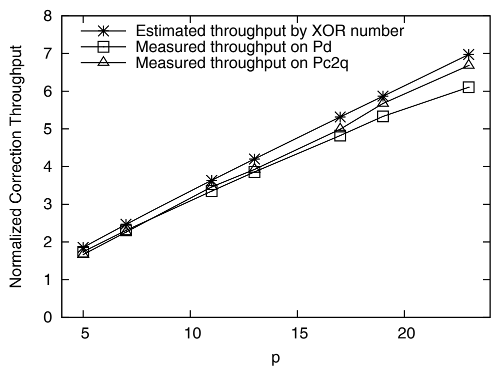
4. The *erasure and error recovery* step needs $2p-3$ XORs in solving the erasure symbol E_u and the error symbol E_v, and then $2p-2$ XORs for recovering the erasure column E_u and error column E_v; hence, $4p - 5$ XORs in total.

Adding all the XOR numbers in the above steps, we get the decoding cost, which is $3p^2 + 18p - 16$ XORs. Similar analysis can be conducted for the other erasure and error patterns, as listed in details in Table 2. If a number contains a constant value less than 5, the constant value is ignored. In Table 2, The first two columns specify an erasure and error pattern. In the second column, a no means there is no error. The $3rd$ column is the XOR numbers needed in the error detection step; the $4th$ column is the number of XORs for the erasure and error correction step; the $5th$ column is the total number of XORs performed in decoding process.

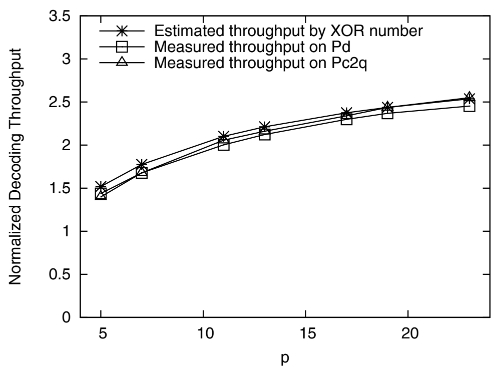
7.1.2 XORs Needed for the Naive Algorithm

The Naive Algorithm also consists of two steps, the *error detection* and the *error correction*. The error detection step is exactly the same as in our decoding algorithm, so we just focus on the error correction step.

We observe that as in the *EEL* Algorithm, the Naive Algorithm can greatly reduce XORs needed in the error correction step by utilizing the results from the error detection step. When the erasure column is a data column, the cost to test whether a data column is an error or not is $13p - 13$ when the results from the error detection step are used. Otherwise that cost would be $3p^2 - 3p$. So the cost reduction is significant. Assume that the average number of *try and test* is $p/2$ since there are in total p data columns, then the total average cost is $(13p - 13) * p/2 = 6.5p^2 - 6.5p$ XORs for correcting when both the erasure and error are data columns. When the error is a parity column, $(13p - 13) * (p - 1)$ XORs are needed on the $p - 1$ data columns, and $4p$ tests on the parity columns, hence the total



(a) Correction throughput of the *EEL* Algorithm



(b) Decoding throughput of the *EEL* Algorithm

Figure 3. Comparison of Throughput

cost is then $13p^2 - 22p$ XORs. Costs for correcting other erasure and error patterns can be counted similarly, as again listed in details in Table 2.

7.1.3 Comparison

When there is one erasure but no error, Table 2 shows that the *EEL* Algorithm performs the same as the Naive Algorithm, since both only conduct the same error detection step without invoking error correction. The real comparison is when there is one error. The table shows that the *EEL* Algorithm greatly outperforms the Naive Algorithm.

7.2 Measured Decoding Throughput

7.2.1 Experiment Setup

We have implemented both decoding algorithms in C language. We then measure their decoding times with random codewords. Codewords are kept in main memory, so there is no disk I/O involved. For a ($p + 3$, p) STAR code, there are in total $p+3$ columns in a codeword. There are thus $p+3$ possible locations for an erasure or an error, and $(p + 3)^2$ combinations for one erasure and one error to occur in a codeword. Note here if the error and the erasure occur on the same column, then that column is treated as an erasure column. In each test, all the $(p + 3)^2$ possible erasure-error patterns are decoded to measure the average decoding time, where the value of an error is randomly generated. Such tests are repeated 3000 times to get stable decoding times to experimentally compare the performance of the two algorithms.

The experiments are conducted on two platforms, one with Intel Pentium Dual Core CPU (named Pd) and the other with Intel Pentium Core 2 Quad CPU (named Pc2d). Both platforms run 64-bit Linux. On both platforms, the decoding algorithms are compiled by gcc with $-O2$ flag, a common choice for the optimization flag [16]. We use $gettimeofday()$ system call to capture the time consumed in decoding, and the time elapsed in the decoding process is used as the decoding time. The ratio of the standard deviation to the average decoding time is at most 5%.

We use an optimization technique employed in [20, 16], where a large packet size can greatly improve encoding/decoding performance. Packet size means how many bytes in one symbol, and large packet size provides good data locality when performing XOR operation. In our experiments, the packet size is set to be 512 bytes so that one symbol naturally maps to one sector in a hard disk.

7.2.2 Decoding Throughput Comparison

Instead of comparing absolute values of XOR numbers and decoding throughput, we normalize the performance of the *EEL* Algorithm by that of the Naive Algorithm. The throughput is defined as the *reciprocal* of the decoding time or the XOR number. Note that both algorithms employ the exact same *error detection* process, and both algorithms include two steps: *error detection* and *error correction*. The error correction step is invoked only when error is detected, which is the focus of this paper.

The decoding throughput in the *error correction* step measured from experiments are plotted in Figure 3(a), together with their corresponding XOR numbers, where the X-axis is the parameter p of a $(p+3, p)$ STAR code, and the Y-axis is the normalized decoding throughput for the error correction step of the corresponding STAR code. On both test platforms, the throughput estimated by the XOR number nicely matches the measured ones in the experiments. As p increases, the normalized throughput of the *EEL* Algorithm also increases.

Finally the *overall* decoding throughput of the *EEL* Algorithm is shown in Figure 3(b), again normalized by that of the Naive Algorithm, together with the corresponding XOR number. The overall decoding operation includes both the error detection step and the error correction step. Again the normalized overall decoding throughput of the *EEL* Algorithm is always greater than one, and also increases as p increases.

8 Conclusions

This paper presents an efficient decoding algorithm for the STAR code to *simultaneously* tolerate one whole disk failure and another silent disk error in a storage system. In addition to a correctness proof of the algorithm, both theoretical analysis and experimental measurement show our decoding algorithm can outperform the best naive decoding algorithm we can think of by large factors in overall decoding throughput, and more in the error correction process.

Our future work is to improve error detection performance. Although the *EEL* Algorithm achieves much better error correction performance than the naive algorithm, the need of performing error correction is relatively rare given that the probability of silent disk errors is low. (Certainly, error correction performance is very important when errors are detected.) A more general operation in a storage system is error detection since it is on regular I/O path and performed more often. Therefore, improving error detection performance would have higher impact in storage system's performance.

Acknowledgments

We would like to thank James Plank for his insightful suggestions on an earlier version of the paper. We also thank our shepherd, Jay Wylie, and anonymous reviewers for their constructive feedback and comments. This research was supported in part by the National Science Foundation under grant IIS-0541527.

References

[1] L. Bairavasundaram, G. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07: Proc. of ACM International Conference on Measurement and Modeling of Computer Systems*, June 2007.

[2] L. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[3] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *EuroSys '06: 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2006.

[4] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers*, 44 (2):192–202, February 1995.

[5] V. Bohossian, C. C. Fan, P. S. LeMahieu, M. D. Riedel, J. Bruck, and L. Xu. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Transaction on Parallel and Distributed Systems*, 12(2):99–114, 2001.

[6] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://www.opensolaris.org/os/community/zfs/docs.

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

[8] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04: Proc. of the 3rd USENIX Conference on File and Storage Technologies*, March 2004.

[9] J. G. Elerath and M. Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *DSN '07: International Conference on Dependable Systems and Networks*, June 2007.

[10] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. In *SOSP '03: Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[11] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao. Undetected Disk Errors in RAID Arrays. *IBM Journal of Research and Development*, 52(4/5):413–425, July/September 2008.

[12] C. Huang and L. Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. *IEEE Transactions on Computer*, 57(7):889–901, July 2008.

[13] A. Krioukov, L. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST '08: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[14] A. Leventhal. Triple-Parity RAID and Beyond. http://queue.acm.org/detail.cfm?id=1670144.

[15] J. Luo, C. Huang, and L. Xu. Decoding STAR Code for Tolerating Simultaneous Disk Failure and Silent Errors. *Tech. Report, http://nisl.wayne.edu*, December 2009.

[16] J. Luo, L. Xu, and J. S. Plank. An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding. In *DSN '09: The International Conference on Dependable Systems and Networks*, June 2009.

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

[18] E. Pinheiro, W. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07: Proc. of the 5th USENIX Conference on File and Storage Technologies*, February 2007.

[19] J. S. Plank. The RAID-6 Liberation Codes. In *FAST '08: Proc. of the 6th Usenix Conference on File and Storage Technologies*, February 2008.

[20] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *FAST '09: Proc. of the 7th Usenix Conference on File and Storage Technologies*, February 2009.

[21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, A. C. Arpaci-Dusseau H. S. Gunawi, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05: Proc. of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[22] B. Schroeder, S. Damouras, and P. Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *FAST '10: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2010.

[23] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST '07: Proc. of the 5th USENIX Conference on File and Storage Technologies*, February 2007.

[24] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, September 2004.

[25] Stephen B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, 1994.

[26] L. Xu. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 45 (1):272–276, January 1999.

[27] L. Xu and J. Bruck. Low Density MDS Code and Factors of Complete Graphs. *IEEE Transactions on Information Theory*, 45:1817–1826, 1999.

[28] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *FAST '10: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2010.