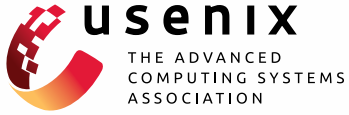


STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures in Practical Storage Systems

Mingqiang Li and Patrick P. C. Lee, ***The Chinese University of Hong Kong***

https://www.usenix.org/conference/fast14/technical-sessions/presentation/li-mingqiang

This paper is included in the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14).

February 17–20, 2014 • Santa Clara, CA USA

ISBN 978-1-931971-08-9

Open access to the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14) is sponsored by

STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures in Practical Storage Systems

Mingqiang Li and Patrick P. C. Lee
The Chinese University of Hong Kong
mingqiangli.cn@gmail.com, pclee@cse.cuhk.edu.hk

Abstract

Practical storage systems often adopt erasure codes to tolerate device failures and sector failures, both of which are prevalent in the field. However, traditional erasure codes employ device-level redundancy to protect against sector failures, and hence incur significant space overhead. Recent sector-disk (SD) codes are available only for limited configurations due to the relatively strict assumption on the coverage of sector failures. By making a relaxed but practical assumption, we construct a general family of erasure codes called *STAIR codes*, which efficiently and provably tolerate both device and sector failures without any restriction on the size of a storage array and the numbers of tolerable device failures and sector failures. We propose the *upstairs encoding* and *downstairs encoding* methods, which provide complementary performance advantages for different configurations. We conduct extensive experiments to justify the practicality of STAIR codes in terms of space saving, encoding/decoding speed, and update cost. We demonstrate that STAIR codes not only improve space efficiency over traditional erasure codes, but also provide better computational efficiency than SD codes based on our special code construction.

1 Introduction

Mainstream disk drives are known to be susceptible to both *device failures* [25, 37] and *sector failures* [1, 36]: a device failure implies the loss of all data in the failed device, while a sector failure implies the data loss in a particular disk sector. In particular, sector failures are of practical concern not only in disk drives, but also in emerging solid-state drives as they often appear as worn-out blocks after frequent program/erase cycles [8, 14, 15, 43]. In the face of device and sector failures, practical storage systems often adopt *erasure codes* to provide data redundancy [32]. However, existing erasure codes often build on tolerating device failures and provide device-level redundancy only. To tolerate additional sector failures, an erasure code must be constructed with extra parity disks. A representative example is RAID-6, which uses two parity disks to tolerate one device failure together with one sector failure in another non-failed device [21, 39]. If the sector failures can span a number of devices, the same number of parity disks must be provisioned. Clearly, dedicating an entire parity disk for tolerating a sector failure is too extravagant.

To tolerate both device and sector failures in a space-efficient manner, sector-disk (SD) codes [27, 28] and the earlier PMDS codes [5] (which are a subset of SD codes) have recently been proposed. Their idea is to introduce parity sectors, instead of entire parity disks, to tolerate a given number of sector failures. However, the constructions of SD codes are known only for limited configurations (e.g., the number of tolerable sector failures is no more than three), and some of the known constructions rely on exhaustive searches [6, 27, 28]. An open issue is to provide a general construction of erasure codes that can efficiently tolerate both device and sector failures without any restriction on the size of a storage array, the number of tolerable device failures, or the number of tolerable sector failures.

In this paper, we make the first attempt to develop such a generalization, which we believe is of great theoretical and practical interest to provide space-efficient fault tolerance for today's storage systems. After carefully examining the assumption of SD codes on failure coverage, we find that although SD codes have relaxed the assumption of the earlier PMDS codes to comply with how most storage systems really fail, the assumption remains too strict. By reasonably relaxing the assumption of SD codes on sector failure coverage, we construct a general family of erasure codes called *STAIR codes*, which efficiently tolerate both device and sector failures.

Specifically, SD codes devote s sectors per stripe to coding, and tolerate the failure of any s sectors per stripe. We relax this assumption in STAIR codes by limiting the number of devices that may simultaneously contain sector failures, and by limiting the number of simultaneous sector failures per device. The new assumption of STAIR codes is based on the strong locality of sector failures found in practice: sector failures tend to come in short bursts, and are concentrated in small address space [1, 36]. Consequently, as shown in §2, STAIR codes are constructed to protect the sector failure coverage defined by a vector $\mathbf{e}$, rather than all combinations of s sector failures.

With the relaxed assumption, the construction of STAIR codes can be based on existing erasure codes. For example, STAIR codes can build on Reed-Solomon codes (including standard Reed-Solomon codes [26, 30, 34] and Cauchy Reed-Solomon codes [7, 33]), which have no restriction on code length and fault tolerance.

We first define the notation and elaborate how the sector failure coverage is formulated for STAIR codes in §2. Then the paper makes the following contributions:

- We present a baseline construction of STAIR codes. Its idea is to run two orthogonal encoding phases based on Reed-Solomon codes. See §3.
- We propose an *upstairs decoding* method, which systematically reconstructs the lost data due to both device and sector failures. The proof of fault tolerance of STAIR codes follows immediately from the decoding method. See §4.
- Inspired by upstairs decoding, we extend the construction of STAIR codes to regularize the code structure. We propose two encoding methods: *upstairs encoding* and *downstairs encoding*, both of which reuse computed parity results in subsequent encoding. The two encoding methods provide complementary performance advantages for different configuration parameters. See §5.
- We extensively evaluate STAIR codes in terms of space saving, encoding/decoding speed, and update cost. We show that STAIR codes achieve significantly higher encoding/decoding speed than SD codes through parity reuse. Most importantly, we show the versatility of STAIR codes in supporting any size of a storage array, any number of tolerable device failures, and any number of tolerable sector failures. See §6.

We review related work in §7, and conclude in §8.

2 Preliminaries

We consider a storage system with n devices, each of which has its storage space logically segmented into a sequence of continuous *chunks* (also called *strips*) of the same size. We group each of the n chunks at the same position of each device into a *stripe*, as depicted in Figure 1. Each chunk is composed of r sectors (or blocks). Thus, we can view the stripe as a $r \times n$ array of sectors. Using coding theory terminology, we refer to each sector as a *symbol*. Each stripe is independently protected by an erasure code for fault tolerance, so our discussion focuses on a single stripe.

Storage systems are subject to both device and sector failures. A device failure can be mapped to the failure of an entire chunk of a stripe. We assume that the stripe can tolerate at most m ($< n$) chunk failures, in which all symbols are lost. In addition to device failures, we

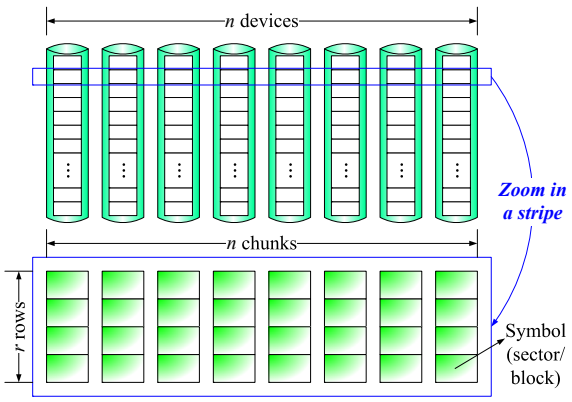


Figure 1: A stripe for $n = 8$ and $r = 4$.

assume that sector failures can occur in the remaining $n - m$ devices. Each sector failure is mapped to a lost symbol in the stripe. Suppose that besides the m failed chunks, the stripe can tolerate sector failures in at most m' ($\leq n - m$) remaining chunks, each of which has a maximum number of sector failures defined by a vector $\mathbf{e} = (e_0, e_1, \cdots, e_{m'-1})$. Without loss of generality, we arrange the elements of $\mathbf{e}$ in monotonically increasing order (i.e., $e_0 \leq e_1 \leq \cdots \leq e_{m'-1}$). For example, suppose that sector failures can only simultaneously appear in at most three chunks (i.e., $m' = 3$), among which at most one chunk has two sector failures and the remaining have one sector failure each. Then, we can express $\mathbf{e} = (1, 1, 2)$. Also, let $s = \sum_{i=0}^{m'-1} e_i$ be the total number of sector failures defined by $\mathbf{e}$. Our study assumes that the configuration parameters n, r, m, and $\mathbf{e}$ (which then determines m' and s) are the inputs selected by system practitioners for the erasure code construction.

Erasure codes have been used by practical storage systems to protect against data loss [32]. We focus on a class of erasure codes with optimal storage efficiency called *maximum distance separable (MDS)* codes, which are defined by two parameters η and κ ($< \eta$). We define an (η, κ)-code as an MDS code that transforms κ symbols into η symbols collectively called a *codeword* (this operation is called *encoding*), such that any κ of the η symbols can be used to recover the original κ uncoded symbols (this operation is called *decoding*). Each codeword is encoded from κ uncoded symbols by multiplying a row vector of the κ uncoded symbols with a $\kappa \times \eta$ *generator matrix* of coefficients based on Galois Field arithmetic. We assume that the (η, κ)-code is *systematic*, meaning that the κ uncoded symbols are kept in the codeword. We refer to the κ uncoded symbols as *data* symbols, and the $\eta - \kappa$ coded symbols as *parity* symbols. We use systematic MDS codes as the building blocks of STAIR codes. Examples of such codes are standard Reed-Solomon codes [26, 30, 34] and Cauchy Reed-Solomon codes [7, 33].

Given parameters n, r, m, and $\mathbf{e}$ (and hence m' and s), our goal is to construct a STAIR code that tolerates both m failed chunks and s sector failures in the remaining $n-m$ chunks defined by $\mathbf{e}$. Note that some special cases of $\mathbf{e}$ have the following physical meanings:

- If $\mathbf{e} = (1)$, the corresponding STAIR code is equivalent to a PMDS/SD code with $s = 1$ [5, 27, 28]. In fact, the STAIR code is a new construction of such a PMDS/SD code.
- If $\mathbf{e} = (r)$, the corresponding STAIR code has the same function as a systematic $(n, n-m-1)$-code.
- If $\mathbf{e} = (\epsilon, \epsilon, \cdots, \epsilon)$ with $m' = n-m$ and some constant $\epsilon < r$, the corresponding STAIR code has the same function as an intra-device redundancy (IDR) scheme [10, 11, 36] that adopts a systematic $(r, r-\epsilon)$-code.

We argue that STAIR codes can be configured to provide more general protection than SD codes [6, 27, 28]. One major use case of STAIR codes is to protect against bursts of contiguous sector failures [1, 36]. Let β be the maximum length of a sector failure burst found in a chunk. Then we should set $\mathbf{e}$ with its largest element $e_{m'-1} = \beta$. For example, when $\beta = 2$, we may set $\mathbf{e}$ as our previous example $\mathbf{e} = (1, 1, 2)$, or a weaker and lower-cost $\mathbf{e} = (1, 2)$. In some extreme cases, some disk models may have longer sector failure bursts (e.g., with $\beta > 3$) [36]. Take $\beta = 4$ for example. Then we can define $\mathbf{e} = (1, 4)$, so that the corresponding STAIR code can tolerate a burst of four sector failures in one chunk together with an additional sector failure in another chunk. In contrast, such an extreme case cannot be handled by SD codes, whose current construction can only tolerate at most three sector failures in a stripe [6, 27, 28]. Thus, although the numbers of device and sector failures (i.e., m and s, respectively) are often small in practice, STAIR codes support a more general coverage of device and sector failures, especially for extreme cases.

STAIR codes also provide more space-efficient protection than the IDR scheme [10, 11, 36]. To protect against a burst of β sector failures in *any* data chunk of a stripe, the IDR scheme requires β additional redundant sectors in each of the $n-m$ data chunks. This is equivalent to setting $\mathbf{e} = (\beta, \beta, \cdots, \beta)$ with $m' = n-m$ in STAIR codes. In contrast, the general construction of STAIR codes allows a more flexible definition of $\mathbf{e}$, where m' can be less than $n-m$, and all elements of $\mathbf{e}$ except the largest element $e_{m'-1}$ can be less than β. For example, to protect against a burst of $\beta = 4$ sector failures for $n = 8$ and $m = 2$ (i.e., a RAID-6 system with eight devices), the IDR scheme introduces a total of $4 \times 6 = 24$ redundant sectors per stripe; if we define $\mathbf{e} = (1, 4)$ in STAIR codes as above, then we only introduce five redundant sectors per stripe.

3 Baseline Encoding

For general configuration parameters n, r, m, and $\mathbf{e}$, the main idea of STAIR encoding is to run two orthogonal encoding phases using two systematic MDS codes. First, we encode the data symbols using one code and obtain two types of parity symbols: *row parity symbols*, which protect against device failures, and *intermediate parity symbols*, which will then be encoded using another code to obtain *global parity symbols*, which protect against sector failures. In the following, we elaborate the encoding of STAIR codes and justify our naming convention.

We label different types of symbols for STAIR codes as follows. Figure 2 shows the layout of an exemplary stripe of a STAIR code for $n = 8$, $r = 4$, $m = 2$, and $\mathbf{e} = (1, 1, 2)$ (i.e., $m' = 3$ and $s = 4$). A stripe is composed of $n-m$ data chunks and m row parity chunks. We also assume that there are m' intermediate parity chunks and s global parity symbols outside the stripe. Let $d_{i,j}$, $p_{i,k}$, $p'_{i,l}$, and $g_{h,l}$ denote a data symbol, a row parity symbol, an intermediate parity symbol, and a global parity symbol, respectively, where $0 \le i \le r-1$, $0 \le j \le n-m-1$, $0 \le k \le m-1$, $0 \le l \le m'-1$, and $0 \le h \le e_l - 1$.

Figure 2 depicts the steps of the two orthogonal encoding phases of STAIR codes. In the first encoding phase, we use an $(n+m', n-m)$-code denoted by $\mathcal{C}_{row}$ (which is an (11,6)-code in Figure 2). We encode via $\mathcal{C}_{row}$ each row of $n-m$ data symbols to obtain m row parity symbols and m' intermediate parity symbols in the same row:

Phase 1: For $i = 0, 1, \cdots, r-1$,

$$d_{i,0}, d_{i,1}, \cdots, d_{i,n-m-1} \overset{\mathcal{C}_{row}}{\Longrightarrow} p_{i,0}, p_{i,1}, \cdots, p_{i,m-1}, p'_{i,0}, p'_{i,1}, \cdots, p'_{i,m'-1},$$

where $\overset{\mathcal{C}}{\Longrightarrow}$ describes that the input symbols on the left are used to generate the output symbols on the right using some code $\mathcal{C}$. We call each $p_{i,k}$ a "row" parity symbol since it is only encoded from the same row of data symbols in the stripe, and we call each $p'_{i,l}$ an "intermediate" parity symbol since it is not actually stored but is used in the second encoding phase only.

In the second encoding phase, we use a $(r+e_{m'-1}, r)$-code denoted by $\mathcal{C}_{col}$ (which is a (6,4)-code in Figure 2). We encode via $\mathcal{C}_{col}$ each chunk of r intermediate parity symbols to obtain at most $e_{m'-1}$ global parity symbols:

Phase 2: For $l = 0, 1, \cdots, m'-1$,

$$p'_{0,l}, p'_{1,l}, \cdots, p'_{r-1,l} \overset{\mathcal{C}_{col}}{\Longrightarrow} \overbrace{g_{0,l}, g_{1,l}, \cdots, g_{e_l-1,l}, *, \cdots, *}^{e_{m'-1}},$$

where "$*$" represents a "dummy" global parity symbol that will not be generated when $e_l < e_{m'-1}$, and we only need to compute the "real" global parity symbols $g_{0,l}, g_{1,l}, \cdots, g_{e_l-1,l}$. The intermediate parity symbols will be discarded after this encoding phase. Note that

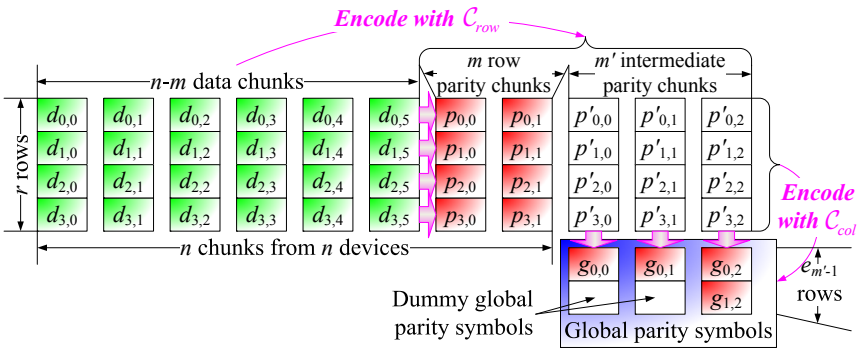


Figure 2: Exemplary configuration: a STAIR code stripe for $n = 8$, $r = 4$, $m = 2$, and $\mathbf{e} = (1, 1, 2)$ (i.e., $m' = 3$ and $s = 4$). Throughout this paper, we use this configuration to explain the operations of STAIR codes.

each $g_{h,l}$ is in essence encoded from all the data symbols in the stripe, and thus we call it a "global" parity symbol.

We point out that $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ can be any systematic MDS codes. In this work, we implement both $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ using Cauchy Reed-Solomon codes [7, 33], which have no restriction on code length and fault tolerance.

From Figure 2, we see that the logical layout of global parity symbols looks like a stair. This is why we name this family of erasure codes *STAIR codes*.

In the following discussion, we use the exemplary configuration in Figure 2 to explain the detailed operations of STAIR codes. To simplify our discussion, we first assume that the global parity symbols are kept outside a stripe and are always available for ensuring fault tolerance. In §5, we will extend the encoding of STAIR codes when the global parity symbols are kept inside the stripe and are subject to both device and sector failures.

4 Upstairs Decoding

In this section, we justify the fault tolerance of STAIR codes defined by m and $\mathbf{e}$. We introduce an *upstairs decoding* method that systematically recovers the lost symbols when both device and sector failures occur.

4.1 Homomorphic Property

The proof of fault tolerance of STAIR codes builds on the concept of a *canonical stripe*, which is constructed by augmenting the existing stripe with additional *virtual parity symbols*. To illustrate, Figure 3 depicts how we augment the stripe of Figure 2 into a canonical stripe. Let $d^*_{h,j}$ and $p^*_{h,k}$ denote the virtual parity symbols encoded with $\mathcal{C}_{col}$ from a data chunk and a row parity chunk, respectively, where $0 \le j \le n-m-1$, $0 \le k \le m-1$, and $0 \le h \le e_{m'-1}-1$. Specifically, we use $\mathcal{C}_{col}$ to generate virtual parity symbols from the data and row parity chunks as follows:

For $j = 0, 1, \cdots, n-m-1$,

$$d_{0,j}, d_{1,j}, \cdots, d_{r-1,j} \overset{\mathcal{C}_{col}}{\Longrightarrow} d^*_{0,j}, d^*_{1,j}, \cdots, d^*_{e_{m'-1}-1,j};$$

and for $k = 0, 1, \cdots, m-1$,

$$p_{0,k}, p_{1,k}, \cdots, p_{r-1,k} \overset{\mathcal{C}_{col}}{\Longrightarrow} p^*_{0,k}, p^*_{1,k}, \cdots, p^*_{e_{m'-1}-1,k}.$$

The virtual parity symbols $d^*_{h,j}$'s and $p^*_{h,k}$'s, along with the real and dummy global parity symbols, form $e_{m'-1}$ augmented rows of $n + m'$ symbols. To make our discussion simpler, we number the rows and columns of the canonical stripe from 0 to $r + e_{m'-1} - 1$ and from 0 to $n + m' - 1$, respectively, as shown in Figure 3.

Referring to Figure 3, we know that the upper r rows of $n + m'$ symbols are codewords of $\mathcal{C}_{row}$. We argue that each of the lower $e_{m'-1}$ augmented rows is in fact also a codeword of $\mathcal{C}_{row}$. We call this the *homomorphic property*, since the encoding of each chunk in the column direction preserves the coding structure in the row direction. We formally prove the homomorphic property in Appendix. We use this property to prove the fault tolerance of STAIR codes.

4.2 Proof of Fault Tolerance

We prove that for a STAIR code with configuration parameters n, r, m, and $\mathbf{e}$, as long as the failure pattern is within the failure coverage defined by m and $\mathbf{e}$, the corresponding lost symbols can always be recovered (or decoded). In addition, we present an *upstairs decoding* method, which systematically recovers the lost symbols for STAIR codes.

For a stripe of the STAIR code, we consider the worst-case recoverable failure scenario where there are m failed chunks (due to device failures) and m' additional chunks that have $e_0, e_1, \cdots, e_{m'-1}$ lost symbols (due to sector failures), where $0 < e_0 \le e_1 \le \cdots \le e_{m'-1}$. We prove that all the m' chunks with sector failures can be recovered with global parity symbols. In particular, we show that these m' chunks can be recovered in the order of $e_0, e_1, \cdots, e_{m'-1}$. Finally, the m failed chunks due to device failures can be recovered with row parity chunks.

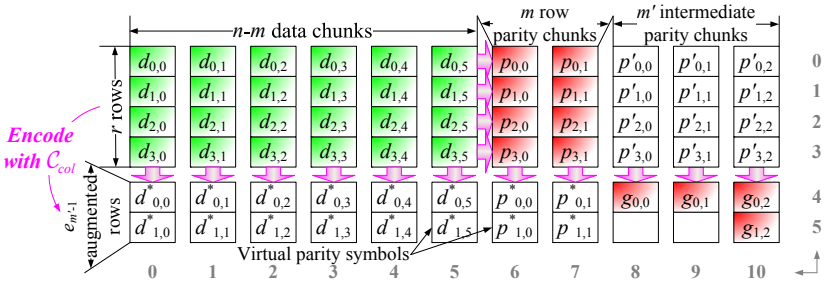


Figure 3: A canonical stripe augmented from the stripe in Figure 2. The rows and columns are labeled from 0 to 5 and 0 to 10, respectively, for ease of presentation.

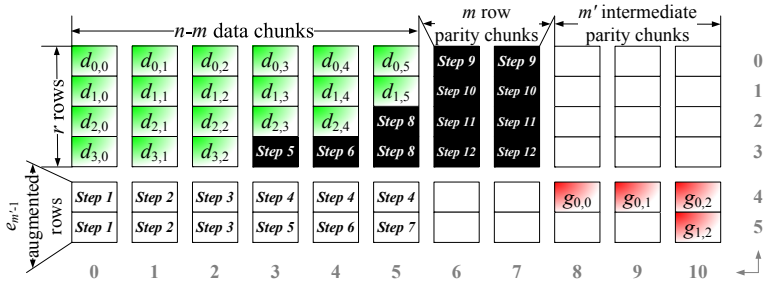


Figure 4: Upstairs decoding based on the canonical stripe in Figure 3.

4.2.1 Example

We demonstrate via our exemplary configuration how we recover the lost data due to both device and sector failures. Figure 4 shows the sequence of our decoding steps. Without loss of generality, we logically assign the column identities such that the m' chunks with sector failures are in Columns $n-m-m'$ to $n-m-1$, with e_0, e_1, $\cdots$, $e_{m'-1}$ lost symbols, respectively, and the m failed chunks are in Columns $n-m$ to $n-1$. Also, the sector failures all occur in the bottom of the data chunks. Thus, the lost symbols form a stair, as shown in Figure 4.

The main idea of upstairs decoding is to recover the lost symbols from left to right and bottom to top. First, we see that there are $n-m-m'=3$ good chunks (i.e., Columns 0-2) without any sector failure. We encode via $\mathcal{C}_{col}$ (which is a (6,4)-code) each such good chunk to obtain $e_{m'-1}=2$ virtual parity symbols (Steps 1-3). In Row 4, there are now six available symbols. Thus, all the unavailable symbols in this row can be recovered using $\mathcal{C}_{row}$ (which is a (11,6)-code) due to the homomorphic property (Step 4). Note that we only need to recover the $m'=3$ symbols that will later be used to recover sector failures. Column 3 (with $e_0=1$ sector failure) now has four available symbols. Thus, we can recover one lost symbol and one virtual parity symbol using $\mathcal{C}_{col}$ (Step 5). Similarly, we repeat the decoding for Column 4 (with $e_1=1$ sector failure) (Step 6). We see that Row 5 now contains six available symbols, so we can recover one unavailable virtual parity symbol (Step 7). Then Column 5 (with $e_2=2$ sector failures) now has four available symbols, so we can recover two lost symbols (Step 8). Now all chunks with sector failures are recovered. Finally, we recover the $m=2$ lost chunks row by row using $\mathcal{C}_{row}$ (Steps 9-12). Table 1 lists the detailed decoding steps of our example in Figure 4.

Steps	Detailed Descriptions
1	$d_{0,0}, d_{1,0}, d_{2,0}, d_{3,0} \Rightarrow d^*_{0,0}, d^*_{1,0}$
2	$d_{0,1}, d_{1,1}, d_{2,1}, d_{3,1} \Rightarrow d^*_{0,1}, d^*_{1,1}$
3	$d_{0,2}, d_{1,2}, d_{2,2}, d_{3,2} \Rightarrow d^*_{0,2}, d^*_{1,2}$
4	$d^*_{0,0}, d^*_{0,1}, d^*_{0,2}, g_{0,0}, g_{0,1}, g_{0,2} \Rightarrow d^*_{0,3}, d^*_{0,4}, d^*_{0,5}$
5	$d_{0,3}, d_{1,3}, d_{2,3}, d^*_{0,3} \Rightarrow d_{3,3}, d^*_{1,3}$
6	$d_{0,4}, d_{1,4}, d_{2,4}, d^*_{0,4} \Rightarrow d_{3,4}, d^*_{1,4}$
7	$d^*_{1,0}, d^*_{1,1}, d^*_{1,2}, d^*_{1,3}, d^*_{1,4}, g_{1,2} \Rightarrow d^*_{1,5}$
8	$d_{0,5}, d_{1,5}, d^*_{0,5}, d^*_{1,5} \Rightarrow d_{2,5}, d_{3,5}$
9	$d_{0,0}, d_{0,1}, d_{0,2}, d_{0,3}, d_{0,4}, d_{0,5} \Rightarrow p_{0,1}, p_{0,2}$
10	$d_{1,0}, d_{1,1}, d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5} \Rightarrow p_{1,1}, p_{1,2}$
11	$d_{2,0}, d_{2,1}, d_{2,2}, d_{2,3}, d_{2,4}, d_{2,5} \Rightarrow p_{2,1}, p_{2,2}$
12	$d_{3,0}, d_{3,1}, d_{3,2}, d_{3,3}, d_{3,4}, d_{3,5} \Rightarrow p_{3,1}, p_{3,2}$

Table 1: Upstairs decoding: detailed steps for the example in Figure 4. Steps 4, 7, and 9-12 use $\mathcal{C}_{row}$, while Steps 1-3, 5-6, and 8 use $\mathcal{C}_{col}$.

4.2.2 General Case

We now generalize the steps of upstairs decoding.

(1) *Decoding of the chunk with e_0 sector failures:* It is clear that there are $n-(m+m')$ good chunks without any sector failure in the stripe. We use $\mathcal{C}_{col}$ to encode each such good chunk to obtain $e_{m'-1}$ virtual parity symbols. Then each of the first e_0 augmented rows must now have $n-m$ available symbols: $n-(m+m')$

virtual parity symbols that have just been encoded and m' global parity symbols. Since an augmented row is a codeword of $\mathcal{C}_{row}$ due to the homomorphic property, all the unavailable symbols in this row can be recovered using $\mathcal{C}_{row}$. Then, for the column with e_0 sector failures, it now has r available symbols: $r - e_0$ good symbols and e_0 virtual parity symbols that have just been recovered. Thus, we can recover the e_0 sector failures as well as the $e_{m'-1} - e_0$ unavailable virtual parity symbols using $\mathcal{C}_{col}$.

(2) *Decoding of the chunk with e_i sector failures ($1 \leq i \leq m' - 1$):* If $e_i = e_{i-1}$, we repeat the decoding for the chunk with e_{i-1} sector failures. Otherwise, if $e_i > e_{i-1}$, each of the next $e_i - e_{i-1}$ augmented rows now has $n - m$ available symbols: $n - (m + m')$ virtual parity symbols that are first recovered from the good chunks, i virtual parity symbols that are recovered while the sector failures are recovered, and $m' - i$ global parity symbols. Thus, all the unavailable virtual parity symbols in these $e_i - e_{i-1}$ augmented rows can be recovered. Then the column with e_i sector failures now has r available symbols: $r - e_i$ good symbols and e_i virtual parity symbols that have been recovered. This column can then be recovered using $\mathcal{C}_{col}$. We repeat this process until all the m' chunks with sector failures are recovered.

(3) *Decoding of the m failed chunks:* After all the m' chunks with sector failures are recovered, the m failed chunks can be recovered row by row using $\mathcal{C}_{row}$.

4.3 Decoding in Practice

In §4.2, we describe an upstairs decoding method for the worst case. In practice, we often have fewer lost symbols than the worst case defined by m and $\mathbf{e}$. To achieve efficient decoding, our idea is to recover as many lost symbols as possible via row parity symbols. The reason is that such decoding is local and involves only the symbols of the same row, while decoding via global parity symbols involves almost all data symbols within the stripe. In our implementation, we first locally recover any lost symbols using row parity symbols whenever possible. Then, for each chunk that still contains lost symbols, we count the number of its remaining lost symbols. Next, we globally recover the lost symbols with global parity symbols using upstairs decoding as described in §4.2, except those in the m chunks that have the most lost symbols. These m chunks can be finally recovered via row parity symbols after all other lost symbols have been recovered.

5 Extended Encoding: Relocating Global Parity Symbols Inside a Stripe

We thus far assume that there are always s available global parity symbols that are kept outside a stripe. However, to maintain the regularity of the code structure and to avoid provisioning extra devices for keeping the global parity symbols, it is desirable to keep all global parity symbols inside a stripe. The idea is that in each stripe, we store the global parity symbols in some sectors that originally store the data symbols. A challenge is that such *inside global parity symbols* are also subject to both device and sector failures, so we must maintain their fault tolerance during encoding. In this section, we propose two encoding methods, namely *upstairs encoding* and *downstairs encoding*, which support the construction of inside global parity symbols, while preserving the homomorphic property and hence the fault tolerance of STAIR codes. These two encoding methods produce the same values for parity symbols, but differ in computational complexities for different configurations. We show how to deduce parity relations from the two encoding methods, and also show that the two encoding methods have complementary performance advantages for different configurations.

5.1 Two New Encoding Methods

5.1.1 Upstairs Encoding

We let $\hat{g}_{h,l}$ ($0 \leq l \leq m' - 1$ and $0 \leq h \leq e_l - 1$) be an inside global parity symbol. Figure 5 illustrates how we place the inside global parity symbols. Without loss of generality, we place them at the bottom of the rightmost data chunks, following the stair layout. Specifically, we choose the $m' = 3$ rightmost data chunks in Columns 3-5 and place $e_0 = 1$, $e_1 = 1$, and $e_2 = 2$ global parity symbols at the bottom of these data chunks, respectively. That is, the original data symbols $d_{3,3}$, $d_{3,4}$, $d_{2,5}$, and $d_{3,5}$ are now replaced by the inside global parity symbols $\hat{g}_{0,0}$, $\hat{g}_{0,1}$, $\hat{g}_{0,2}$, and $\hat{g}_{1,2}$, respectively.

To obtain the inside global parity symbols, we extend the upstairs decoding method in §4.2 and propose a recovery-based encoding approach called *upstairs encoding*. We first set all the outside global parity symbols to be zero (see Figure 5). Then we treat all $m = 2$ row parity chunks and all $s = 4$ inside global parity symbols as lost chunks and lost sectors, respectively. Now we "recover" all inside global parity symbols, followed by the $m = 2$ row parity chunks, using the upstairs decoding method in §4.2. Since all outside global parity symbols are set to be zero, we need not store them. The homomorphic property, and hence the fault tolerance property, remain the same as discussed in §4. Thus, in failure mode, we can still use upstairs decoding to reconstruct lost symbols. We call this encoding method "upstairs encoding" because the parity symbols are encoded from bottom to top as described in §4.2.

5.1.2 Downstairs Encoding

In addition to upstairs encoding, we present a different encoding method called *downstairs encoding*, in which we generate parity symbols from top to bottom and right to left. We illustrate the idea in Figure 6, which depicts

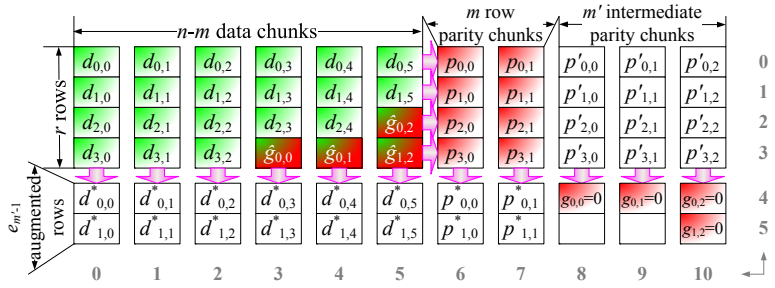


Figure 5: Upstairs encoding: we set outside global parity symbols to be zero and reconstruct the inside global parity symbols using upstairs decoding (see §4.2).

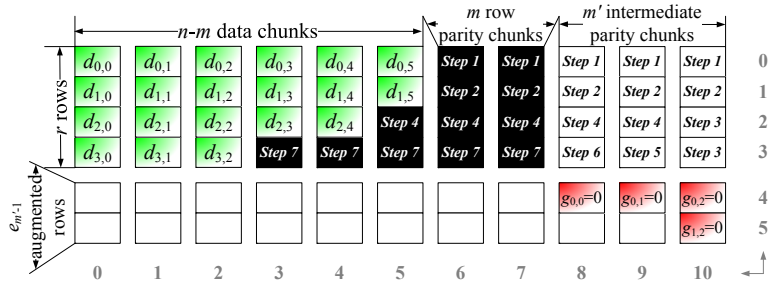


Figure 6: Downstairs encoding: we compute the parity symbols from top to bottom and right to left.

the sequence of generating parity symbols. We still set the outside global parity symbols to be zero. First, we encode via $\mathcal{C}_{row}$ the $n - m = 6$ data symbols in each of the first $r - e_{m'-1} = 2$ rows (i.e., Rows 0 and 1) and generate $m + m' = 5$ parity symbols (including two row parity symbols and three intermediate parity symbols) (Steps 1-2). The rightmost column (i.e., Column 10) now has $r = 4$ available symbols, including the two intermediate parity symbols that are just encoded and two zeroed outside global parity symbols. Thus, we can recover $e_{m'-1} = 2$ intermediate parity symbols using $\mathcal{C}_{col}$ (Step 3). We can generate $m + m' = 5$ parity symbols (including one inside global parity symbol, two row parity symbols, and two intermediate parity symbols) for Row 2 using $\mathcal{C}_{row}$ (Step 4), followed by $e_{m'-2} = 1$ and $e_{m'-3} = 1$ intermediate parity symbols in Columns 9 and 8 using $\mathcal{C}_{col}$, respectively (Steps 5-6). Finally, we obtain the remaining $m + m' = 5$ parity symbols (including three global parity symbols and two row parity symbols) for Row 3 using $\mathcal{C}_{row}$ (Step 7). Table 2 shows the detailed steps of downstairs encoding for the example in Figure 6.

In general, we start with encoding via $\mathcal{C}_{row}$ the rows from top to bottom. In each row, we generate $m + m'$ symbols. When no more rows can be encoded because of insufficient available symbols, we encode via $\mathcal{C}_{col}$ the columns from right to left to obtain new intermediate parity symbols (initially, we obtain $e_{m'-1}$ symbols, followed by $e_{m'-2}$ symbols, and so on). We alternately encode rows and columns until all parity symbols are

Steps	Detailed Descriptions
1	$d_{0,0}, d_{0,1}, d_{0,2}, d_{0,3}, d_{0,4}, d_{0,5} \Rightarrow p_{0,0}, p_{0,1}, p'_{0,0}, p'_{0,1}, p'_{0,2}$
2	$d_{1,0}, d_{1,1}, d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5} \Rightarrow p_{1,0}, p_{1,1}, p'_{1,0}, p'_{1,1}, p'_{1,2}$
3	$p'_{0,2}, p'_{1,2}, g_{0,2} = 0, g_{1,2} = 0 \Rightarrow p'_{2,2}, p'_{3,2}$
4	$d_{2,0}, d_{2,1}, d_{2,2}, d_{2,3}, d_{2,4}, p'_{2,2} \Rightarrow \hat{g}_{0,2}, p_{2,0}, p_{2,1}, p'_{2,0}, p'_{2,1}$
5	$p'_{0,1}, p'_{1,1}, p'_{2,1}, g_{0,1} = 0 \Rightarrow p'_{3,1}$
6	$p'_{0,0}, p'_{1,0}, p'_{2,0}, g_{0,0} = 0 \Rightarrow p'_{3,0}$
7	$d_{3,0}, d_{3,1}, d_{3,2}, p'_{3,0}, p'_{3,1}, p'_{3,2} \Rightarrow \hat{g}_{0,0}, \hat{g}_{0,1}, \hat{g}_{1,2}, p_{3,0}, p_{3,1}$

Table 2: Downstairs decoding: detailed steps for the example in Figure 6. Steps 1-2, 4, and 7 use $\mathcal{C}_{row}$, while Steps 3 and 5-6 use $\mathcal{C}_{col}$.

formed. We can generalize the steps as in §4.2.2, but we omit the details in the interest of space.

It is important to note that the downstairs encoding method cannot be generalized for decoding lost symbols. For example, referring to our exemplary configuration, we consider a worst-case recoverable failure scenario in which both row parity chunks are entirely failed, and the data symbols $d_{0,3}$, $d_{1,4}$, $d_{2,2}$, and $d_{3,2}$ are lost. In this case, we cannot recover the lost symbols in the top row first, but instead we must resort to upstairs decoding as described in §4.2. Upstairs decoding works because we limit the maximum number of chunks with lost symbols (i.e., at most $m+m'$). This enables us to first recover the leftmost virtual parity symbols of the augmented rows first and gradually reconstruct lost symbols. On the other

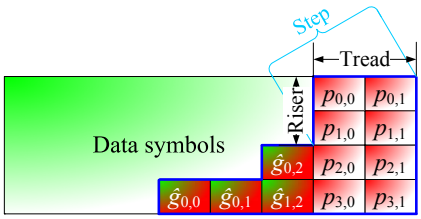


Figure 7: A stair step with a tread and a riser.

hand, we do not limit the number of rows with lost symbols in our configuration, so the downstairs method cannot be used for general decoding.

5.1.3 Discussion

Note that both upstairs and downstairs encoding methods always generate the *same* values for all parity symbols, since both of them preserve the homomorphic property, fix the outside global parity symbols to be zero, and use the same schemes $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ for encoding.

Also, both of them reuse parity symbols in the intermediate steps to generate additional parity symbols in subsequent steps. On the other hand, they differ in encoding complexity, due to the different ways of reusing the parity symbols. We analyze this in §5.3.

5.2 Uneven Parity Relations

Before relocating the global parity symbols inside a stripe, each data symbol contributes to m row parity symbols and all s outside global parity symbols. However, after relocation, the parity relations become uneven. That is, some row parity symbols are also contributed by the data symbols in other rows, while some inside global parity symbols are contributed by only a subset of data symbols in the stripe. Here, we discuss the uneven parity relations of STAIR codes so as to better understand the encoding and update performance of STAIR codes in subsequent analysis.

To analyze how exactly each parity symbol is generated, we revisit both upstairs and downstairs encoding methods. Recall that the row parity symbols and the inside global parity symbols are arranged in the form of stair steps, each of which is composed of a *tread* (i.e., the horizontal portion of a step) and a *riser* (i.e., the vertical portion of a step), as shown in Figure 7. If upstairs encoding is used, then from Figure 4, the encoding of each parity symbol does not involve any data symbol on its right. Also, among the columns spanned by the same tread, the encoding of parity symbols in each column does not involve any data symbol in other columns. We can make similar arguments for downstairs encoding. If downstairs encoding is used, then from Figure 6, the encoding of each parity symbol does not involve any data symbol below it. Also, among the rows spanned by the same riser, the encoding of parity symbols in each row does not involve any data symbol in other rows.

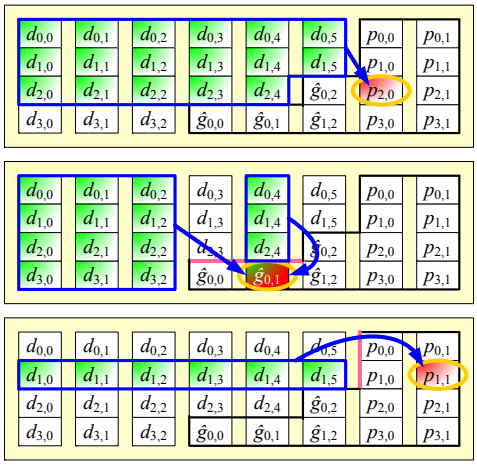


Figure 8: The data symbols that contribute to parity symbols $p_{2,0}$, $\hat{g}_{0,1}$, and $p_{1,1}$, respectively.

As both upstairs and downstairs encoding methods generate the same values of parity symbols, we can combine the above arguments into the following property of how each parity symbol is related to data symbols.

Property 1 (Parity relations in STAIR codes): In a STAIR code stripe, a (row or inside global) parity symbol in Row i_0 and Column j_0 (where $0 \leq i_0 \leq r-1$ and $n-m-m' \leq j_0 \leq n-1$) depends only on the data symbols $d_{i,j}$'s where $i \leq i_0$ and $j \leq j_0$. Moreover, each parity symbol is unrelated to any data symbol in any other column (row) spanned by the same tread (riser).

Figure 8 illustrates the above property. For example, $p_{2,0}$ depends only on the data symbols $d_{i,j}$'s in Rows 0-2 and Columns 0-5. Note that $\hat{g}_{0,1}$ in Column 4 is unrelated to any data symbol in Column 3, which is spanned by the same tread as Column 4. Similarly, $p_{1,1}$ in Row 1 is unrelated to any data symbol in Row 0, which is spanned by the same riser as Row 1.

5.3 Encoding Complexity Analysis

We have proposed two encoding methods for STAIR codes: upstairs encoding and downstairs encoding. Both of them alternately encode rows and columns to obtain the parity symbols. We can also obtain parity symbols using the standard encoding approach, in which each parity symbol is computed directly from a linear combination of data symbols as in classical Reed-Solomon codes. We now analyze the computational complexities of these three methods for different configuration parameters of STAIR codes.

STAIR codes perform encoding over a Galois Field, in which linear arithmetic can be decomposed into the basic operations `Mult_XORs` [31]. We define

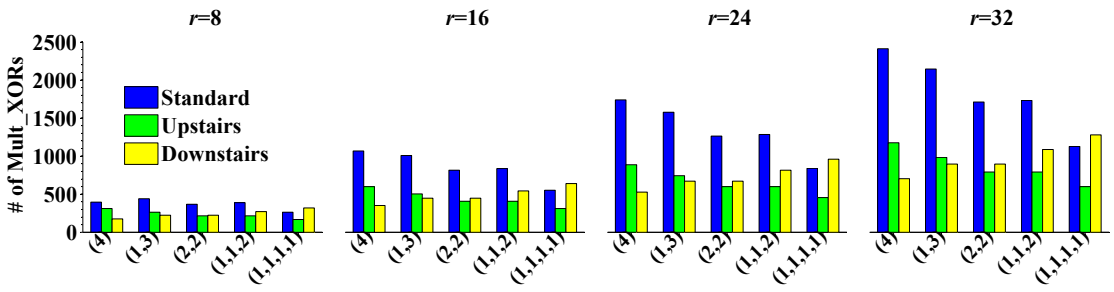


Figure 9: Numbers of Mult_XORs (per stripe) of the three encoding methods for STAIR codes versus different $\mathbf{e}$'s when $n = 8$, $m = 2$, and $s = 4$.

Mult_XOR$(\mathbf{R}_1, \mathbf{R}_2, \alpha)$ as an operation that first multiplies a region $\mathbf{R}_1$ of bytes by a w-bit constant α in Galois Field $GF(2^w)$, and then applies XOR-summing to the product and the target region $\mathbf{R}_2$ of the same size. For example, $\mathbf{Y} = \alpha_0 \cdot \mathbf{X}_0 + \alpha_1 \cdot \mathbf{X}_1$ can be decomposed into two Mult_XORs (assuming $\mathbf{Y}$ is initialized as zero): Mult_XOR$(\mathbf{X}_0, \mathbf{Y}, \alpha_0)$ and Mult_XOR$(\mathbf{X}_1, \mathbf{Y}, \alpha_1)$. Clearly, fewer Mult_XORs imply a lower computational complexity. To evaluate the computational complexity of an encoding method, we count its number of Mult_XORs (per stripe).

For upstairs encoding, we generate $m \cdot r$ row parity symbols and s virtual parity symbols along the row direction, as well as s inside global parity symbols and $(n-m) \cdot e_{m'-1} - s$ virtual parity symbols along the column direction. Its number of Mult_XORs (denoted by $\mathcal{X}_{up}$) is:

$$\mathcal{X}_{up} = \overbrace{(n-m) \times (m \cdot r + s)}^{\text{row direction}} + \overbrace{r \times [(n-m) \cdot e_{m'-1}]}^{\text{column direction}}. \quad (1)$$

For downstairs encoding, we generate $m \cdot r$ row parity symbols, s inside global parity symbols, and $m' \cdot r - s$ intermediate parity symbols along the row direction, as well as s intermediate parity symbols along the column direction. Its number of Mult_XORs (denoted by $\mathcal{X}_{down}$) is:

$$\mathcal{X}_{down} = \overbrace{(n-m) \times \left[(m+m') \cdot r\right]}^{\text{row direction}} + \overbrace{r \times s}^{\text{column direction}}. \quad (2)$$

For standard encoding, we compute the number of Mult_XORs by summing the number of data symbols that contribute to each parity symbol, based on the property of uneven parity relations discussed in §5.2.

We show via a case study how the three encoding methods differ in the number of Mult_XORs. Figure 9 depicts the numbers of Mult_XORs of the three encoding methods for different $\mathbf{e}$'s in the case where $n = 8$, $m = 2$, and $s = 4$. Upstairs encoding and downstairs encoding incur significantly fewer Mult_XORs than standard encoding most of the time. The main reason is that both upstairs encoding and downstairs encoding often reuse the computed parity symbols in subsequent encoding steps. We also observe that for a given s, the number of Mult_XORs of upstairs encoding increases with $e_{m'-1}$ (see Equation (1)), while that of downstairs encoding increases with m' (see Equation (2)). Since larger m' often implies smaller $e_{m'-1}$, the value of m' often determines which of the two encoding methods is more efficient: when m' is small, downstairs encoding wins; when m' is large, upstairs encoding wins.

In our encoding implementation of STAIR codes, for given configuration parameters, we always pre-compute the number of Mult_XORs for each of the encoding methods, and then choose the one with the fewest Mult_XORs.

6 Evaluation

We evaluate STAIR codes and compare them with other related erasure codes in different practical aspects, including storage space saving, encoding/decoding speed, and update penalty.

6.1 Storage Space Saving

The main motivation for STAIR codes is to tolerate simultaneous device and sector failures with significantly lower storage space overhead than traditional erasure codes (e.g., Reed-Solomon codes) that provide only device-level fault tolerance. Given a failure scenario defined by m and $\mathbf{e}$, traditional erasure codes need $m + m'$ chunks per stripe for parity, while STAIR codes need only m chunks and s symbols (where $m' \leq s$). Thus, STAIR codes save $r \times m' - s$ symbols per stripe, or equivalently, $m' - \frac{s}{r}$ devices per system. In short, the saving of STAIR codes depends on only three parameters s, m', and r (where s and m' are determined by $\mathbf{e}$).

Figure 10 plots the number of devices saved by STAIR codes for $s \leq 4$, $m' \leq s$, and $r \leq 32$. As r increases, the number of devices saved is close to m'. The saving reaches the highest when $m' = s$.

We point out that the recently proposed SD codes [27,28] are also motivated for reducing the storage space

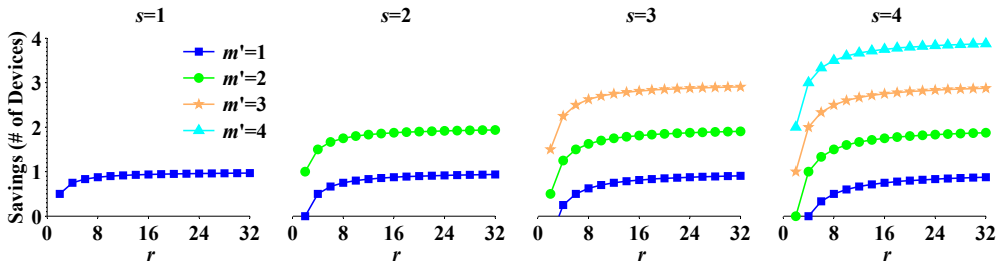


Figure 10: Space saving of STAIR codes over traditional erasure codes in terms of s, m', and r.

over traditional erasure codes. Unlike STAIR codes, SD codes always achieve a saving of $s - \frac{s}{r}$ devices, which is the maximum saving of STAIR codes. While STAIR codes apparently cannot outperform SD codes in space saving, it is important to note that the currently known constructions of SD codes are limited to $s \leq 3$ only [6,27,28], implying that SD codes can save no more than three devices. On the other hand, STAIR codes do not have such limitations. As shown in Figure 10, STAIR codes can save more than three devices for larger s.

6.2 Encoding/Decoding Speed

We evaluate the encoding/decoding speed of STAIR codes. Our implementation of STAIR codes is written in C. We leverage the GF-Complete open source library [31] to accelerate Galois Field arithmetic using Intel SIMD instructions. Our experiments compare STAIR codes with the state-of-the-art SD codes [27, 28]. At the time of this writing, the open-source implementation of SD codes encodes stripes in a decoding manner without any parity reuse. For fair comparisons, we extend the SD code implementation to support the standard encoding method mentioned in §5.3. We run our performance tests on a machine equipped with an Intel Core i5-3570 CPU at 3.40GHz with SSE4.2 support. The CPU has a 256KB L2-cache and a 6MB L3-cache.

6.2.1 Encoding

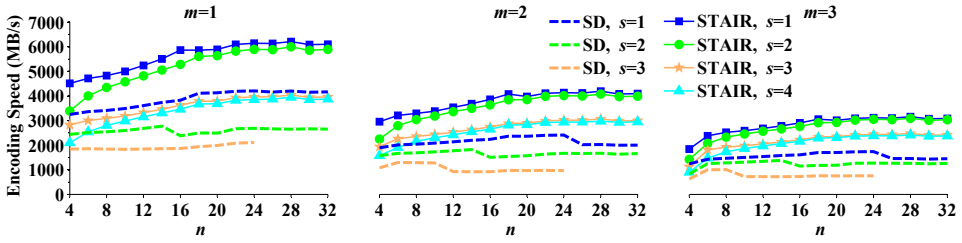
We compare the encoding performance of STAIR codes and SD codes for different values of n, r, m, and s. For SD codes, we only consider the range of configuration parameters where $s \leq 3$, since no code construction is available outside this range [6, 27, 28]. In addition, the SD code constructions for $s = 3$ are only available in the range $n \leq 24$, $r \leq 24$, and $m \leq 3$ [27, 28]. For STAIR codes, a single value of s can imply different configurations of $\mathbf{e}$ (e.g., see Figure 9 in §5.3), each of which has different encoding performance. Here, we take a conservative approach to analyze the worst-case performance of STAIR codes, that is, we test all possible configurations of $\mathbf{e}$ for a given s and pick the one with the lowest encoding speed.

Note that the encoding performance of both STAIR codes and SD codes heavily depends on the word size w of the adopted Galois Field $GF(2^w)$, where w is often set to be a power of 2. A smaller w often means a higher encoding speed [31]. STAIR codes work as long as $n + m' \leq 2^w$ and $r + e_{m'-1} \leq 2^w$. Thus, we choose $w = 8$ since it suffices for all of our tests. However, SD codes may choose among $w = 8$, $w = 16$, and $w = 32$, depending on configuration parameters. We choose the smallest w that is feasible for the SD code construction.

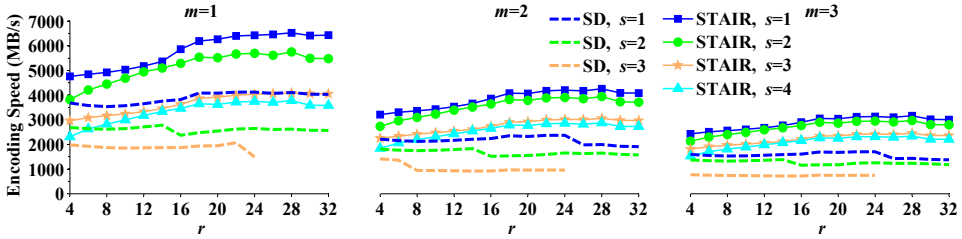
We consider the metric *encoding speed*, defined as the amount of data encoded per second. We construct a stripe of size roughly 32MB in memory as in [27, 28]. We put random bytes in the stripe, and divide the stripe into $r \times n$ sectors, each mapped to a symbol. We obtain the averaged results over 10 runs.

Figures 11(a) and 11(b) present the encoding speed results for different values of n when $r = 16$ and for different values of r when $n = 16$, respectively. In most cases, the encoding speed of STAIR codes is over 1000MB/s, which is significantly higher than the disk write speed in practice (note that although disk writes can be parallelized in disk arrays, the encoding operations can also be parallelized with modern multi-core CPUs). The speed increases with both n and r. The intuitive reason is that the proportion of parity symbols decreases with n and r. Compared to SD codes, STAIR codes improve the encoding speed by 106.03% on average (in the range from 29.30% to 225.14%). The reason is that STAIR codes reuse encoded parity information in subsequent encoding steps by upstairs/downstairs encoding (see §5.3), while such an encoding property is not exploited in SD codes.

We also evaluate the impact of stripe size on the encoding speed of STAIR codes and SD codes for given n and r. We fix $n = 16$ and $r = 16$, and vary the stripe size from 128KB to 512MB. Note that a stripe of size 128KB implies a symbol of size 512 bytes, the standard sector size in practical disk drives. Figure 12 presents the encoding speed results. As the stripe size increases, the encoding speed of both STAIR codes and SD codes first increases and then drops, due to the mixed effects of SIMD instructions adopted in GF-Complete [31] and CPU cache. Nevertheless, the encoding speed advantage of STAIR codes over SD codes remains unchanged.



(a) Varying n when $r = 16$



(b) Varying r when $n = 16$

Figure 11: Encoding speed of STAIR codes and SD codes for different combinations of n, r, m, and s.

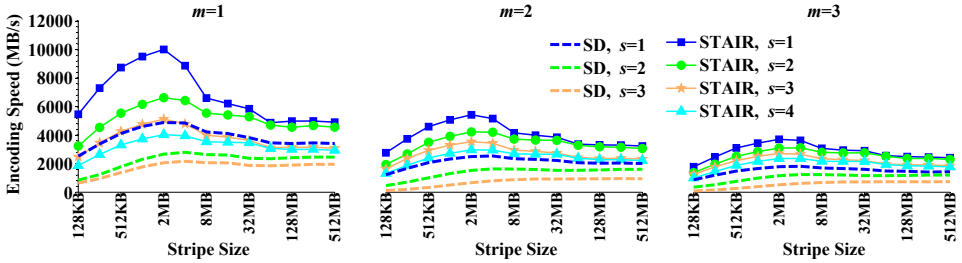


Figure 12: Encoding speed of STAIR codes and SD codes for different stripe sizes when $n = 16$ and $r = 16$.

6.2.2 Decoding

We measure the decoding performance of STAIR codes and SD codes in recovering lost symbols. Since the decoding time increases with the number of lost symbols to be recovered, we consider a particular worst case in which the m leftmost chunks and s additional symbols in the following m' chunks defined by $\mathbf{e}$ are all lost. The evaluation setup is similar to that in §6.2.1, and in particular, the stripe size is fixed at 32MB.

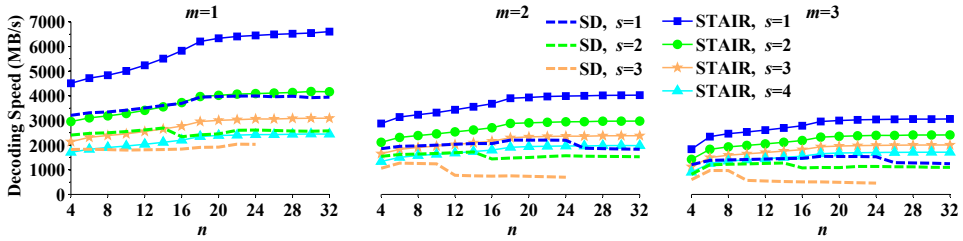
Figures 13(a) and 13(b) present the decoding speed results for different n when $r = 16$ and for different r when $n = 16$, respectively. The results of both figures can be viewed in comparison to those of Figures 11(a) and 11(b), respectively. Similar to encoding, the decoding speed of STAIR codes is over 1000MB/s in most cases and increases with both n and r. Compared to SD codes, STAIR codes improve the decoding speed by 102.99% on average (in the range from 1.70% to 537.87%).

In practice, we often have fewer lost symbols than the worst case (see §4.3). One common case is that there are only failed chunks due to device failures (i.e., $s = 0$), so the decoding of both STAIR and SD codes is identical to that of Reed-Solomon codes. In this case, the decoding speed of STAIR/SD codes can be significantly higher than that of $s = 1$ for STAIR codes in Figure 13. For example, when $n = 16$ and $r = 16$, the decoding speed increases by 79.39%, 29.39%, and 11.98% for $m = 1$, 2, and 3, respectively.

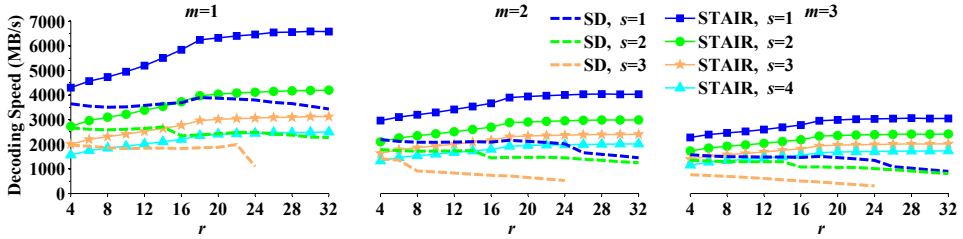
6.3 Update Penalty

We evaluate the update cost of STAIR codes when data symbols are updated. For each data symbol in a stripe being updated, we count the number of parity symbols being affected (see §5.2). Here, we define the *update penalty* as the average number of parity symbols that need to be updated when a data symbol is updated.

Clearly, the update penalty of STAIR codes increases with m. We are more interested in how $\mathbf{e}$ influences the update penalty of STAIR codes. Figure 14 presents the update penalty results for different $\mathbf{e}$'s when $n = 16$ and $s = 4$. For different $\mathbf{e}$'s with the same s, the update penalty of STAIR codes often increases with $e_{m'-1}$. Intuitively, a larger $e_{m'-1}$ implies that more rows of row parity symbols are encoded from inside global parity



(a) Varying n when $r = 16$



(b) Varying r when $n = 16$

Figure 13: Decoding speed of STAIR codes and SD codes for different combinations of n, r, m, and s.

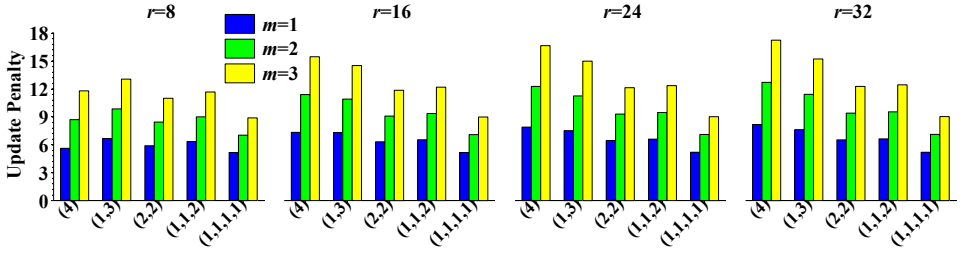


Figure 14: Update penalty of STAIR codes for different $\mathbf{e}$'s when $n = 16$ and $s = 4$.

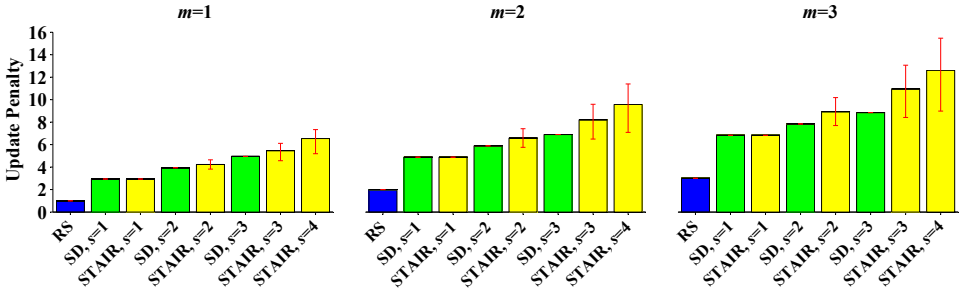


Figure 15: Update penalty of STAIR codes, SD codes, and Reed-Solomon (RS) codes when $n = 16$ and $r = 16$. For STAIR codes, we plot the error bars for the maximum and minimum update penalty values among all possible configurations of $\mathbf{e}$.

symbols, which are further encoded from almost all data symbols (see §5.2).

We compare STAIR codes with SD codes [27,28]. For STAIR codes with a given s, we test all possible configurations of $\mathbf{e}$ and find the average, minimum, and maximum update penalty. For SD codes, we only consider s between 1 and 3. We also include the update penalty results of Reed-Solomon codes for reference. Figure 15 presents the update penalty results when $n = 16$ and $r = 16$ (while similar observations are made for other n and r). For a given s, the range of update penalty of STAIR codes covers that of SD codes, although the average is sometimes higher than that of SD codes (same for $s = 1$, by 7.30% to 14.02% for $s = 2$, and by 10.47% to 23.72% for $s = 3$). Both STAIR codes and SD codes have higher update penalty than Reed-Solomon codes due to more parity symbols in a stripe, and hence are suitable for storage systems with rare updates (e.g., backup

or write-once-read-many (WORM) systems) or systems dominated by full-stripe writes [27, 28].

7 Related Work

Erasure codes have been widely adopted to provide fault tolerance against device failures in storage systems [32]. Classical erasure codes include standard Reed-Solomon codes [34] and Cauchy Reed-Solomon codes [7], both of which are MDS codes that provide general constructions for all possible configuration parameters. They are usually implemented as systematic codes for storage applications [26,30,33], and thus can be used to implement the construction of STAIR codes. In addition, Cauchy Reed-Solomon codes can be further transformed into *array codes*, whose encoding computations purely build on efficient XOR operations [33].

In the past decades, many kinds of array codes have been proposed, including MDS array codes (e.g., [2–4,9, 12,13,20,22,29,41,42]) and non-MDS array codes (e.g., [16, 17, 23]). Array codes are often designed for specific configuration parameters. To avoid compromising the generality of STAIR codes, we do not suggest to adopt array codes in the construction of STAIR codes. Moreover, recent work [31] has shown that Galois Field arithmetic can be implemented to be extremely fast (sometimes at cache line speeds) using SIMD instructions in modern processors.

Sector failures are not explicitly considered in traditional erasure codes, which focus on tolerating device-level failures. To cope with sector failures, ad hoc schemes are often considered. One scheme is *scrubbing* [24, 36, 38], which proactively scans all disks and recovers any spotted sector failure using the underlying erasure codes. Another scheme is *intra-device redundancy* [10, 11, 36], in which contiguous sectors in each device are grouped together to form a segment and are then encoded with redundancy within the device. Our work targets a different objective and focuses on constructing an erasure code that explicitly addresses sector failures.

To simultaneously tolerate device and sector failures with minimal redundancy, SD codes [27, 28] (including the earlier PMDS codes [5], which are a subset of SD codes) have recently been proposed. As stated in §1, SD codes are known only for limited configurations and some of the known constructions rely on extensive searches. A relaxation of the SD property has also been recently addressed as a future work in [27], which assumes that each row has no more than a given number of sector failures. It is important to note that the relaxation of [27] is different from ours, in which we limit the maximum number of devices with sector failures and the maximum number of sector failures that simultaneously occur in each such device. It turns out that our relaxation enables us to derive a general code construction. Another similar kind of erasure codes is the family of locally repairable codes (LRCs) [18, 19, 35]. Pyramid codes [18] are designed for improving the recovery performance for small-scale device failures and have been implemented in archival storage [40]. Huang *et al.*'s and Sathiamoorthy *et al.*'s LRCs [19, 35] can be viewed as generalizations of Pyramid codes and are recently adopted in commercial storage systems. In particular, Huang *et al.*'s LRCs [19] achieve the same fault tolerance property as PMDS codes [5], and thus can also be used as SD codes. However, the construction of Huang *et al.*'s LRCs is limited to $m = 1$ only. To our knowledge, STAIR codes are the first general family of erasure codes that can efficiently tolerate both device and sector failures.

8 Conclusions

We present STAIR codes, a general family of erasure codes that can tolerate simultaneous device and sector failures in a space-efficient manner. STAIR codes can be constructed for tolerating any numbers of device and sector failures subject to a pre-specified sector failure coverage. The special construction of STAIR codes also makes efficient encoding/decoding possible through parity reuse. Compared to the recently proposed SD codes [5, 27, 28], STAIR codes not only support a much wider range of configuration parameters, but also achieve higher encoding/decoding speed based on our experiments.

In future work, we explore how to correctly configure STAIR codes in practical storage systems based on empirical failure characteristics [1, 25, 36, 37].

The source code of STAIR codes is available at **http://ansrlab.cse.cuhk.edu.hk/software/stair**.

Acknowledgments

We would like to thank our shepherd, James S. Plank, and the anonymous reviewers for their valuable comments. This work was supported in part by grants from the University Grants Committee of Hong Kong (project numbers: AoE/E-02/08 and ECS CUHK419212).

References

[1] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289–300, San Diego, CA, June 2007.

[2] M. Blaum. A family of MDS array codes with minimal number of encoding operations. In *Proceedings of the 2006 IEEE International Symposium on*

Information Theory (ISIT '06), pages 2784–2788, Seattle, WA, July 2006.

[3] 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, 1995.

[4] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2):529–542, 1996.

[5] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory*, 59(7):4510–4519, July 2013.

[6] M. Blaum and J. S. Plank. Construction of sector-disk (SD) codes with two global parity symbols. IBM Research Report RJ10511 (ALM1308-007), Almaden Research Center, IBM Research Division, Aug. 2013.

[7] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug. 1995.

[8] S. Boboila and P. Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 115–128, San Jose, CA, Feb. 2010.

[9] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, CA, Mar. 2004.

[10] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. Rao. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM Transactions on Storage*, 4(1):1–42, 2008.

[11] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. Rao. Disk scrubbing versus intradisk redundancy for RAID storage systems. *ACM Transactions on Storage*, 7(2):1–42, 2011.

[12] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers*, 54(9):1071–1080, 2005.

[13] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers*, 54(12):1473–1483, 2005.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO '09)*, pages 24–33, New York, NY, Dec. 2009.

[15] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, pages 17–24, San Jose, CA, Feb. 2012.

[16] J. L. Hafner. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 211–224, San Francisco, CA, Dec. 2005.

[17] J. L. Hafner. HoVer erasure codes for disk arrays. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN '06)*, pages 1–10, Philadelphia, PA, June 2006.

[18] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage*, 9(1):1–28, Mar. 2013.

[19] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, Boston, MA, June 2012.

[20] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 889–901, San Francisco, CA, Dec. 2005.

[21] Intel Corporation. Intelligent RAID 6 theory — overview and implementation. White Paper, 2005.

[22] M. Li and J. Shu. C-Codes: Cyclic lowest-density MDS array codes constructed using starters for RAID 6. IBM Research Report RC25218 (C1110-004), China Research Laboratory, IBM Research Division, Oct. 2011.

[23] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage*, 4(4):1–22, 2009.

[24] A. Oprea and A. Juels. A clean-slate look at disk scrubbing. In *Proceedings of the 8th USENIX Con-*

ference on File and Storage Technologies (FAST '10), pages 1–14, San Jose, CA, Feb. 2010.

[25] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, pages 17–28, San Jose, CA, Feb. 2007.

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

[27] J. S. Plank and M. Blaum. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems. Technical Report CS-13-708, University of Tennessee, May 2013.

[28] J. S. Plank, M. Blaum, and J. L. Hafner. SD codes: Erasure codes designed for how storage systems really fail. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 95–104, San Jose, CA, Feb. 2013.

[29] J. S. Plank, A. L. Buchsbaum, and B. T. Vander Zanden. Minimum density RAID-6 codes. *ACM Transactions on Storage*, 6(4):1–22, May 2011.

[30] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software — Practice & Experience*, 35(2):189–194, 2005.

[31] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 299–306, San Jose, CA, Feb. 2013.

[32] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, Feb. 2013.

[33] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications (NCA '06)*, pages 173–180, Cambridge, MA, July 2006.

[34] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[35] M. Sathiamoorthy, M. Asteris, D. Papailiopoulous, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB '13)*, pages 325–336, Trento, Italy, Aug. 2013.

[36] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 71–84, San Jose, CA, Feb. 2010.

[37] 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 *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, pages 1–16, San Jose, CA, Feb. 2007.

[38] T. J. E. Schwarz, Q. Xin, E. L. Miller, and D. D. E. Long. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418, Volendam, Netherlands, Oct. 2004.

[39] J. White and C. Lueth. RAID-DP: NetApp implementation of double-parity RAID for data protection. Technical Report TR-3298, NetApp, Inc., May 2010.

[40] A. Wildani, T. J. E. Schwarz, E. L. Miller, and D. D. Long. Protecting against rare event failures in archival systems. In *Proceedings of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 1–11, London, UK, Sept. 2009.

[41] L. Xu, V. Bohossian, J. Bruck, and D. G. Wagner. Low-density MDS codes and factors of complete graphs. *IEEE Transactions on Information Theory*, 45(6):1817–1826, Sept. 1999.

[42] L. Xu and J. Bruck. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[43] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, CA, Feb. 2013.

Appendix: Proof of Homomorphic Property

We formally prove the homomorphic property described in §4.1. We state the following theorem.

Theorem 1 *In the construction of the canonical stripe of STAIR codes, the encoding of each chunk in the column direction via C_{col} is homomorphic, such that each*

augmented row in the canonical stripe is a codeword of $\mathcal{C}_{row}$.

Proof: We prove by matrix operations. We define the matrices $\mathbf{D} = [d_{i,j}]_{r\times(n-m)}$, $\mathbf{P} = [p_{i,k}]_{r\times m}$, and $\mathbf{P'} = [p'_{i,l}]_{r\times m'}$. Also, we define the generator matrices $\mathbf{G}_{row}$ and $\mathbf{G}_{col}$ for the codes $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$, respectively, as:

$$\begin{aligned}\mathbf{G}_{row} &= \left(\mathbf{I}_{(n-m)\times(n-m)} \mid \mathbf{A}_{(n-m)\times(m+m')}\right), \\ \mathbf{G}_{col} &= \left(\mathbf{I}_{r\times r} \mid \mathbf{B}_{r\times e_{m'-1}}\right),\end{aligned}$$

where $\mathbf{I}$ is an identity matrix, and $\mathbf{A}$ and $\mathbf{B}$ are the submatrices that form the parity symbols. The upper r rows of the stripe can be expressed as follows:

$$(\mathbf{D} \mid \mathbf{P} \mid \mathbf{P'}) = \mathbf{D}\cdot\mathbf{G}_{row}.$$

The lower $e_{m'-1}$ augmented rows are expressed as follows:

$$\begin{aligned}\left((\mathbf{D} \mid \mathbf{P} \mid \mathbf{P'})^T \cdot \mathbf{B}\right)^T &= \mathbf{B}^T \cdot (\mathbf{D}\cdot\mathbf{G}_{row}) \\ &= \left(\mathbf{B}^T\cdot\mathbf{D}\right)\cdot\mathbf{G}_{row}\end{aligned}$$

We can see that each of the lower $e_{m'-1}$ rows can be calculated using the generator matrix $\mathbf{G}_{row}$, and hence is a codeword of $\mathcal{C}_{row}$. □