Published in proceedings of DSN 2007, The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 25 - June 28, 2007, Edinburgh, UK

Determining fault tolerance of XOR-based erasure codes efficiently

Jay J. Wylie and Ram Swaminathan
Hewlett-Packard Labs
jay.wylie@hp.com, ram.swaminathan@hp.com

Abstract

We propose a new fault tolerance metric for XOR*-based erasure codes: the minimal erasures list (*MEL*). A minimal erasure is a set of erasures that leads to irrecoverable data loss and in which every erasure is necessary and sufficient for this to be so. The* MEL *is the enumeration of all minimal erasures. An* XOR*-based erasure code has an irregular structure that may permit it to tolerate faults at and beyond its Hamming distance. The* MEL *completely describes the fault tolerance of an* XOR*-based erasure code at and beyond its Hamming distance; it is therefore a useful metric for comparing the fault tolerance of such codes. We also propose an algorithm that efficiently determines the* MEL *of an erasure code. This algorithm uses the structure of the erasure code to efficiently determine the* MEL*. We show that, in practice, the number of minimal erasures for a given code is much less than the total number of sets of erasures that lead to data loss: in our empirical results for one corpus of codes, there were over* 80 *times fewer minimal erasures. We use the proposed algorithm to identify the most fault tolerant* XOR*-based erasure code for all possible systematic erasure codes with up to seven data symbols and up to seven parity symbols.*

1. Introduction

Storage systems must be fault tolerant. Traditionally, tolerating a single disk failure via simple replication or RAID 5 has provided sufficient reliability. In storage arrays, ever increasing disk capacity leads to ever increasing recovery times which leads to sector or second disk failures being encountered during recovery [2]. Cluster-based and grid storage systems are built with commodity components and rely on network-attached components; the former have lower reliability and the latter lower availability than the components traditionally employed in storage arrays. The trends in storage arrays, cluster-based storage, and grid storage demand that storage schemes with higher degrees of fault tolerance be developed and be well understood.

Erasure codes are the means by which storage systems are typically made fault tolerant (i.e., tolerant of disk failures). There are many types of erasure codes, such as replication, RAID 5, and Reed-Solomon codes, each of which trades off between computation (encode & decode) costs, fault tolerance, and space efficiency. Reed-Solomon codes provide the best tradeoff between fault tolerance and space efficiency, but are computationally the most demanding type of erasure code. Erasure codes that rely solely on XOR operations to generate redundancy are computationally cheap. However, such codes offer a non-uniform tradeoff between space efficiency and fault tolerance. In practice, the exact degree of fault tolerance such codes provide in storage systems is not yet well understood, although there is much recent activity towards this end [14, 13, 6, 4, 5, 7].

To completely understand the fault tolerance of an XOR-based erasure code, we must enumerate *every* set of erasures that leads to data loss. This is necessary because of the irregular structure of such codes. For example, if the smallest *erasure pattern*—set of erasures that leads to data loss—for a given code is of size 3, then the Hamming distance of the code is 4. However, the code may tolerate many erasures of size 4. The enumeration of all erasure patterns thus completely describes the fault tolerance of an XOR-based erasure code. Unfortunately, there are exponentially many such erasure patterns.

In this paper, we propose enumerating every *minimal erasure* to characterize the fault tolerance of a code. A minimal erasure is a set of erasures that leads to irrecoverable data loss and in which every erasure is necessary and sufficient for this to be so. We call the enumeration of minimal erasures, the *minimal erasures list* (MEL). The minimal erasures list contains all of the fault tolerance information as the list of erasure patterns, but can be much smaller in size. There are also an exponential number of minimal erasures, but our results suggest that, in practice, for most codes, there are many fewer minimal erasures than erasure patterns.

We introduce the Minimal Erasures (ME) Algorithm for efficiently determining the MEL of an XOR-based erasure code. The efficiency of the ME Algorithm is premised on there being few minimal erasures relative to the overall number of erasure patterns, and on using the structure of the XOR-based erasure code to identify the minimal erasures. We have used our implementation of the ME Algorithm to analyze many XOR-based codes. Our empirical results demonstrate that there can be almost two orders of magnitude fewer minimal erasures than erasure patterns (and likely a bigger reduction as k and m increase). We use the MEL to compare different small XOR-based erasure codes of similar size and report the most fault tolerant codes. These results demonstrate both the efficacy and utility of the ME Algorithm for determining the fault tolerance of XOR-based erasure codes.

The outline of the paper is as follows. In §2, we introduce terminology and review some related work. We present the ME Algorithm and prove its correctness in §3. In §4, we describe our implementation of the ME Algorithm, our method of validating the correctness of our implementation, empirical results that demonstrate the efficiency of the ME Algorithm, and identify the most fault tolerant systematic XOR-based erasure codes with up to seven data symbols and up to seven parity symbols. We discuss the ME Algorithm in relation to specific other recent work in §5 and then conclude in §6.

2. Background

Table 1 lists some symbols and acronyms used in this paper. An XOR-based erasure code consists of n symbols, k of which are *data symbols*, and m of which are *parity symbols* (redundant symbols). We refer to redundant symbols as parity symbols because our focus is on XOR-based erasure codes. We only consider *systematic* erasure codes: codes that store the data and parity symbols. In storage systems, data symbols are called "stripes." The use of systematic erasure codes in storage systems is generally considered a necessity to ensure good common case performance.

A set of erasures f is a set of erased symbols; it may contain either data symbols or parity symbols and it may or may not be possible to recover these symbols. An *erasure pattern* $\hat{f}$ is a set of erasures that result in at least one data symbol being irrecoverable (i.e., impossible to recover via any decoding method). The *erasures list* EL for an erasure code is the list of all its erasure patterns. A *minimal erasure* $\tilde{f}$ is an erasure pattern in which every erasure is necessary for it to be an erasure pattern; if any erasure is removed from $\tilde{f}$, then it is no longer an erasure pattern. The *minimal erasures list* MEL for an erasure code is the list of all its minimal erasures. A more compact representation of the EL and MEL are respectively the erasures vector EV, and the minimal erasures vector MEV. An erasures vector is a vector of length m in which the ith element is the total number of erasure patterns of size i in the EL; the minimal erasures vector is defined similarly with regard to the MEL. The EV and MEV vectors only need m entries because all erasure sets greater than m in length are necessarily erasure patterns.

Symbol	Definition
n	Total number of symbols in the erasure code.
k	Number of *data* symbols in the code.
m	Number of *parity* symbols in the code.
f	A *set of erasures.*
$\hat{f}$	An *erasure pattern.*
$\tilde{f}$	A *minimal erasure.*
EL	The *erasures list*: a list of $\hat{f}$.
MEL	The *minimal erasures list*: a list of $\tilde{f}$.
EV	The *erasures vector* for the EL.
MEV	The *minimal erasures vector* for the MEL.

Table 1. Terminology

2.1. Erasure codes

Plank's tutorial on erasure codes is a great introduction to erasure codes in general, and their applicability in storage systems in particular [12]. A Reed-Solomon erasure code uses m redundant symbols to tolerates all erasures of size m or less; it is therefore perfectly space efficient. Unfortunately, Reed-Solomon encode and decode require k operations to generate each redundant symbol, or to decode any data symbols using redundant symbols. The operations required by Reed-Solomon codes are based on arithmetic operations in Galois Fields (GF), and such operations are computationally more demanding than simple XOR operations. Cauchy Reed-Solomon codes implement Galois Field operations only using XOR operations, but require many XOR operations per Galois Field operation. XOR-based erasure codes are appealing because of the computational efficiency of encode and decode.

Two well known sub-classes of XOR-based erasure codes are low-density parity-check (LDPC) codes and array codes. LDPC codes trade imperfect space efficiency for improved performance. Luby et al. [8] identified methods of constructing LDPC codes, and efficiently encoding and decoding them; such codes were originally identified by Gallager [3]. Plank has briefly surveyed LDPC code constructions for their applicability to storage systems [14].

An LDPC code can be represented as a Tanner graph: a bipartite graph with k constraint nodes on one side and $k + m$ data and parity symbols on the other. The efficiency of LDPC codes hinges on bounding the degree of the nodes in the Tanner graph and consequently on iterative decoding. The efficacy of iterative decoding is significantly affected by *stopping sets*: erasure patterns that prevent iterative decoding from recovering symbols (e.g., see the work of Schwartz and Vardy [17]). We note that every minimal erasure *is* a stopping set but that the converse is *not* true. Stopping sets are defined with regard to iterative decoding; minimal erasures are defined with regard to decodability, i.e., without regard to any particular decoding method.

Array codes are specialized erasure codes for storage arrays (e.g., RAID 5 is an array code). Two well known double disk fault tolerant array codes are EVENODD by Blaum et al. [1] and Row-Diagonal Parity (RDP) by Corbett et al. [2]. Hafner has generalized the concept of XOR-based array codes to HoVer codes: codes with parity symbols in both *Ho*rizontal and *Ver*tical dimensions of the array [5]. He has also proposed Weaver codes, an XOR-based erasure code construction that chains parity symbols among a subset of servers in a redundancy group [4]. The ME Algorithm can be applied to any XOR-based erasure code, for example, to LDPC codes, Weaver codes, or array codes.

2.2. Evaluating erasure codes

The seminal RAID analysis by Patterson et al. [11] provides the framework that most storage system reliability analyses follow: identify an appropriate Markov model, plug in failure and recovery rates, and determine mean time to data loss (MTTDL). Saito et al. [16] and Rao et al. [15] both applied such a framework to analyze the reliability of erasure codes. The former considered Reed-Solomon erasure codes and the latter array codes.

Plank et al. analyzed the *read overhead* of moderate-sized LDPC codes using Monte Carlo methods [14] and of small-sized LDPC codes using deterministic methods [13]. Read overhead is a performance measure of a client random read policy; it measures the number of symbols beyond k that must be read, on average, to decode all data symbols. Recent work, done concurrently to our work, by Hafner and Rao investigates the reliability of XOR-based erasure codes [7]. The MEL and MEV output by the ME Algorithm are "threshold" measures of fault tolerance. A reliability measure requires additional assumptions about component failure and recovery rates. We discuss the ME Algorithm in the context of both of the above bodies of work in §5.

3. The Minimal Erasures (ME) Algorithm

The ME Algorithm uses the structure of an erasure code to efficiently generate the MEL. We rely on two representations of the XOR-based erasure code: the Generator matrix and the systematic Tanner graph. The *Generator matrix* of a (k, m)-code is a $k \times (k+m)$ matrix in GF(2). Addition of rows and columns in the Generator matrix is done modulo 2 (i.e., the XOR operation). The Generator matrix consists of a $k \times k$ identity matrix (the *data submatrix*) with m columns of dimension $k \times 1$ appended (the *parity submatrix*). Each of the k columns in the data submatrix corresponds to a stored data symbol. Each of the m columns in the parity submatrix corresponds to a stored parity symbol. Parity column p has a one in row i if, and only if, data symbol s_i is XOR'ed to determine p. For example, if $p = s_2 \oplus s_4$, then parity column p has a one in rows 2 and 4, and a zero in all other rows. We refer to the erasure pattern induced by the ones in the ith row of the Generator matrix as the ith *base erasure* $\tilde{f}_i$. (We show in §3.3 that a base erasure is a minimal erasure.) The structure of an erasure code is also captured by its Tanner graph $\mathcal{T}$. Since we exclusively consider systematic erasure codes—erasure codes that store the k data symbols and m parity symbols—we use a simplified Tanner graph representation. In the representation we use, we collapse the k data symbols from the one side into the constraint nodes on the other. In doing so, we end up with what we call a *systematic Tanner graph*: a bipartite graph that has k data symbols on one side and m parity symbols on the other.

At a high level, the ME Algorithm operates as follows. It begins by identifying the k base erasures (one for each data symbol) and adding them to the MEL. The ME Algorithm then proceeds, in an iterative fashion. For every minimal erasure it finds, it generates *child* erasure patterns. A minimal erasure has a child erasure pattern for every *adjacent data symbol*. Adjacency is defined with regard to the systematic Tanner graph. A data symbol is adjacent to a minimal erasure if it is connected to a parity symbol in the minimal erasure. To generate a child erasure pattern, the base erasure from the Generator matrix that corresponds to the adjacent data symbol is XOR'ed with the parent minimal erasure. A child erasure pattern is either a minimal erasure not yet in the MEL, a minimal erasure already in the MEL, or an erasure pattern that can be partitioned into minimal erasures that are already in the MEL. We refer to the last case as a composite erasure and discuss it in the next section. The algorithm recurses upon child erasure patterns until all minimal erasures in the MEL have no more children that are minimal erasures (not in the MEL), or have no adjacent data symbols (i.e., the minimal erasure contains all of the data symbols from some component of the Tanner graph).

3.1. ME Algorithm Pseudo-code

The pseudo-code for the ME Algorithm is given in Figure 1. Variables used in the pseudo-code are listed on lines 100–107. The function **me_search** enumerates the minimal erasures and stores them in the minimal erasures data structure M, which it returns. This function has two phases: in the first phase, the base erasures are enumerated

Published in proceedings of DSN 2007, The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 25 - June 28, 2007, Edinburgh, UK

```
100: $s, S$          /* Data symbol $s$; Set of data symbols $S$. */
101: $p, P$          /* Parity symbol $p$; Set of parity symbols $P$. */
102: $e, E$          /* Edge $e$ is a data/parity pair ($e = sp$); Set of edges $E$. */
103: $\mathcal{T}$   /* Tanner graph. Has structure $\mathcal{T}.(S, P, E)$. */
104: $\tilde{f}$     /* A minimal erasure. Has structure $\tilde{f}.(S, P)$. */
105: $M$             /* Minimal erasures data structure: a set of $\tilde{f}$. */
106: $\overline{M}$  /* Erasure patterns cache: a set of $\tilde{f}$. */
107: $Q$             /* Erasures queue: a FIFO queue of $\tilde{f}$. */

/* Search systematic Tanner graph $\mathcal{T}$ for minimal erasures. */
me_search($\mathcal{T}$) :
200: $M \leftarrow \emptyset, \overline{M} \leftarrow \emptyset, Q \leftarrow \emptyset$          /* Initialize data structures. */
201: /* Generate the $k$ base erasures. */
202: for all $(s' \in \mathcal{T}.S)$ do
203:     /* $P'$ contains all parities connected to $s'$. */
204:     $P' \leftarrow \{\forall p \in \mathcal{T}.P, \exists e \in \mathcal{T}.E : e = s'p\}$
205:     $\tilde{f} \leftarrow (\{s'\}, P')$          /* A base erasure. */
206:     $Q$.enqueue($\tilde{f}$)          /* Enqueue $\tilde{f}$ to process its children. */
207:     $M \leftarrow M \cup \{\tilde{f}\}$   /* Add $\tilde{f}$ to minimal erasures data structure. */
208: end for
209: /* Process children of enqueued erasure patterns. */
210: /* Repeat until no more erasure patterns are enqueued. */
211: while $(Q$.length$() > 0)$ do
212:     $\tilde{f} \leftarrow Q$.dequeue()          /* Get next erasure pattern to process. */
213:     $(M, \overline{M}, Q) \leftarrow$ me_children$(\mathcal{T}, M, \overline{M}, Q, \tilde{f})$
214: end while
215: return $(M)$

/* Generate children of $\tilde{f}$ and enqueue them in $Q$. */
me_children$(\mathcal{T}, M, \overline{M}, Q, \tilde{f})$
300: /* $S'$ contains all data symbols that are adjacent to $\tilde{f}$. */
301: $S' \leftarrow \{\forall s \in \mathcal{T}.S, \exists p \in \tilde{f}.P, \exists e \in \mathcal{T}.E : e = sp\} \setminus \tilde{f}.S$
302: for all $(s' \in S')$ do
303:     $\tilde{f}'.S \leftarrow \tilde{f}.S \cup \{s'\}$
304:     /* $P'$ contains all parities in $\mathcal{T}.P$ that are connected to $s'$. */
305:     $P' \leftarrow \{\forall p \in \mathcal{T}.P, \exists e \in \mathcal{T}.E : e = s'p\}$
306:     /* $\overline{P}$ contains all parities in $\tilde{f}.P$ that are connected to $s'$. */
307:     $\overline{P} \leftarrow \{\forall p \in \tilde{f}.P, \exists e \in \mathcal{T}.E : e = s'p\}$
308:     $\tilde{f}'.P \leftarrow (\tilde{f}.P \cup P') \setminus \overline{P}$
309:     if $(\tilde{f}' \in \overline{M})$ then continue
310:     $\overline{M} \leftarrow \overline{M} \cup \{\tilde{f}'\}$
311:     $Q$.enqueue($\tilde{f}'$)
312:     $\tilde{f}_{\text{MIN}} \leftarrow l : l \in M, \forall r \in M, |l| \leq |r|$
313:     if $(|\tilde{f}'| \geq 2|\tilde{f}_{\text{MIN}}|)$ then
314:         if (is_composite$(\mathcal{T} \setminus \{\tilde{f}'\})$) then continue
315:     end if
316:     $M \leftarrow M \cup \{\tilde{f}'\}$
317: end for
318: return $(M, \overline{M}, Q)$
```

Figure 1. ME Algorithm pseudo-code.

(cf. lines 201–208), and in the second phase, child erasure patterns are repeatedly enumerated (cf. lines 209–214). Each base erasure corresponds to one row of the Generator matrix; it consists of a single data symbol s' and the parities connected to it in the Tanner graph $\mathcal{T}$ (cf. line 205). Every base erasure is enqueued on the erasures queue Q, a FIFO queue, and inserted into M.

In the second phase of **me_search**, minimal erasures are dequeued from Q, and then **me_children** generates and processes the child erasure patterns. In **me_children**, line 301 determines which data symbols are adjacent to $\tilde{f}$. There is a child erasure pattern for every adjacent data symbol s'. Child erasure patterns are the XOR of $\tilde{f}$ with $\tilde{f}_{s'}$, the base erasure corresponding to data symbol s'. The pseudocode is written in set notation to make the relationship to the structure of $\mathcal{T}$ more apparent. A child is created as follows: s' is added to the parent (cf. line 303); parities connected to s' are added to the parent (cf. line 305); parities connected to s' and to some data symbol in the parent are removed (cf. lines 307 and 308). Regarding $\overline{P}$, there exists at least one such parity to remove, otherwise the data symbol s' would not be adjacent to $\tilde{f}$ (cf. line 301). If the child erasure pattern has previously been generated by **me_children**, i.e., it is already in the erasure patterns cache $\overline{M}$, then $\tilde{f}'$ is not processed (cf. line 309). If the child erasure pattern has not yet been processed, then it is added to $\overline{M}$ and to the erasures queue Q (cf. lines 310 and 311 respectively). Each child erasure pattern that is a minimal erasure is inserted into M (cf. line 316). Once all of the children are processed, **me_children** returns the updated M, $\overline{M}$ and Q.

Lines 312–315 deal with composite erasures: those child erasure patterns that can be partitioned into multiple minimal erasures and thus are not added to M. Only child erasure patterns at least twice as big as the shortest known minimal erasure can possibly be a composite erasure. Line 313 determines the length of the shortest minimal erasure in M. A child erasure pattern shorter than this length must be a minimal erasure since it is too small to be a composite erasure. A child erasure pattern twice or more this length is analyzed to determine if it is a minimal erasure or a composite erasure. The function **is_composite** called on line 314 determines if $\tilde{f}'$ is a composite erasure or not. It does so by testing the rank of the matrix that corresponds to $\{\mathcal{T} \setminus \tilde{f}'\} \cup \{e \in \tilde{f}'\}$. It is necessary and sufficient to remove a single symbol e from $\tilde{f}'$ to test for minimality: $\tilde{f}'$ is either a minimal erasure, and so removing e yields a matrix of full rank, or a composite erasure, and so removing e yields a matrix not of full rank. (Note that *removing* erasure e from $\tilde{f}'$ makes the symbol corresponding to e available for decoding.)

Once the ME Algorithm completes, the minimal erasures data structure M is trivially transformed into both the MEL and the MEV.

3.2. Example execution

Consider the XOR-based erasure code with $k = 4$ and $m = 4$ defined by the following Generator matrix:

$$\mathcal{C} = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{array}{cccccccc} s_1 & s_2 & s_3 & s_4 & p_1 & p_2 & p_3 & p_4 \\ \end{array}$$

$$\mathcal{C} = \begin{array}{cccccccc|c} s_1 & s_2 & s_3 & s_4 & p_1 & p_2 & p_3 & p_4 & \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \tilde{f}_{s_1} \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & \tilde{f}_{s_2} \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \tilde{f}_{s_3} \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & \tilde{f}_{s_4} \end{array}$$

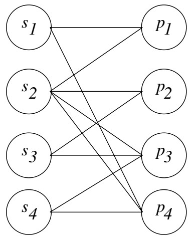

Figure 2. Systematic Tanner graph for code $\mathcal{C}$

The columns in $\mathcal{C}$ are labeled as either a data column or a parity column, and the rows are labeled as base erasures.

Figure 2 shows the systematic Tanner graph $\mathcal{T}$ for code $\mathcal{C}$. Table 2 summarizes the execution of the ME Algorithm for the code defined by $\mathcal{C}$. The first column lists the erasure pattern $\hat{f}$ being processed by **me_children**, the second column lists the children of $\hat{f}$ (possibly over multiple rows), and the third column indicates (via a checkmark) if $\hat{f}$ is inserted into M and enqueued in Q. The first four rows illustrate the base erasures $\tilde{f}_{s_1}, \tilde{f}_{s_2}, \tilde{f}_{s_3}, \tilde{f}_{s_4}$ processed by **me_search**. The remaining rows illustrate the children generated by dequeuing erasure patterns from Q and indicates which children are inserted into M. If a child is already in M, then it is not inserted again. Consider the first erasure pattern dequeued from Q: (s_1, p_1, p_4). From the systematic Tanner graph, it is easy to see that the adjacent data symbols are s_2 and s_4. The final steps of the ME Algorithm are elided because children generated by the remaining erasure patterns in Q are already in M. The MEL output by the ME Algorithm is $\{(s_1, p_1, p_4), (s_3, p_2, p_3), (s_4, p_3, p_4), (s_1, s_2, s_3), (s_1, s_2, p_2, p_3), (s_1, s_2, p_2, p_3), (s_2, s_3, p_1, p_4), (s_2, s_4, p_1, p_2), (s_2, s_3, p_1, p_4), (s_3, s_4, p_2, p_4)\}$. Any $\tilde{f}$ longer than $m = 4$ is elided from the MEL. The MEV output by the ME Algorithm is $(0, 0, 4, 5)$.

3.3. Correctness

In this section, we prove that the MEL of an XOR-based erasure code completely describes its fault tolerance (i.e., contains all of the information that the EL does), and that the ME Algorithm generates a complete MEL and so is correct.

We first prove that the MEL is a complete description of an XOR-based erasure code's fault tolerance.

Theorem 3.1 *If the fault tolerance of an* XOR*-based erasure code can be obtained through the* EL*, it can also be obtained through the* MEL*.*

Proof. By definition of minimality, every minimal erasure is an erasure pattern. Now for each erasure pattern $\hat{f}$ in EL, write down all possible minimal erasures. This can be done by exhaustively removing data and parity symbols from $\hat{f}$.

Q.**dequeue**	$\hat{f}$	M
	(s_1, p_1, p_4)	✓
	$(s_2, p_1, p_2, p_3, p_4)$	✓
	(s_3, p_2, p_3)	✓
	(s_4, p_3, p_4)	✓
(s_1, p_1, p_4)	(s_1, s_2, p_2, p_3)	✓
	(s_1, s_4, p_1, p_3)	✓
$(s_2, p_1, p_2, p_3, p_4)$	(s_1, s_2, p_2, p_3)	×
	(s_2, s_3, p_1, p_4)	✓
	(s_2, s_4, p_1, p_2)	✓
(s_3, p_2, p_3)	(s_2, s_3, p_1, p_4)	×
	(s_3, s_4, p_2, p_4)	✓
(s_4, p_3, p_4)	(s_1, s_4, p_1, p_3)	×
	(s_2, s_4, p_1, p_2)	×
	(s_3, s_4, p_2, p_4)	×
(s_1, s_2, p_2, p_3)	(s_1, s_2, s_3)	✓
	$(s_1, s_2, s_4, p_2, p_4)$	✓
...	...	...

Table 2. Example ME Algorithm execution.

Clearly, from each of these minimal erasures, the original erasure pattern $\hat{f}$ can be generated by adding back the respective data and parity symbols that were deleted. Now take the union of all these minimal erasures. Since this union is the MEL, and every erasure pattern in the EL can be generated from some minimal erasure in MEL, the theorem now follows. □

Consider the Generator matrix for the code and some erasure set f. We refer to the matrix that corresponds to the Generator matrix with all of the data and parity columns that correspond to f removed as the *recovery matrix.* We refer to a recovery matrix as a *defective recovery matrix* if it cannot be used to recover some data symbol. We refer to the rows in the recovery matrix that correspond to the data symbols in f—which we refer to via the notation $f.S$—as the *lost data rows.*

Proposition 3.2 *A recovery matrix is defective if and only if its rank is less than k.*

Proof. Follows from the definitions of erasure pattern and defective recovery matrix. □

Lemma 3.3 *Every base erasure is a minimal erasure.*

Proof. We first argue that a base erasure $\tilde{f}_b$ is an erasure pattern. By definition, a base erasure precisely corresponds to a row of the Generator matrix. The recovery matrix induced by $\tilde{f}_b$ has rank of $k - 1$ because it has an all zero row—the single lost data row—and an $(k-1)\times(k-1)$ identity matrix in the data submatrix. Therefore, by Proposition 3.2, the recovery matrix is defective.

To show that a base erasure is a minimal erasure, we need to establish that if any of the columns in the Generator matrix, that are not in the recovery matrix, is added into the recovery matrix, then the recovery matrix would have rank k. If the data column s_b corresponding to $\tilde{f}_b$ is added into the recovery matrix, then the data submatrix would contain an $k \times k$ identity matrix, thus have full rank, and not be defective. If one of the parity columns corresponding to some parity symbol $p \in \tilde{f}_b.P$ is added into the recovery matrix, then a column swap operation can move it into the data submatrix. Column additions within the data submatrix can then be performed until an identity matrix of size $k \times k$ is established. □

Lemma 3.4 *The addition (*XOR*) of any subset of the collection of base erasures is an erasure pattern.*

Proof. Let $\{\tilde{f}_1,\ldots,\tilde{f}_k\}$ denote the set of base erasures, and $f = \bigoplus_{i=1}^{k}\tilde{f}_i$ denote their sum. Consider the recovery matrix induced by f. Each of the lost data rows of the recovery matrix can be written as a linear combination of all of the other lost data rows. The lost data rows of the data submatrix are all zeroes and so are linear combinations of one another. The lost data rows of all the parity columns in the recovery matrix have even parity. In each lost data row then, each parity column is either a zero or a one: in either case, it can be written as a linear combination of the other lost data rows. If the column is a zero, then the other rows have even parity, and their sum is zero; if it is a one, then the other rows have odd parity and their sum is one. Therefore the rank of the recovery matrix must be less than k and so is defective; f is an erasure pattern. □

Unfortunately, the above lemma cannot be strengthened to say that f is a minimal erasure. The XOR of some sets of base erasures in some codes result in composite erasures comprised of two or more minimal erasures. However, all minimal erasures can be generated through the addition of base erasures. We prove this next.

Theorem 3.5 *Every minimal erasure can be obtained by the addition of some set of base erasures.*

Proof. Consider a minimal erasure $\tilde{f}$. By Proposition 3.2, the rank of the recovery matrix induced by $\tilde{f}$ is less than k, and therefore, at least one lost data row in the recovery matrix is linearly dependent on the other lost data rows, or all zeroes. Below we strengthen this observation.

Claim 3.1 *Every lost data row in the recovery matrix induced by $\tilde{f}$ is linearly dependent on the rest of the lost data rows, or is all zeroes.*

Proof. Suppose there exist lost data rows in the recovery matrix that are not linearly dependent on the rest of the lost data rows. Ignore all such rows; the remaining lost data rows are either linearly dependent on one another or a single lost data row remains. If a single lost data row remains, by Lemma 3.3, it is a base erasure (and is all zeroes), a contradiction to the minimality of $\tilde{f}$. On the other hand, let S' be the set of data symbols that correspond to the lost data rows that are linearly dependent on one another. Then, $S' \subset \tilde{f}.S$. Now, by Proposition 3.2, $\{S' \cup \tilde{f}.P\}$ is an erasure pattern, a contradiction to the minimality of $\tilde{f}$. □

Claim 3.2 *In any subset of rows corresponding to lost data symbols in the recovery matrix, if every row is linearly dependent on the rest of the rows, then the respective parity columns must have even parity.*

Proof. Suppose there is a column that has odd parity. A lost data row with a zero in this column cannot be linearly dependent on all other lost data rows. □

By Claims 3.1 and 3.2, the parity columns of the recovery matrix induced by $\tilde{f}$ must have even parity. So, by an argument similar to the proof of Lemma 3.4, $\tilde{f}$ can be written as the sum of the base erasures that correspond to the data symbols in $\tilde{f}.S$. This proves the theorem. □

3.4. Bounds on |MEL| and |EL|

The bound on the size of the EL is as follows:

$$|\mathrm{EL}| \leq \sum_{i=1}^{m} \binom{k+m}{i} < 2^{k+m}$$

We use the fact that all erasure patterns of interest are less than or equal to m in length to tighten the bound on $|\mathrm{EL}|$.

Let BEL be the base erasures list: the XOR of each set in the powerset of base erasures, except the null set. It is thus the union of all minimal erasures and all composite erasures. The bound on the size of the BEL is as follows:

$$|\mathrm{BEL}| \leq \begin{cases} \sum_{i=1}^{m}\binom{k}{i} & \text{if } m < k, \\ 2^k & \text{if } m \geq k. \end{cases}$$

The bound on the size of the MEL is as follows:

$$|\mathrm{MEL}| \leq \begin{cases} \min\left(\binom{k+m}{m}, \sum_{i=1}^{m}\binom{k}{i}\right) & \text{if } m < k, \\ \min\left(\binom{k+m}{\lfloor (k+m)/2 \rfloor}, 2^k\right) & \text{if } m \geq k. \end{cases}$$

The first term in the minimum follows from the bound on $|\mathrm{EL}|$ and the fact that a minimal erasure of size i precludes many potential minimal erasures of size $i+1$: $|\mathrm{MEL}|$ is thus bound by the largest possible term in the summation that bounds $|\text{EL}|$. The second term in the minimum follows from the bounds on $|\text{BEL}|$.

The difference in the bounds on $|\text{EL}|$ and $|\text{MEL}|$ suggests that we can expect many more erasure patterns than minimal erasures. Unfortunately, the bound on $|\text{MEL}|$ indicates that we can still expect the number of minimal erasures to grow exponentially in k.

Consider the example execution from §3.2. For that code, the size of the MEL is 9. This is less than the size of the BEL for the code: $9 < 2^4 = 16$. It is also much less than the size of the EL for the code: 29. (We used a program described in §4.1 to determine the EL for the code.) The value 29 fits the bound for $|\text{EL}|$: $29 < \sum_{i=1}^{4}\binom{8}{i} = 162$.

3.5. Using EV or MEV to compare two codes

The EV and MEV can be used to compare the fault tolerance of any two XOR-based erasure codes. An erasure vector is written as $(j_1, j_2, \ldots, j_m)$ and j_i indicates the number of erasure patterns of size i. A minimal erasure vector is written similarly. Note that the first non-zero entry, j_i, in the MEV and the EV for a code are identical and indicate that the Hamming distance for the code is i (i.e., that the code tolerates all erasures of size $i-1$).

To compare two erasure vectors, the value in the vector are compared, from shortest to longest (i.e., from 1 to m). If the value of the first entry in EV that is distinct from that in EV$'$ is greater, then $\text{EV} < \text{EV}'$. For example, if $\text{EV} = (0,4,5)$ and $\text{EV}' = (0,0,10)$, then $\text{EV} < \text{EV}'$, because $4 > 0$; and, if $\text{EV} = (0,4,5)$ and $\text{EV}' = (0,4,4)$, then $\text{EV} < \text{EV}'$, because $5 > 4$. If $\text{EV} < \text{EV}'$, then the code corresponding to EV$'$ is the more fault tolerant. Minimal erasure vectors can be compared similarly. For two codes with the same k and m, the result of comparing the MEV is the same as the result of comparing the EV. For two codes that differ in k and/or m, the result of comparing the MEV is only necessarily the same as the result of comparing the EV if the codes have different Hamming distances, or if the codes have the same Hamming distance but different values in the MEV at the Hamming distance.

4. Evaluation

In this section, we evaluate our implementation of the ME Algorithm: `mela`. In §4.1, we describe how we validated the correctness of the `mela` implementation. In §4.2, we use `mela` to determine the MEV of all XOR-based erasure codes with $1 \le k, m \le 7$. One major result from this section is that we identify the most fault tolerant XOR-based erasure codes over these parameters. Another major result from this section is that, in practice, the average of the ratio $\frac{|\text{EL}|}{|\text{MEL}|}$ tends to increase with k and m. The average of this ratio is over 80 for one corpus of codes we evaluated. This result supports our claim that it is more efficient to determine the MEL than the EL (especially as k and m increase).

We used Python 2.4.3 to implement `mela` and the rest of our tool suite. Python code is easy to modify and so allows us to quickly prototype modifications and extensions of the ME Algorithm. The minimal erasures data structure and cache, M and $\overline{M}$ respectively, are implemented via m dictionaries. The kth dictionary only stores erasure patterns of length k. We store erasure patterns as bitmaps because they are concise and efficient to compare. Testing for membership in a dictionary is efficient in Python. Dictionaries keep track of the number of elements they store and so conversion from M to the MEV is trivial. The minimal erasures queue Q is a FIFO queue implemented via a list.

We also implemented a tool, `mel2el`, that efficiently transforms a MEL into a EL using set operations. `mel2el` must perform O($|\text{EL}|$) operations and is efficient only in that it does not perform any matrix rank tests. We use `mel2el` to determine the EL and EV for some codes in this section.

We use `nauty` version 2.2 with `gtools` to generate Tanner graphs for `mela` to evaluate [10, 9]. Specifically, we use `genbg` to generate non-isomorphic bipartite graphs that we translate into systematic Tanner graphs. Isomorphic Tanner graphs have similar fault tolerance characteristics and so we only evaluate non-isomorphic Tanner graphs. We refer to the set of all possible non-isomorphic systematic Tanner graphs with a common k and m as the (k,m)-code corpus. For a given k and m, there are up to 2^{km} possible systematic XOR-based codes; there are dramatically fewer codes in the (k,m)-code corpus because we only include non-isomorphic Tanner graphs.

All execution times in this section are based on execution on an HP DL360 computer with a 2.8 GHz Intel Xeon processor and 4 GB of RAM.

4.1. Validation of implementation

To validate the correctness of the `mela` implementation we generate the MEL via "brute force" for some codes. We implemented a program, `ela`, that generates the EL for a given code. Roughly speaking, `ela` performs a matrix rank test for every erasure pattern with one erasure, then all erasure patterns with two erasures, and so on, up to all erasure patterns with m erasures. A matrix rank test indicates if the erasure pattern is decodable. We developed another program, `el2mel`, that filters the EL output by `ela` to produce the MEL. The implementation of `el2mel` is based on inserting the erasure patterns in the EL into a data structure that checks for the subset or equal to relationship. If an erasure pattern is a superset of an erasure pattern that is already in the data structure, then it is not inserted. If an erasure pattern is successfully inserted, then all larger erasure patterns in the data structure are checked to see if they are a superset of the inserted erasure pattern; any that are a superset are removed from the data structure.

We validated the correctness of `mela` for the $(4, 4)$-code corpus; it contains 179 codes. The MEL output by `mela` exactly matches the MEL output by `ela` piped to `el2mel`. It took less than one second for `mela` to complete, 231 seconds for `ela` to complete, and less than one second for `el2mel` to complete.

4.2. Fault tolerance of XOR-based codes

We evaluated all (k, m)-code corpi for $1 \leq k, m \leq 7$ with `mela`. Table 3 lists the results. The first two columns list k and m for each code corpus. The third column lists the number of codes in the (k, m)-code corpus. The fourth column, # w MEV*, lists the number of codes in the corpus that share the best minimal erasures vector. The fifth column, # w d^*, lists the number of codes in the corpus that share the best Hamming distance. The sixth column lists a parity submatrix for a code from the corpus that has an MEV equal to MEV*. The parity submatrix is presented in "bitmap" representation: each integer represents a column of the parity submatrix (i.e., if the parity symbol includes data symbol s_j, then add 2^{j-1} to the integer representation). For example, for the $(4, 3)$-code corpus, "7, 11, 13" means that the first parity is the XOR of $s_1, s_2, \& s_3$, the second parity is the XOR of $s_1, s_2, \& s_4$, and the third parity is the XOR of $s_1, s_3, \& s_4$. The seventh column, MEV*, lists the value of the best MEV in the corpus. The eighth column, d^*, lists the best Hamming distance in the corpus (i.e., the index of the first non-zero entry in MEV*). The ninth column, $\frac{|\text{EL}|}{|\text{MEL}|}$, lists the average of the ratio $|\text{EL}|$ to $|\text{MEL}|$ for the corpus.

We do not list the rows for $k = 1$ or the columns for $m = 1$ in Table 3 because each such corpus has only one code: replication for $k = 1$ and RAID 4 for $m = 1$. Replication has no erasure patterns that lead to irrecoverable data loss of size less than or equal to m and so both the MEL and EL are empty. RAID 4 tolerates all erasure patterns of size 1 and so again, both the MEL and EL are empty.

We make the following observations about the results:

- The average of the ratio $\frac{|\text{EL}|}{|\text{MEL}|}$ tends to increase with both k and m. For the $(5, 7)$-code corpus, directly calculating the EL rather than the MEL requires $84.9\times$ more steps. We were surprised though that the $(7, 7)$-code corpus did not have the greatest average ratio.

- As m increases with regard to some fixed k, the best MEV improves; this makes sense since there is more redundancy. As k increases with regard to some fixed m, the best MEV degrades; this makes sense since the same amount of redundancy is protecting more data.

- As k and m increase, very few codes share the absolute best MEV. For example, in the $(7, 7)$-code corpus, only 1 in 1.48 million codes has the best fault tolerance. However, 1 in 30 codes does share the best Hamming distance.

- The data symbols for the best codes are included by at least d^* parity symbols. This is because we focus exclusively on systematic codes. The best codes thus have higher Hamming weight (are more connected) than would be expected from reading the LDPC literature. The LDPC literature, in general, does not consider systematic codes.

- We were surprised by the code corpi in which a specific parity symbol, the XOR of all data symbols, is replicated (cf. corpi with $k = 2$ & $m > 2$, and $k = 3$ & $m > 4$). We wonder if this is generally true: if m sufficiently exceeds k, then the "RAID 4 parity symbol" is replicated many times in the best code.

- We were surprised to find that there does not exist a systematic XOR-based erasure code with $5 \leq k \leq 7$ and $m = 7$ that tolerates all erasures of size 4.

To appreciate the value of considering only non-isomorphic Tanner graphs in a corpus, compare the number of codes in any (k, m)-code corpus to the value of 2^{km}. For the $(7, 7)$-code corpus the reduction is of over a factor of 19 million times. We had hoped to consider all code corpi up to $k = 10$ and $m = 10$ but the growth in the size of such corpi was prohibitive. To identify the best codes in larger corpi we may need to develop additional theory to reduce the number of codes we need to evaluate in each corpus, improve the efficiency of the `mela` implementation, or get a larger compute cluster. Our approach to enumerating all non-isomorphic codes for a given k and m to evaluate is quite different from the traditional approach of identifying *families* of codes: most coding theorists focus on identifying a code family, parameterized on k and m, that constructs "good" codes given k, m, and possibly a random seed.

5. Discussion

Concurrently to our work, Hafner and Rao investigated the reliability of irregular erasure codes [7]. They did so in the "standard" RAID framework: they developed a Markov model with failure and recovery rates for various components. Irregular XOR-based codes do not simply "plug" into such a model though. They calculate the conditional probabilities q_j that a state in their Markov model with j failures results in irrecoverable data loss (note that they use subscript k not j for their notation). They do so by counting the number of erasure sets of size j that do not lead to data loss for all $j \leq m$ (resulting in vector s_j). To get q_j, they divide s_j by $\binom{k+m}{j}$. The vector s_j is the complement of the

k	m	# in corpus	# w MEV*	# w d^*	A parity submatrix w MEV*	MEV*	d^*	$\frac{\lvert\text{EL}\rvert}{\lvert\text{MEL}\rvert}$
2	2	3	2	3	1, 3	(0, 1)	2	1.8
2	3	5	1	2	1, 3, 3	(0, 0, 1)	3	2.9
2	4	8	1	2	1, 3, 3, 3	(0, 0, 0, 1)	4	4.9
2	5	11	1	2	1, 3, 3, 3, 3	(0, 0, 0, 0, 1)	5	8.4
2	6	15	1	2	1, 3, 3, 3, 3, 3	(0, 0, 0, 0, 0, 1)	6	14.2
2	7	19	1	2	1, 3, 3, 3, 3, 3, 3	(0, 0, 0, 0, 0, 0, 1)	7	24.0
3	2	5	2	5	3, 5	(0, 2)	2	2.0
3	3	17	2	2	3, 5, 6	(0, 0, 4)	3	2.8
3	4	42	1	1	3, 5, 6, 7	(0, 0, 0, 3)	4	6.7
3	5	91	1	1	3, 5, 6, 7, 7	(0, 0, 0, 0, 3)	5	13.9
3	6	180	1	1	3, 5, 6, 7, 7, 7	(0, 0, 0, 0, 0, 3)	6	26.4
3	7	328	1	1	3, 5, 6, 7, 7, 7, 7	(0, 0, 0, 0, 0, 0, 3)	7	47.7
4	2	8	1	8	7, 11	(0, 3)	2	2.0
4	3	42	1	1	7, 11, 13	(0, 0, 7)	3	3.4
4	4	179	1	1	7, 11, 13, 14	(0, 0, 0, 14)	4	6.3
4	5	633	1	14	3, 5, 9, 14, 15	(0, 0, 0, 6, 1)	4	16.2
4	6	2001	2	166	3, 5, 7, 9, 14, 15	(0, 0, 0, 2, 2, 1)	4	36.7
4	7	5745	1	10	3, 5, 7, 11, 13, 14, 15	(0, 0, 0, 0, 1, 2, 1)	5	75.5
5	2	11	2	11	7, 27	(0, 5)	2	2.3
5	3	91	3	91	7, 11, 29	(0, 1, 10)	2	4.0
5	4	633	3	35	7, 11, 19, 29	(0, 0, 4, 14)	3	7.6
5	5	3835	2	14	7, 11, 19, 29, 30	(0, 0, 0, 10, 16)	4	14.7
5	6	20755	4	542	3, 5, 15, 23, 25, 30	(0, 0, 0, 4, 14, 1)	4	36.3
5	7	102089	1	11890	7, 11, 13, 14, 19, 21, 25	(0, 0, 0, 1, 8, 0, 1)	4	84.9
6	2	15	2	15	15, 51	(0, 7)	2	2.4
6	3	180	6	180	7, 27, 45	(0, 2, 14)	2	4.6
6	4	2001	6	35	7, 27, 45, 56	(0, 0, 8, 18)	3	8.8
6	5	20755	7	12	7, 25, 42, 52, 63	(0, 0, 0, 25, 0)	4	16.9
6	6	200082	5	1338	7, 27, 30, 45, 53, 56	(0, 0, 0, 6, 24, 16)	4	31.8
6	7	1781941	19	118130	7, 11, 21, 25, 45, 51, 62	(0, 0, 0, 2, 16, 18, 1)	4	75.3
7	2	19	1	19	31, 103	(0, 9)	2	2.6
7	3	328	7	328	15, 51, 85	(0, 3, 19)	2	5.1
7	4	5745	10	28	15, 54, 90, 113	(0, 0, 12, 26)	3	10.1
7	5	102089	8	10	7, 57, 90, 108, 119	(0, 0, 0, 38, 0)	4	19.1
7	6	1781941	57	2610	7, 46, 56, 75, 85, 118	(0, 0, 0, 14, 28, 24)	4	35.7
7	7	29610804	20	965097	7, 27, 45, 51, 86, 110, 120	(0, 0, 0, 3, 24, 36, 16)	4	65.3

Table 3. Evaluation of all (k, m)-code corpi for $1 \leq k, m \leq 7$.

EV in our work; it counts the number of erasure sets of size j that are not an erasure pattern, whereas the EV counts the number of erasure patterns of size j.

Hafner and Rao claim to compute s_j by "straightforward calculation" using techniques they previously developed [6]. (We note that those previous techniques are based on the pseudo-inverse of the Generator matrix and efficiently determines if a set of erasures is an erasure pattern or not, and if not, outputs how to reconstruct data.) The ME Algorithm efficiently calculates the MEL and MEV. We ran `mela` and `mel2el` on the Tanner code from Plank's RAID tutorial [12] that Hafner and Rao analyze in §3.2 of [7]. Our analysis took less than one second and produced an EV compatible with the s_j they produce.

Plank et al. analyzed the *read overhead* for LDPC constructions [14]. Read overhead is the expected number of symbols that must be read to recover all of the data symbols, assuming a random read order. Read overhead is a good performance metric for irregular XOR-based erasure codes deployed in grid storage environments; in LAN settings, we expect that storage systems would employ systematic codes and read data symbols ("stripes") before reading any parity symbols. More recently, Plank et al. [13] analyzed the read overhead for codes with small m. Our analyses overlap for the (k, m)-code corpi with $k, m \leq 5$. We believe that the EV can be transformed into a read overhead metric, but have not yet determined this transformation.

6. Conclusions

We identified a new fault tolerance metric for XOR-based erasure codes, the minimal erasures list (MEL), a concise representation of that metric, the minimal erasures vector (MEV), and the Minimal Erasures (ME) Algorithm which efficiently determines the MEL. We applied the implementation of the ME Algorithm to all systematic XOR-based erasure codes with $1 \leq k, m \leq 7$, and so identified the most fault tolerant such codes. We presented empirical evidence that the ME Algorithm requires less work (over a factor of $80\times$) than an algorithm that directly generates all erasure patterns.

Acknowledgements

We thank our colleagues Vinay Deolalikar, Xiaozhou Li, Craig Soules, Krishnamurthy Viswanathan, and Pascal Vontobel for their feedback. We also thank the anonymous DSN reviewers for their thorough reviews and suggestions.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput.*, 44(2):192–202, 1995.

[2] 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-2004: 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2004.

[3] R. G. Gallager. *Low density parity-check codes*. MIT Press, 1963.

[4] J. L. Hafner. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th USENIX Conference on File and Storage Technologies*, pages 212–224. USENIX Association, December 2005.

[5] J. L. Hafner. HoVer erasure codes for disk arrays. In *DSN-2006: The International Conference on Dependable Systems and Networks*, pages 217–226. IEEE, June 2006.

[6] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th USENIX Conference on File and Storage Technologies*, pages 183–196. USENIX Association, December 2005.

[7] J. L. Hafner and K. Rao. Notes on reliability models for non-MDS erasure codes. Technical Report RJ–10391, IBM, October 2006.

[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Practical loss-resilient codes. In *STOC 1997: Proceedings of the 29th annual ACM Symposium on Theory of Computing*, pages 150–159. ACM Press, 1997.

[9] B. McKay. `nauty` version 2.2 (including `gtools`). `http://cs.anu.edu.au/~bdm/nauty/`.

[10] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[11] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[12] J. S. Plank. Erasure codes for storage applications. Tutorial slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, `http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html`, December 2005.

[13] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason. Small parity-check erasure codes - exploration and observations. In *DSN-2005: The International Conference on Dependable Systems and Networks*. IEEE, July 2005.

[14] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN-2004: The International Conference on Dependable Systems and Networks*, pages 115–124. IEEE, June 2004.

[15] K. Rao, J. L. Hafner, and R. A. Golding. Reliability for networked storage nodes. In *DSN-2006: The International Conference on Dependable Systems and Networks*, pages 237–248. IEEE, June 2006.

[16] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM Press, 2004.

[17] M. Schwartz and A. Vardy. On the stopping distance and the stopping redundancy of codes. *IEEE Trans. on Inf. Theory*, 52(3):922–932, 2006.