WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems

James Lee Hafner
IBM Almaden Research Center
hafner@almaden.ibm.com

Abstract

We present the WEAVER codes, new families of simple highly fault tolerant XOR-based erasure codes for storage systems (with fault tolerance up to 12). The design features of WEAVER codes are (a) placement of data and parity blocks on the same strip, (b) constrained parity in-degree and (c) balance and symmetry. These codes are in general not maximum distance separable (MDS) but have optimal storage efficiency among all codes with constrained parity in-degree. Though applicable to RAID controller systems, the WEAVER codes are probably best suited in dRAID systems (distributed Redundant Arrangement of Independent Devices). We discuss the advantages these codes have over many other erasure codes for storage systems.

1 Introduction

It has become increasingly clear in the storage industry that RAID5 does not provide sufficient reliability against loss of data either because of multiple concurrent disk losses or disk losses together with sector losses (e.g., due to medium errors from the disks). The reasons are primarily due to the dramatic increase in single disk capacity together with a fairly constant per-bit error rate. Additional factors, as mentioned in [5], include increasing number of disks per system, and use of less reliable disks such as ATA (vs. SCSI). The cited paper makes the case for double fault tolerance; by extrapolation, there is (or will be) a need for higher fault tolerant codes if these trends continue. Furthermore, as the industry moves into very long-term archival storage or dRAID (distributed Redundant Arrangement of Independent Devices) node-based systems, the need for higher fault-tolerance erasure codes will likely become more acute.

N-way mirroring can clearly be used to provide additional redundancy in any system, but the storage efficiency (ratio of user data to the total of user data plus redundancy data) of mirroring is very low. (We prefer the term "efficiency" in storage contexts instead of the equivalent term "rate" which is more suitable for communication channels.) On the other hand, codes like Reed-Solomon (RS) [14] provide optimal storage efficiency (that is, are maximum distance separable, or MDS) and arbitrarily high fault tolerance, but require special purpose hardware to enable efficient computation of the finite field arithmetic on which the codes are based (or, if formulated as a binary XOR code, generally have higher computational costs and complexities).

Other erasure codes have been proposed for better fault tolerance than RAID5, but none has emerged as a clear winner even in the RAID controller market – the industry has not even settled on a *de facto* standard for 2 fault tolerance after 40+ years (since RS codes were first proposed). We believe no such "perfect" code can exist; every code requires some trade-offs in efficiency, performance or fault tolerance.

In this paper, we present the WEAVER codes, so called because the parity/redundancy values are computed by XOR formulas defined by patterns that weave through the data. There are three design principles that characterize WEAVER codes: (a) every strip (stripe unit) contains both data and parity from the same stripe (we call these vertical codes because data and parity are arranged together vertically on each strip), (b) the number of data values that contribute to each parity value (parity in-degree) is fixed and, most importantly, is independent of the stripe size (number of strips) and (c) balance and symmetry. The second property enables flexibility in choices of stripe sizes without altering computational costs (in both XOR and IO). In addition, it bounds the computational costs of many operations (e.g., short writes, rebuild). More details on these points are given in Section 3 and elsewhere.

The WEAVER codes are designed with balance and symmetry in three aspects. First, every parity is constructed from some fixed number of data values (we call this number the "parity in-degree" – as noted, it is independent of the stripe size). Second, each data value contributes to a fixed number of parity values (we call this the "data out-degree"; it is also independent of the stripe size). The data out-degree for WEAVER codes is set to the fault tolerance, the theoretical minimum number for the given fault tolerance. For additional symmetry, some of the codes have the parity in-degree equal to the data out-degree – this provides a certain duality between data and parity. Third, all the code constructions are specified by a weave-pattern which is repeated by simple rotation of a base configuration. That is, they are rotationally symmetric.

We bound the parity in-degree by the fault tolerance to control the complexity of parity computations and improve other properties such as localization (see Section 3.1.2). There are undoubtedly other whole families of codes yet to be discovered that relax this requirement, while keeping the parity in-degree fixed (some Wiencko codes [15, 6] may have this property as well).

There are two general families and one ad-hoc family of WEAVER codes, which we describe in detail later. We briefly mention here that there are constructions of WEAVER codes that tolerate up to 12 device failures (and perhaps beyond). A key feature of all WEAVER codes is the "localization" property that for large stripe sizes limits the scope of most operations (including, for example, rebuild) to small subsets of the stripe. This is discussed in more detail in Section 3.1.2.

The WEAVER codes are in general *not* MDS codes (though some special cases are). Consequently, the main disadvantage of these codes is their storage efficiency. However, these codes are optimally efficient for the given fault tolerance and parity in-degree constraint (see Section 3.2). Of the three families of WEAVER codes, one family has efficiency 50% for all levels of fault tolerance (up to 10 in our constructions); the other families have lower efficiency which decreases with increasing fault tolerance. In all cases, these codes have significantly higher efficiency than N-way mirroring. The WEAVER codes, by their symmetry, have a certain simplicity of implementation (though not as simple as N-way mirroring). As such these codes provide a way for a system designer to select highly fault tolerant codes that interpolate between N-way mirroring with its performance advantages, exceptional simplicity but minimal efficiency and MDS codes with somewhat lower performance and somewhat greater complexity but optimal efficiency.

Unfortunately, we do not have many theoretical results concerning specific constructions. Generally, for small fault tolerance, these codes can be tested by hand (in some cases, we give the proof). For other cases, our constructions were tested by computer program using the generator matrix (see Section 4 and [10] for related methodology).

The paper is organized as follows. We close this introduction with some definitions and notation. In Section 2 we describe the various constructions for each family of WEAVER codes. Section 3 lists the key advantages and disadvantages of these codes. Related work and comparisons with other published codes are discussed in detail in Section 5. Section 4 outlines our testing methodology. We conclude with a short summary.

1.1 Vocabulary and Notations

The literature contains some inconsistency concerning the use of common storage and erasure code terms, so we state our definitions here to avoid confusion. We use the term "system" to refer to either a dRAID storage system of node-type devices or to a controller array of disks (RAID). The term "device" will refer to the "independent" storage device in the system (a node in dRAID or a disk in RAID).

element: a fundamental unit of data or parity; this is the building block of the erasure code. In coding theory, this is the data that is assigned to a bit within the symbol. For XOR-based codes, this is typically one or more sequential sectors on a disk (or logical sectors on a storage node).

stripe: a complete (connected) set of data and parity elements that are dependently related by parity computation relations. In coding theory, this is a code word; we use "code instance" synonymously.

strip: a unit of storage consisting of all contiguous elements (data and/or parity) from the same device and stripe (also called a stripe unit). In coding theory, this is a code symbol. The set of strips in a code instance form a stripe. Typically, the strips are all of the same size (contain the same number of elements).

vertical code: an erasure code in which a (typical) strip contains both data elements and parity elements (e.g., X-code [17] or these WEAVER codes). Contrast this notion with a "horizontal code" in which each strip contains either data elements or parity elements, never both (e.g., EVENODD [2]).

We use the symbol t exclusively to represent the fault tolerance of a code, the symbol n for the size of the stripe (the number of strips, or equivalently, the number of devices in a code instance), and k for the maximum parity in-degree. For WEAVER codes, all parity have in-degree exactly k and $k \leq t$. In addition, all data have out-degree equal to t. We therefore parameterize our codes as WEAVER(n,k,t), and we provide constructions for different values of these parameters. We let r denote the number of data elements and q the number of parity elements per strip. (We see how r and q may be determined from k and t in Section 2.)

We define a "short write" as a host write to any sequential subset of an element (e.g., a single sector); a "multiple strip write" as a host write to a sequential subset of the strips in a stripe (that is, the user data portion of the strips). The "write lock zone" is the set of elements that should be locked during a short write so as to provide data/parity consistency in case of failures encountered during a write operation. The "rebuild zone" is the subset of strips within the stripe which are needed during

a rebuild of one or more lost strips. We see why these notions are relevant to WEAVER codes in Section 3.

2 WEAVER code definitions

In this section we describe our WEAVER constructions. We use a graph representation to visualize the constrained parity-in degree and fixed data-out degree. A table format is used to visualize the data and parity layout on strips (and so devices). Formulas for "parity defining sets" (see below) are used to precisely define each construction. Each of these presentations exhibit some of the balance and symmetry of the WEAVER codes.

Figure 1 shows a directed, bipartite graphical representation of a general WEAVER code; the nodes on top represent the data elements in the stripe and the nodes on the bottom represent the parity elements. An edge connects a data element to a parity element if that data element contributes to the parity value computation (we say that the data element "touches" this parity element). For WEAVER(n,k,t) codes, each data element has data out-degree equal to t. In addition, each parity element has parity in-degree exactly k, where $k \leq t$. As mentioned, this constraint is the key to many of the good properties of these codes.

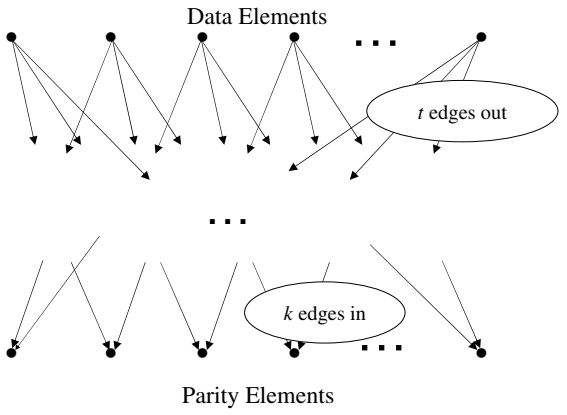

Figure 1: Graphical representation of a general WEAVER(n,k,t) code. Each parity element has in-degree equal to k; each data element has out-degree equal to t.

Generally, a graphical representation like that of Figure 1 can be used for any XOR-based erasure code (for example, see the Tornado codes [11]). In addition, Tanner graphs may also be used (see the description of the Low-Density Parity-Check codes in [13, 12]). In Tanner graphs the nodes on one side represent data and parity and the opposite nodes represent parity checks. The systematic and regular nature of our WEAVER codes makes the Tanner representation less useful for visualization. We use our graphs to show the encoding of data/parity relations, and not for decoding as in [11]. We also draw our graph with nodes on top and bottom (not left/right) to suggest a relationship to the data/parity layout onto strips as described next.

Figure 1 only provides a partial description of the code in the context of storage systems. The physical layout of data and parity on the strips within the stripe must also be specified. This is given in Figure 2 where we see the vertical nature of the WEAVER codes. As noted, other codes share this vertical layout (see, for example, the X-code [17] and the BCP code [1]). We view the logical addressing of the data elements from the host's viewpoint as first within a strip and then strip to strip.

S_0	S_1	$\cdots$	S_j	$\cdots$	S_{n-1}
$d_{0,0}$	$d_{0,1}$	$\cdots$	$d_{0,j}$	$\cdots$	$d_{0,n-1}$
$d_{1,0}$	$d_{1,1}$	$\cdots$	$d_{1,j}$	$\cdots$	$d_{1,n-1}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$d_{r-1,0}$	$d_{r-1,1}$	$\cdots$	$d_{r-1,j}$	$\cdots$	$d_{r-1,n-1}$
$p_{0,0}$	$p_{0,0}$	$\cdots$	$p_{0,j}$	$\cdots$	$p_{0,n-1}$
$p_{1,0}$	$p_{1,0}$	$\cdots$	$p_{1,j}$	$\cdots$	$p_{1,n-1}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$p_{q-1,0}$	$p_{q-1,0}$	$\cdots$	$p_{q-1,j}$	$\cdots$	$p_{q-1,n-1}$

Figure 2: Stripe/strip layout of general WEAVER(n,k,t) code. Each strip contains r data elements and q parity elements. S_j denotes a strip label; $d_{i,j}$ is a labeled data element; $p_{i,j}$ is a labeled parity element. Each strip is stored on a different device in the system.

Of course, there is no requirement that the parity elements be placed below the data elements; they must be placed however on the same device. WEAVER codes always have both data and parity elements on each strip. Essential for performance is that the data elements be logically contiguous, as must the parity elements.

We can also represent our codes by sets of indices. A parity element $p_{i,j}$ can be relabeled $p_{\kappa(i,j)}$ where $\kappa(i,j)$ is the set of (ordered pair) indices of the data elements that touch this parity element, That is,

$$p_{i,j} \rightarrow p_{\kappa(i,j)} = \bigoplus_{(u,v)\in\kappa(i,j)} d_{u,v}. \qquad (1)$$

Conversely, we can relabel the data elements as $d_{\tau(i,j)}$ where $\tau(i,j)$ is the set of (ordered pair) indices of the parity elements that are touched by $d_{\tau(i,j)}$. That is,

$$\tau(i,j) = \{(u,v) \ : \ (i,j) \in \kappa(u,v)\}. \qquad (2)$$

In the graph of Figure 1, $\kappa(i,j)$ can label the set of edges into parity node $p_{i,j}$, and, similarly, $\tau(i,j)$ can label the set of edges out of data node $d_{i,j}$. This notation is used to provide formulas to define specific constructions.

For notation, define for any set of ordered pairs of indices κ and any offset s, a new set of indices referred to as $\kappa + s$ using the operation

$$\kappa + s \stackrel{\text{def}}{=} \{(u, v + s \bmod n) \ : \ (u, v) \in \kappa\},$$

so each column index in $\kappa + s$ is offset by s modulo n.

Using this notation, we provide rotational symmetry (a key design feature of WEAVER codes). For $0 \le j \le n-1$ and $0 \le i \le q-1$, set

$$\kappa(i, j) = \kappa(i, 0) + j. \tag{3}$$

In other words, a specification for the parities $p_{\kappa(i,0)}$ for $0 \le i \le q-1$ (on the first strip), together with rotation to the right (with wrap-around) provides a complete specification of WEAVER(n,k,t) erasure codes. We call the sets $\kappa(i, j)$ "parity defining sets". A similar rotational formula can be derived for the sets $\tau(i, j)$.

By counting edges in two ways, it is easy to see from Figures 1 and 2 that $rt = qk$. Generally, a WEAVER(n,k,t) code will have r and q minimal (to minimize overall complexity): so $r = k/m$ and $q = t/m$ where $m = \gcd(k, t)$. This is assumed throughout unless otherwise noted.

Given these parameters, the storage efficiency for these codes is given by

$$\text{Eff} = \frac{nr}{nr + nq} = \frac{r}{r + q} = \frac{k}{k + t}. \tag{4}$$

The first two are obvious from Figure 2, the latter comes from the relation $r = qk/t$. Since we assume $k \le t$, the maximum efficiency for any WEAVER code is 50%.

In the next few subsections, we describe specific constructions of parity defining sets that provide for prescribed fault tolerance.

2.1 WEAVER codes of efficiency 50%

For our first family of WEAVER codes we set $k = t$ so that efficiency is 50%. We have $\gcd(k, t) = k = t$ so that $r = q = 1$ and the layout of Figure 2 has only one row of data and one row of parity (see the example below). For this family of codes, we suppress the first component of each index pair and refer to our parity defining sets simply as $\kappa(j) = \kappa(0, j)$ for $0 \le j \le n-1$. We use the following additional notation. Let $\kappa_1(i)$ be an increasing sequence of k integers with initial value 1 and let s be an "offset". We can specify a parity defining set $\kappa(j)$ from $\kappa_1(j)$ and s by the relation

$$\kappa(j) \stackrel{\text{def}}{=} \kappa_1(j) + s = \{i + s \bmod n \ : \ i \in \kappa_1(j)\}.$$

As we will see, simple $\kappa_1(j)$ sets, together with different offsets s provide a convenient way to specify good parity defining sets. As before, if we impose rotational symmetry (see equation (3), then we need only specify $\kappa_1(0)$ and s to completely determine the code. We overload the term "parity defining set" to include a set $\kappa_1(0)$ and an offset s.

For example, with $\kappa_1(0) = \{1, 2, 4\}$ and $s = 2$ (see Table 1, $t = 3$, second entry), the following diagram provides a valid WEAVER code provided $n \ge 7$.

S_0			S_j		S_{n-1}
d_0	d_1	$\cdots$	d_j	$\cdots$	d_{n-1}
$p_{\{3,4,6\}}$	$p_{\{4,5,7\}}$	$\cdots$	$p_{\{j+3,j+4,j+6\}}$	$\cdots$	$p_{\{2,3,5\}}$

Table 1 gives a partial listing of parity defining sets ($\kappa_1(0)$ and offset s) and valid stripe sizes n for fault tolerance $1 \le t \le 10$. (We say a stripe size is "valid" for a given parity defining set if the code on that stripe size has the required fault tolerance.) The entries tagged with an asterisk are discussed in the remarks below.

We make the following remarks concerning Table 1.

- The first entry in the table is a simple RAID1 mirror code, but with a non-standard data layout. This code easily and uniformly provides simple mirroring on any number of devices (at least 2), including an odd number of devices. In addition it provides load-balancing; every device is equally burdened by data and a parity mirror. See Section 5 for further comments on the $t = 2$ entry of the table.
- The first three rows in the table show valid codes for $t \le 3$ with $n \ge 2t$. When $n = 2t$, these codes are in fact MDS. For larger t we could not find constructions that maintained this property. One can measure the "space efficiency penalty" as the difference between the optimal efficiency of an MDS codes on n strips and the actual efficiency; for these WEAVER codes, this is:

$$\frac{n - t}{n} - \frac{1}{2} = \frac{1}{2} - \frac{t}{n}.$$

As can easily be calculated, this ranges from 0.0 to 0.23 for the values in the table, using the smallest valid n for each t. It also increases as n increases for a fixed fault tolerance t (but larger n improves the localization effects as in Section 3.1.2).

- The table provides only a small subset of all the constructions we discovered. For this work, we tested validity for all cases of stripe sizes n, offsets s and parity defining sets $\kappa_1(0) \subseteq [1, w]$ of various ranges. See Section 4 for the methodology we used to test configurations. For $t \le 7$, we covered the ranges $n \le t^2 + 2t$, $0 \le s \le 8$ and $w = 3t$. For $t = 8$, we ran a preliminary filter to find good candidate parity defining sets, then processed the most promising ones in the range $n \le t^2 = 64$, $0 \le s \le 8$ and $w = 2t = 16$. For $t = 9, 10$, we did a preliminary search with $0 \le s \le 8$ and $w = 2t$, with $n \le 4t + 4$ (see the next remark) and

t	$\kappa_1(0)$	**offset** s	**Stripe Size** n
1	$\{1\}^*$	0	2+
2	$\{1,2\}^*$	0	4+
3	$\{1,2,3\}^*$	1	6,8+
	$\{1,2,4\}$	2	7+
4	$\{1,3,5,6\}$	1	10+
	$\{1,2,3,6\}^*$	0,2,3	11+
5	$\{1,3,4,5,7\}$	2	12,15+
	$\{1,5,6,8,9\}$	3	13+
	$\{1,2,3,4,7\}$	1	14+
	$\{1,2,3,6,9\}^*$	2	15+
6	$\{1,5,8,9,10,12\}$	2	17,19,21+
	$\{1,6,8,9,11,12\}$	7	17,20+
	$\{1,2,3,6,9,10\}$	0	18+
	$\{1,2,3,4,6,9\}^*$	5	19+
7	$\{1,4,5,6,7,8,11\}$	4	20,23-24,26,28+
	$\{1,2,4,7,10,12,13\}$	3	20,24+
	$\{1,3,5,6,7,11,12\}$	1	22+
	$\{1,2,3,4,6,9,14\}^*$	6	23+
8	$\{1,2,4,8,10,11,12,13\}$	0	26,28+
	$\{1,2,6,7,8,9,12,14\}$	0	27+
	$\{1,2,3,4,6,7,9,14\}^*$	0	28+
9	$\{1,4,5,6,7,12,13,15,18\}$	2	30,32,34+
	$\{1,4,5,8,11,12,13,14,15\}$	6	31+
	$\{1,2,3,4,6,7,9,14,15\}^*$	5	32+
10	$\{1,2,5,6,7,10,13,15,19,20\}$	3	35,40+
	$\{1,2,4,7,8,9,13,14,17,18\}$	0	37+
	$\{1,2,3,4,6,7,9,14,15,19\}^*$	3	40+

Table 1: Partial listing of parity defining sets for WEAVER(n,t,t) codes. See the remarks for a description of the entries tagged with an asterisk. A stripe size n_0+ means $n \geq n_0$.

then verified the table entries up to $n \leq 64$ for $t = 9$ and $n \leq 56$ for $t = 10$. For $t \leq 3$, it is fairly easy to prove that the constructions work for all n in the described range. For $t \geq 4$, the implication that the codes work for n outside the tested range is not theoretically established; see Section 6 and Theorem 1 in particular. Note that, theoretically, we could have extended the offset range up to $n - w - t$, but that would have made our search spaces even larger, particularly for large n (we did limit it for small n when appropriate).

• The search space for these experiments is actually quite large. For a given t and w, there are $\binom{w-1}{t-1}$ sets $\kappa_1(0)$ and 9 offsets s each to examine. For each such parity defining set and each n, there are up to $\binom{n-1}{t-1}$ individual matrix rank tests to perform (see Section 4). These numbers grow rapidly with higher fault tolerance. For example, the $t = 6$ search completed at least 1.6 trillion matrix rank tests. For $t \geq 6$ it was prohibitive to do this on a standard workstation. Instead, we implemented our search to run on a portion (only 1024 processors) of an IBM Blue Gene/L system. Each processor was given a subcollection of the search space of sets $\kappa_1(0)$ and, for each $\kappa_1(0)$ in its subcollection, ran the tests for every offset s and n in the ranges mentioned above. The $t = 6$ case mentioned above took approximately 12 hours on 1024 processors. The $t = 10$ preliminary search (for $n \leq 44$) took approximately 37.6 days and completed more than 64 trillion matrix rank tests!.

• For fault tolerance $t \geq 4$, there are gaps in the sequence defining the set $\kappa_1(0)$. This is a requirement as the following argument illustrates. Suppose $k = t = 4$ and there are four consecutive integers in $\kappa(0)$, say, $i, i+1, i+2, i+3$. Consider the data element labels $i+1$ and $i + 2$. Both appear together in $\kappa(n - 1) = \kappa(-1)$, $\kappa(0)$ and $\kappa(1)$. But $i + 1$ appears by itself in $\kappa(2)$ and $i + 2$ appears by itself in $\kappa(-2)$. If we loose strips $i + 1$ and $i + 2$ (so data elements d_{i+1} and d_{i+2}), and strips 2 and $(n-2)$ (with $p_{\kappa(2)}$ and $p_{\kappa(-2)}$), then every surviving parity contains either both of the data elements $i + 1$ and $i + 2$ or neither. Consequently, there is no way to distinguish between the two values and this 4 strip failure case cannot be tolerated. It is not clear what additional heuristics (or theorems) define "good" or "bad" sets $\kappa(0)$ (see Section 6).

• We listed only entries that have valid stripe sizes for all $n \geq n_0$ (identified in the table as n_0+), with perhaps

a few isolated valid stripe sizes below n_0. For example, for $t = 5$, the entry $\kappa_1(0) = \{1, 3, 4, 5, 7\}$ with offset $s = 2$ has valid stripe sizes $n = 12$ and $n \geq 15$ but not $n = 13, 14$. We typically observed similar behavior for almost all sets we tested though there were anomalies. The set $\kappa_1(0) = \{1, 3, 6, 10, 15, 21\}$ had invalid stripe sizes for every n divisible by 9 regardless of offset. We do not have a proof that this persists, but we believe that a proof of such negative results would not be difficult.

• The entries marked with $*$ form a single chain of supersets for $\kappa_1(0)$ as t increases. The usefulness of this chain is described in more detail in Section 3.1.4, but briefly it enables changing of fault tolerance on-the-fly with minimal parity recomputation.

2.1.1 The special case of 2 fault tolerance

Consider the $t = 2$ element from Table 1 where $\kappa(0) = \kappa_1(0) = \{1, 2\}$ (and $s = 0$). We will describe this code in somewhat different terms and prove the claimed fault tolerance. With $k = t = 2$, each parity value is the XOR sum of a pair of data values, one from each of the two strips immediately to the right of the parity element (with wrap-around, of course). Alternatively, each data element touches two parity elements and so is paired with two other data elements (the other data element in each of its two parity elements). From this viewpoint, a given data element D is paired with its west W and east E neighbor elements (graphically):

$$W \leftarrow D \rightarrow E$$

The parity value computed from $W \oplus D$ is stored in the strip to the left of W; the parity value computed from $D \oplus E$ is stored on the strip to the left of D, namely, the same strip as W.

We now give a proof that this code has the required 2 fault tolerance, provided $n \geq 4$. It is clear that this is a necessary condition (if $n \leq 3$ and two strips are lost, there is at most only one strip remaining and that is clearly insufficient). It is also clear that we only need to recover lost data values, as parity values can be recomputed from all the data.

There are three cases to consider.

Case 1: Suppose one strip is lost (say, strip S_j). We observe that $\kappa(j-1) = \{j, j+1\}$ so that d_j can be recovered by reading d_{j+1} (from the strip S_{j+1} to the right of strip S_j), reading $p_{\kappa(j-1)}$ (from the strip S_{j-1} to the left of S_j), and using the formula:

$$d_j = d_{j+1} \oplus p_{\kappa(j-1)}. \tag{5}$$

Alternatively, we can recover d_j from d_{j-1} and $p_{\kappa(j-2)}$ since $\kappa(j-2) = \{j-1, j\}$.

Case 2: If two adjacent strips are lost (say, S_j and S_{j+1}), we read d_{j-1} and $p_{\kappa(j-1)}$ (in one operation from strip S_{j-1}) and $p_{\kappa(j-2)}$ from S_{j-2}. Then, recursively,

$$\begin{aligned} d_j &= p_{\kappa(j-2)} \oplus d_{j-1} \\ d_{j+1} &= p_{\kappa(j-1)} \oplus d_j. \end{aligned}$$

Note that even though we needed to read three elements (two parity and one data), we only needed to access two devices (for disks, this is a single IO seek per device) because of the vertical arrangement of data and parity.

Case 3: If two non-adjacent strips are lost, then we reconstruct as two independent cases of a single strip loss (using (5)), because the data can always be reconstructed from its left neighboring parity and right neighboring data (neither of which is on a lost strip). For $n \geq 4$, most of the dual failure cases are of this type.

In this proof, we saw two examples of the "localization" property of WEAVER codes (see Section 3.1.2). For all cases, only a few devices in the stripe need to be accessed for reconstruction; this number is independent of the stripe size n. In addition, in Case 3 the reconstruction problem split into two smaller, easier and independent reconstruction problems.

We also saw how the vertical layout reduces device access costs, by combining some multi-element reads into a single device access. For disk arrays, this implies fewer disk seeks and better performance.

2.2 Other constructions with one data row

The constructions of the previous section are "ad hoc"; that is, (with the exception of the case $t = 2, 3$) they were found by computer search and not by a parametric formulation of the parity defining sets $\kappa(0)$. In this section, we give a different and formulaic construction.

We start by making the assumption that k divides t. Then $\gcd(k, t) = k$ so we can take $r = 1$ (one data row) and $q = t/k$ (q parity rows). We use the term consecutive-i to mean consecutive numbers with difference i. (For example, the consecutive-2 numbers starting at 1 are $1, 3, 5, \ldots$.) We say a set of parity elements are consecutive-i if they are in the same row and their corresponding strip numbers are consecutive-i, modulo n.

The constructions are described as follows. In parity row 0, data element d_j touches k consecutive-1 parity elements to the left, ending at some left-offset s from d_j (with wrap-around). In each parity row i, $1 \leq i \leq q-1$, data element d_j touches the set of k consecutive-$(i+1)$ parity elements ending one strip to the left of the first parity element touched in the previous row (again, with wrap-around). A data element touches exactly k parity elements in each row so that each parity is composed of k data elements (that is, parity in-degree is k).

Figure 3 shows a graph representation for the special case when $k = 3$, $t = 9$ and so $r = 1$ and $q = 3$. This graph is that subgraph of the general graph in Figure 1

corresponding to the out-edges for the one data element d_j (and by rotational symmetry, implies the subgraph for all other data elements).

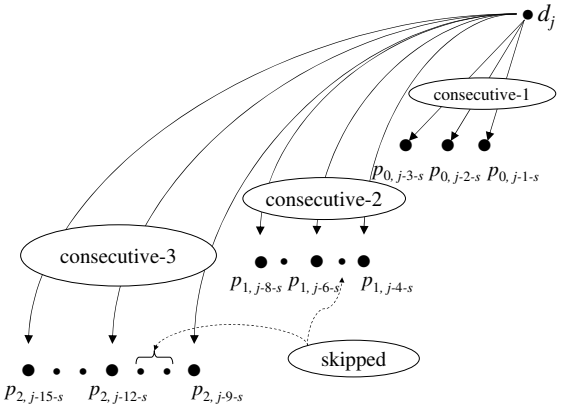

Figure 3: The subgraph of the data/parity relations graph for $k = 3$, $t = 9$ and offset s. The relative position suggests approximate placement in the rows of the data/parity layout of Figure 2. Small dots represent parity elements in the gaps that not touched by d_j.

Put in mathematical terms, the set $\tau(j) = \tau(0, j)$ of (ordered pairs of) indices of parity elements touched by $d_j = d_{0,j}$ is given by the formula

$$\tau(j) = \bigcup_{i=0}^{q-1} \{(i,\ \langle j - \sigma(i,k,u) - s\rangle_n)\ :\ 1 \le u \le k\}\,. \tag{6}$$

where, as shorthand, $\langle x\rangle_n = x \bmod n$ and

$$\sigma(i,k,u) = (k-1)i(i+1)/2 + u(i+1).$$

From the parity viewpoint, the equivalent formulation is

$$\kappa(i,j) = \{\langle j + \sigma(i,k,u) + s\rangle_n\ :\ 1 \le u \le k\}\,. \tag{7}$$

In these expressions, the term $u(i+1)$ for $1 \le u \le k$ provides the k consecutive-$(i+1)$ parity elements. The term s provides the initial offset. The term $(k-1)i(i+1)/2 + s$ provides for the starting point relative to d_j.

Table 2 provides a list of some examples found by testing all $s \le 8$ and all $n \le 64$ for $t \le 10$ and $n \le 48$ for $t = 12$. We also give the efficiency, Eff $= 1/(q+1)$ by (4); in the previous section, all codes had efficiency 50%. Notice the examples with fault tolerance as high as $t = 12$. We conjecture that this construction (with a suitable offset s and sufficiently large n) should work for arbitrarily large t, though not necessarily all k.

The examples with $t = 9, k = 3$ and $t = 12, k = 4$ are interesting because the fault tolerances are so high, but the efficiency is 250% higher and 325% higher than

t	k	q	s	**Stripe Size** n	**Efficiency**
2	2	1	0,1	4+	50%
4	2	2	0	6+	33%
6	2	3	0,1,2	9+, excl. 9+s	25%
8	2	4	0,1,2,3	15+, excl. 14+s	20%
10	2	5	0,3,4	16,20+, excl. 20+s	17%
12	2	6	0,1	23,27+, excl. 27+s	14%
3	3	1	1	6,8+	50%
6	3	2	2,4 3	11,13,15+ 12,14+	33%
9	3	3	1 3	15,17+ 16-17,19+	25%
12	3	4	1,3	24+, excl. 24+s	20%
12	4	3	2	21,25+	25%

Table 2: Partial listing of WEAVER(n,k,t) codes where k divides t and parity defining sets given by (6) or (7).

corresponding 10-way or 13-way mirrors. The two examples in Table 2 with $t = k = 2$ and $t = k = 3$ are identical to the WEAVER(n,t,t) codes given in Table 1 with $\kappa_1(0) = \{1, 2\}$ and $\kappa_1(0) = \{1, 2, 3\}$, respectively.

There are two codes in Table 2 with the same fault tolerance $t = 6$; they have different parity in-degree k and so different efficiency. The code with $k = 2$ has very simple parity computations but efficiency only 25%. The code with $k = 3$ has somewhat more complex parity computations but better efficiency 33%. This exemplifies one of the trade-offs of erasure codes (performance vs. efficiency) and the fact that the rich set of WEAVER codes provide a means to balance these trade-offs for a given system's constraints. (Similar remarks apply to the three codes with $t = 12$.)

The remark made in the comments of Section 2.1 about consecutive parity elements when $k = t$ precludes this construction from working for $k = t \ge 4$. For $k = 4$ and $t > 4$ the situation is undecided. Preliminary experiments suggest that $t = 8$ suffers from a similar obstacle; surprisingly $t = 12$ does have some valid configurations. Clearly, alternatives to consecutive-1 may be required if $k \ge 4$ (though we have not tested any of these cases).

There is considerably more structure to the patterns of valid offsets and stripe sizes for this construction compared to those of the previous section. It is likely that this construction can be analyzed in most cases theoretically. We conjecture that other parity defining sets will also provide valid WEAVER codes of similar efficiency and design, perhaps on even smaller stripe sizes or will fill in the gaps in the stripe sizes in Table 2. We leave these two issues to future work.

2.3 Parity in-degree 2

The constructions we have presented so far have one data row ($r = 1$, because k divides t in all cases). In the next

two sections, we give two ad-hoc constructions (with 3 and 4 fault tolerance, respectively) that have $r = 2$. The key feature of these codes is that we restrict $k = 2$, so that the parity computations are quite simple.

These codes generalize the 2 fault tolerant code discussed in Section 2.1.1. They each contain two copies of this 2 fault tolerant code, plus some additional cross relations between them (hence, again the WEAVER name). All three codes use "data neighbor" relations to determine the parity defining sets. Adding a second row allows for relations in different directions than just east/west; this in turn enables more parity defining sets containing a given data element; and this increases the (potential) fault tolerance.

2.3.1 Three fault tolerance

The WEAVER(n,2,3) code presented in this section has a data/parity layout given by

S_0		S_j		S_{n-1}
$d_{0,0}$	$\cdots$	$d_{0,j}$	$\cdots$	$\cdots$
$d_{1,0}$	$\cdots$	$d_{1,j}$	$\cdots$	$\cdots$
$p_{\{(0,1),(0,2)\}}$	$\cdots$	$p_{\{(0,j+1),(0,j+2)\}}$	$\cdots$	$\cdots$
$p_{\{(1,1),(1,2)\}}$	$\cdots$	$p_{\{(1,j+1),(1,j+2)\}}$	$\cdots$	$\cdots$
$p_{\{(1,n-2),(0,n-1)\}}$	$\cdots$	$p_{\{(1,j-2),(0,j-1)\}}$	$\cdots$	$\cdots$

Each parity element is labeled by its parity defining set. The first parity row is the WEAVER(n,2,2) code built from the first data row. The second parity row is again the WEAVER(n,2,2) code built from the second data row. The third row weaves between these two along nearest neighbor up-diagonals, placing the parity to the right of the up-neighbor (other placements are possible).

Visually, data elements D_0 from the first row and D_1 from the second row are paired with neighbors as in the following diagram (each pair computes a different parity value):

$$\begin{array}{ccc} W_0 \leftarrow D_0 \rightarrow E_0 & & \\ \swarrow & & \\ SW & & \end{array} \qquad \begin{array}{ccc} & & NE \\ & \nearrow & \\ W_1 \leftarrow D_1 \rightarrow & E_1 & \end{array}$$

The $D_1 \rightarrow NE$ relation is just a reverse perspective on the $D_0 \rightarrow SW$ relation above ($SW = D_1$ and $NE = D_0$). Each data element touches three different parity elements on three different strips, twice in one parity row, and once in the last row parity row. This provides the necessary condition for 3 fault tolerance.

It can be proven that this construction is also sufficient for 3 fault tolerance provided $n \geq 6$ but we leave out the proof as it is similar to the one we gave for WEAVER(n,2,2).

This construction provides another example of a 3 fault tolerant code on as few as 6 devices. Compare this with the first $t = 3$ entry in Table 1 where $k = 3$; the difference is again a performance/efficiency tradeoff. See Section 5 for more comments.

2.3.2 Four fault tolerance

In this section we provide another WEAVER(n,2,4) code with two data rows. Contrast this construction with the code of Section 2.2, which also has $k = 2$ and $t = 4$ but has only one data row (see Table 2). This is the only code we present that drops the "minimalist" condition $r = k/\gcd(k, t)$.

The layout and parity defining sets can be seen in the following chart:

S_0		S_j		S_{n-1}
$d_{0,0}$	$\cdots$	$d_{0,j}$	$\cdots$	$\cdots$
$d_{1,0}$	$\cdots$	$d_{1,j}$	$\cdots$	$\cdots$
$p_{\{(0,1),(0,2)\}}$	$\cdots$	$p_{\{(0,j+1),(0,j+2)\}}$	$\cdots$	$\cdots$
$p_{\{(1,1),(1,2)\}}$	$\cdots$	$p_{\{(1,j+1),(1,j+2)\}}$	$\cdots$	$\cdots$
$p_{\{(1,n-3),(0,n-2)\}}$	$\cdots$	$p_{\{(1,j-3),(0,j-2)\}}$	$\cdots$	$\cdots$
$p_{\{(0,n-3),(1,n-2)\}}$	$\cdots$	$p_{\{(0,j-3),(1,j-2)\}}$	$\cdots$	$\cdots$

The first five rows are essentially identical to the WEAVER(n,2,3) code of the previous section, with the exception that the fifth row is cyclically shifted right by one. The last row of parity elements is computed by weaving the down-diagonals and placing the parity value in the parity element two strips to the right.

This time, each data element is paired with four other data elements: west, east, and two diagonal neighbors (southwest, southeast for an element D_0 in row 0 and northeast, northwest for a an element D_1 in row 1). Graphically, this looks like:

$$\begin{array}{ccccc} W_0 \leftarrow D_0 \rightarrow E_0 & & & & \\ \swarrow \quad \searrow & & & & \\ SW \qquad SE & & & & \end{array} \qquad \begin{array}{ccccc} & NW & & NE & \\ & \nwarrow & & \nearrow & \\ & & W_1 \leftarrow D_1 \rightarrow & & E_1 \end{array}$$

Each data element is now stored in four parity elements: twice in one of the first two parity rows (and not in the other), and once in each of the last two parity rows.

Again, we leave out the proof that this is 4 fault tolerant provided $n \geq 8$. This code has the same fault tolerance, efficiency and parity in-degree as the WEAVER(n,2,4) code in Table 2, but requires more strips for the minimal configuration (and has six rows in total versus three for the previous code). However, it has better localization properties. For example, for this construction the write lock zone comprises a total of eight data and parity elements on six neighboring strips (centered on the target strip) versus the same number of data and parity elements on seven strips (five neighboring strips similarly centered plus two additional strips) for the $k = 2, t = 4$ code of Section 2.2.

Note that by dropping the first two rows of parity, we get yet another WEAVER(n,2,2) code with two data

rows, two parity rows and efficiency 50%. It can be made equivalent to the code described in Section 2.1.1 and so is not very interesting by itself.

3 Features

Now that we have defined the WEAVER codes and given many examples, we next discuss the key features of these codes, the advantages in the next subsection and a brief discussion of the primary disadvantage following.

3.1 Advantages

3.1.1 High Fault Tolerance

The WEAVER codes have instances of exceptionally high fault tolerance (we gave constructions with fault tolerance 12 and conjecture that other constructions should be possible). There are few codes in the storage system literature that meet these fault tolerance levels. The only viable options for very high fault tolerance to date seem to be Reed-Solomon codes, with their high computational costs, or N-way mirroring with their very low efficiency. See also the remarks in Section 5 on the LDPC and Wiencko codes.

3.1.2 Localization Effects

The design characteristics of constant parity in-degree and rotational symmetry are key features of the WEAVER codes. They enable the "localization" of many operations on the stripe. We have seen two examples of this: (a) in Section 2.1.1 we saw reconstruction requiring access to a small bounded (independent of stripe size) subset of the stripe; and (b) in Section 2.3.2 we saw write lock zones for two codes that are also small, bounded subsets of the stripe.

These two examples are typical of any WEAVER code (in fact, any code with parity in-degree bounded independent of the stripe size). The write lock zone (see Section 1.1) can be determined by examining the 2-neighborhood of the target element in the data/parity graph (see Figure 1 – the 2-neighborhood is the set of nodes in the graph within a distance two of the target element's node). With t parities each having k out edges (one of which is the target element), this bounds the write lock zone to at most $t(k-1)+t = tk$ data and parity elements (so at most tk devices as well – for some WEAVER codes, the actual number of devices is smaller). This is independent of the stripe size n, providing a proof of the localized write lock zone.

In contrast, even RAID5 has a write lock zone that is effectively the entire stripe, since the 2-neighborhood of an element is the entire remainder of the stripe. This is a consequence of the parity in-degree determined as a function of the stripe size.

Similar localization effects occur during rebuild. A rebuild of one or more lost strips in a WEAVER code only requires access to a fixed and bounded set of strips. This set is at most the union of the 2-neighborhoods for all the data elements on all the lost strips (and is independent of n as well). Also, as we saw in Section 2.1.1, certain multi-strip failures may partition themselves into independent smaller failure scenarios. Reconstruction algorithms generally get more complicated with the number of inter-related failures so partitioning a multiple failure case into two or more independent failure cases can have a significant performance advantage and enable parallelism. For example, consider the recovery costs of EVENODD(p,n) [2] and WEAVER(n,2,2) code when two strips fail. For $n \geq 6$, the EVENODD always requires accessing $n-2$ devices, whereas most cases of WEAVER recovery involve only 4 devices, and the other cases only require 2 devices to be accessed!

3.1.3 Symmetry

The vertical layout of data and parity together with the symmetry properties (balanced parity in-degree and data out-degree and the rotational pattern) provide natural load balancing across all the devices. Multiple instances of the WEAVER codes can be stacked on the same set of devices with a simple host-to-strip addressing and also a simple logical strip-to-physical-device labeling. In contrast, the parity rotation of RAID5 vs RAID4 requires more complicated logical/physical addressing models.

Furthermore, multiple WEAVER code instances with different fault tolerances can be easily stacked on the same collection of devices (provided the number of devices is sufficiently large). This enables different reliability classes of logical volumes on the same set of devices. This is possible with other codes, but generally requires more complex logical/physical addressing.

3.1.4 Variability of Stripe Size and Fault Tolerance

The localization property mentioned above enables the WEAVER stripes to be expanded or shrunk with only local effects to the stripe; that is, not all devices need to be accessed and data or parity moved around. For example, in the WEAVER(n,2,t) codes, new devices/strips can be inserted into the stripe, and only the nearby devices need to have parity recomputed. See the additional comments in Section 5.

With the WEAVER(n,t,t) codes, it is further possible to change, *on-the-fly*, the fault tolerance of a single stripe in the system (either up or down) by simply recomputing the parity values. No remapping of either the host addressing or the strip labeling is needed. The only requirement is that the stripe size is supported for the higher fault tolerance. This enables more autonomic adaptability and is not possible with (almost) any other

code. In addition, by using a chain of $\kappa_1(0)$ subsets (e.g., those marked by a asterisk in Table 1), the recomputation step involves only adding a single new data value into each parity value and then storing the new parity values in the appropriate strips (which changes only if the offset changes). This is significantly more efficient than recomputing all the parity from scratch. Note that "adding a data value" can be used to either lower or raise the fault tolerance.

We believe that these features in particular make the WEAVER codes best suited for dRAID (distributed Redundant Arrangement of Independent Devices) systems involving network-connected storage nodes. Such systems will likely have data sets with varying reliability and performance requirements. Such sets may be distributed across different but intersecting sets of nodes. The WEAVER variability of stripe size and fault tolerance enable a dRAID data distribution algorithm to focus on user data layout (e.g., for load-balancing) and to achieve a balanced parity distribution as a natural consequence of the code itself. In addition, the chain of design sets for WEAVER(n,t,t) codes allows the system to change fault tolerance with minimal network bandwidth utilization. Each node reads both its data and parity values, and sends only a single data value over the network, performs a single XOR operation, sends the recomputed parity value over the network and then performs a single disk write operation of the new parity). The parity "send" step is only required if the offset changes; more interestingly, the data "send" step may be skipped for some parity defining set subset chains. This operation is then both load-balanced and disk and network efficient.

3.1.5 Short Write IOs

For most of the WEAVER(n,k,t) codes, the short write IO cost in device accesses (e.g., disk seeks) is equal to $2(t+1)$. For the parity update algorithm that uses the parity delta, this seek cost is optimal for any t fault tolerant code. Many codes have even higher short write IO costs, when a given data element touches more than t parity elements (and strip sizes are large – see Section 3.1.7). For example, the EVENODD codes [2, 3] have this property for some elements.

Furthermore, only codes of efficiency (approximately) 50% can achieve better short write IO seek costs than the typical $2(t+1)$. For example, a t-fault tolerant Reed-Solomon code can perform a short write in $2t$ seeks but only if $n = 2t$ (so efficiency 50%) or in $(2t+1)$ seeks only if $n = 2t+1$ (so efficiency close to 50%). In these cases, the stripe size is fixed as a function of the fault tolerance t. Mirroring achieves the best short write IO seek costs $(t+1)$ but also has the lowest efficiency.

In contrast, some WEAVER codes achieve better short write IO seek costs for a given fault tolerance and for *any* valid stripe size. For example, the WEAVER(n,2,2) code (see Section 2.1.1) enables an implementation of a short write with 5 IOs (one less than is typical). This is achieved by reading the west and east neighbors of the target data element, computing the two new parities (from the parity equations) and writing the two new parities and one new data.

Similarly, by amortizing two neighbor data element reads into one longer read, the other two ad hoc WEAVER codes with parity in-degree equal to 2 can achieve a short write IO seek cost of 6 (for fault tolerance 3, Section 2.3.1) and 7 (for fault tolerance 4, Section 2.3.2). We emphasize that these IOs are not all of the same length, and a fairer comparison should take this into account (we do not do that here since seeks dominate device access costs; but see [9] for a more thorough analysis of these issues).

3.1.6 Multiple Strip Writes

Many of the WEAVER codes can amortize a significant number of device accesses required for consecutive multi-strip host writes. We explain this in detail for the WEAVER(n,2,2) code (Section 2.1.1) and leave it to the reader to see how this principle can be applied in other WEAVER constructions. We will contrast the WEAVER behavior with EVENODD [2] which is also 2 fault tolerant, though this analysis applies to many other 2-fault tolerant codes, including Reed-Solomon [14].

Suppose the system gets a host write for m consecutive strips in a stripe of size n. For the EVENODD code, there are two efficient implementations. The first implementation reads all the m strips and the 2 parity strips, computes the new parity strips and writes the m data strips and 2 parity strips for a total of $2m+4$ device accesses. The second implementation reads the $n-2-m$ other (2-neighborhood) data strips, computes the parity and writes $m+2$ data and parity strips for a total of n device accesses. The optimum number of device accesses is then $\min(2m+4, n)$.

In contrast, the following diagram shows how this could be implemented for a WEAVER(n,2,2) code. In the diagram, an R indicates a data element that we read (only), a W indicates a data element that we write (this is a target of the host IO), P indicates a parity element that we write (we do not read any parity or old data).

	R	W	W	$\cdots$	W	R
P	P	P	$\cdots$	P		

For the same m consecutive multi-strip write, we read only two data elements (the two indicated by R on the west and east ends of the top row), and write $m-1$ strips with *both* data and parity (as a single seek), one data element W (on the right side of the top row) and the two parity elements labeled P on the left, bottom row. This

totals $2 + m - 1 + 1 + 2 = m + 4$ device accesses and is better than $\min(2m + 4, n)$ for EVENODD provided $m \leq n - 4$. (For $m = n - 3, n - 2, n - 1, n$ the device accesses counts are $n + 1, n + 2, n + 1, n$, respectively, indicating a small disadvantage in these cases.)

3.1.7 Host IO Size Uniformity

XOR-based erasure codes have typically two alternative implementations with respect to size (number of bytes) that are mapped to each strip. One choice is to map a strip to a small unit of the device, e.g., 512B or perhaps 4KB. In this case, a host short write maps to a full strip write and the parity computations involve all (or most) of the parity elements. Generally, Reed-Solomon codes are implemented as extreme examples of this where a strip is a byte. XOR-based codes may be implemented in this way as well, but multiple rows impose fragmentation of the parity computations.

The alternative is to map a strip to a unit comparable to a moderate size host IO (say, 256KB as is typically done in RAID5). In these cases, elements are much larger units and a host short write affects only a subportion of an element. With this implementation, the host short write costs can scale to larger host IO lengths, up to the size of the element, incurring additional costs that are only linear functions of the length. There are no additional seeks or other computational costs such as additional XOR formulas.

For a fixed strip size, more data rows in the code imply smaller element size (for a fixed strip size), and hence limitations on the advantages of this uniform and linear scaling. In this regard, (most of) the WEAVER codes are optimal because they have only one data row (the ad hoc constructions have two data rows, so are near optimal). They also do not suffer from excessive XOR fragmentation for the same reason. (Clearly, RAID5 and N-way mirroring have these properties as well but they are at opposite ends of the fault-tolerance/efficiency spectrum, with WEAVER codes occupying the middle ground. See Section 5 for additional comments.)

3.2 Disadvantages – Efficiency

The primary disadvantage of the WEAVER codes is their limited efficiency (at most 50%). On the other hand, WEAVER codes are optimally efficient among all possible codes of fault tolerance t and parity in-degree $k \leq t$ as the following argument shows. Suppose an erasure code with fault tolerance t has N data elements and Q parity elements and that each parity element has parity in-degree bounded above by k. Each data element must touch at least t parity elements. Counting edges in the parity graph (as in the WEAVER example in Figure 1), we see that the number of edges E is at least Nt (counting the edges coming out of the data nodes) and at most Qk (counting the edges coming into the parity nodes); that is, $Nt \leq E \leq Qk$. The efficiency is

$$\frac{N}{N+Q} = \frac{Nt}{Nt+Qt} \leq \frac{Qk}{Qk+Qt} = \frac{k}{k+t},$$

with equality in the case of the WEAVER codes. In addition, we clearly see the trade-off of efficiency for simplicity, fault tolerance and the other positive features of these codes. (Other codes, including RAID5 have performance/efficiency trade-offs, but those trade-offs are as functions of the stripe size with constant fault tolerance – for WEAVER codes, it is a function of the fault tolerance regardless of the stripe size.)

4 Testing methodology

We have mentioned above that we search parameter spaces for valid configurations, that is, configurations that provide the requisite fault tolerance. Our methodology is the following (see also [10]). For each fault tolerance t and each choice of parity defining set (including offset), and for each stripe size n, we construct the binary generator matrix for the WEAVER code of those parameters. This matrix has rows indexed by the data elements and columns indexed by the data and parity elements. A column with a single one indicates a data element; a column with at least 2 ones indicates a parity element (and the formula used to compute it – that is, the data elements touching it). We view the generator matrix in column block form where each block corresponds to a strip; the columns in a block correspond to data and parity elements on the same strip. The blocks are indexed by the numbers $\{0, 1, \ldots, n-1\}$ since there are n strips. The generator matrix then maps the input user data into data and parity elements, and the block structure shows how they are organized on strips.

We can simulate strip loss by removing (or zeroing) the columns of the corresponding block. Consequently, to test this code for the required fault tolerance t, we execute the following pseudo-code:

1. For each t-sized subset $T \subseteq \{0, 1, \ldots, n-1\}$:
   (a) Remove the column blocks from the generator matrix indexed by the elements of T (simulate strip loss).
   (b) If the binary row rank of the reduced matrix equals the number of rows, continue; else return "Invalid".
2. Return "Valid"

The algorithm returns "Valid" if and only if the given generator matrix defines a t fault tolerant code. Each reduced matrix represents a set of equations that maps the "unknown" data values into the "known" data and parity values (those that are not lost). The row rank of the reduced matrix equals the number of rows if and only if

this system of equations is solvable; that is, if and only if all the "unknown" data values can be reconstructed from the "known" data and parity values. This test succeeds for all specific failure cases T if and only if the code is t fault tolerant.

For rotationally symmetric codes like the WEAVER codes, one can restrict the search space a bit by requiring that $0 \in T$. Other such restrictions may be used to reduce the search space further (e.g., see [6]).

As n and t grow, this search space grows rather rapidly, as there are $\binom{n-1}{t-1}$ such cases to consider (and the matrices get larger as $O(n^2)$). Other optimizations are possible that reduce these costs.

5 Related Work – Other codes

The WEAVER codes can be compared to any other erasure code suitable for storage systems (we have mentioned some already such as Reed-Solomon and EVENODD). For ease of comparison, we divide the set of known erasure codes into different categories and give (non-exhaustive) examples in each category. In a category by themselves are the Reed-Solomon codes [14], which are MDS but require complex finite field arithmetic. Second are XOR-based codes that are MDS. These come in two types: vertical codes such as the X-code [17], BCP [1] or ZZS codes [18] and horizontal codes such as EVENODD [2, 3], Blaum-Roth [4], or Row-Diagonal Parity codes [5]. Finally, there are non-MDS codes that are XOR-based. These subdivide into three categories based on efficiency: N-way mirroring (trivially XOR-based) with efficiency less than 50%, the Gibson, *et al*, codes [8] with efficiency larger than 50%, and two codes with efficiency 50% exactly. In the last category, we have the LSI code [16] (in fact a subcode of one of the Gibson *et al* codes) and one 3 fault tolerant Blaum-Roth binary code [4]. In the second category, we also have the LDPC codes (see [7, 11, 13, 12]) and the Wiencko codes [15, 6] which we address separately.

LDPC codes [7, 11] where originally designed for communication channels but have recently been studied in the context of storage applications over wide-area networks [13, 12]. In these applications, random packet loss (or delay) is the dominant erasure model (not total device failure), so a typical "read" *a priori* assumes random erasures and hence is always in reconstruction mode. Because of this, good LDPC codes have highly irregular graph structures but can have high fault tolerance and near optimal efficiency (both in the sense of expected value, however). In contrast, the WEAVER codes are designed for a traditional storage model where reads involve reconstruction algorithms *only if* the direct read the user data fails. In addition, WEAVER codes have very regular graphs, and relatively high fault tolerance over a wide range of stripe sizes.

The Wiencko codes are presented in the patent [15] and patent application [6] as methodologies for formulating and testing a specific code instance code via design patterns. This is similar to the BCP [1] patent. The Wiencko codes are vertical codes (with layout as in Figure 2) and can utilize rotational symmetry. Valid constructions meet certain parameter constraints, and include MDS design possibilities. Few examples are given in these references, however. The construction methodology differs from WEAVER codes in that no *a priori* restrictions are placed on the parity in-degree. In fact, some examples have differing degree for different parity values; so are less regular and uniform than required in WEAVER codes. Essentially any vertical code such as the X-code [17] and BCP [1] can also be (re)constructed by a Wiencko formulation.

With the exception of the Reed-Solomon codes, N-way mirroring, LDPC and Wiencko codes, none of the codes have exceptionally high fault tolerance. There are variations of EVENODD [3] that are ≥ 3 fault tolerant; there is one instance of a BCP code of three fault tolerance on 12 strips; the Blaum-Roth [4] binary code of efficiency 50% and two codes in [8] are 3 fault tolerant. As far as we know, none of the other codes have variants that can tolerate more than 2 failures. The WEAVER codes can have very high fault tolerance (up to 12 and perhaps beyond). Compared to Reed-Solomon codes, they are significantly simpler but less efficient. Compared to N-way mirroring, they are more complex but more efficient. The WEAVER codes provide alternative interpolating design points between Reed-Solomon and N-way mirroring over a long range of fault tolerances.

As we mentioned in Section 3.1.5, only codes with efficiency approximately 50% can implement a host short write with fewer IO seeks than $2(t+1)$; the implementation in fact must compute parity from new data and the 2-neighborhood dependent data. To achieve IO seeks costs less than $2(t+1)$, this 2-neighborhood must be small. Special subcodes of the horizontal codes (both MDS and non-MDS) can achieve this but only if the stripe size is bounded as a function the fault tolerance: $n = 2t$ or $n = 2t+1$. The Blaum-Roth [4] three fault tolerant (binary) code is equivalent to a very special case of Reed-Solomon with $t = 3$ and so can be implemented with a 6 IO seeks (at efficiency 50%). In these implementations, the strip size must be comparable to the short write IO size (see Section 3.1.7) so that a short write contains a strip. Only the LSI code [16] and the WEAVER(n,2,t) codes support variable stripe sizes of fixed fault tolerance and improved short write IO seek costs. These can both be implemented with large single element strips gaining the advantages of host IO size uniformity over a longer range of sizes (see Section 3.1.7).

All the MDS codes have the property that parity in-

degree increases with stripe size, for constant t. Consequently, the advantages of WEAVER codes that result from bounded parity in-degree (see Section 3.1.2) can not be achieved with MDS codes of similar fault tolerance. Here again is a performance/efficiency trade-off.

All the 2 fault tolerant XOR-based MDS codes (and some 3 fault tolerant codes as well) share the property that the number of elements per row (row count) increases with increasing stripe size. For example, for EVENODD, the row count is $p-1$ were $p \geq n+2$ and p is prime; for the X-code, the row count equals the stripe size (and must be a prime as well). This has consequences for stripe size flexibility. For horizontal codes such as EVENODD, Row-Diagonal Parity, or Blaum-Roth, flexibility in stripe size can be attained either by selecting a large row count to start with, or by changing the row count with strip size. The latter is prohibitively complex in practice. The former, initial large row count, increases the fragmentation and XOR-complexity of parity computations. For example, the scalability of host IO size (see Section 3.1.7) rapidly degrades with increasing row count. For the vertical codes, changing stripe sizes implies changing row counts and that is prohibitive for on-the-fly changes (this may not be true for certain Wiencko codes, though this has not been established). In contrast, the WEAVER codes maintain constant XOR complexity with changes in stripe size (XOR complexity only increases as fault tolerance increases, which is a necessary effect).

The Gibson *et al* codes (of efficiency greater than 50%) share a number of the good qualities of the WEAVER codes, including the host IO size uniformity (because they have only one row). They are, however, horizontal codes and so require parity rotation for load balancing, only tolerate at most 3 failures and have large minimum stripe sizes. Furthermore, to maintain balance and symmetry, they must restrict stripe sizes to specific values. We believe these codes are reasonable choices for performance/efficiency trade-offs for 2 or 3 fault tolerant codes if efficiency beyond 50% is required. As we have seen, though, the WEAVER codes have a number of additional advantages, a greater range of fault tolerance and better natural balance and symmetry.

The LSI code is very similar to the special case WEAVER(n,2,2) code detailed in Section 2.1.1, and as mentioned is a subcode of a Gibson *et al* code. Each parity value is computed from two data elements, but instead of being placed below in a new row (and new strip), each parity value is placed on a new device separate from any data in the stripe (so it is a horizontal code). Besides being limited to tolerating only 2 failures, the specific layout of the LSI code implies two restrictions on stripe size: $n \geq 6$ and n must be even. The WEAVER(n,2,2) code requires only $n \geq 4$ and has no such even/odd restriction. In addition, the WEAVER vertical code layout again provides natural load balancing under mixed read/write host IOs without any special parity rotation as would be required for the LSI code.

6 Open Problems

There are still a number of open questions and missing constructions. We list a few here:

- Find constructions of WEAVER codes where k divides t, $4 \leq k < t \leq 8$ (see Section 2.2).
- For $k = t$ (Section 2.1), determine the minimum valid stripe size and the parity defining sets that achieve this minimum. More generally, resolve the same issue for any t and $k \leq t$.
- For a given parity defining set, determine the stripe size n_0 so that all stripe sizes $n \geq n_0$ are valid (see the next theorem); more generally determine the complete set of valid stripe sizes.

Theorem 1. *Let a WEAVER(n,k,t) be constructed from a collection of parity defining sets $\{\kappa(j) : 0 \leq j \leq q-1\}$ (one for each parity row). Let w be the largest element in these sets. Then the fault tolerance of the code is constant for all $n > tw$.*

Proof. (Idea) The fault tolerance should not change once n is larger than the largest window of devices that are (a) touched by some data element and its 2-neighborhood and (b) can be affected by t failures. This window defines the localization of the code (write lock zone and rebuild zones) and is dependent only on w, k, t. For a given data element, the parity it touches are within a neighborhood of at most w strips. There are at most t such neighborhoods that can be affected. Consequently, once $n > tw$, the failures are localized to a zone independent of n and the result follows. Note, we claim only constant, but not necessarily t, fault tolerance. □

7 Summary

In this paper, we introduced the WEAVER codes, families of XOR-based erasure codes suitable for storage systems (either RAID arrays or dRAID node-based systems). These codes have a number of significant features: (a) they are designed with simplicity and symmetry for easy implementation; (b) they have constrained parity in-degree for improved computational performance; (c) they are vertical codes for inherent load-balance; (d) they have constructions with very high fault tolerance; (e) they support all stripe sizes above some minimum (determined as a function of each specific construction, but generally dependent on the fault tolerance); (f) there are families with efficiency equal to 50% as well as families of lower efficiency (independent of fault tolerance and stripe size). The WEAVER codes

provide system designers with great flexibility for fault tolerance and performance trade-offs versus previously published codes. They provide a middle ground between the performance advantages but low efficiency of N-way mirroring and the lower performance but higher efficiency of codes such as Reed-Solomon. All these features make the WEAVER codes suitable for any storage system with high fault tolerance and performance requirements; they are perhaps best suited to dRAID systems where flexibility in stripe sizes, fault tolerance and autonomic considerations drive design choices.

8 Acknowledgements

The author extends his thanks and appreciation to Jeff Hartline, Tapas Kanungo and KK Rao. Jeff, in particular, showed the author the relationship between parity in-degree and efficiency, thereby indirectly offering the challenge to construct optimal codes under these constraints. We also want to thank the Blue Gene/L support team at IBM's Almaden Research Center for the opportunity to run many of the larger experiments on their system (and their assistance). Testing stripe sizes in 40-60 ranges, with fault tolerance 10 and tens of thousands of parity defining sets (as we did for the results in Table 1) required considerable computing power. Finally, we thank the reviewers for reminding us about LDPC codes and for pointing us to the Wiencko codes.

References

[1] S. Baylor, P. Corbett, and C. Park. Efficient method for providing fault tolerance against double device failures in multiple device systems, January 1999. U. S. Patent 5,862,158.

[2] 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*, 44:192–202, 1995.

[3] M. Blaum, J. Brady, J. Bruck, J. Menon, and A. Vardy. The EVENODD code and its generalization. In J. Jin, T. Cortest, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 14, pages 187–208. IEEE and Wiley Press, New York, 2001.

[4] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45:46–59, 1999.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[6] L. J. Dickson. Data redundancy methods and apparatus, October 2003. U. S. Patent Application US2003/0196023 A1.

[7] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.

[8] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, 1989.

[9] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and KK Rao. Performance metrics for erasure codes in storage systems. Technical Report RJ 10321, IBM Research, San Jose, CA, 2004.

[10] J. L. Hafner, V. Deenadhayalan, KK Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, San Francisco, CA USA, December 2005.

[11] M. G. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, 2001.

[12] J. S. Plank, R. L. Collins, A. L. Buchsbaum, and M. G. Thomason. Small parity-check erasure codes – exploration and observations. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005.

[13] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004.

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

[15] J. A. Wiencko, Jr., K. Land, and L. J. Dickson. Data redundancy methods and apparatus, April 2003. U. S. Patent 6,557,123 B1.

[16] A. Wilner. Multiple drive failure tolerant raid system, December 2001. U. S. Patent 6,327,672 B1.

[17] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, IT-45:272–276, 1999.

[18] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems in Information Transmission*, 19:29–37, 1983.