Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs

Kevin M. Greenan
ParaScale
kgreenan@parascale.com

Xiaozhou Li
HP Labs
xiaozhou.li@hp.com

Jay J. Wylie
HP Labs
jay.wylie@hp.com

Abstract—**Large scale storage systems require multi-disk fault tolerant erasure codes. Replication and RAID extensions that protect against two- and three-disk failures offer a stark tradeoff between how much data must be stored, and how much data must be read to recover a failed disk. *Flat XOR-codes*—erasure codes in which parity disks are calculated as the XOR of some subset of data disks—offer a tradeoff between these extremes. In this paper, we describe constructions of two novel flat XOR-code, *Stepped Combination* and *HD-Combination* codes. We describe an algorithm for flat XOR-codes that enumerates *recovery equations*, i.e., sets of disks that can recover a failed disk. We also describe two algorithms for flat XOR-codes that generate *recovery schedules*, i.e., sets of recovery equations that can be used in concert to achieve efficient recovery. Finally, we analyze the key storage properties of many flat XOR-codes and of MDS codes such as replication and RAID 6 to show the cost-benefit tradeoff gap that flat XOR-codes can fill.**

I. INTRODUCTION

Erasure codes such as replication, RAID 5, and other Reed-Solomon codes, are the traditional means by which storage systems are typically made reliable. Such codes are Maximum Distance Separable (MDS). This means that they offer optimal space-efficiency for a given fault tolerance: each additional disk of redundancy allows the system to tolerate an additional disk (or sector) failure. As big data systems become more prevalent, more systems are comprised of huge populations of disks, and the commodity-based architecture leads to higher disk failure rates. Indeed, there are many alarming trends in disk failure rates [1], [2], [3], [4], [5]. These storage trends motivate the move towards two- and three-disk fault tolerant storage systems for big data, archival, and cloud storage.

As we move towards more and more fault tolerant storage systems, we believe that non-MDS codes ought to be considered for deployment. We have studied *flat XOR-codes* and identified features of such codes that may benefit a wide range of large-scale storage systems. Flat XOR-codes are erasure codes in which each parity disk is the XOR of a distinct subset of data disks. Such codes are non-MDS and so incur additional storage overhead, and, in some cases, additional small write costs relative to MDS codes. In return for these costs, such codes offer short, diverse *recovery equations*, i.e., many distinct sets of disks that can recover a specific disk.

We believe that the recovery equations of flat XOR-codes can be exploited in many different ways. First, short, diverse

recovery equations offers the potential for more efficient recovery of failed disks relative to MDS codes. By efficient, we mean fewer bytes need to be read and the read recovery load is spread evenly over most disks in the stripe. Efficient disk recovery of a flat XOR-code can be faster and can interfere less with the foreground workload than for a similarly fault tolerant MDS code. For flat XOR-codes with a *static* layout (like a RAID 4 layout with specific parity disks), a *simultaneous recovery schedule* uses multiple, distinct recovery equations to achieve efficient recovery. For flat XOR-codes with a *rotated* layout (like a RAID 5 layout with data and parity on each disk), a *rotated recovery schedule* uses a different recovery equation for each failed stripe to achieve efficient recovery. Our proposed *intra-stripe* recovery schedules complement techniques such as distributed sparing [6] and parity declustering [7].

We also believe there are opportunities in low-power (archival) storage and storage-efficient big data computing. Flat XOR-codes may offer a richer set of tradeoffs for power-aware storage systems that leverage redundancy to schedule disk power-downs: EERAID [8], PARRAID [9], eRAID [10], and Power-Aware Coding [11]. Big data computing systems that use triple-replication (e.g., Google File System [12] and the Hadoop File System [13]) may be able to achieve significantly higher storage-efficiency with nominal additional read costs incurred by computation scheduled on parity nodes.

We make the following contributions in this paper. First, we propose the following novel flat XOR-code *Combination* code constructions: *Stepped Combination* codes and *HD-Combination* codes. Both code constructions have two- and three-disk fault tolerant variants. We also describe a prior flat XOR-code construction, *Chain* code, and a *flattening* method to convert previously known array codes such as EVENODD into a flat XOR-code. Second, we develop the following algorithms for *flat XOR-codes*: the Recovery Equations (RE) Algorithm, the Simultaneous Recovery Schedule (SRS) Algorithm, and the Rotated Recovery Schedule (RRS) Algorithm. Third, we analyze key properties of flat XOR-codes and MDS codes for storage: storage overhead, small write costs, recovery equation size, recovery read load, and fault tolerance.

Even though replication is an MDS code, there is a gap between the properties of replicated storage (high space overhead, small recovery equations) and other traditional MDS codes such as RAID 6 (low space overhead, long recovery equations). Our analysis leads us to believe that the Combi-

nation codes and Chain codes delineate the possible tradeoff space that flat XOR-codes can achieve, and that this tradeoff space fills a large part of the gap between replicated storage and other MDS storage.

In Section II, we provide an extensive background on erasure-coded storage. We describe all of the flat XOR-code constructions, including our novel constructions, in Section III. In Section IV, we describe our algorithms for producing recovery equations and schedules for flat XOR-codes. We analyze key storage properties of two- and three-disk fault tolerant MDS and flat XOR-codes in Section V.

II. BACKGROUND

Over a decade ago, a class of non-MDS erasure codes, low-density parity-check (LDPC) codes, were discovered [14] (actually, re-discovered [15]). This class of erasure code has had a significant impact in networked and communication systems, starting with Digital Fountain content streaming [16], and now widely adopted in modern IEEE standards such as 10GBase-T Ethernet [17] and WiMAX [18]. The appeal of LDPC codes is that, for large such codes, a small amount of space-efficiency can be sacrificed to significantly reduce the computation costs required to encode and decode data over lossy channels. Networked systems can take advantage of the asymptotic nature of LDPC codes to great effect.

To date, the only non-MDS codes that have been used in practice in storage systems are replicated RAID configurations such as RAID 10, RAID 50, and RAID 60. Storage systems have not taken advantage of LDPC style codes. There are many reasons for this. First, storage systems only use *systematic* codes—erasure codes in which the stored data is striped across the first k disks. This allows *small reads* to retrieve a portion of some stored data by reading a minimal subset of the data disks. This constraint distinguishes storage systems from network systems. Second, though storage systems continue to get larger and larger, they do not take advantage of the asymptotic properties of LDPC codes because any specific stored object is striped over a relatively small number of disks (e.g., 8–20, maybe 30). This leads to the final reason: "small" LDPC codes are not well understood. There are not many well-known methods of constructing small LDPC codes for storage, though some specific constructions are known (e.g., [19], [20], [21], [22], [23]). The performance, fault tolerance, and reliability of small LDPC codes is also not well understood, though there is much progress (e.g., [24], [25], [26], [27], [28], [29], [11]). In this paper we provide many constructions of flat XOR-codes, which in some sense are small LDPC codes, and an analysis to put the storage properties of such constructions in perspective relative to traditional MDS codes.

A. Terminology

An erasure code consists of n symbols, k of which are *data symbols*, and m of which are *redundant symbols*. For XOR-codes, we refer to redundant symbols as *parity symbols*. We only consider *systematic* erasure codes: codes that store the data symbols along with the parity symbols. Whereas

coding theorists consider individual bits as symbols, symbols in storage systems correspond to device blocks. To capture this difference in usage of erasure codes by storage systems from elsewhere, we refer to *elements* rather than symbols. We use the terms *erase* and *fail* synonymously: an element is erased if the disk on which it is stored fails.

The fault tolerance of an erasure code is defined by d its *Hamming distance*. An erasure code of Hamming distance d tolerates all failures of fewer than d elements, data or parity. In storage systems, erasure codes are normally described as being one-, two-, or three-disk fault tolerant. These respectively correspond to Hamming distances of two, three, and four.

MDS codes use m redundant elements to tolerate any m erasures. MDS codes are optimally space-efficient: every redundant element is necessary to achieve the Hamming distance. Replication, RAID 5, RAID 6, and Reed-Solomon codes are all examples of MDS codes. To be specific, for replication $k = m = 1$, for RAID 5 $m = 1$, and for RAID 6 $m = 2$. Reed-Solomon codes can be instantiated for any value of k and m [30] and can be implemented in a computationally efficient manner (e.g., Liberation [31] and Liber8Tion codes [32]).

Still, much work has gone into determining constructions of MDS codes that use only XOR operations to generate redundant elements. Such constructions are based on parity techniques developed by Park [33]. We refer to this class of erasure code constructions as *parity-check array codes*. Examples of such codes include EVENODD [34], generalized EVENODD [35], Row-Diagonal Parity (RDP) [36], X-Code [37], Star [38], and P-Code [39]. Parity-check array codes have a two-dimensional structure in which some number of elements are placed together "vertically" in a *strip* on each disk in the *stripe*. This two-dimensional structure allows such codes to be MDS and so does not offer interesting recovery equation possibilities. In Section III-B though, we describe a method of converting parity-check array codes into non-MDS flat XOR-codes.

B. Flat XOR-codes

Flat XOR-codes are small low-density parity-check (LDPC) codes [14]. Because such codes are flat—each strip consists of a single element—and each parity element is simply the XOR of a subset of data elements, they cannot be MDS. The singular exception to this rule is RAID 5 which is a one-disk fault tolerant MDS flat XOR-code with $m = 1$.

A concise way of describing an LDPC code is with a Tanner graph: a bipartite graph with data elements on one side, and parity elements on the other. A parity element is calculated by XORing each data element to which it is connected together. Figure 1 illustrates a simple Tanner graph with the corresponding parity equations. The Tanner graph can also be used for efficient decoding of LDPC codes[1].

---

[1] A technique called *iterative decoding* based on the Tanner graph can efficiently decode with high probability. *Stopping sets* [40], specific sets of failures, can prevent iterative decoding from successfully decoding even though a more expensive matrix-based decode operation could succeed. In storage systems, we expect the expensive matrix-based decode method to be used in such cases.

When k is very large, LDPC codes can achieve near MDS level space-efficiency for a given fault tolerance, and significantly lower computational complexity for both encoding and decoding. In storage systems, these asymptotic properties do not apply since k tends not to be sufficiently large. Traditionally, LDPC codes are constructed by randomly generating a Tanner graph based on some distributions of the expected edge count for data and parity elements. Unfortunately, such techniques only work (with high probability) when k is sufficiently large. Therefore, storage systems practitioners need constructions of flat XOR-codes with well understood properties.

C. Related work

Parity declustering [7] (and chained declustering [41]) distribute the recovery read load among many disks. Parity declustering lays out many stripes on distinct sets of devices such that all available devices participate equally in reconstruction when some device fails; it needs to be used with more disks than the stripe width to distribute the recovery read load. Distributed sparing allows spare capacity on each available disk in the system to recover a portion of a failed device and so distributes the recovery write load [6]. The recovery schedules we propose are complementary to these techniques, and differ from these techniques in that they operate wholly within a stripe.

Recently, non-MDS self-adaptive parity-check array codes have been proposed [23]. Such codes allow disks to be rebuilt within the stripe so as to preserve two-disk fault tolerance, until fewer than two redundant disks are available. In some sense, distributed sparing is built into the array code itself. SSPiRAL codes also include techniques for rebuilding elements within a stripe as disks fail [22], [42]. Our techniques do not yet do anything analogous to distributed sparing within a stripe for flat XOR-codes. Pyramid codes [43] are non-MDS codes constructed from traditional MDS codes; we are interested in comparing flat XOR-codes to Pyramid constructions in the future. Weaver codes are non-MDS vertical (i.e., not flat) XOR-codes that Hafner discovered via computational methods [20]. Weaver codes have a regular, symmetric structure which limits the recovery read load after a failure to nearby "neighbor" disks. We are interested in distributing the recovery read load widely rather than localizing it.

There is much prior work on evaluating erasure codes in storage systems. Hafner et al. [44] outlined a comprehensive performance evaluation regime for erasure-coded storage. There is a long history of evaluating the reliability of RAID variants in storage systems (e.g., [45], [46], [47], [48], [1]). Some aspects of small LDPC code instances suitable for storage systems have been previously studied, e.g., read performance (e.g., [24], [25]), fault tolerance (e.g., [28], [27]), and reliability (e.g., [26], [29], [11]). For additional background on erasure coding in storage, see Plank's tutorial slides [49].
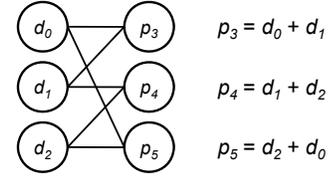

Fig. 1. Two-disk fault tolerant Chain code construction ($k = m = d = 3$).

III. CONSTRUCTIONS

In this section, we describe Chain codes a variant of a previously known flat XOR-code and *flattening* a technique to convert any parity-array check code into a flat XOR-code. We describe these variants of previously known codes in detail so that others can easily reproduce these constructions, and because we include these constructions in our analysis. We then describe our novel combination constructions for Stepped Combination and HD-Combination codes.

A. Chain codes

For a code to tolerate at least two disk failures, it must have a Hamming distance d of at least three. For a systematic XOR-based code, this means that each data element must be in at least two parity equations. Starting with this minimal constraint, two-disk fault tolerant *Chain* codes[2] can be constructed [19].

A Chain code construction requires that $k = m$. As originally proposed, a Chain code can be thought of as alternating data elements and parity elements in a row, with each parity element being the XOR of the data element on either side of it. Figure 1 illustrates our Chain code construction for $k = m = d = 3$ with the Tanner graph on the left and parity equations on the right. Let d_i be data element i, where $0 \le i < k$, and let p_j be parity element j, where $0 \le j < m$. For Hamming distance d, the parity equation for p_j is as follows:

$$p_j = \sum_{i=j}^{j+d-1} d_{i \bmod k}.$$

Simply put, each parity equation is the XOR sum of $d-1$ contiguous data elements, wrapping around appropriately after the kth data element.

The Chain code construction requires that $k = m \ge d = 3$. If k is less than d, then all of the parity equations end up being the same (the XOR sum of all data elements) and the constructed Chain code does not achieve a Hamming distance of d. If $d = 2$, then the Chain code construction replicates each data element once (i.e., RAID 10), and if $d = 1$, then the Chain code construction stripes the data (i.e., RAID 0). The Chain code construction can be extended to three-disk fault tolerance: if $k = m \ge d = 4$, then connect each parity element to three contiguous data elements.

---

[2]"Chain code" is our name for these codes. The patent by Wilner does not explicitly name these codes [19].
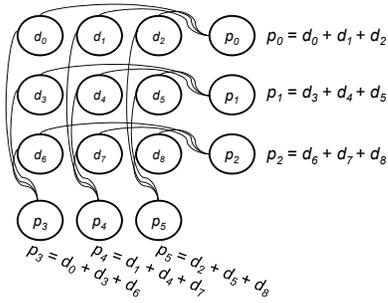
Fig. 2. SPC with $k = 9$ and $m = 6$. Hamming distance is three when flattened.

Fig. 3. Flattened RDP code for prime $p = 5$ yields $k = 16$, $m = 8$, and $d = 3$.

Unfortunately, we have not found such simple Chain code construction variants for a Hamming distance greater than four. Note that the Weaver$(n, 2, 2)$ codes [20] discovered by Hafner can be converted into the two-disk fault tolerant Chain code by flattening (discussed below). We believe that flattening the other Weaver(n, t, t) code constructions may produce other flat XOR-codes with greater Hamming distance, though the construction will not be as simple as the Chain codes for $d = 3$ and $d = 4$. Note that the SSPiRAL codes are another variant/extension of the Chain code construction [22].

B. Flattened parity-check array codes

Parity-check array code constructions can be *flattened* to produce a flat XOR-code with the same Hamming distance as the original array code. By flattening, we mean taking some two (or more) dimensional code and placing each element on a distinct device. Whereas the original parity-check array code construction produces an MDS, flattening the code yields a non-MDS code.

To illustrate the flattening technique, we apply it to two parity-check array code constructions: Simple Product Code (SPC) and Row-Diagonal Parity (RDP). Flattening can be applied to other two-disk fault tolerant array codes such as EVENODD, X-CODE, P-CODE, and so on. Flattening can also be applied to three-disk fault tolerant array codes such as STAR codes and generalized EVENODD codes, or even to non-MDS parity-check array codes such as Weaver and HoVer [21] codes.

1) Simple Product Code (SPC): A Simple Product Code (SPC) [50] is constructed by grouping data elements into a rectangle and then performing RAID 4 on each row and each column. Figure 2 illustrates a specific SPC construction in which nine data elements are layed out in a square pattern. There is a parity element for each row and for each column of data elements. When flattened, this code uses $k = 9$ data disks and $m = 6$ parity disks to tolerate any two element failures (i.e., $d = 3$).

We only consider flattened *square* SPC constructions (i.e., those with the same number of rows as columns) for the purposes of this paper, even though any rectangular layout of data elements can be used. For an SPC code with q rows/columns, $k = q^2$ and $m = 2q$. The storage overhead is therefore $(q + 2)/q$, and each data element participates in
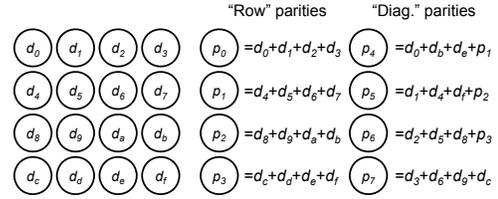
two parity equations each of which is of length q elements. Note that the *full-2* and *full-3* codes proposed by Gibson et al. [51] are similar to the SPC construction, and to the HoVer$_{1,1}^2$ construction given by Hafner [21].

2) Row-Diagonal Parity (RDP): Figure 3 illustrates a flattened RDP code. This figure is based on Figure 1 from the RDP paper [36] which illustrates the RDP construction for prime $p = 5$. The figure is setup to illustrate the columns (or strips) of elements that would be on each disk of an RDP instance. For example the first data disk would store elements d_0, d_4, d_8, d_c, and the first parity disk would store elements p_0, p_1, p_2, and p_3. Once flattened, each element of the RDP code is on a distinct disk producing a non-MDS flat XOR-code with $k = 16$, $m = 8$, and $d = 3$.

In general, for prime p, a flattened RDP code has $k = (p-1)^2$ and $m = 2(p-1)$. The space overhead is therefore $(p+1)/(p-1)$. Note that some of the "diagonal" parity equations are given in terms of "row" parity equations in the figure. This makes it look like each parity equation has $p - 1$ elements. In fact, p parity equations are $p - 1$ elements in length, and $p - 2$ parity equations are $2p - 3$ elements in length.

C. Combination codes

In this section, we present four novel Combination code constructions. We present two- and three-disk fault tolerant *Stepped Combination* codes. We then describe a variant of each of these codes called *HD-Combination* codes (where HD means Hamming Distance). We refer to these code constructions as Combination codes because they construct a Tanner graph by assigning distinct combinations of parity elements to each data element. The difference between the two constructions is in the size of the combinations assigned. We believe Combination constructions minimize the storage overhead for flat XOR-codes. Thus far, we have only developed constructions for $d = 3$ and $d = 4$. We expect that constructions for greater Hamming distances do exist, but have not yet found them. We include a proof sketch that combination codes achieve the specified Hamming distance in Appendix I.

1) Two-disk fault tolerant Stepped Combination: Stepped Combination codes are constructed by assigning all combinations of parity elements that are of size $d - 1$ to a different data element, then all combinations of parity elements that are of size d, and so on, up until the one combination of all m parity elements is assigned. Figure 4 lists the procedure for constructing a Stepped Combination code with $d = 3$. Given

```
100: for i := 2, 3, ..., m do
101:     for each m-choose-i combination of parities, P do
102:         for next unconnected data element D do
103:             connect D to the i parities in P
```

Fig. 4. Stepped Combination code construction for $d = 3$.

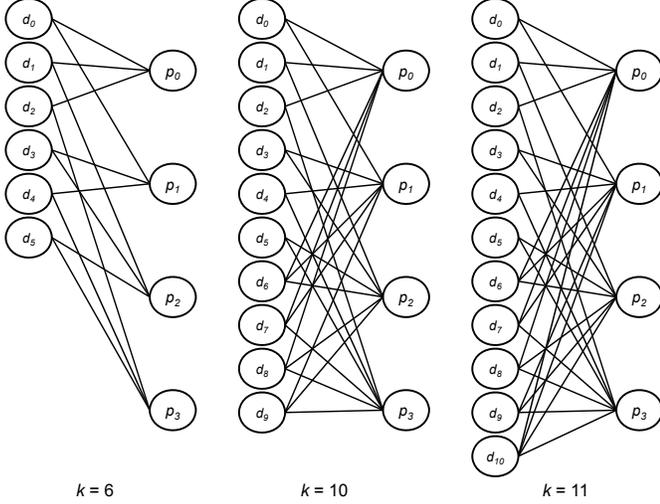
Fig. 5. Two-disk fault tolerant Stepped Combination constructions for $d = 3$ and $m = 4$.

the Hamming distance d and a number of parity elements m, the construction assigns combinations of increasing size to the available data elements. There is a maximum number of data elements for which this construction works. If $d = 3$ and $m = 4$, then there are $\binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 6 + 4 + 1 = 11$ distinct combinations of parity elements that can be assigned to data elements. In general, $k \leq \sum_{i=2}^{m} \binom{m}{i} = 2^m - m - 1$. In some sense, this also implies a lower bound on k: if a Stepped Combination code can be constructed for the desired value of k with a smaller m, then that smaller value of m should be used. Therefore, $2^{m-1} - m < k$.

Figure 5 illustrates the Tanner graphs of three Stepped Combination code constructions with $d = 3$ and $m = 4$. The values of k are chosen based on the three different sizes of combinations: 6 of size two, $6 + 4 = 10$ of size two or three, and $6 + 4 + 1 = 11$ of size two through four. In the Tanner graph for $k = 6$, notice that each data element participates in two distinct parity equations (connects to two parity elements). For the Tanner graph with $k = 10$ the first six data elements are the same as for $k = 6$. The next four data elements each participate in three parity equations. Finally, the Tanner graph with $k = 11$ has the same first ten data elements as for $k = 10$ and then one additional data element that connects to all four parity elements. These Tanner graphs illustrate the key idea behind the construction of Stepped Combination codes.

2) Three-disk fault tolerant Stepped Combination: The Stepped Combination code construction for Hamming distance four is quite similar to that for three, but with one key difference. Consider the construction algorithm for the two-disk fault tolerant Stepped Combination code in Figure 4. The

first difference for the three-disk fault tolerant construction is that variable i on line 100 starts at 3: to achieve a Hamming distance of four, all data elements must be connected to at least three parity elements. The second difference is that variable i increments by two each iteration, i.e., $i := 3, 5, 7, \ldots, m$. This difference is not intuitive, though the proof sketch in Appendix I provides some explanation.

Given some number of parity elements m, the three-disk fault tolerant Stepped Combination code has an upper and lower limit on k. For example, if $m = 5$, then $k \leq \binom{5}{3} + \binom{5}{5} = 10 + 1 = 11$. In general, $k \leq \sum_{3 \leq i \leq m, i \text{ odd}} \binom{m}{i} = 2^{m-1} - m$. The lower bound is based on $m - 1$ and so $2^{m-2} - m + 1 < k$. Compared with the two-disk fault tolerant construction, the maximum value of k drops by about half: $2^m - m - 1$ vs. $2^{m-1} - m$.

3) HD-Combination codes: HD-Combination codes are a variant of Stepped Combination codes. As with Stepped Combination codes, we have developed two- and three-disk fault tolerant constructions. Whereas the Stepped Combination construction steps up the size of parity combinations, the HD-Combination construction only uses parity combinations of the minimum size necessary to achieve the Hamming distance. As with the Stepped Combination code, we have not found HD-Combination constructions for larger Hamming distances yet.

The HD-Combination construction code differs from the algorithm described in Figure 4 only on line 100. For an HD-Combination construction with either $d = 3$ or $d = 4$, this line is simply $i = d - 1$. This difference leads to different bounds on k. For the two-disk fault tolerant HD-Combination code construction, $\binom{m-1}{2} < k \leq \binom{m}{2}$. Whereas for the three-disk fault tolerant construction, $\binom{m-1}{3} < k \leq \binom{m}{3}$.

IV. RECOVERY EQUATIONS AND SCHEDULES

In this section, we describe *recovery equations* (i.e., sets of elements that can be used to recover a specific other element) and *recovery schedules* (i.e., sets of recovery equations that can be used in concert to make recovery more efficient). To illustrate the basic ideas, we first consider multi-disk fault tolerant MDS codes. We then describe examples of recovery equations and schedules for flat XOR-codes that illustrate some key differences with MDS codes. We then present algorithms to determine recovery equations and schedules for flat XOR-codes. Finally, we discuss some limitations of our current algorithms.

A. MDS Examples

For MDS codes, any combination of k elements can be used to recovery any other element. Therefore, each element in an MDS code has exactly $\binom{k+m}{m}$ recovery equations, each consisting of k elements. For two-disk fault tolerant codes like RAID 6, this is only $k + m - 1$ recovery equations.

Figure 6 illustrates the recovery equations for element e_0 of a RAID 6 MDS code with $k = 4$ and $m = 2$. In this example, the code is static (not rotated), so disks 0 through 3 are data disks, and disks 4 and 5 are parity disks. If disk 0 were inaccessible, and an element e_0 from some stripe needs
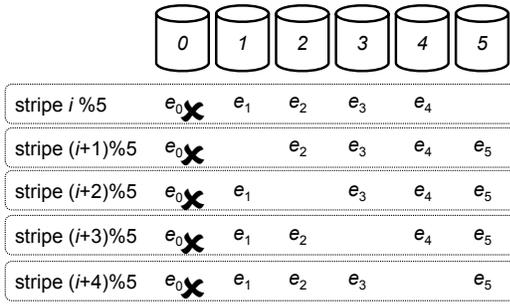
	0	1	2	3	4	5
stripe i %5	e_0✗	e_1	e_2	e_3	e_4	
stripe $(i+1)$%5	e_0✗		e_2	e_3	e_4	e_5
stripe $(i+2)$%5	e_0✗	e_1		e_3	e_4	e_5
stripe $(i+3)$%5	e_0✗	e_1	e_2		e_4	e_5
stripe $(i+4)$%5	e_0✗	e_1	e_2	e_3		e_5

Fig. 6.   Recovery equations for RAID 6.

	0	1	2	3	4	5
stripe i %4	d_0✗	d_1		p_3		
stripe $(i+1)$%4	d_0✗		d_2	p_3	p_4	
stripe $(i+2)$%4	d_0✗		d_2			p_5
stripe $(i+3)$%4	d_0✗	d_1			p_4	p_5

Fig. 7.   Simultaneous recovery schedule for Chain code.

to be read, any of the five sets of four other elements in that stripe can be read and decoded instead to recover e_0.

Now, consider if disk 0 fails, as is illustrated in Figure 6. A *recovery schedule* is a set of recovery equations that can be used in concert to make recovery more efficient. Since the code is static, we want a *simultaneous recovery schedule* that uses many different recovery equations for the same element in concert. This figure illustrates such a schedule: each of the five recovery equations is used to recover one fifth of the stripes on the failed disk. This simultaneous recovery schedule distributes the recovery read load evenly over all the available disks.

If the RAID 6 code in Figure 6 was rotated, then elements e_0 through e_5 would be stored in the various stripes on disk 0. For this layout, a *rotated recovery schedule* is required. For MDS codes, a simultaneous recovery schedule is easily converted to a rotated recovery schedule since any k elements can recover *any* element. I.e., the recovery schedule is the same, but the element recovered on disk 0 depends on which stripe is being recovered.

B. Flat XOR-code examples

Recovery equations for flat XOR-codes are more complicated than for MDS codes. Each element of a flat XOR-code may have a different number of recovery equations, and each of these may differ in size. For example, consider the recovery equations of the the two-disk fault tolerant Chain code illustrated in Figure 1 with $k = m = 3$:

$$d_0 = d_1 \oplus p_3 = d_2 \oplus p_3 \oplus p_4 = d_2 \oplus p_5 = d_1 \oplus p_4 \oplus p_5$$
$$d_1 = d_0 \oplus p_3 = d_2 \oplus p_4 = d_2 \oplus p_3 \oplus p_5 = d_0 \oplus p_4 \oplus p_5$$
$$d_2 = d_1 \oplus p_4 = d_0 \oplus p_3 \oplus p_4 = d_0 \oplus p_5 = d_1 \oplus p_3 \oplus p_5$$
$$p_3 = d_0 \oplus d_1 = d_0 \oplus d_2 \oplus p_4 = d_1 \oplus d_2 \oplus p_5 = p_4 \oplus p_5$$
$$p_4 = d_1 \oplus d_2 = d_0 \oplus d_2 \oplus p_3 = d_0 \oplus d_1 \oplus p_5 = p_3 \oplus p_5$$
$$p_5 = d_0 \oplus d_2 = d_1 \oplus d_2 \oplus p_3 = d_0 \oplus d_1 \oplus p_4 = p_3 \oplus p_4$$

We can list these recovery equations exhaustively because the code is small and simple. Contrast these recovery equations with those of the similarly sized MDS code discussed above. Some of these recovery equations consist of only two elements, which is less than $k = 3$. This difference is more pronounced in flat XOR-codes with larger k. Another property that emerges with larger k is that there are more distinct
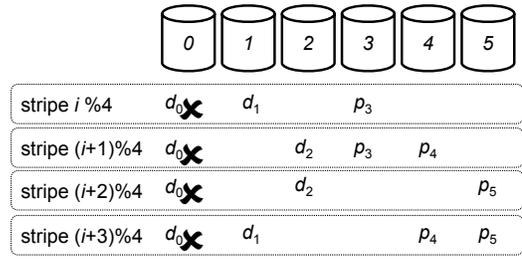
recovery equations to choose from than for a similarly sized MDS code.

Figure 7 illustrates a simultaneous recovery schedule for the two-disk fault tolerant Chain code discussed previously. If disk 0 fails, and the Chain code has a static layout, then each of the four recovery equations for d_0 can be used in a simultaneous recovery schedule. Such a schedule has two interesting features. First, 2.5 disk's worth of data are read to recover the failed disk. This is less than $k = 3$, the minimum amount of data required to be read by an MDS code with $k = 3$. Second, each available disk reads only one half a disk's worth of data, and the recovery read load is evenly distributed among available disks. These two properties, reducing the minimum total amount of data required to be read for recovery, and distributing that recovery read load evenly among available devices, are the properties we mean by efficient recovery.

If distributing the recovery read load evenly among available disks is not a priority, then just the shortest recovery equations can be used: $d_0 = d_1 \oplus p_3 = d_2 \oplus p_5$. This simultaneous recovery schedule requires two disk's worth of data to be read, less than the above schedule. It also requires four of the five available disks to read half a disk's worth of data, and the one remaining available disk, disk 4, to read none.

Figure 8 illustrates a rotated recovery schedule for the two-disk fault tolerant Chain code discussed previously. Six rotated stripes are illustrated with all elements labeled. In the figure, disk 0 is failed and bold-faced elements are in the recovery equation used to recover the failed element, whereas grayed out elements are not. A total of two disk's worth of data are read to recover failed disk 0. Of the five available disks, two read half a disk's worth of data (disks 1 and 5) and three read one third of a disk's worth of data (disks 2, 3, and 4). Disks 1 and 5 could thus be a bottleneck in this recovery.

C. Recovery Equations (RE) Algorithm

Before describing the RE Algorithm, consider a simplistic, brute-force algorithm which enumerates the powerset of all possible recovery equations (i.e., all other elements) for each element in the code. Each possible recovery equation is tested to see if it recovers the desired element; if it does so, then it is retained as a recovery equation. (By tested, we mean decoding would be attempted which is essentially a matrix rank test.) Such a brute force algorithm is very expensive since it evaluates $(k + m) \cdot 2^{k+m-1}$ possible recovery equations.
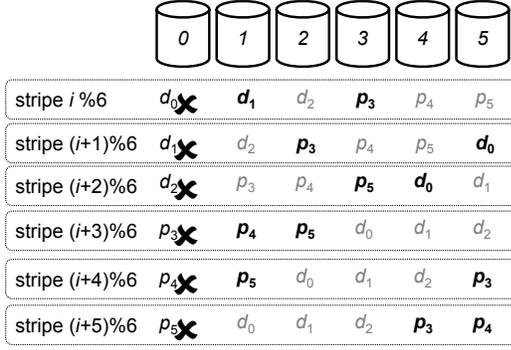
Fig. 8. Rotated Recovery for Chain code.

```
re_algorithm ()
200: for i ← 0, ..., k + m − 1 do
201:     BRE[i] ← base recovery equations for e_i
202: for i ← 0, ..., k + m − 1 do
203:     RE[i] ← re_element(i, BRE)
204: return RE
re_element(i, BRE)
300: RE_i ← BRE[i]
301: for all re ∈ RE_i do
302:     for all e_j ∈ re do
303:         for all bre ∈ BRE[j] do
304:             if e_i ∉ bre then
305:                 re' ← re ⊕ bre
306:                 if re' ∉ RE_i then
307:                     RE_i ← RE_i ∪ re'
308: return RE_i
```

Fig. 9. Recovery Equations (RE) Algorithm pseudocode.

```
fsrs(srs, SRS, re_off, RE_i, D)
400: for re_ndx ∈ (re_off, ..., |RE_i|) do
401:     srs_nxt ← srs ∪ {RE_i[re_ndx]}
402:     if max_depth(srs_nxt) ≤ D then
403:         SRS ← fsrs(srs_nxt, SRS, re_ndx + 1, RE_i, D)
404: return SRS ∪ {srs}
```

Fig. 10. Part of the Simultaneous Recovery Schedule (SRS) Algorithm pseudocode that enumerates feasible schedules for element e_i with depth D.

Figure 9 illustrates the pseudocode for our current RE Algorithm. The RE Algorithm first generates *base recovery equations* for each element (line 201). The Tanner graph is used to generate the base recovery equations. Each parity element has exactly one base recovery equation: the parity equation itself. Each data element has at least one base recovery equation per parity element to which it is connected. Each parity equation in which the data element appears is rearranged to become a base recovery equation for that data element. Note that if any *odd set* of recovery equations for some element e_i (i.e., any $3, 5, \ldots$ equations) are added together (via XOR), then the resulting recovery equation also solves for e_i. So, if a data element is connected to three or more parity elements, the RE Algorithm includes all odd sets of the initial base recovery equations in that data element's base recovery equations.

The RE Algorithm generates all of the other recovery equations for each element in turn by invoking **re_element**. The set of recovery equations RE_i for element e_i is initialized with its base recovery equations (line 300). Every recovery equation re in RE_i is used to generate additional recovery equations to add to RE_i (line 301). Each element e_j of re is substituted by every one of its base recovery equations that does not contain element e_i (lines 302–305). The XOR notation as applied to the sets on line 305 means that only the unique elements are retained (i.e., $(re \cup bre) \setminus (re \cap bre)$). If a new recovery equation is found, it is added to the set RE_i (lines 306–307). Note that this new recovery equation will eventually also be processed by this loop (i.e., by line 300).

To make this algorithm a bit more concrete, consider the recovery equations we list above for the two-disk fault tolerant Chain code. For d_0, the base recovery equations are $d_1 \oplus p_3$ and $d_2 \oplus p_5$. For d_1, the base recovery equations are $d_0 \oplus p_3$ and $d_2 \oplus p_4$. In the first iteration of the loop in **re_element** for $i = 0$, the element d_1 in the first base recovery equation of d_0 is substituted. The second base recovery equation of d_1 is substituted because the first includes d_0. This results in recovery equation $d_2 \oplus p_3 \oplus p_4$ being added to RE_i.

D. Recovery schedule algorithms

The Simultaneous Recovery Schedule (SRS) Algorithm uses the recovery equations enumerated by the RE Algorithm to generate simultaneous recovery schedules. Remember that a recovery schedule is simply a set of recovery equations. The SRS Algorithm works similarly to an iterative, depth-first search of possible schedules, where "depth" is the number of recovery equations in which each disk participates in a schedule. Given the recovery equations and some maximum depth D_{max}, the SRS Algorithm enumerates all the feasible schedules for each depth up to D_{max}. A *feasible* schedule for depth D being one in which no disk participates in more than D distinct recovery equations. The SRS Algorithm post-processes the feasible schedules for all depths to find the most efficient.

Figure 10 illustrates **fsrs**, the greedy and recursive manner in which feasible schedules are enumerated for depth D. For the initial invocation of **fsrs**, both srs and SRS are $\emptyset$. The greedy and recursive parts of this algorithm are lines 400 and 403 respectively. Together, these steps ensure that all feasible schedules are found and returned in variable SRS. The method **max_depth** returns the maximum depth across all elements for the schedule $srs \cup \{RE_i\}$.

The SRS Algorithm finds the most efficient schedule by post-processing all of the feasible schedules found up to D_{max}. An efficient schedule primarily minimizes the amount of data read by any one disk and secondarily minimizes the overall amount of data read. For the two-disk fault tolerant Chain code discussed above, the best simultaneous recovery schedule for d_0, $\{d_1 \oplus p_3, d_2 \oplus p_5\}$, would be found by **fsrs** at $D = 1$. The schedule illustrated in Figure 7 would be found by **fsrs** at $D = 2$.

The Rotated Recovery Schedule (RRS) Algorithm is a

straight forward variant of the SRS Algorithm: whereas the SRS Algorithm consider all recovery equations for a single element at a time, the RRS Algorithm considers all recovery equations for all elements. Unfortunately, this makes the RRS Algorithm much slower than the SRS Algorithm. There are two reasons for this slowdown. First, the RRS Algorithm must consider many more recovery equations. Second, the RRS Algorithm needs to run to a greater depth to find a schedule which recovers all of the elements.

E. Discussion

The number of recovery equations can grow exponentially. The RE Algorithm can be configured to terminate after some amount of time, or some number of recovery equations have been found. Developing more efficient versions of the RE Algorithm, SRS Algorithm, and the RRS Algorithm is an open question.

Hafner et al. have developed heuristic methods of constructing a pseudoinverse of a parity-check array code that determines whether a specific configuration of disk and sector failures is recoverable [52], and if so, provides a short recovery equation to recover the lost data. We believe that an algorithm based on Hafner's could more effectively find "short" recovery equations than the RE Algorithm. We believe that stochastic optimization techniques could be incorporated into the SRS Algorithm or RRS Algorithm to tame some of the state space explosion. We also believe that recovery schedules for flat XOR-codes are related to network coding for erasure-coded storage systems [53], [54].

We have only presented recovery schedules for single disk failures even though the example Chain code is two-disk fault tolerant. The algorithms we have developed easily extend to recovering a single disk in the face of a multi-disk failure: recovery equations that include failed elements are not passed into the scheduling algorithms. We have also developed extensions to the SRS Algorithm to produce recovery schedules for static layouts that re-use elements within a strip to recover multiple failed elements in that stripe. Note that MDS codes are much simpler in this regard: any k elements can recover any number of other failed elements.

V. ANALYSIS

In our analysis, we focus on two- and three-disk fault tolerant codes. We compare flat XOR-codes to MDS codes for stripes based on $1 \leq k \leq 30$. When $k = 1$, MDS codes are three- and four-fold replication (with $m = 2$ and $m = 3$ respectively). When $k > 1$, the MDS codes are RAID 6 ($m = 2$) and beyond ($m = 3$). Table I lists the codes we analyze. Checkmarks indicate which codes are included in two- and three-disk fault tolerant analysis. Pointers to the appropriate sections of this paper and to the original description of the code are listed as well. Flat parity-check array code constructions depend on some prime number or other constant. When we analyze such codes for a given k, we select the most space-efficient construction that is large enough to store k data elements. I.e., we use the smallest prime number or other

Code	2DFT	3DFT	Comment
MDS	✓	✓	Replication, RAID 6, and beyond.
StepComb	✓	✓	Section III-C
HDComb	✓	✓	Section III-C
Chain	✓	✓	Section III-A
Flat (square) SPC	✓		Section III-B and [50]
Flat RDP	✓		Section III-B and [36]
Flat XCODE	✓		Section III-B and [37]
Flat STAR		✓	Section III-B and [38]

TABLE I
ERASURE CODES ANALYZED.

constant that generates a code that we can then shorten to the appropriate k value. For SPC codes, we only analyze "square" constructions.

We analyze the following storage properties of the erasure code constructions:

1) Relative storage overhead (Section V-B): The amount of data stored relative to a single replica. I.e., the cost of a full stripe write.
2) Small write cost (Section V-C): The average number of parity elements that need to be updated when a single data element is updated.
3) Average size of shortest recovery equation (Section V-D): The fewest number of disks that must be accessed to recover an element. I.e., the number of disk's worth of data that must be read to recover one failed disk.
4) Average recovery read load (Section V-E): The amount of data relative to an entire disk's worth of data, that an available disk must read to recover one failed disk within the stripe.
5) Fault tolerance at the Hamming distance (Section V-F): The fraction of three- and four-disk failures that lead to data loss for two- and three-disk fault tolerant codes respectively.

We summarize all of our analyses for a specific number of data disks ($k = 15$) in Section V-G.

A. Computation

Even though computation costs of erasure codes receive the bulk of attention from coding theorists, we believe that they are rarely the bottleneck in storage systems. Whether the CPU that performs erasure coding is firmware in a RAID controller or a library on a general-purpose CPU, we believe that bandwidth constraints in cache, the memory bus, on the network, or from the storage devices dictate performance. Beyond this, most reads in erasure-coded storage systems read data elements directly and so require no computation to decode.

To provide a sense of the computational demands of the various codes being analyzed, we counted the minimum number of distinct mathematical operations that must be performed to encode an entire stripe. For a given d, both the computational cost of MDS and non-MDS codes both increase linearly with k and are within a small constant factor of one another. In

practice, the non-MDS codes are likely to be computationally faster to encode (or decode) for two main reasons: one, many of the XOR computations are common across parity elements and so XOR-*Scheduling* can be used [55]; two, the actual mathematical operations performed while encoding an MDS code are heavier weight than the simple XOR performed in the non-MDS codes [30], [56], [57]. In summary, there are moderate differences between computational costs among the codes we are analyzing, but we do not believe these costs affect the decision of which codes to employ in a given storage system.

B. Relative storage overhead

Figure 11 and Figure 12 show the relative storage overhead for two- and three-disk fault-tolerant codes respectively. The y axis on each graph goes up to the relative storage overhead of d-fold replication. A key property of MDS codes is optimal storage overhead. As k increases, the relative storage overhead of MDS codes decreases monotonically.

The Chain code construction has a fixed relative storage overhead of 2. This is because it is constructed with $m = k$. I.e., the Chain code has the same relative storage overhead as two-fold replication, but can achieve either two- or three-disk fault tolerance (depending on the construction).

As the stripe widens (k increases), the relative storage overhead of all the other flat XOR-code constructions reduces. The Stepped Combination construction provides the best relative storage overhead. We believe the Stepped Combination construction uses the minimal number of parity elements possible to achieve the desired fault tolerance using only simple XOR operations upon data elements to produce parity elements. If k gets large enough, then a construction like the Stepped Combination code exhibits storage overhead like LDPC codes [14]. At $k = 30$, the three-disk fault tolerant StepComb code has 12.4% more storage overhead than the MDS code, and the four-disk fault tolerant StepComb has 11.8% more.

The storage overhead of the flattened parity-check array codes and combination codes do not decrease monotonically. This is because we plot the most space-efficient construction for each of these specific code constructions that can store k data elements.

C. Small write cost

Two key performance metrics of erasure-coded storage are small write costs and strip write costs [44]. A small write is a write that updates a single data element and all necessary parity elements. A strip write is a write that updates a column of (data) elements and all necessary parity elements. In traditional parity-check array codes, these performance metrics are different. For flat XOR-based codes and MDS codes, these performance metrics are identical because each strip consists of a single element.

To calculate the small write cost, we count the number of parity elements that each data element must update if it
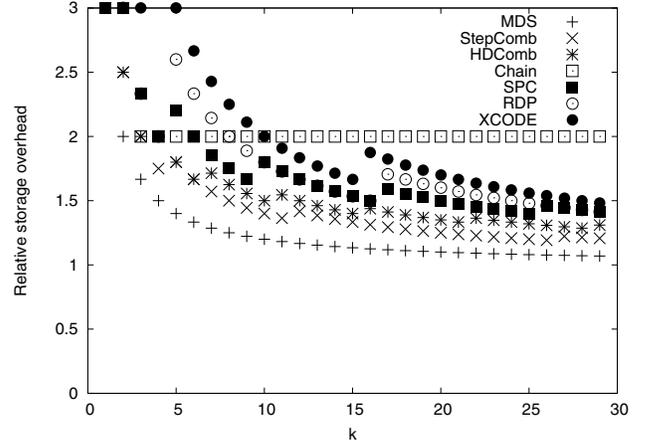

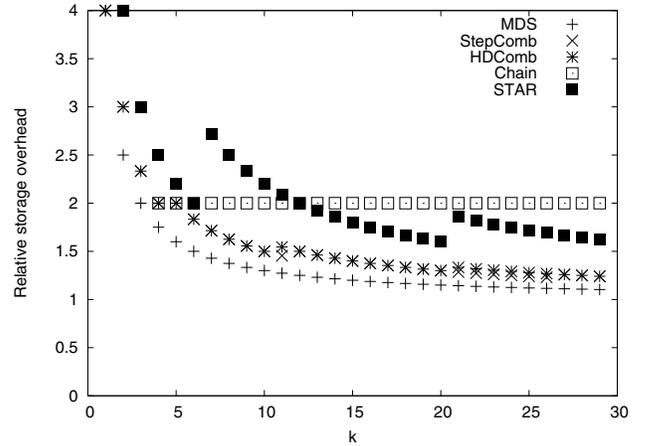Fig. 11.   Relative storage overhead for $d = 3$.

Fig. 12.   Relative storage overhead for $d = 4$.

is updated. An average is reported because for flat XOR-codes, different data elements have different small write costs. Figure 13 and Figure 14 show the average small write cost for two- and three-disk fault-tolerant codes respectively.

As with relative storage overhead, MDS codes are optimal in this property: exactly $d-1$ parity elements must be updated whenever a data element is updated. Many of the flat XOR-codes are also optimal: HDComb, Chain, SPC, and XCODE. Each of these XOR-codes was constructed to achieve optimal update complexity and so this result is not surprising. For some values of k, the remaining XOR-codes are also optimal: StepComb, RDP, and Star.

For values of k that the StepComb construction is identical to the HDComb construction, it achieves optimal small write costs. Over the range of k values analyzed, three-disk fault tolerant StepComb is always less than 50% worse than optimal. Whereas four-disk fault tolerant StepComb is always less than 17% worse than optimal.
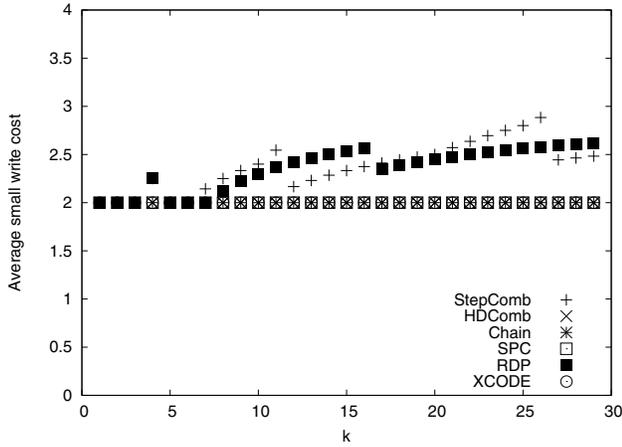
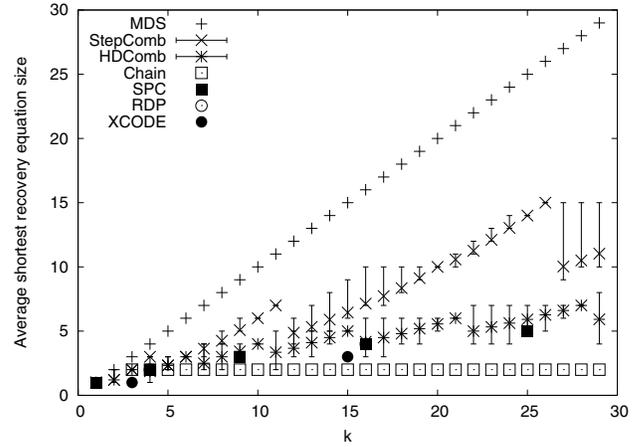Fig. 13. Small write costs for $d = 3$.

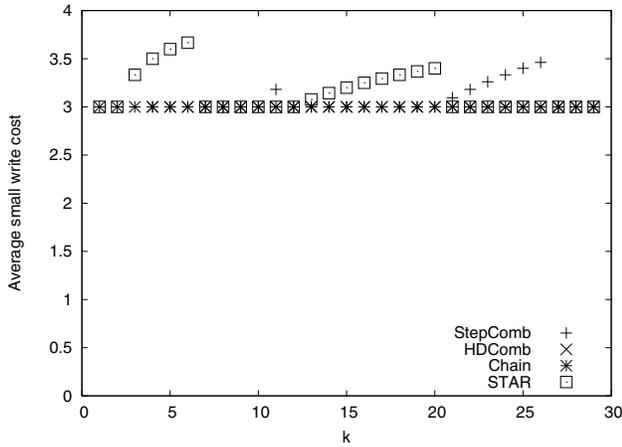Fig. 15. Average shortest recovery equation size $d = 3$.

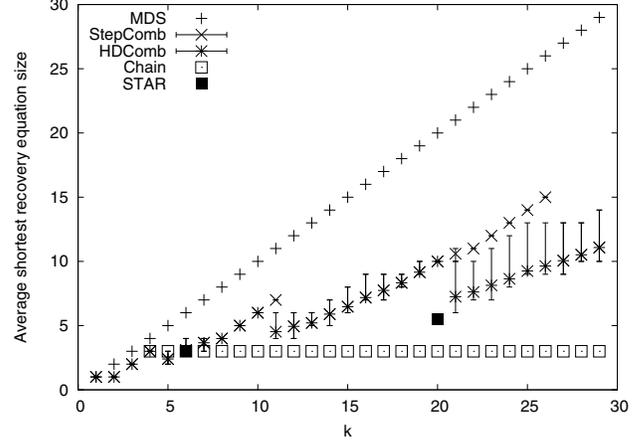Fig. 14. Small write costs for $d = 4$.

Fig. 16. Average shortest recovery equation size for $d = 4$.

D. Recovery equation size

We use the RE Algorithm to analyze all the flat XOR-codes to find all of the recovery equations for each element. We then calculate the average size of the smallest recovery equation. We are interested in the size of the smallest recovery equations because they indicate the potential for choosing a recovery equation over a small read, and are the best-case for how many disk's of data must be read to recover a failed disk. To put this in context, consider replication: each element in x-fold replication has $x - 1$ recovery equations of size one. I.e., one disk's worth of data needs to be read to recover a failed disk. Replication has the smallest recovery equations of any code.

Figure 15 and Figure 16 show the results for two- and three-disk fault tolerant codes respectively. The shortest recovery equation for MDS codes increases linearly with k since k elements are required to recover any element. The Chain code has the shortest recovery equations of $d - 1$ since all parity elements are connected to exactly $d - 1$ data elements, and vice versa. Of course, this is only possible due to the storage overhead incurred by the Chain code construction. The flat

parity-check array code constructions also offer quite short recovery equations also due to the storage overhead cost.

For most of the codes, the minimum, average, and maximum size of shortest recovery equations across all data elements are the same. For the combination codes, different data elements have different shortest recovery equations and so we use error bars to indicate the size of the minimum and maximum values of the shortest recovery equation across all k data elements. The inverse relationship between storage overhead and shortest recovery equation size continues with the combination codes falling between MDS codes and the other flat XOR-codes, and with HDComb codes having shorter average recovery equations than StepComb codes.

All of these results are based on exactly one element being inaccessible—if multiple elements are inaccessible, then the average shortest recovery equation size will likely increase for flat XOR-codes.

E. Recovery read load

The recovery read load is the amount of data relative to an entire disk's worth of data, that an available disk must
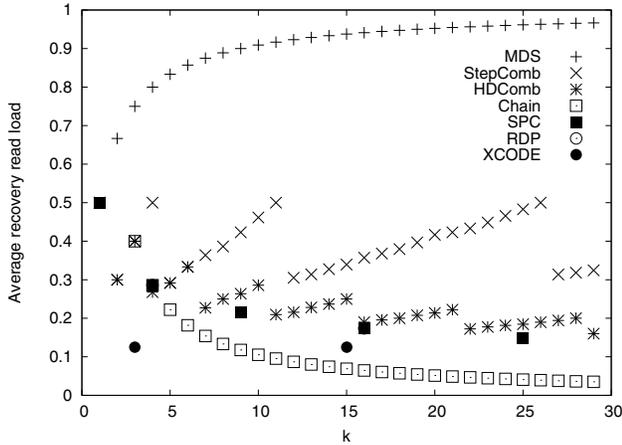
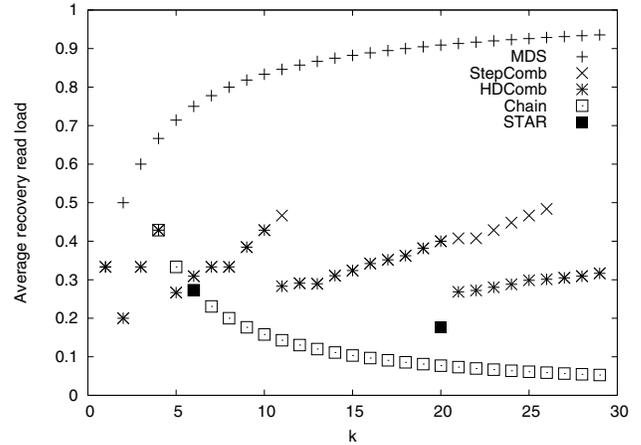Fig. 17. Average recovery read load for $d = 3$.

Fig. 18. Average recovery read load for $d = 4$.

read to recover a failed disk within the stripe. The smaller this value is, the quicker recovery can complete, or the less impact recovery has on foreground workload, or a combination of both. Figure 17 and Figure 18 shows the results for two- and three-disk fault tolerant codes respectively. This analysis is again based on a single disk failure. Beyond this, these results assume that code rotation perfectly distributes the recovery read load across available devices (i.e., that the recovery schedule takes full advantage of the average smallest recovery equation).

As k increases, MDS codes approach the case of each available disk having to be read in its entirety to recover a single failed disk. The flat XOR-codes exhibit radically different behavior! Storage overhead has a compound affect: it creates the opportunity for small recovery equations *and* allows the recovery read load to be spread over more disks. The end result is that Chain codes approach the case of each available disk within a stripe having to read a nominal amount of data to recover a single failed disk. The flat parity-check array codes exhibit the same trend, though in a less pronounced manner.

The combination codes are again in between the MDS codes and the other flat XOR-codes. At two-disk fault tolerance, HDComb codes range from 18% to 40% and StepComb codes range from 30% to 50% of a disk's worth of data being read by each available disk. At three-disk fault tolerance, the combination codes range from 25% to 50% along this metric.

Consider each available disk having to read 25% of a disk's worth of data to recover an element. If recovery read disk bandwidth is the bottleneck for recovering the failed disk, then the failed disk can be recovered $4\times$ faster. Alternately, for a given recovery time objective, the recovery workload can be reduced by this same factor.

Note that for d-fold replication ($k = 1$) each available device must read $1/(d - 1)$ of a disk's worth of data. For three-fold replication, this is 0.5 and for four-fold replication, this is 0.33.

We have used the RRS Algorithm and SRS Algorithm to produce simultaneous and rotated recovery schedules for many of the codes we analyze in this section. The rotated schedules we have produced thus far have matched the analysis in this section. The simultaneous schedules we have produced thus far have yielded two distinct recovery equations (or the equivalent schedule) for all the flat XOR-codes. Because our current algorithms are computationally intensive, we are still working on producing complete recovery schedules for all of the codes for stripes with k up to 30.

F. Fault tolerance

Flat XOR-codes offer fault tolerance at and beyond the Hamming distance. By this, we mean that a two-disk fault tolerant code may be able to tolerate *some* triple-disk failures (but not all). MDS codes do not tolerate any failures at or beyond the Hamming distance. This is because such codes are optimally space-efficient. The storage overhead of non-MDS codes is what makes fault tolerance at and beyond the Hamming distance possible

Figure 19 and Figure 20 show the fault tolerance, at the Hamming distance, for two- and three-disk fault tolerant codes respectively. We plot the fraction of failures at the Hamming distance that lead to data loss. Notice that the y axes of these graphs are logarithmic and so these graphs are somewhat like the "nines" graphs used to illustrate availability. Results for the flat parity-check array codes are only plotted for constructions that are not shortened (i.e., for the largest possible value of k given the prime used to construct the code).

The fault tolerance at the Hamming distance is inversely correlated with the storage overhead: the more storage overhead of the code construction, the fewer faults at the Hamming distance that lead to data loss. In particular, the Chain code, with the highest storage overhead, offers the most fault tolerance at the Hamming distance.

We do not include fault tolerance beyond the Hamming distance in our analysis because additional fault tolerance at the Hamming distance has the largest affect on overall reliability. Translating fault tolerance of codes into reliability is
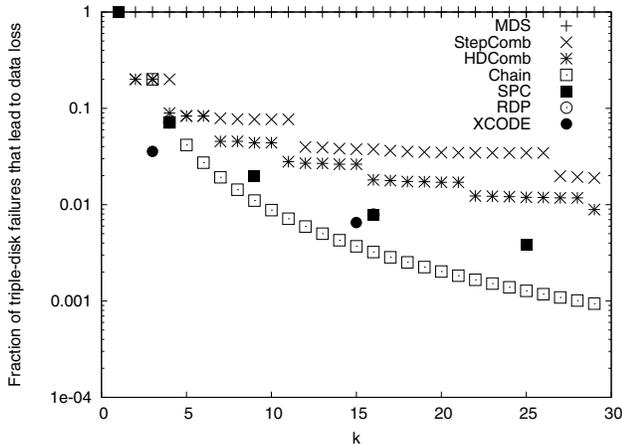
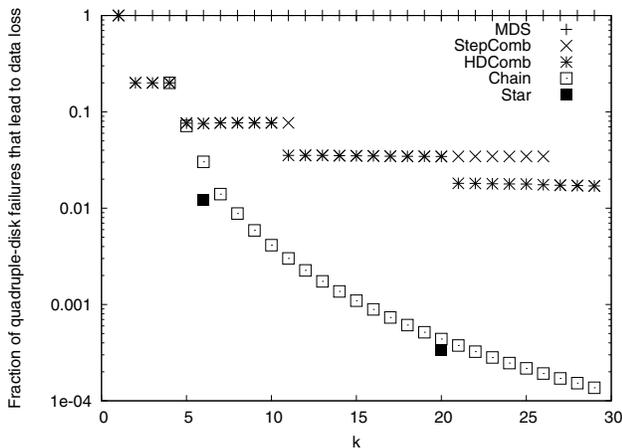Fig. 19.   Fraction of triple-disk failures that lead to data loss.

Fig. 20.   Fraction of quadruple-disk failures that lead to data loss.

Code	d	Stor. over.	Small write	Min. RE	Read load	% fail at d
MDS	3	1.13	2.00	15.00	0.94	100
StepComb	3	1.33	2.33	6.45	0.34	3.8
HDComb	3	1.40	2.00	5.00	0.25	2.6
Chain	3	2.00	2.00	2.00	0.069	0.37
Rep ($k = 1$)	3	3.00	2.00	1.00	0.50	100
MDS	4	1.20	3.00	15.00	0.88	100
StepComb	4	1.40	3.00	6.48	0.32	3.5
HDComb	4	1.40	3.00	6.48	0.32	3.5
Chain	4	2.00	3.00	3.00	0.10	1.1
Rep ($k = 1$)	4	4.00	3.00	1.00	0.33	100

TABLE II
SUMMARY OF RESULTS FOR $k = 15$.

beyond the scope of this paper (though there is prior work [26], [29], [11]).

G. Results summary

At a high level, flat XOR-codes pay some cost in storage overhead, small write, or both relative to MDS codes. By incurring this cost, flat XOR-codes are able to achieve shorter recovery equations, reduced recovery read load, and increased fault tolerance at the Hamming distance. Table II summarizes our analysis for specific codes with $k = 15$: MDS codes, StepComb, HDComb, and Chain codes. We also include Replication (Rep) on a distinct line from MDS in the table. For Replication, we use $k = 1$ which is a much smaller stripe than for the other codes. This set of codes illustrates the various tradeoffs possible between MDS codes and flat XOR-codes. We believe that the properties demonstrated by the StepComb, HDComb, and Chain constructions delineate the possible tradeoff space for flat XOR-codes. I.e., we believe that there exists other flat XOR-codes we did not analyze, and which no one may yet know how to construct, that achieve tradeoff points between these three code constructions.

Compare two-disk fault tolerant StepComb to MDS: an 18% additional storage overhead and a 17% small write overhead provides recovery equations that are on average 43% the size, a 64% reduction in per-disk read load during recovery, and the ability to tolerate 96.2% of all triple-disk failures. A slightly larger storage overhead allows the HDComb code to achieve optimal small write costs, even shorter recovery equations, further reduced read load during recovery, and slightly better fault tolerance. The Chain code takes this tradeoff to the limit: for the storage overhead of two-fold replication, the Chain code achieves optimal small write costs, and other fantastic properties. Compared to the MDS code, the Chain code decreases recovery equation size 87%, per-disk read load 93%, and tolerates 99.6% of all triple-disk failures.

The comparison of these codes at $d = 4$ demonstrate similar tradeoffs. Though the HDComb and StepComb constructions for this specific value of k are identical.

The replication results for read load and failure are somewhat misleading because of the smaller stripe width. Consider if parity declustering [7] is used with replication to take advantage of 30 disks, the number of disks that the Chain code uses when $k = 15$. The read load for both two- and three-disk fault tolerant replication is then 0.034, even less than for the Chain code. Again considering 30 disks, it is possible to setup 10 distinct three-fold replication stripes. Such a layout only loses data in 0.15% of the possible three disk failures. Again, this is better than the Chain code. Note how the replica layouts for these two 30-disk comparisons differ: replication does not achieve both properties simultaneously like Chain code does.

VI. CONCLUSIONS

As the scale of storage systems increases, two- and three-disk fault tolerant protection, or more, is needed. There is a pronounced gap between the cost-benefit propositions of replication and other MDS codes such as RAID 6 in multi-disk fault tolerant systems. In such systems, flat XOR-codes offer cost-benefit tradeoffs that cover a large portion of the gap between replication and other MDS codes.

In this paper, we described a number of two- and three-disk fault tolerant non-MDS flat XOR-code constructions: Chain

codes, flattened parity-check array codes, Stepped Combination codes, and HD-Combination codes. Chain codes are based on previously known constructions, and flattening is a technique that applies to previously known codes. The combination codes are novel constructions. We expect additional flat XOR-code constructions will be discovered that fill in more of the tradeoff space between replication and other MDS codes.

We introduced recovery equations, recovery schedules and algorithms to determine such equations and schedules for flat XOR-codes. We expect that the initial algorithms we describe can be improved significantly.

A key property of flat XOR-codes is that they trade an increased storage overhead for shorter recovery equations which can be leveraged to create efficient recovery schedules. We analyzed key storage properties of these flat XOR-codes and MDS codes: storage overhead, small write cost, size of shortest recovery equation, recovery read load, and fault tolerance at the Hamming distance.

MDS codes achieve optimal storage overhead and small write costs. Recovery equations for such codes though increase in size with stripe width. The Stepped Combination code achieves minimal storage overhead for a flat XOR-code construction at a given fault tolerance. Whereas, the HD-Combination code achieves minimal storage overhead *and* optimal small write costs for a flat XOR-code construction at a given fault tolerance. The recovery read load is the fraction of a disk's worth of data each available disk in a stripe must read to recover a single failed disk. Combination codes, depending on stripe width and fault tolerance, have a recovery read load of between 0.2 and 0.5. These values are competitive with replication and much better than other MDS codes. Chain codes provide optimal small write costs and optimal shortest recovery equation size, but require storage overhead equivalent to two-fold replication. This storage overhead cost enables Chain codes to achieve a recovery read load that decreases with stripe width and so is better than replication and significantly better than other MDS codes.

REFERENCES

[1] M. Baker, M. A. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *EuroSys-2006: 1st EuroSys Conference*. ACM, April 2006, pp. 221–234.

[2] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *FAST-2007: 5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007, pp. 1–16.

[3] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *FAST-2007: 5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 289–300, 2007.

[5] J. Elerath, "Hard-disk drives: The good, the bad, and the ugly," *ACM Queue*, 2009.

[6] J. Menon and D. Mattson, "Distributed sparing in disk arrays," in *COMPCON '92: Proceedings of the thirty-seventh international conference on COMPCON*. IEEE Computer Society Press, 1992, pp. 410–421.

[7] M. Holland and G. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 23–25.

[8] D. Li and J. Wang, "EERAID: Energy Efficient Redundant and Inexpensive Disk array," in *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2004, p. 29.

[9] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning, "PARAID: A gear-shifting power-aware RAID," *Trans. Storage*, vol. 3, no. 3, p. 13, 2007.

[10] J. Wang, H. Zhu, and D. Li, "e-RAID: Conserving energy in conventional disk-based RAID system," in *IEEE Transactions on Computers*, 2008.

[11] K. Greenan, "Reliability and power-efficiency in erasure-coded storage systems," UC Santa Cruz, Tech. Rep. UCSC–SSRC–09–08, 2009.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles"*, October 2003, pp. 29–43.

[13] The Apache Software Foundation, "Hadoop File System (HDFS))," http://hadoop.apache.org/hdfs/.

[14] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *STOC-1997*. ACM Press, 1997, pp. 150–159.

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

[16] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *SIGCOMM '98*. ACM, 1998, pp. 56–67.

[17] "IEEE standard for local and metropolitan area networks part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications - section one," *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pp. c1–597, 2008.

[18] "IEEE standard for local and metropolitan area networks part 16: Air interface for broadband wireless access systems," *IEEE Std 802.16-2009 (Revision of IEEE Std 802.16-2004)*, pp. C1–2004, 2009.

[19] A. Wilner, "Multiple drive failure tolerant RAID system," United States Trademark and Patent Office, December 2001, patent number 6,327,627 B1.

[20] J. L. Hafner, "WEAVER Codes: Highly fault tolerant erasure codes for storage systems," in *FAST-2005*. USENIX Association, December 2005, pp. 212–224.

[21] ——, "HoVer erasure codes for disk arrays," in *DSN-2006: The International Conference on Dependable Systems and Networks*. IEEE, June 2006, pp. 217–226.

[22] A. Amer, J.-F. Pâris, and T. Schwarz, "Outshining mirrors: MTTDL of fixed-order spiral layouts," in *Storage Network Architecture and Parallel I/Os, 2007. SNAPI.International Workshop on*, Sept. 2007, pp. 11–16.

[23] J.-F. Pâris, T. J. E. Schwarz, and D. D. E. Long, "Self-adaptive two-dimensional raid arrays," in *International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, April 2007, pp. 246–253.

[24] 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*. IEEE, June 2004, pp. 115–124.

[25] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason, "Small parity-check erasure codes - exploration and observations," in *DSN-2005*. IEEE, July 2005.

[26] J. L. Hafner and K. Rao, "Notes on reliability models for non-MDS erasure codes," IBM, Tech. Rep. RJ–10391, October 2006.

[27] M. Woitaszek and H. M. Tufo, "Tornado codes for MAID archival storage," in *24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 221–226.

[28] J. J. Wylie and R. Swaminathan, "Determining fault tolerance of XOR-based erasure codes efficiently," in *DSN-2007*. IEEE, June 2007, pp. 206–215.

[29] K. M. Greenan, E. L. Miller, and J. J. Wylie, "Reliability of flat XOR-based erasure codes on heterogeneous devices," in *DSN-2008*. IEEE, June 2008.

[30] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[31] J. S. Plank, "The RAID-6 Liberation codes," in *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[32] ——, "The RAID-6 Liber8Tion code," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 242–251, 2009.

[33] C.-I. Park, "Efficient placement of parity and data to tolerate two disk failures in disk array systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 11, pp. 1177–1184, 1995.

[34] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, 1995.

[35] M. Blaum, J. Bruck, and A. Vardy, "MDS array codes with independent parity symbols," *Information Theory, IEEE Transactions on*, vol. 42, no. 2, pp. 529–542, Mar 1996.

[36] 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*. USENIX Association, March 2004, pp. 1–14.

[37] L. Xu and J. Bruck, "X-Code: MDS array codes with optimal encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, 1999.

[38] C. Huang and L. Xu, "Star: an efficient coding scheme for correcting triple storage node failures," in *FAST'05: Proceedings of the 4th conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–15.

[39] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-Code: a new RAID-6 code with optimal properties," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 360–369.

[40] M. Schwartz and A. Vardy, "On the stopping distance and the stopping redundancy of codes," *IEEE Trans. on Inf. Theory*, vol. 52, no. 3, pp. 922–932, 2006.

[41] H.-I. Hsiao and D. DeWitt, "Chained Declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of 6th International Data Engineering Conference*, 1990, pp. 456–465.

[42] A. Amer, J.-F. Pâris, D. Long, and T. Schwarz, "Progressive parity-based hardening of data stores," in *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International*, Dec. 2008, pp. 34–42.

[43] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, July 2007, pp. 79–86.

[44] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and K. Rao, "Performance metrics for erasure codes in storage systems," IBM, Tech. Rep. RJ–10321, 2004.

[45] 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*, June 1988, pp. 109–116.

[46] W. A. Burkhard and J. Menon, "Disk array storage system reliability," in *Symposium on Fault-Tolerant Computing*, 1993, pp. 432–441. [Online]. Available: citeseer.ist.psu.edu/burkhard93disk.html

[47] J. F. Elerath and M. Pecht, "Enhanced reliability modeling of raid storage systems," in *DSN-2007*. IEEE, June 2007, pp. 175–184.

[48] 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*. IEEE, June 2006, pp. 237–248.

[49] J. S. Plank, "Erasure codes for storage applications," Tutorial slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.

[50] P. Elias, "Error free coding," *IRE Trans. Inform. Theory*, vol. IT-44, pp. 29–37, Sept. 1954.

[51] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson, "Failure correction techniques for large disk arrays," in *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1989, pp. 123–132.

[52] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin, "Matrix methods for lost data reconstruction in erasure codes," in *FAST-2005*. USENIX Association, December 2005, pp. 183–196.

[53] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network coding: An instant primer," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 63–68, 2006.

[54] A. Dimakis, V. Prabhakaran, and K. Ramchandran, "Decentralized erasure codes for distributed networked storage," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2809–2816, June 2006.

[55] J. Luo, L. Xu, and J. S. Plank, "An efficient XOR-Scheduling algorithm for erasure codes encoding," in *DSN-2009: The International Conference on Dependable Systems and Networks*. Lisbon, Portugal: IEEE, June 2009.

[56] K. Greenan, E. L. Miller, and T. Schwarz, "Optimizing Galois Field arithmetic for diverse processor architectures," September 2008.

[57] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *NCA-2006: 5th IEEE International Symposium on Network Computing Applications*. IEEE, July 2006, pp. 173–180.

APPENDIX I
CORRECTNESS SKETCHES FOR COMBINATION CODES

Below we outline correctness sketches for the two- and three-disk fault tolerant Stepped Combination code constructions. These sketches also apply to the corresponding HD-Combination code constructions.

A. Stepped Combination - Hamming distance 3

Consider the following three scenarios of disk failures: (1) Two parity disks fail. In this case, since no data disks fail, all parity disks can be easily recovered. (2) One data disk fails and one parity disk fails. Since every data disk is connected to at least two parity disks, there exists a working parity disk that is connected to the failed data disk. Therefore, the failed data disk can be recovered, and then so can the failed parity disk. (3) Two data disks fail. Since no two data disks are connected to the same set of parity disks, there exists a parity disk that is connected to exactly one of the failed disks. Therefore, that disk can be recovered, and then so can the other one.

This construction is storage-efficient optimal in the sense that it achieves the maximum value of k. The reason is that in order to recover two disk failures, all data disks have to be connected to at least two parity disks and each data disk must be connected to a a unique combination of parity disks. The above construction includes all such possible connections.

B. Stepped Combination - Hamming distance 4

Again, we consider all the disk failure scenarios: (1) Three parity disks fail. Since no data disks fail, all parity disks can be easily recovered. (2) Two parity disks and one data disk fail. Since every data disk is connected to at least three parity disks, there exists a working parity disk that is connected to the failed data disk. Therefore, that data disk can be recovered, and then so can the two failed parity disks. (3) One parity disk and two data disks fail. Since the two sets of parity disks connected to any two data disks differ by at least two parity disks, there exists a working parity disk that is connected to exactly one of the failed data disk. Therefore, that data disk can be recovered, and then so can the other failed data disk and the failed parity disk. (4) Three data disks fail. Since all three data disks are connected to an odd number of parity disks, there exists a parity disk that is connected to exactly an odd number (i.e., one or three) of these three failed data disks. We further consider two cases: (4.1) Suppose no parity disks are connected to exactly one failed data disk. Then there exists a parity disk that is connected to all three failed data disks, and there exists another parity disk that is connected to exactly two of the three failed data disks (but not to the third one). In this case, the third data disk can be first recovered, and then so can the other two. (4.2) Suppose there exists a parity disk that is connected to exactly one of the failed data disks. In this case, that data disk can be recovered first, and then so can the other two.