Coding Techniques for Handling Failures in Large Disk Arrays[1]

Garth A. Gibson, Lisa Hellerstein, Richard M. Karp,
Randy H. Katz and David A. Patterson

Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract: *The ever increasing need for I/O bandwidth will be met with ever larger arrays of disks. These arrays require redundancy to protect against data loss. This paper examines alternative choices for encodings, or codes, that reliably store information in disk arrays. Codes are selected to maximize mean time to data loss or minimize disks containing redundant data, but are all constrained to minimize performance penalties associated with updating information or recovering from catastrophic disk failures. We show codes that give highly reliable data storage with low redundant data overhead for arrays of 1000 information disks.*

With the proliferating processing power provided by advanced VLSI technology and parallel architectures comes an inflating demand for Input/Output (I/O) performance. The mainstay of online secondary storage, the magnetic disk, is providing neither the data rates required for applications that process large amounts of sequential data nor the access rates required for applications that process large numbers of random accesses [Boral83]. This widening gap has led to I/O systems that achieve performance through disk parallelism, using such techniques as disk striping (also known as disk interleaving) for higher data rates and data distributing for greater access rates [Kim85, Kim87, Klietz88, Livny87, Park86, Salem86].

Even though disk performance has not kept pace with processor performance, magnetic disk technology has not been idle; densities have been growing exponentially and physical packaging has achieved amazing volume reductions. The 5.25-inch form-factor drives may currently have the best cost per megabyte, and the 3.5-inch form-factor is expected to overtake it soon [Vasudeva88]. This trend has led some to explore the replacement of individual large form-factor drives with many smaller form-factor drives [Jilke86, Maginnis87, Patterson88], giving yet another reason to expect that future I/O systems will contain large

---

[1] A subset of this paper (UCB CSD 88-477) will appear in the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), Boston March 1989. The authors may be reached electronically at gibson@ernie.Berkeley.EDU.

numbers of disks.

While performance improves with increasing numbers of disks, the catch is that the chance of data loss also increases. A simple model for device lifetime, used by the disk industry [Murnan87] and for electronics in general [Siewiorek82], is an exponential random variable. In this model the rate of failures in an I/O system is directly proportional to the number of disks; even with disks 10 times as reliable as the best on the market today, the first unrecoverable failure in a non-redundant array of a thousand disks can be expected in less than two weeks. Such high rates of data loss impel the inclusion of data redundancy to allow information to survive hardware failures.

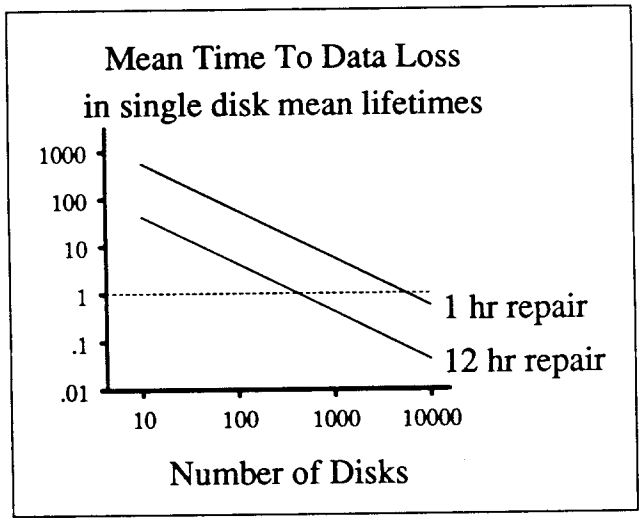

Figure 1. *The mean time to data loss in a single-erasure-correcting array is* $MTTDL = (MTTF_{disk})^2 / (\#Disks \times (Groupsize - 1) \times MTTR)$, *where* $MTTF_{disk}$ *is the mean lifetime of an individual disk, #Disks is the total number of disks, Groupsize is 1 + the number of information (user data) disks associated with each redundant data disk, and MTTR is the mean time required to repair and reconstruct a failed disk [Patterson88]. In this figure we show MTTDL in terms of the number of individual disk's mean lifetimes expected to pass before the array suffers an unrecoverable failure. We assume highly reliable disks* ($MTTF_{disk}$ *= 50,000 hours), and 10% as many redundant data disks as information disks (Groupsize = 11). Two values for mean repair and reconstruction time are shown; although 1 hour repair is feasible, it requires online "hot spare" disks or continuous human maintenance. In a simpler scheme, repair is carried out during daily visits by maintenance personnel. In this case mean time to repair would be 12 hours.*

Today's magnetic disk drives suffer from three primary types of failures. *Transient* or noise-related errors are corrected by repeating the offending operation or by applying per sector error-correction facilities. *Media defects* are permanent errors usually detected and masked at the factory. For very large arrays, we are interested in *catastrophic* failures: head crashes and read/write or controller electronics failures. Controller microprocessors detect mechanical failures and the complex host/controller ("thin-wire") protocol ensures that failed controllers are identifiable. These self-identifying catastrophic failures are called *erasures* in coding theory to distinguish them from arbitrary data changes, which are called errors [Berlekamp68]. This paper is concerned with techniques for erasure recovery.

We are primarily concerned with avoiding loss of user data. Disk drive failures are but one way that user data loss occurs. Power, cabling, memory, and processor failures can also cause data loss. In a related paper, we examine the effect of including disk drive support hardware failures on data reliability and find it to have a large effect [Schulze89]. To make matters worse, it has been reported that the dominant cause of data loss is incorrect or misused software [Gray85]. Our purpose in this paper is not to resolve all causes of data loss, but to make sure that, as the number of disk drives per system grows, the factor limiting data reliability is not independent, catastrophic disk failures.

Figure 1 describes an estimate for the mean time to data loss (MTTDL) in single-erasure-correcting I/O systems suffering catastrophic disk failures [Patterson88]. We can see that as the number of disks soars, reliability plummets; even with single-erasure-correction, an I/O system of more than a thousand disks is only a fraction as reliable as a single disk! The goal of this paper is to make such large arrays at least as reliable as an individual disk.

This paper explores more powerful encodings for redundancy in large disk arrays while seeking high reliability at low cost. It emphasizes double- and triple-erasure-correcting approaches. We show the implementation of codes that have very high reliability, as measured by the mean time to data loss, and also obtain excellent trade-offs among the following cost and performance metrics: the fraction of disks that contain redundant data, the number of disks that must be accessed when data is updated, and the number of disks that

must be accessed when lost data is reconstructed. In particular we show codes that improve mean time to data loss by multiple orders of magnitude and require less than 10% of disks to be used for redundant data.

Our paper begins with a discussion of practical metrics for redundancy in large disk arrays. We then describe an introductory family of codes for correcting multiple erasures. To probe further, we recall the matrix formulation of error-correcting code (ECC) theory and show how erasure-correcting codes can be implemented in disk arrays. Then we motivate and describe best possible codes for our implementation metrics. Finally we discuss our results and conclude with an indication of future directions.

1. Metrics for Redundancy in Disk Arrays.

There is a variety of schemes for encoding redundant information into a collection of disks. To avoid read performance penalties associated with decoding when there are no failures, we restrict ourselves to schemes that leave the original data unmodified on some disks and define a redundant encoding for that data on other disks. We call the former, *information* disks and the latter, *check* disks. Calculations used to form the data on check disks are restricted to modulo 2 arithmetic; that is, parity[2] operations. This ensures that

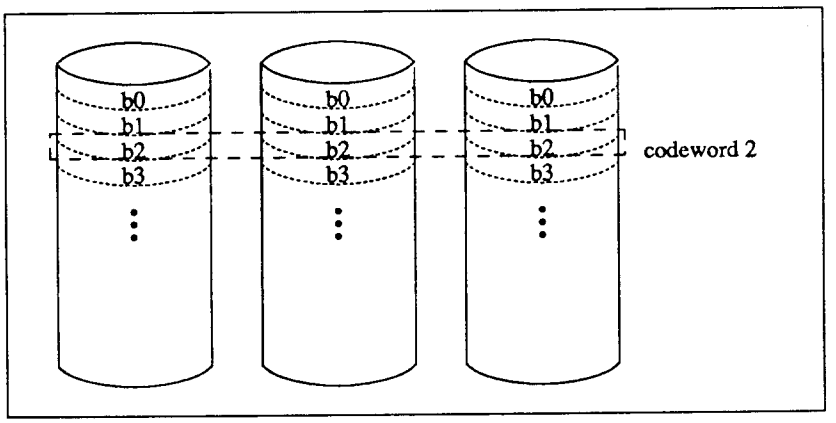

Figure 2. *A codeword consists of the set of bits at the same position on each disk. A code determines which bits in a codeword are check bits and how to calculate these from the other information bits.*

[2] Parity is the exclusive OR of a set of bits.

check data can be quickly and simply manipulated.

We view a disk as a stack of bits as shown in Figure 2. The jth bit from each disk forms the jth codeword in the redundancy encoding. Reconstruction of lost data is logically one codeword at a time, although in practice a block of codewords is processed in parallel. For simplicity we will discuss only one of the codewords and refer to the disks as bits in this single codeword.

Because non-redundant disk array unreliability induces our need for redundancy, the *mean time to data loss* (MTTDL) is a primary metric for the choice of a code. Associated with each code are three further important metrics: check disk overhead, update penalty and group size.

The *check disk overhead* for a code is the ratio of the number of check disks to information disks.

The *update penalty* of a code is the number of check disks whose contents must be changed when the contents of a given information disk is changed. The update penalty is an important metric because redundant updates are the dominant performance penalty in the choice of a code and must be minimized.

The information and check disks that must be accessed during the reconstruction of a failed disk form a group. The *group size* is an important metric because, in very large arrays, individual disk failure will be frequent enough that highly available systems must continue operation during repair and reconstruction. In such systems, throughput requirements will allow reconstruction only a fraction of the total I/O bandwidth; thus, the duration of reconstruction depends on the group size. In addition to affecting performance, the longer repair of larger groups increases the "window of vulnerability" to other failures, decreasing reliability.

Finally, pragmatism requires that users be allowed to add new disks to their array. These additional disks should be accommodated without a complete recalculation of all check data. Such a recalculation would induce a huge strain on bandwidth and a large window of vulnerability to unrecoverable failures each time an array is expanded.

2. A First Example: Parity in N Dimensions.

So that later implementation discussions and our advanced codes will be more explicable, we begin with an example. In [Patterson88], a single-erasure-correction scheme was described in which every set of G information disks was associated with a single check disk that contained the parity of all G information disks. In Figure 3(a) we show these $G+1$ disks (for $G=4$) in a row with their parity in the disk on the right side. This code has an overhead of $1/G$, since each group has one check disk and G information disks, an update penalty of 1, since one check disk update is needed for every information disk update, and has a group size of $G+1$, since $G+1$ disks are involved in the reconstruction of any failure. Two failures in any one group leads to lost data. The reliability of this scheme, which we call *1d-parity*, is given in Figure 1.

A simple extension of 1d-parity, called *2d-parity*, is shown in Figure 3(b) (for G=4). A set of G^2 information disks are arranged in a two dimensional array. On the end of each row and column a check disk stores parity for that row or column. Since a failed disk belongs to two groups – its row and its column, – it can be reconstructed from the data of either; thus, 2d-parity is double-erasure-correcting. This code has a check disk overhead of $2G/G^2 = 2/G$, an update penalty of 2, since it requires 2 check disk updates for every

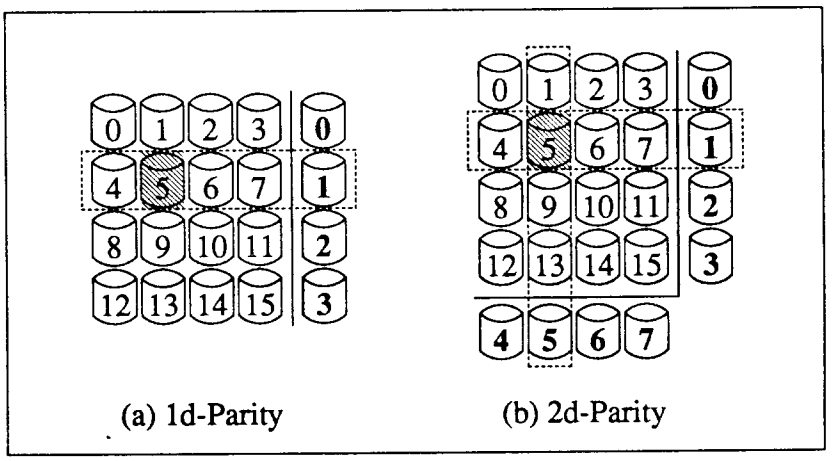

Figure 3. *With parity computed only along rows of disks we have a single-erasure-correcting code called 1d-parity, and with parity computed both along rows and along columns we have a double-erasure-correcting code called 2d-parity. In this example we have used groups with 4 information disks and marked the groups that contain disk 5.*

information disk update, and a group size of $G+1$, since $G+1$ disks are involved in the reconstruction of any single failure. We will discuss the extensibility and reliability of the 2d-parity and higher dimensional codes in Sections 6, and 7.

The 1d-parity and 2d-parity codes extend to N-erasure-correction by logically arranging G^N information disks in an N-dimensional array and recording parity on a disk at the end of each dimensional group. In these codes each information disk belongs to N groups, and the updates penalty is N. These codes have a check disk overhead of $NG^{(N-1)}/G^N = N/G$, since there are $NG^{(N-1)}$ check disks and G^N information disks. However, the group size remains $G+1$ because only $G+1$ disks are involved in the reconstruction of any single failure. We do not envision practical disk arrays large enough to require N greater than 2 or 3, because of the large check disk update penalty paid on every information disk update.

N-dimensional parity, Nd-parity, is very similar to a technique, called *iterated codes*, that forms the basis of error-correcting codes that have been commonly used in magnetic tape systems for three decades [Peterson61]. Although Nd-parity has an easily visualized structure, it is not necessarily the best code for our metrics. Before introducing better codes we will revisit error-correction code theory [Peterson61, Berlekamp68] for a convenient framework[3] that will allow us to describe and compare codes.

3. Linear Codes.

Linear codes contain the original information unmodified within each codeword, and compute the check bits (disks) of each codeword as the *parity* of subsets of the information bits. These codes can be defined in terms of a $c \times (k+c)$ *parity check matrix*, $H = [P \mid I]$, shown in Figure 4, where c is the number of check bits, k is the number of information bits, I is the $c \times c$ identity matrix and P is a $c \times k$ matrix that determines the equations for the check disks. In Figure 5 we show a parity check matrix for the 1d-parity and 2d-parity codes described in the previous section.

[3] Error-correction theory is built on linear algebra. A good general textbook on linear algebra is [Friedberg79].

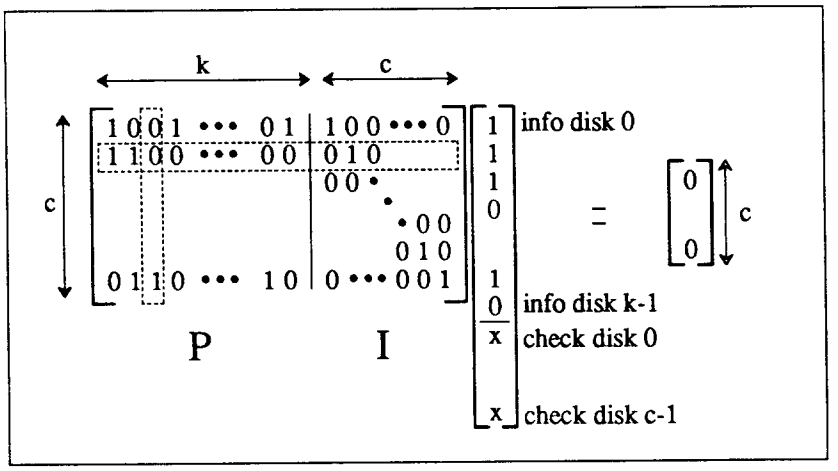

Figure 4. *The parity check matrix, $H = [P \mid I]$, for a linear code with k information disks and c check disks is composed of a sub-matrix, P, and an identity sub-matrix, I. A column of P represents an information disk and a column of I represents a check disk. A row represents a check group; each information disk (column) that has a one in a given row contributes to the check disk (column) for that row. When H is matrix multiplied by a codeword (one bit from every disk) the result is required to be a vector of zeros. This defines the value of the check bits in this codeword.*

The k columns of P represent the k information disks and the c columns of I represent the c check disks. Each row in the parity check matrix, H, corresponds to a group; the columns that have ones in a row correspond to the disks in that group. If the vector X represents a codeword (one bit from all disks in the same order as the columns of H), then, using modulo 2 arithmetic, the system of linear equations $HX = 0$ is another way of saying that each check disk contains the parity of its group's information disks.

Any parity check matrix, H, has four equivalent properties expressed in terms of a parameter, t, whose value is between 0 and c. [Peterson61] (Theorem 1; we include a proof in the appendix for the reader's convenience). (1). H will allow any t erasures to be corrected. (2). H will allow any t errors (arbitrary changes in t disks' values) to be detected.[4] (3). The minimum number of bits in which any two codewords differ, known as the *distance* of the code, is at least $t+1$. (4). Any set of t columns selected from H will be *linearly independent*. In modulo 2 arithmetic on vectors, linear independence means that the element-by-element exclusive OR of a set t columns, or any subset of these t columns, will not result in a vector of zeros. In fact,

[4] If we were interested in correcting errors rather than erasures, this H will allow any $\lfloor t/2 \rfloor$ errors to be corrected.

whether any set of disk failures can be repaired and reconstructed depends on whether the corresponding set of columns in H is linearly independent (see the next section). The equivalence of t-erasure-correcting and t-error-detecting means that we could borrow any code used in memory systems; however, many of these codes will not perform well for disk arrays because they will generally have large update penalties.

Three of our metrics from Section 1 are easily expressed in terms of parity check matrices. The check disk overhead is c/k, the ratio of the number of check disks to information disks. The size of a group, which determines performance degradation during reconstruction, is the number of ones in the row for that group. The number of ones in any vector is known as the weight of that vector, so a group's size is the weight of its row.

The update penalty for any information disk, which is the number of groups including that disk, is the weight of its column. Because the update penalty is the dominant performance cost in our redundant disk array, we restrict our attention to codes that minimize it. Since any code that corrects t erasures must leave

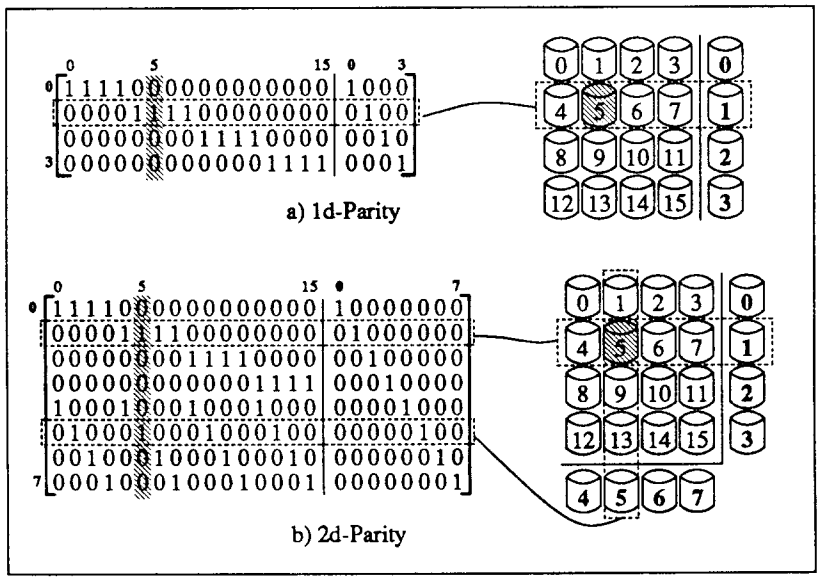

Figure 5. *We contrast the parity check matrices for 1d-parity and 2d-parity to the physical models given in Section 2 for groups of 4 information disks. In both cases disk 5 is shaded so you can see the column that represents it and the rows that represent the groups that contain it.*

evidence of every write on at least t different check disks, its minimum update penalty is t. When searching for a good t-erasure-correcting code, we demand that the column corresponding to each information disk have weight t. The Nd-parity code given in Section 2 has this property.

A nice property of linear codes is that extending an array with new information disks can be done without performance penalty. If a new disk's contents are all zeros, then adding it to a group will not change the parity of that group. This means that a new column can be added to P when a new, zeroed disk is brought online, and no recalculation of check disks is required. In practice the columns of P are constrained by the properties of H we mentioned above; when an I/O system is first installed, the matrix P should be picked with many more columns than are needed, and the extra columns should be reserved until new disks are installed.

Unfortunately, our measure of reliability, the mean time to data loss, *MTTDL*, is not easily calculated from the parity check matrix. It depends on the way disks fail and are repaired. For our calculations, we assume that disk lifetimes are identical independent exponential random variables and that repair is done periodically. With this model we can estimate *MTTDL* as the expected number of repair periods until an unrecoverable set of failures occur. Although a t-erasure-correcting code certainly recovers all t or fewer failed disks, it will also be able to recover some of the larger sets of failures. For example, in a 1d-parity array, 1 disk in every group can fail recoverably (for the example in Figure 3, up to 4 disks can be recovered if they fail one to a group). We use Monte Carlo simulation [Rubinstein81] on subsets of columns from H to estimate the probability that any particular size subset is unrecoverable. In Section 7, we apply this technique to evaluate the mean time to data loss for the Nd-parity codes and codes to be discussed in Section 5.

4. Implementing Reconstruction.

We turn our attention to the process of reconstructing lost data onto a repaired or replaced disk. Paralleling this discussion we present a specific example in Figures 6(a), 6(b), and 6(c). Recall that $HX = 0$ is satisfied when no disk has failed and X is a vector containing a bit from each disk. If m disks fail then the columns of H and the entries of X are divided into two types: those that do and those that do not represent

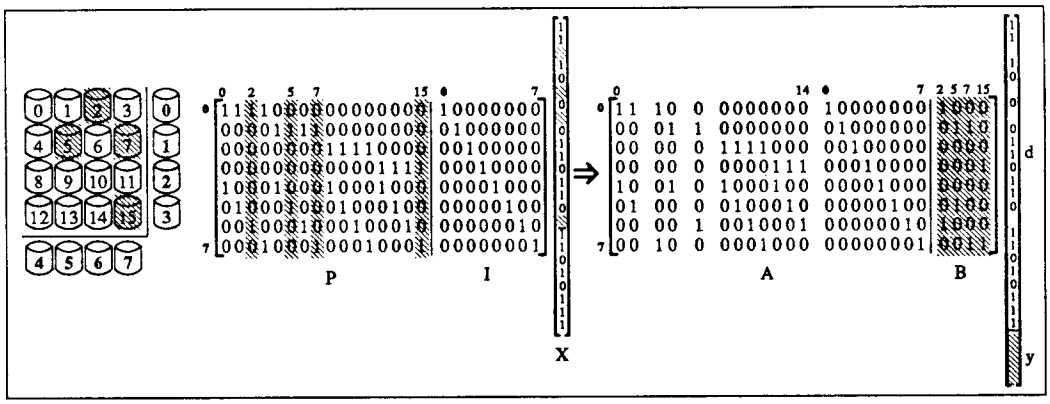

Figure 6(a). *When 4 disks fail in a 16 information disk 2d-parity array, the controllers allow us to identify which disks need to be repaired and reconstructed. In this example, information disks 2, 5, 7 and 15 have failed and are shaded. $H = [P \mid I]$ and any codeword, X, are then rearranged into $[A \mid B]$ and $[d \mid y]$ to isolate the erased bits into y. Notice that more disks have failed in this example than are guaranteed recoverable (2). The successful recovery of such sets of failures is important to a high MTTDL.*

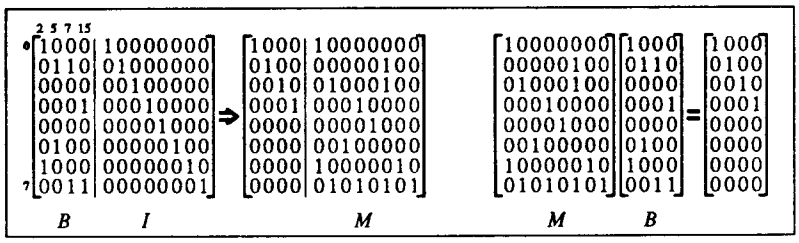

Figure 6(b). *When the columns of H that represent failed disks are grouped into sub-matrix B we can apply "elementary row operations" (the essence of Gaussian elimination) to find a matrix, M, such that the product MB has the 4x4 identity matrix in its first 4 rows. The row operations that were performed to get M were (in order): add row 0 into row 6, add row 1 into row 5, exchange row 5 and row 2, add row 2 into row 1, add row 2 into row 7, and add row 3 into row 7.*

disks that have failed. As we have done for our example in Figure 6(a), if we rearrange the columns of H and the entries of X so that the failures are on the right and the functional disks are on the left, then $H = [A \mid B]$ and $X = [d \mid y]$. The lost data are the m entries of y, and the columns of H representing these disks are in B. We must determine the value of y such that $HX = Ad + By = 0$. This means that we must solve the c equations in m unknowns described by $Ad = By$, because addition and subtraction give the same result in modulo 2 arithmetic.

If the code for parity check matrix H is t-erasure-correcting and the number of failed disks, m, is less than or equal to the number guaranteed recoverable, t, then B is a subset of at most t columns from H. Since

H is t-erasure-correcting, all subsets of at most t columns are linearly independent. This means that there exists a c x c matrix, M, that is determined by Gaussian elimination[5] on B, such that the first m rows of the product MB are the mxm identity matrix. In this case the first m entries of the product MAd are precisely the failed disks' data. In Figure 6(b) we show M for our example, and in Figure 6(c) we show MAd for our example.

Moreover, if the number of failures, m, is greater than the number guaranteed recoverable, t, as in our example, then some, but not all, sets of failures are recoverable. A set of $m > t$ failures is recoverable if and only if a matrix M can be found by Gaussian elimination. For more failures than check disks, ($m > c$), there is never a way to reconstruct all lost data.

Codes with t-erasure-correction can be implemented in software that runs in an I/O processor. This processor may need some hardware support for fast exclusive OR on blocks of data, but can otherwise be a traditional microprocessor. Software learns of failures directly from disk controllers or by lack of response from disk controllers. This identifies our variables A, B, d and y, where d and y are placeholders for each of the

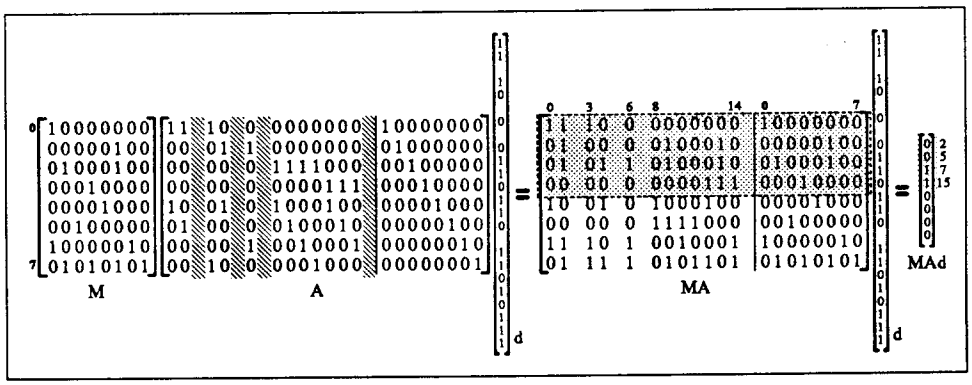

Figure 6(c). *The first 4 rows of MA describe the operations that must be performed to reconstruct our 4 disks. Disk 2 is the parity of 0, 1, 3 and check disk 0. Disk 5 is the parity of 1, 9, 13 and check disk 5. Disk 7 is the parity of 1, 4, 6, 9, 13 and check disks 1 and 5. And disk 15 is the parity of 12, 13, 14 and check disk 3. For the example codeword, X = [d | y], in Figure 6(a), we see from the first four entries in MAd that bit 2 is 0, bit 5 is 0, bit 7 is 1 and bit 15 is 1. These parity calculations must be performed on every codeword of all involved disks, but they can occur in parallel as bandwidth allows.*

[5] For our purposes, Gaussian elimination is not an expensive algorithm because it is applied to relatively small arrays describing the failures, and because operations are modulo 2.

billions of codewords that make up the disks' data. An algorithm based on Gaussian elimination determines if all failed disks can be reconstructed, and if so, what M should be used. It then computes MA as we show for our example in Figure 6(c).

Once MA is known, we can determine the set of functional disks that must be read to reconstruct lost data. The failed disk represented by the first position of y is reconstructed as the exclusive OR of every disk whose column in MA has a one in the first row. Similarly the second entry of y is reconstructed as the exclusive OR of the disks whose columns in MA have ones in the second row, and so on. In Figure 6(c) we demonstrate this process for our example failure set. The set of functional disks required to reconstruct lost data is read in parallel, large blocks at a time, and the reconstructed data are written either to repaired or replaced disks.

5. Double-Erasure- and Triple-Erasure-Correcting Codes.

Now that we have the linear code framework, we can ask: what are the codes that behave better than Nd-parity under our metrics? Within the class of 2- and 3-erasure-correcting codes with minimum possible update penalty (the information columns of H have weight 2 or 3, respectively) we will describe four codes, each the best possible, or nearly so, with respect to either check disk overhead or mean time to data loss. The four codes are: the *full-2* code, the already introduced *2d-parity* code, the *full-3* code, and the *additive-3* code. Theorems referenced in the following section are presented in the appendix. Examples of the parity check matrices for these codes are found in Figures 7 and 8. A fifth code, the *steiner* code, of theoretical interest only, is also mentioned.

Full-2: The full-2 code is a 2-erasure-correcting code (Theorem 2). It can be defined in terms of its parity check matrix $H_{full2}=[P_{full2}|I]$. P_{full2} consists of all possible distinct columns of weight 2. Codewords in the full-2 code consist of c check bits and $\binom{c}{2} = c(c-1)/2$ information bits.

2d-parity: The 2d-parity code is the 2-erasure-correcting code (Theorem 3) described in Section 2. It has an even number, c, of check bits and $c^2/4$ information bits.

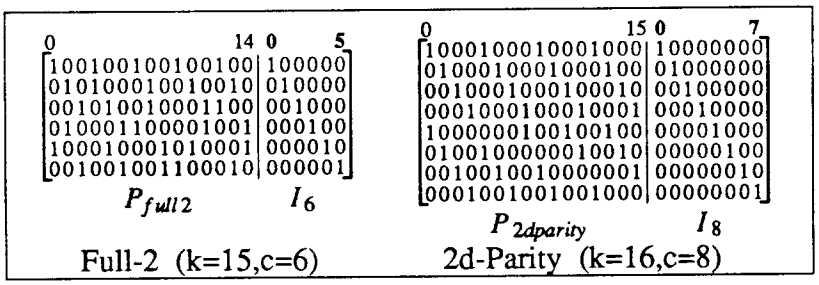

Figure 7. *Example parity check matrices for the two 2-erasure-correcting codes discussed in this section are shown in this figure. Notice that although the full-2 code has 2 fewer check disks it only has 1 fewer information disk, so its check disk overhead, 6/15, is less than 2d-parity's, 8/16. You may notice that the order of the columns in 2d-parity differs from the example in Figures 5 and 6; the order of columns in these examples is explained in Section 6.*

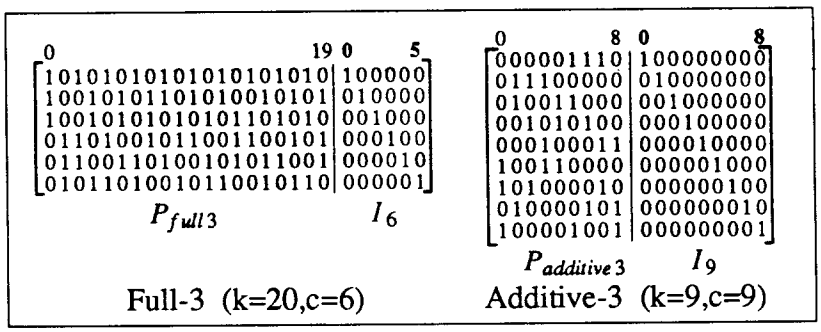

Figure 8. *Example parity check matrices for the two 3-erasure-correcting codes discussed in this section are shown in this figure.*

Full-3: The full-3 code is a 3-erasure-correcting code (Theorem 4) similar to the full-2 code. P_{full3} consists of all possible distinct columns of weight 3. Codewords in the full-3 code contain c check bits and $\binom{c}{3} = c(c-1)(c-2)/6$ information bits.[6]

Additive-3: The additive-3 code is a 3-erasure-correcting code (Theorem 5). Choose c, the number of check bits, to be an odd multiple of 3. Number the rows of $P_{additive3}$ from 0 to $c-1$. $P_{additive3}$ consists of all distinct weight 3 columns that contain 1's in rows q, r, and s, where $q + r + s \equiv 1 \bmod c$. The number of information bits is $c(\frac{c-1}{2} - 1)/3$ (Theorem 11).

[6] If you extend the full-2 and full-3 codes to a full-N code that includes all distinct columns of weight N, then the resulting code is not N-erasure-correcting for N ≥ 4, because there are linearly dependent sets of 4 columns of weight 4.

The full-2 and the full-3 codes are the best possible with respect to check disk overhead. *For a given number of check bits, the full-2 code has the highest number of information bits of any 2-erasure-correcting code with minimum possible update penalty. Similarly, for a given number of check bits, the full-3 code has the highest number of information bits possible for a 3-erasure-correcting code with minimum possible update penalty* (Theorem 8).

The 2d-parity code and the additive-3 code have excellent reliability properties. Because the 2d-parity code is a 2-erasure-correcting code, any set of 2-erasures can be corrected. In determining the overall reliability of the 2d-parity code, the fraction of sets of 3-erasures that can be corrected is a major contributor to high reliability. It is impossible to correct all sets of 3-erasures in a 2-erasure-correcting code with minimum possible update penalty (Theorem 6). In particular, because every information bit is associated with two check bits, it is impossible to correct the set of 3-erasures consisting of an information bit and its two check bits. Call any set of 3-erasures that involve an information bit and its two associated check bits a *bad* 3-erasure. A 2-erasure-correcting code with minimum possible update penalty will be unable to correct any bad 3-erasure. The 2d-parity code has the property that it corrects all sets of 3-erasures except bad 3-erasures. Moreover, no code with lower disk overhead can make this claim; i.e. *for a given number of check bits, if a 2-erasure-correcting code has minimum possible update penalty, and corrects all sets of 3-erasures except bad 3-erasures, then it cannot have more information bits than the 2d-parity code* (Theorem 9).

The additive-3 code can correct all sets of 4-erasures except bad 4-erasures (an information bit and its 3 associated check bits) (Theorem 7). *For a given number, c, of check bits, if a 3-erasure-correcting code has minimum possible update penalty, and corrects all sets of 4-erasures except bad 4-erasures, then it can have at most $c(c-1)/6$ information bits* (Theorem 10). This means that the additive-3 code's check disk overhead, $6/(c-3)$, is vanishingly larger than optimal, $6/(c-1)$, as disk arrays get larger.

In the very restrictive case where c is a power of 3, there exists a code with properties like the additive-3 code that achieves minimal check disk overhead (Theorem 12). The construction of this code is based on a class of Steiner triple systems (see the proof in Theorem 12 for an explanation). We call this code the *steiner*

code. Because this code applies only when the number of check disks is a power of 3, and because the additive-3 code has nearly as good check disk overhead, we do not think the steiner code has much practical usefulness.

6. Controlling Groupsize with Additional Overhead.

A major disadvantage of the full-2 and full-3 codes is that they have extremely large group sizes. In fact, large group size is an inevitable result of low disk overhead. Any t-erasure-correcting code with minimum possible update penalty, k information bits, and c check bits, has average group size $(tk+c)/c$, because the parity check matrix $H=[P \mid I]$ has t 1's in each column of the $c \times k$ matrix, P, and one 1 in each column of the $c \times c$ matrix, I. Therefore, if k is large relative to c, the check disk overhead (c/k) will be small, but the average group size will be large. The inherent trade-off between check disk overhead and average group size is expressed by the following inequality: $\textit{check disk overhead} \times (\textit{average group size} - 1) \geq t$. Fortunately, it is possible to derive new codes from the full-2 and full-3 codes which will allow us to trade disk overhead for smaller group sizes.

Given any linear code, we obtain a new linear code by simply deleting some of its information bits; by deleting the corresponding information columns from the original code's parity check matrix, we obtain the new code's parity check matrix. If the original code was t-erasure-correcting, then the new code will also have this property. To see this, notice that any set of columns from the new code's parity check matrix is also a set of columns of the original code's parity check matrix. Since the original code is t-erasure-correcting, all sets of up to t of its columns are linearly independent; therefore, all sets of up to t of the new code's columns are also linearly independent, and the new code is t-erasure-correcting.

We can use this fact to design t-erasure-correcting codes that trade disk overhead for smaller group sizes. Since the new code has fewer information columns, it has higher check disk overhead, but it also has lower *average* group size. However, it does not necessarily have lower maximum group size. We would like to be able to choose the information columns to be deleted in such a way that the maximum group size of the new code is close to its average group size.

Let us say that a parity check matrix and its associated code have *balanced group size* if the following hold: when the average group size is an integer, all groups are the same size; when the average group size is not an integer, the maximum group size is one greater than the minimum group size. The four codes discussed in the last section all have balanced group size. Moreover, if the number of check disks, c, in the full-2 code is even, then it is possible to arrange the columns of P_{full2} so that for any k (less than the number of columns in P_{full2}), a new code with information columns identical to the first k columns of P_{full2} will have balanced group size (Theorem 12). We call such an ordering of the columns of a parity check matrix a *balanced ordering*.

It is also possible to achieve a balanced ordering of the columns of P_{full3} (when c is divisible by 3) and $P_{2dparity}$ (when c is divisible by 2) (Theorem 12 and 13). Thus codes with balanced group size can also be derived from the full-3 code and the 2d-parity code in the manner described above. We show balanced orderings for the full-2, 2d-parity, and full-3 codes in Figures 7 and 8. We can prove that there does not exist a balanced ordering for the columns of $P_{additive3}$ (Theorem 13). However, it is possible to find an ordering which is almost balanced.

7. Discussion.

As we mentioned in Figure 1, a simple implementation for repair is periodic visits (for example, daily or weekly) by maintenance personnel.[7] With this model the mean time to data loss (MTTDL) is the expected number of repair periods until an unrecoverable set of failures occurs. Using exponential disk lifetimes we can calculate the probability of y failures in a repair period. We have used Monte Carlo simulation on the columns of a code's parity check matrix to estimate the fraction of y failures that are unrecoverable (linearly dependent) in each of our sample codes. The probability of an unrecoverable set of failures occurring in an given repair period is then the summation, over all y, of the probability that exactly y failures occur times the

[7] As this section shows, high reliability does not require immediate, automatic reconstruction to idle "hot spare" disks. However, if a failed disk is the target of frequent accesses then the performance degradation of reconstructing each request's data until the next maintenance personnel visit may justify the more complex automatic approach.

Code	Update Penalty	Check Disk Overhead	Group Size	MTTDL (years) Daily Repair	MTTDL (years) Weekly Repair
1d-parity	1	10.0%	10	2.1	0.3
full-2	2	4.4%	46	1,500	28
2d-parity	2	6.3%	33	25,000	260
3d-parity	3	30.0%	10	130,000	180
full-3	3	1.8%	172	15,000	38
additive-3	3	7.7%	40	$> 10^6$	$> 10^4$

Table 1: Comparing Codes for an Array of about 1000 Information Disks

This table contrasts the codes described in Sections 2 and 5 when these codes are applied to an array of about 1000 information disks. The update penalty is the number of additional accesses to check disks that accompany each information disk write. The check disk overhead is the ratio of the number of check disks to information disks. The group size is the number of disks that must be accessed to reconstruct a single failed disk. The mean time to data loss (MTTDL) is based on a periodic repair model and on Monte Carlo simulation of the probability that the set of failures at the end of the repair period is unrecoverable.

probability (fraction) that a set of y failures is unrecoverable. The mean number of repair periods until data loss is just the reciprocal of the probability that an unrecoverable failure occurs in a single repair period.

For our simulation we use codes that have close to 1000 information disks and we assume highly reliable disks; disks with a mean time to failure of 50,000 hours or about 5.7 years. Our results are shown in Table 1.

First, these results reaffirm our introductory comments about single-erasure-correction: the 1d-parity code is less reliable than a single disk, even if repair is daily. So we turn to double-erasure-correcting codes and pay the additional check disk update penalty on every write. The 2d-parity code has many times the reliability of a single disk, even if repair is weekly. The full-2 code does better in overhead and still has very good reliability.

The MTTDL for the additive-3 code is much higher than all other codes because it has so few unrecoverable failure sets. There is no chance of data loss until at least 4 failures occur in one repair period, but then only about 1 in 10^9 4-failures is unrecoverable. Even if 4-failures occurred every day, the mean time to data loss would exceed 10^6 years. The steiner code shares this very high reliability.

Even though 3d-parity has a much higher MTTDL than 2d-parity with daily repair, it has lower MTTDL with weekly repair. This happens because, for short repair periods, the unrecoverable 3-erasures in 2d-parity dominate, but, as the repair period gets longer, the fraction of unrecoverable 4-erasures, 5-erasures, etc., dominate. And 2d-parity has a lower fraction of unrecoverable N-erasures than 3d-parity for all N greater than 3. The same effect causes 2d-parity to have higher MTTDL than full-3 even with daily repair.

For arrays of about 1000 disks it is not necessary to pay the cost of a third check disk update penalty on every write. However, if very much larger arrays are considered, then eventually triple-erasure-correction may be necessary. Regardless of immediate need, triple-erasure-correcting codes dramatically demonstrate the differences in our codes. The 3d-parity is very expensive in overhead, the additive-3 code has phenomenal reliability with reasonable overhead, and the full-3 code has very low overhead yet retains very good reliability.

As we noted in the introduction to this paper, a serious source of data loss in I/O subsystems is the support hardware: power supplies, cooling, cabling and host memory ports. In a related paper we observe that this support hardware is likely to be shared among a subset of the disks. We then show that if parity groups are organized orthogonally to support hardware groups, data redundancy provides protection against support hardware failures as well as catastrophic disk failures [Schulze89]. This technique is easily extended for the 2d-parity code by organizing support hardware groups on the diagonals of the 2d array.

8. Conclusion.

Arrays of disks are a promising solution to the increasing demand for I/O bandwidth and access parallelism. If high reliability is to be preserved as the size of these arrays grows, redundancy encodings may be required to guarantee correction of double and perhaps triple failures.

This paper has explored the choice and implementation of redundancy codes for the practical constraints of disk arrays. We summarize the characteristics of our codes in Table 2. Our codes all minimize the number of check disks that must be updated whenever an information disk is updated. Beyond this requirement we

Code	Erasures Corrected = Update Penalty	Check Disk Overhead (c check disks)	High MTTDL?	Distinguishing Feature
1d-parity	1	$1/Groupsize$	No	Low Update Cost
2d-parity	2	$4/c$	Yes	Few Triple Failures
full-2	2	$2/(c-1)$	Yes	Low Overhead
3d-parity	3	$5.2/\sqrt{c}$	Very	None
full-3	3	$6/(c^2-3c+2)$	Yes	Low Overhead
additive-3	3	$6/(c-3)$	Extremely	Few Quadruple Failures
steiner	3	$6/(c-1)$	Extremely	c Values Restricted

Table 2: Characteristics of Erasure Correcting Codes Discussed.

have explored codes that minimize check disk overhead. The reliability of our double-erasure-correcting codes for arrays of about 1000 information disks is so good that triple-erasure-correction is unnecessary.

9. Future Work

We are interested in further questions about the reconstruction process: If online "hot spares" are included and reconstruction is done immediately on failure, how do reliability and I/O bandwidth degradation interact?

Updates to check disks are likely to be much more frequent than updates to information disks, so to avoid bottlenecks, we need to explore distributing the check data across all disks, and, in general, search for ways to reduce the cost of writes.

We are in the process of constructing a prototype disk array that will allow us to implement and evaluate the performance of various codes under various workloads.

10. Acknowledgements.

We would like to acknowledge the people whose comments about drafts of this paper greatly contributed: Peter Chen, Fred Douglis, Susan Eggers, Mark Hill, Corinna Lee, Ken Lutz, John Ousterhout, and Martin Schulze. We are also indebted to the mathematicians, who, over two decades ago, laid the foundation of error-correcting codes. The work described here was supported in part by the National Science Foundation

under grant no. MIP-8715235.

11. References.

[Berlekamp68] Berlekamp, E.R., *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[Bollobas86] Bollobas, B., *Combinatorics, Set Systems, Hypergraphs, Families of Vectors, and Combinatorial Probability*, Cambridge University Press, 1986.

[Boral83] Boral, H., DeWitt, D., "Database machines: an idea whose time has passed?," *Database Machines*, ed. H.O. Leilich, M. Missikoff, Springer-Verlag, September 1983.

[Gray85] Gray, J., "Why do computers stop and what can be done about it?," Tandem Technical Report 85.7, June 1985.

[Friedberg79] Friedberg, S.H., A.J. Insel, L.E. Spence, *Linear Algebra*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Hall67] Hall, M. Jr., *Combinatorial Theory*, Blaisdell Publishing Co., 1967.

[Murnan87] Murnan, T., "Challenges in Winchester Disk Drives, High Performance Disk Drives," *Challenges in Winchester Technology, A Short Course*, IIST, Santa Clara University, Dec 1987.

[Jilke86] Jilke, W., "Disk array mass storage systems: the new opportunity," Amperif Corp., Sept 1986.

[Kim85] Kim, M.Y., "Parallel operation of magnetic disk storage devices: synchronized disk interleaving," *Database Machines, Fourth Int. Workshop on*, ed., D.J. DeWitt, H. Boral, Springer-Verlag, March 1985.

[Kim87] Kim, M.Y., A.N. Tantawi, "Asynchronized disk interleaving," IBM T.J. Watson Research Center Technical Report RC-12497, February 1987.

[Klietz88] Klietz, A., J. Turner, and T.C. Jacobson, "TurboNFS: fast shared access for Cray disk storage," *Proc. of Cray User Group Convention*, Apr. 1988.

[Livny87] Livny, M., S. Khoshafian, H. Boral, "Multi-disk management algorithms," *Proc. of ACM SIGMETRICS*, May 1987.

[Siewiorek82] Siewiorek, D.P., R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[Maginnis87] Maginnis, N.B., "Store more, spend less: mid-range options around," *Computerworld*, Nov. 16, 1987, p. 71.

[Park86] Park, A., K. Balasubramanian, "Providing fault tolerance in parallel secondary storage systems," Princeton Technical Report CS-TR-057-86, November 1986.

[Patterson88] Patterson, D.A., G.A. Gibson, R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD 88*, Chicago, June 1-3, 1988.

[Peterson61] Peterson, W.W., *Error-Correcting Codes*, M.I.T. Press and John Wiley & Sons, 1961.

[Rubinstein81] Rubinstein, R.Y., *Simulation and the Monte Carlo Method*, John Wiley & Sons, 1981.

[Salem86] Salem, K., H. Garcia-Molina, "Disk striping," *IEEE 1986 Int. Conf. on Data Engineering*, 1986.

[Schrijver79] Schrijver, A., ed., "Packing and Covering in Combinatorics," *Mathematical Centre Tracts 106*, Mathematisch Centrum, Amsterdam 1979.

[Schulze89] Schulze. M., G. Gibson, R. Katz, D. Patterson, "How reliable is a RAID?," *COMPCON Spring 89*, San Francisco, 1989.

[Vasudeva88] Vasudeva, A., "A case for disk array storage system," *Systems Design and Networks Conference Proc., Mass Storage Trends and Systems Integration*, ed. Kenneth Majithia, April 1988.

12. Appendix: Proofs of Theorems Regarding t-Erasure-Correcting Codes.

In this section, the *sum* of two columns means the element-by-element exclusive OR.

Theorem 1: In a linear code S with parity check matrix $H=[P\,|I]$ the following properties are equivalent: 1) The distance between any two codewords is at least $t+1$; 2) The code is t error detecting; 3) Every set of t columns from H is linearly independent; 4) The code is t-erasure-correcting. **Pf:** The equivalence of properties 1-3 is proved in most coding theory textbooks. We include a proof here for the convenience of the reader. If the distance between any two codewords of a code S is at least $t+1$, then any change in at most t components of a codeword of S will not result in a different valid codeword. So, S detects at least t errors.

The zero vector is a codeword of S because $H0=0$ for all possible H. Let Y be a vector of weight z, $z\le t$. If S is t error detecting, then X cannot be a codeword in S, because otherwise the code would not be able to detect the t errors which turn the 0 vector into X. Therefore, $HX\neq 0$, which implies that the sum of any subset of t or fewer of the columns of H is never the zero vector. It follows that any set of t columns of H is linearly independent.

Let X be a codeword of S with $z\le t$ components erased. Rearrange the columns of H and entries of X in the same way so that $H=[A\,|B]$ and $X=[d\,|y]$, where y represents the z erased components, and B contains the z corresponding columns of H. Since X is a codeword, $HX=0$, which implies that $Ay=Bd$. If every set of t columns of H is linearly independent, then B has rank z, and there is a unique solution for y. Therefore, S is t-erasure-correcting.

If S is t-erasure-correcting, then if t entries are erased from a codeword X, there is only one way to fill in the erased entries to form a valid codeword of S. Suppose Y was a codeword that differed from X in fewer than t positions. If the values in those positions in X were erased, then the erased positions could be completed in two ways, either to form X or to form Y, which is a contradiction. Therefore the distance between any two codewords of S is at least $t+1$.

Lemma 1: The sum of an even weight column and an odd weight column is a column with odd weight. **Pf:** Let the even weight column have weight $2a$ and the odd weight column have weight $2b+1$. Suppose the columns contain q 1's which occur in a common row in both columns Then the weight of the sum of the two columns is $2a-q+2b+1-q=2(a+b-q)+1$, which is odd.

Theorem 2: The full-2 code is a 2-erasure-correcting code. **Pf:** A code is t-erasure-correcting if and only if any set of t columns selected from its parity check matrix are linearly independent. An equivalent condition is that the sum of any t or fewer of its columns is never the zero vector. Consider $H=[P_{full2}\,|I]$, the parity check matrix of the full-2 code. It

does not contain the zero vector as a column. The sum of any two columns of I is a column of weight 2. Columns in P_{full2} have even weight, and columns in I have odd weight, so the sum of a column from I and a column from P_{full2} is a column of odd weight. Finally, consider a pair of columns from P_{full2}. These columns are distinct and have weight 2, so their sum either has weight 2 or weight 4.

Theorem 3: The 2d-parity code is a 2-erasure-correcting code. **Pf:** The columns of the $c \times c^2/4$ parity check matrix $H_{2dparity}=[P_{2dparity} | I]$ are a subset of the columns of the $c \times (\binom{c}{2}+c)$ parity check matrix $H_{full2}=[P_{full2} | I]$. Because any 2 columns of H_{full2} are linearly independent, it follows that any 2 columns of $H_{2dparity}$ are linearly independent, and 2d-parity is a 2-erasure-correcting code.

Theorem 4: The full-3 code is a 3-erasure-correcting code. **Pf:** Similar to the proof that the full-2 code is 2-erasure-correcting.

Theorem 5: The additive-3 code is a 3-erasure-correcting code. **Pf:** Follows directly from the fact that the full-3 code is 3-erasure-correcting.

Theorem 6: The 2d-parity code can correct all sets of 3-erasures except bad 3-erasures. **Pf:** A code can correct a set of erasures provided that the erased bits correspond to a set of linearly independent columns of the parity check matrix. We must therefore show that any set of 3 columns of $H_{2dparity}=[P_{2dparity} | I]$ which do not correspond to an information bit and its two check bits, are linearly independent. Remember that 2d-parity can be described in terms of a two dimensional array of information disks, with check disks storing the parity of each row and column of the array. For simplicity, assume that the top $c/2$ rows of $P_{2dparity}$ correspond to the check disks which store the parity of the rows of the disk array, and the bottom $c/2$ rows of $P_{2dparity}$ correspond to the check disks which store the parity of the columns of the disk array. Then $P_{2dparity}$ consists of all possible columns of weight 2 with the property that one of the 1's in the column occurs in the first $c/2$ rows, and the other occurs in the last $c/2$ rows. Consider any two columns of $P_{2dparity}$. If the two columns do not contain a 1 in a common row, then the sum will have weight 4. If the two columns do contain a 1 in a common row, then their sum will be a column which either contains two 1's in its first $c/2$ rows, or two 1's in its last $c/2$ rows. It follows that the sum of any three columns of $P_{2dparity}$ cannot be the zero vector, and the sum of two columns of $P_{2dparity}$ and one column of I cannot be the zero vector either. A set of two columns from I and one column from $P_{2dparity}$ can only sum to zero if the columns correspond to an information disk and its two associated check disks. Finally, any three columns of I are linearly independent.

Theorem 7: The additive-3 code corrects all sets of 4-erasures except bad 4-erasures. **Pf:** We have already shown that any three columns of $H_{additive\,3}=[P_{additive\,3}|I]$ are linearly independent. Therefore, the theorem holds if and only if no set of four columns of $H_{additive\,3}$ sums to zero unless the columns correspond to a data bit and its three associated check bits. We begin by proving a result about the columns of $P_{additive\,3}$. A column of $P_{additive\,3}$ can be represented by a set {q,r,s} where

$$q+r+s \equiv 1 \bmod c$$

and the three 1's in the column occur in rows q, r, and s. Our claim is that if $\{q,r,s\}$ and $\{a,b,d\}$ represent two different columns of $P_{additive\,3}$, then the two sets contain at most one element in common. Assume not. Then without loss of generality, $a=q$ and $b=r$. It follows that

$$a+b+s \equiv 1 \bmod c \;;\quad a+b+d \equiv 1 \bmod c$$

which implies that $s=d$, and $\{q,r,s\}=\{a,b,d\}$. The two sets represent the same column, which is a contradiction.

Consider a set consisting of 4 columns from $P_{additive\,3}$. We want to show that these columns do not sum to zero. Let $\{a,b,d\}$ and $\{q,r,s\}$ be the sets which represent two of the columns. These sets either contain one element in common, or no elements in common. Suppose first that they do not contain any elements in common. In this case, the sum of the two columns is the column $\{a,b,d,q,r,s\}$. If the 4 columns from $P_{additive\,3}$ did sum to zero, then the sets representing the other 2 columns would have to be three element subsets of $\{a,b,d,q,r,s\}$. However, any three element subset of $\{a,b,d,q,r,s\}$ contains at least two elements in common with either $\{a,b,d\}$ or $\{q,r,s\}$, which leads to a contradiction. Therefore, if the four columns from $P_{additive\,3}$ sum to zero, then $\{a,b,d\}$ and $\{q,r,s\}$ must contain exactly one element in common. Assume without loss of generality that $d=s$. In this case the sum of columns $\{a,b,d\}$ and $\{q,r,d\}$ is the weight 4 column represented by $\{a,b,q,r\}$. If the 4 columns of $P_{additive\,3}$ sum to zero, then the sum of the other 2 columns must also be the column $\{a,b,q,r\}$. The sets representing these other two columns can contain at most 1 element in common with $\{a,b\}$ and $\{q,r\}$. It follows that the sets must be of the form $\{a,q,z\}$ and $\{b,r,z\}$ or $\{a,r,z\}$ and $\{q,b,z\}$. Without loss of generality assume they are of the form $\{a,q,z\}$ and $\{b,r,z\}$. The following is true:

$$a+b+d \equiv 1 \bmod c \;;\quad q+r+d \equiv 1 \bmod c$$

$$a+q+z \equiv 1 \bmod c \;;\quad b+r+z \equiv 1 \bmod c$$

These equations imply that

$$a+b \equiv (q+r) \bmod c \quad \text{and} \quad a+q \equiv (b+r) \bmod c$$

Subtracting the two equations yields $b-q \equiv (q-b) \bmod c$ which implies that $2(b-q) \equiv 0 \bmod c$. Because c is odd, the

only number between 0 and $c-1$ which satisfies the equation $2x \equiv 0 \bmod c$ is 0, which implies that $b=q$. But if $b=q$ then {a,b,d} and {q,r,d} have two elements in common, which is a contradiction. Therefore, a set of 4 columns of $P_{additive\,3}$ cannot sum to zero.

To complete the proof, one must show that other sets of 4 columns of $H_{additive\,3}$ (e.g. sets consisting of 2 columns from $P_{additive\,3}$ and 2 columns from I) do not sum to zero either, unless the columns correspond to a bad 3-erasure. These other cases are easier than the above case, and we omit them here.

Theorem 8: For a given number of check bits, the full-2 code has the highest number of information bits of any 2-erasure-correcting code with minimum possible update penalty. Similarly, for a given number of check bits, the full-3 code has the highest number of information bits possible for a 3-erasure-correcting code with minimum possible update penalty. **Pf:** Consider the parity check matrix $H=[P\,|I]$ of any 2-erasure-correcting code with minimum possible update penalty. The number of information bits in the code is the number of columns of P. The columns of P must have weight 2. Furthermore, they must all be distinct, because if columns of P are identical, then those two columns are not linearly independent, and the code is not 2-erasure-correcting. Therefore, P contains the maximum possible number of columns if it contains all possible distinct columns of weight 2. The proof for the full-3 code is analogous.

Theorem 9: For a given even number of check bits, the 2d-parity code has the highest number of information bits of any 2-erasure-correcting, minimum possible update penalty code that can correct all sets of 3-erasures except bad 3-erasures. **Pf:** Consider any 2-erasure-correcting code with c check bits and minimum possible update penalty. The parity check matrix $H=[P\,|I]$ of the code can be represented as a graph. The graph contains c vertices. The vertices correspond to the c rows of P. The columns of P have weight two. The graph contains an edge between two vertices v_1 and v_2 if and only if there exists a column in P that contains 1's in the rows corresponding to v_1 and v_2.

Suppose the graph contains a clique of size 3. Consider the three columns of P corresponding to the edges in the clique. Each vertex in the clique is adjacent to exactly two edges of the clique. Therefore, if one of the three columns has a 1 in row r, exactly one of the other two columns will also have a 1 in row r. It follows that the three columns sum to zero. The three columns constitute an uncorrectable 3-erasure that is not bad. Therefore, if a 2-erasure-correcting code with minimum possible update penalty corrects all sets of 3-erasures except bad 3-erasures, then the graph corresponding to the parity check matrix of that code cannot contain a clique of size three.

Recall the description of the 2d-parity code in section 2. Some of the check disks compute parity along rows, and some compute parity along columns. Each information disk is checked by a "row check disk," and a "column check

disk.'' The graph of the parity check matrix of the 2d-parity code is a complete bipartite graph. The vertices on one side of the bipartition correspond to the $c/2$ row check disks, and the vertices on the other side correspond to the $c/2$ column check disks. The number of information bits is $c^2/4$, and hence the number of edges in the graph is $c^2/4$. The famous theorem of Turan (see, e.g., [Bollobas86]) says that for even c, a graph with c vertices that does not contain a clique of size three has at most $c^2/4$ edges. Moreover, the only graph on c vertices that actually has $c^2/4$ edges and no cliques of size 3 is the complete bipartite graph on c vertices in which each side of the bipartition consists of $c/2$ vertices. Therefore, for fixed even c, the 2d-parity code contains the highest number of information bits of any 2-erasure-correcting code with minimum possible update penalty that corrects all sets of 3-erasures except bad 3-erasures.

Theorem 10: For a given number of check bits, c, any minimum possible update penalty, 3-erasure-correcting code that corrects all sets of 4-erasures except bad 4-erasures, has at most $c(c-1)/6$ information bits. **Pf:** Omitted.

Theorem 11: Codewords in the additive-3 code contain $c(\frac{c-1}{2}-1)/3$ information bits, where c is the number of check bits. **Pf:** Follows from the fact that c is a multiple of 3.

Definition: the steiner code.

Let $X = \{0,1,2,\ldots,c-1\}$. A *Steiner triple system* of X is a set of subsets $X_0, X_1, \cdots, X_{c(c-1)/6-1}$ of X such that each subset is of size three, and every pair of elements x, y in X appears in exactly one subset X_i ($i \in \{0,1,\ldots,c(c-1)/6-1\}$) [Hall67].

We use a simple, recursively defined, Steiner triple system of $X = \{0,1,\ldots,c-1\}$ to construct our code. We call this Steiner triple system *Steiner*(c). *Steiner*(c) is defined only for c a power of three. Let $k+$*Steiner*(c) denote the Steiner triple system of $\{k,k+1,\ldots,k+c-1\}$ produced by adding k to every element of every subset in *Steiner*(c). The recursive definition of *Steiner*(c) is as follows (c is a power of three):

1) For $c>3$, $\quad$ *Steiner*(c) = *Steiner*$(c/3) \cup 1/3+$ *Steiner*$(c/3) \cup 2/3+$ *Steiner*$(c/3) \cup$

$$\{\{i, 1/3+((i+j) \bmod c/3), 2/3+((i+2j) \bmod c/3)\} \mid i,j \in \{0,1,\ldots,c/3-1\}\}$$

2) Steiner(3) = $\{\{0,1,2\}\}$

It is straightforward to show that *Steiner*(c) is actually a Steiner triple system.

We call our code the *steiner* code. The parity check matrix of the Steiner code has the following form. Let c, the number of check bits, be a power of 3. As in the additive-3 code, number the rows of $P_{steiner}$ from 0 to $c-1$. $P_{steiner}$ con-

sists of all distinct columns that contain 1's in rows q, r, and s, where $\{q,r,s\}$ is a set in $Steiner(c)$.

Theorem 12: For c is a power of 3, the steiner code is the 3-erasure-correcting code with c check bits that has minimum possible update penalty, corrects all sets of 4-erasures except bad 4-erasures, and has the maximum possible number of information bits of any code with these properties. **Pf:** The fact that the steiner code is 3-erasure-correcting with minimum possible update penalty follows directly from the fact that the full-3 code is 3-erasure-correcting with minimum possible update penalty.

Because $Steiner(c)$ contains $c(c-1)/6$ sets, $P_{steiner}$ contains $c(c-1)/6$ columns. It follows directly from Theorem 10 that the Steiner code has the maximum possible number of information bits of any code with the properties in question.

All that is left to show is that the steiner code corrects all sets of 4-erasures except bad 4-erasures. We know that the steiner code is 3-erasure correcting, and thus any set of three columns of $H_{steiner} = [P_{steiner} | I]$ is linearly independent. It is therefore sufficient to prove the following claim.

Claim: If a set of four columns of $H_{steiner}$ does not consist of columns corresponding to an information bit and its three associated check bits, then those four columns do not sum to zero. **Pf:** The proof of the claim is by induction on c. The claim is clearly true for $c = 3$, because when $c = 3$, $H_{steiner}$ consists of exactly four columns: one corresponding to a data bit, and the others corresponding to its three associated check bits. Assume that $c > 3$, c is a power of three, and the claim holds for all smaller powers of three.

Note that the triples of $Steiner(c)$ are of two basic types: those that are subsets of $\{k,k+1,\ldots,k+c/3-1\}$ for some fixed $k \in \{0,1/3,2/3\}$, and those that contain one element from each of $\{0,1,2,\ldots,c/3-1\}$, $\{c/3,c/3+1,\ldots,2c/3-1\}$, and $\{2c/3,2c/3+1,\ldots,c-1\}$. We call the former type *vertical triples* and the latter type *horizontal triples*. The proof is broken into cases. We show the two most difficult cases.

Case 1: The four columns are all contained in $P_{steiner}$, and they all correspond to vertical triples.

We will need the following fact about Steiner triple systems: Given a set of 1, 2, or 3 triples of a Steiner system, there exist at least three elements that are contained in exactly one triple in the set. This fact is a straightforward consequence of the property that any two triples of a Steiner triple system can contain at most one element in common.

Suppose all four of the vertical triples in this case are contained in $\{k,k+1,\ldots,k+c/3-1\}$ for some fixed k in $\{0,1/3,2/3\}$. Then the triples are triples of the Steiner triple system $k+Steiner(c/3)$. It follows from the induction

hypothesis that the four columns do not sum to zero.

Suppose therefore that the four triples are not all contained in a single set $\{k,k+1,\ldots,k+c/3-1\}$ ($k\in\{0,1/3,2/3\}$). Let $\{k_1,k_1+1,\ldots,k_1+c/3-1\}$ be a set that contains at least one of the triples. At most three of the four triples can be contained in this set. It follows from the fact we mentioned above that there exist at least three elements of $\{k_1,k_1+1,\ldots,k_1+c/3-1\}$ that are contained in exactly one of the four triples. Therefore, the four columns do not sum to zero.

Case 2: The four columns are all contained in $P_{steiner}$, and they all correspond to horizontal triples.

Assume that the four columns do sum to zero. We will show that this assumption leads to a contradiction.

Because the columns sum to zero, each element of the corresponding triples appears in an even number of those triples. Let the four triples be $\{r_1,s_1,t_1\}$, $\{r_2,s_2,t_2\}$, $\{r_3,s_3,t_3\}$, and $\{r_4,s_4,t_4\}$, where $r_k\in\{0,1,\ldots,c/3-1\}$, $s_k\in\{c/3,c/3+1,\ldots,2c/3-1\}$, and $t_k\in\{2c/3,2c/3+1,\ldots,c-1\}$ for $k\in\{1,2,3,4\}$.

Suppose that some element appears in all four of the triples. Without loss of generality, assume that element is r_1 (i.e. $r_1=r_2=r_3=r_4$). Two triples have at most one element in common. Hence all other elements but r_1 appear in exactly one of the four triples, and the four columns do not sum to zero. Contradiction.

Therefore, every element appearing in the four triples must appear in exactly two of them. Without loss of generality, assume that $r_1=r_2$. It follows that $r_3=r_4$, and $s_1\neq s_2$. s_1 must appear in exactly two triples. Without loss of generality, assume $s_1=s_3$. It follows that $s_2=s_4$, $t_1=t_4$, and $t_2=t_3$.

Recall that every horizontal triple is equal to $\{i,1/3+((i+j)\bmod c/3),2/3+((i+2j)\bmod c/3)\}$ for some $i,j\in\{0,1,\ldots,c/3-1\}$. Note that $i=2(i+j)-(i+2j)$, which implies that for $k\in\{1,2,3,4\}$,

$$r_k=(2s_k-t_k)\bmod c/3.$$

Therefore,

$$r_1=r_2=(2s_1-t_1)\bmod c/3=(2s_2-t_2)\bmod c/3$$

and

$$r_3=r_4=(2s_1-t_2)\bmod c/3=(2s_2-t_1)\bmod c/3.$$

As a consequence, $2t_1=(2t_2)\bmod c/3$. Because $c/3$ is odd and $t_1,t_2\in\{2c/3,2c/3+1,\ldots,c-1\}$, $t_1=t_2$. But then t_1 appears in all four triples, rather than in exactly two of them. Contradiction.

This completes the proof of Case 2. The remaining cases are simpler, and we omit them.

Theorem 13: There exist balanced orderings of the full-2 code, the full-3 code (when c is a multiple of 3), and the 2d-parity code. **Pf:** Let c be divisible by t. Consider a set of distinct columns of length c and weight t. Number the positions in the columns from 0 to $c-1$. We say that the set of columns is *factorizable* if it is possible to partition the columns into disjoint subsets of c/t columns, such that within each subset, for all i between 0 and $c-1$, there is exactly one column containing a 1 in position i. Such a partition is called a *factorization*. Each subset in the partition is called a *factor*. An example of the factorization of the full-2 code is given in Figure 10.

Given a factorization of a set of columns, those columns can be arranged in a balanced ordering by simply putting down the columns factor by factor. That is, the first c/t columns of the ordering are the columns of one factor, the next c/t columns are the columns of another factor, and so on. There is a theorem (proved by Baranyai) from combinatorics which states that if t divides c, then there exists a factorization of the set consisting of all columns of length c and weight t [Schrijver79, Bollobas86]. This theorem implies that it is possible to achieve a balanced ordering of the full-2 code when c is even, and the full-3 code when c is divisible by 3. An explicit construction for a factorization of the set of weight 2 columns of length c, when c is even is given in [Bollobas86]. This construction can be modified slightly to produce a balanced ordering of the full-2 code when c is odd. It is easy to construct a factorization of the columns of $P_{2dparity}$.

Theorem 14: There is no balanced ordering of the additive-3 code. **Pf:** Omitted.

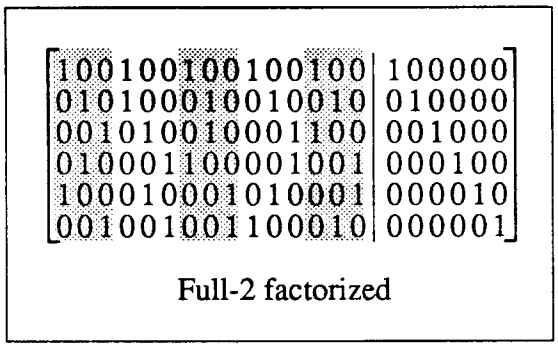

Figure 10. *A factorization of the columns of P_{full2}. Each shaded or unshaded block of 3 contiguous columns is a factor.*