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

Lisa Hellerstein,[2] Garth A. Gibson,[3] Richard M. Karp,[4]
Randy H. Katz[4] and David A. Patterson[4]

Abstract: *A crucial issue in the design of very large disk arrays is the protection of data against catastrophic disk failures. Although today single disks are highly reliable, when a disk array consists of 100 or 1000 disks, the probability that at least one disk will fail within a day or a week is high. In this paper, we address the problem of designing erasure-correcting binary linear codes that protect against the loss of data caused by disk failures in large disk arrays. We describe how such codes can be used to encode data in disk arrays, and give a simple method for data reconstruction. We discuss important reliability and performance constraints of these codes, and show how these constraints relate to properties of the parity check matrices of the codes. In so doing, we transform code design problems into combinatorial problems. Using this combinatorial framework, we present codes and prove they are optimal with respect to various reliability and performance constraints.*

Keywords: Input/Output architecture, redundant disk arrays, RAID, error-correcting codes, reliability, availability.

1. Background

In recent years, processing power has increased dramatically through advanced VLSI technology [Myers86, Gelsinger89] and parallel architectures [Bell85, Bell89]. As processing power increases, so does the demand for increased Input/Output (I/O) performance. The mainstay of on-line 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 [Chen90, Kim87, Klietz88, Livny87, Salem86].

---

[1] This paper is a revised and expanded version of material that appeared in the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), Boston, March 1989 [Gibson89].

[2] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208.

[3] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

[4] Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

Although disks have not been getting much faster, they have been getting significantly smaller and cheaper. Disk densities have been growing exponentially, cost per megabyte has been decreasing in step with density increases, and physical packaging has been achieving amazing reductions in volume [Hoagland89, Kryder89]. This trend has led some to explore the replacement of individual large form-factor drives with many smaller form-factor drives [Jilke86, Patterson88, Gibson92], giving yet another reason to expect that future I/O systems will contain large 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 for electronics in general and for magnetic disks in particular [Lin88, Gibson92], 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 four months. Such high rates of data loss encourage the inclusion of data redundancy to allow information to survive hardware failures.

Today's magnetic disk drives suffer from three primary types of failures. The first type, *transient* or noise-related error, is corrected by repeating the offending operation or by applying per sector error-correction facilities. The second type, *media defects*, are permanent errors and are usually detected and masked at the factory. In this paper, we address the problem of designing codes to protect against the third type of failure, *catastrophic* failures, which are head crashes or failures of the read/write or controller electronics.

When a disk in an array experiences a head crash, the crash is detected by the controller microprocessor. If a controller fails, the failure is detected by a host through violations of the complex host/controller protocol. Thus, following a catastrophic disk failure, the bits on a failed disk can be designated as ''unreadable.'' An unreadable bit is called an *erasure*. The codes we develop for disk arrays are ‘‘erasure-correcting’’ codes. Erasure-correcting codes differ from error-correcting codes in that erasure-correcting codes are designed to recover erased bits in a message word. The positions of the erased bits are known. In contrast, error-correcting codes are designed to correct messages in which some of the bits may have been flipped, but the positions of those bits are unknown. For example, parity is a single-erasure-correcting code, but it is not a single-error-correcting code (though it is single-error-detecting). Since in the parity code, the value of each bit is the parity of all the other bits, to correct a single erasure we can simply compute the parity of all the unerased bits.

There are, of course, other possible causes of data loss in a disk array in addition to disk drive failures. Examples include the failure of power, cabling, memory, and processors (see [Gibson92, Gibson93] for discussions of the effect of these on RAID architecures), and incorrect or misused software [Gray85]. Although the codes presented in this paper do not protect against data loss from these causes, they do ensure that, as the number of disk drives per system grows, catastrophic disk failures will not limit data reliability.

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. A goal in designing codes for these arrays is to make them at least as reliable as an individual disk.

The use of erasure-correcting codes in large disk arrays has also been studied by Rabin [Rabin89], and the method he proposed for recovering lost data is similar to the one we present in Section 5. However, he did not address the problem of designing codes particularly suited for use in disk arrays. This problem is a primary focus of our paper.

We begin by discussing the properties that are desirable in a code for a large disk array. We then present an introductory family of binary linear codes for correcting multiple erasures. We continue with a general discussion of the implementation of such codes in disk arrays. We show that the problem of designing a good code can often be expressed in terms of a combinatorial design problem. We present a number of codes and prove that they are optimal with respect to our implementation metrics. Finally, we discuss our results and conclude with an indication of future directions.

2. Metrics for Redundancy in Disk Arrays

There are many possible schemes for introducing redundancy into a disk array. 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 operations. This ensures that check data can be quickly and simply manipulated. An equivalent way of expressing these restrictions is to say that we restrict ourselves to schemes based on binary linear codes.

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.

There are many metrics that can be used to assess the suitability of a coding scheme for use in a large disk array. Because of the unreliability of a large non-redundant disk array induces our need for redundancy, the *mean time to data loss* (MTTDL) is a primary metric for the choice of a coding scheme. Associated with each coding scheme are three further important metrics: check disk overhead, update penalty, and group size.

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

The *update penalty* of a coding scheme is the number of check disks whose contents must be changed when a minimal change is made in the contents of a given information disk. One of the advantages of large disk arrays lies in the concurrent processing of many random secondary storage accesses. If a code requires $N > 1$ check disks to be involved in every write, then the available parallelism is reduced by up to a factor of $N + 1$. Because parallelism is the reason we want to use disk arrays, the number of disk accesses required to effect a small data update must be minimized.

The set of disks that must be accessed during the reconstruction of a single failed disk form a group. The *group size* is an important metric because the duration of reconstruction is likely to scale linearly with the number of disks to be read. Additionally, in very large arrays, individual disk failure will be frequent enough that highly available systems must continue operation during repair and reconstruction [Muntz90, Holland92]. Until reconstruction is complete, the group size indicates the number of disks that must be accessed to read or write an unreconstructed block on a failed disk. Moreover, the group size also indicates the number of operational disks for which user access performance is degraded by a reconstruction.

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.

3. A First Example: Parity in t Dimensions

To illustrate some of the metrics discussed in the previous section, we present a simple example. One standard single-erasure-correction scheme is to divide up the information disks into sets of G disks, and to associate a parity check disk with each of those sets. In Figure 3(a) we show these G disks (for $G = 4$) in a row with their parity in the disk on the right side. This coding scheme has an overhead of $1/G$, 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 one end of each row and column a check disk stores parity for that row or column. Since a failed disk belongs to two potential groups – its row and its column – it can be reconstructed from the data of either; thus, 2d-parity is double-erasure-correcting. This coding scheme has a check disk overhead of $2G/G^2 = 2/G$, an update penalty of 2, and a group size of $G+1$.

The 2d-parity coding scheme extends to t-erasure correction by logically arranging G^t information disks in a t-dimensional array and recording parity on a disk at one end of each dimensional group. A dimensional group is formed by fixing the coordinates in $t-1$ dimensions, and allowing the remaining coordinate to vary. In these coding schemes each information disk belongs to t groups, and the update penalty is t. These coding schemes have a check disk overhead of $t\,G^{(t-1)}/G^t = t/G$, since there are $t\,G^{(t-1)}$ check disks and G^t information disks. However, the group size remains $G+1$ because only $G+1$ disks are involved in the reconstruction of any single failure. The td-parity coding schemes are a member of the class of product codes that have been commonly used in magnetic tape systems [Peterson72]. A common, but expensive, technique for protecting disk systems from disk failures is known as *shadowing* [Bates89, Bitton88]. A shadowing code is equivalent to a td-parity code with $G = 1$ because the parity of a single bit duplicates the value of that bit.

We do not envision practical disk arrays large enough to require t greater than 2 or 3, because other sources of data loss limit the increase in data reliability and because of the performance degradation associated with increasing numbers of check disk updates.

Although td-parity has an easily visualized structure, it is not necessarily the best coding scheme for our metrics. In the next section, we discuss more general techniques for designing schemes based on linear codes.

4. Use of Linear Codes for Large Disk Arrays

The codewords in a binary linear code can be viewed as vectors of information bits and check bits. The check bits can be computed as the parity of subsets of information bits. Each binary linear code can be defined in terms of a c x$(k+c)$ *parity check matrix,* $H=[P \mid I]$, where c is the number of check bits, k is the number of information bits, I is the c xc identity matrix, and P is a c xk matrix that determines the equations for the check bits. The codewords in the code are the vectors X satisfying the equation $HX=0$. In Figure 4 we show parity check matrices for the 1d-parity and 2d-parity codes described in the previous section.

We can use the parity check matrix $H=[P \mid I]$ as the basis of a coding scheme for a large disk array. The k columns of P represent the k information disks and the c columns of I represent the c check disks. In our coding scheme, each row of H represents a group – the columns that have 1's in a row correspond to the disks in that group. Each group consists of a check disk and the information disks whose parity it stores. In the following sections, when we discuss properties of a ''code,'' we refer to the properties of the coding scheme based on the associated parity check matrix $H=[P \mid I]$.

It is well known that any parity check matrix, H, has three equivalent properties expressed in terms of a parameter, t, whose value is between 0 and c.

(1) H will allow any t errors (arbitrary changes in t disks' values) to be detected.

(2) The minimum number of bits in which any two codewords differ, known as the *distance* of the code, is at least $t+1$.

(3) Any set of t columns selected from H will be linearly independent considered as vectors over GF[2].

A fourth property, which concerns erasure correction (not error correction), is also known to be equivalent.

(4) H will allow any t erasures to be corrected.

Note that a set of t binary vectors is linearly independent over GF[2] if and only if the vector sum, modulo 2, of those columns, or any nonempty subset of those columns, is not equal to the zero vector. In fact, whether any set of disk failures can be repaired and reconstructed depends on whether the corresponding set of columns in H is linearly independent (cf. Section 5). 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 are not suitable for disk arrays because they have large update penalties.

Three of our metrics from Section 2 are easily expressed in terms of parity check matrices. The check disk overhead is c/k, the ratio of the number of rows in P to the number of columns in P. The size of a group, which determines performance degradation during reconstruction of a single disk, is the weight of the row for that group (i.e. the number of 1's in that row). The update penalty for any information disk, which is the number of groups including that disk, is the weight of its column.

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 on-line, 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 [Gibson92] 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. 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 8, we apply this technique to evaluate the mean time to data loss for the t d-parity codes and to the codes introduced in Section 6.

5. Implementing Reconstruction

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 disks that have failed. 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 $HX = Ad + By = 0$ (addition mod 2). This means that we must try and solve the c equations in m unknowns described by $Ad = -By = By$. The failures are recoverable if and only if there is a unique solution to this system of equations, that is, if and only if the columns of B are linearly independent. This proves the fact stated in Section 4 that a code is t-erasure-correcting if and only if every set of t columns of H is linearly independent.

To solve the system $Ad = By$, we could first compute the product Ad, yielding a vector q, and then solve the system $q = By$. Unfortunately, because today's disks contain billions of bits, this method computes q then solves $q = By$ billions of times. It is better to spend additional time performing initial matrix computations on A and B (which are parts of the parity check matrix H stored locally) if this will reduce the cost of the computations to be done billions of times.

We therefore propose the following method of solving the system $Ad = By$. Because the m columns of B are linearly independent if and only if there is a set of m rows from B that is linearly independent, we first attempt to find m linearly independent rows of B (using a standard technique such as Gaussian elimination). If this attempt fails, the failed disks cannot be reconstructed. Otherwise, if m linearly independent rows are found, let B' be the matrix consisting of these rows, and let A' be the corresponding rows of A. The solution to the system $Ad = By$ is the same as the solution to the system $A'd = B'y$. To solve the system $A'd = B'y$, we first compute $(B')^{-1}$ (by some standard technique) and then the product $(B')^{-1}A'$. This determines the operation that must be performed once for each codeword, $X = [d \mid y]$, to correct exactly one of the billions of bits on each of the lost disks; that is, one codeword of the failed disks' data is reconstructed by $y = (B')^{-1}A'd$.

Note that to compute y we only need to access those disks corresponding to non-zero columns of $(B')^{-1}A'$. Because $(B')^{-1}$ is of full rank, a column of $(B')^{-1}A'$ is non-zero if and only if the corresponding column of A' is non-zero. Therefore, where there are multiple ways to select B' from B, this selection should be done to minimize the number of non-zero columns in the corresponding matrix, A'. We leave for further research the development of such specialized methods.

In the case of a single failure, B' will consist of a single row from B. This means that the set of disks involved in reconstructing a single disk will correspond to a row of H; thus, group size is determined by the weight of the rows in H, as claimed in Section 4.

The above method for failure correction can be implemented in software that runs in the host or 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 the variables A, B, d and y, where d and y are placeholders for each of the billions of codewords. Using these values, software can compute the matrix $(B')^{-1}A'$, or determine that the lost data cannot be reconstructed.

The functional disks corresponding to non-zero columns of $(B')^{-1}A'$ are accessed by reading them in parallel, large blocks at a time. The reconstructed data is then computed and written either to repaired or replaced disks.

Any set of t or fewer columns from the parity check matrix of a t-erasure-correcting code is linearly independent, so all sets of m less than or equal to t failures are recoverable. Moreover, if m is greater than t, then some, but not all, sets of failures are recoverable. A set of $m > t$ failures is recoverable if and only if the columns corresponding to the failed disks are linearly independent. The successful recovery of such sets of failures is important to a high MTTDL. For more failures than check disks, ($m > c$), there is never a way to reconstruct all lost data.

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

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 evidence of every write on at least t different check disks, its minimum update penalty is t. Thus, in designing good t-erasure-correcting codes, we demand that the column corresponding to each information disk have weight exactly t. The td-parity code described in Section 3 has this property.

Within the class of 2-erasure-correcting and 3-erasure-correcting codes with minimum update penalty (the information columns of H have weight 2 or 3, respectively) we present, in Subsections 6.1 and 6.2, 2-erasure-correcting and 3-erasure-correcting codes that are designed to achieve low check disk overhead and high reliability. We also prove optimality results concerning these codes. Example parity check matrices for these codes are found in Figures 5 and 6.

6.1 Codes with Optimal Check Disk Overhead

We begin by presenting a 2-erasure-correcting code called the *full-2 code*, and a 3-erasure-correcting code called the *full-3 code*. We then argue that these codes achieve the optimal check disk overhead possible among codes in their class.

Definition: The *full-2 code* is a 2-erasure-correcting code. It is defined by the parity check matrix $H_{full\,2} = [P_{full\,2} \,|\, I]$, where $P_{full\,2}$ consists of all possible distinct columns of weight two.

The full-2 code is 2-erasure-correcting because each column of its parity check matrix is non-zero, and the sum of any two columns is also non-zero (because the columns are all distinct). The check disk overhead of the full-2 code is $c/\binom{c}{2} = 2/(c-1)$, where c is the number of check bits,[5] because the number of distinct columns of weight two is $\binom{c}{2}$.

Definition: The *full-3 code* is a 3-erasure-correcting code with parity check matrix $H_{full\,3} = [P_{full\,3} \,|\, I]$, where $P_{full\,3}$ consists of all possible distinct columns of weight three.

To verify that the full-3 code is indeed 3-erasure correcting, note that the sum of any two columns of its parity check matrix is a non-zero column of even weight, and therefore the sum of any three columns cannot be the zero vector. The check disk overhead of the full-3 code is $c/\binom{c}{3} = 6/((c-1)(c-2))$, where c is the number of check bits.

If a code corrects all sets of 2-erasures, then all the columns of its parity check matrix must be distinct. In a 2-erasure-correcting code with minimum update penalty, the columns of its parity check matrix that are associated with information bits must all have weight two. It follows that the full-2 code protects the maximum number of information disks for a given number of

[5] We generally express a code's check disk overhead in terms of the number of check disks, c, because our codes are designed to maximize the number of information disks, k, for a given number of check disks.

check disks; therefore, it achieves the minimum possible check disk overhead of any 2-erasure-correcting code with minimum update penalty. Similarly, the full-3 code achieves the minimum possible check disk overhead of any 3-erasure-correcting code with minimum update penalty.

6.2 Codes with Good Reliability Properties

Sometimes we are willing to use a code that has less than optimal check disk overhead, but achieves extremely high reliabilty. In a t-erasure-correcting code, the fraction of sets of $t+1$-erasures that can be corrected has a significant effect on the overall reliability of the code. Note that in a t-erasure-correcting code with minimum possible update penalty, it is impossible to correct all sets of $t+1$-erasures. In particular, because every information bit is associated with t check bits, it is impossible to correct the set of $t+1$-erasures consisting of an information bit and its t check bits. We call such sets of erasures *bad* $t+1$-erasures. One way to design a reliable t-erasure-correcting code with minimum update penalty is to ensure that it corrects all sets of $t+1$-erasures except bad $t+1$-erasures. At the same time, it is desirable to keep the check disk overhead as low as possible.

In Section 6.2.1 we will prove an optimality result concerning correctability of 3-erasures and check disk overhead of 2-erasure-correcting codes. We will show that the 2d-parity code presented in Section 3 is optimal in its class. In Section 6.2.2, we will prove an optimality result concerning correctability of 4-erasures and check disk overhead of 3-erasure-correcting codes. We will also present two new 3-erasure-correcting codes, the *additive-3* code and the *steiner* code, prove that they correct all sets of 4-erasures that are not bad, and show that they are optimal or near optimal in their class.

6.2.1 Highly Reliable Double-Erasure-Correcting Codes

Theorem 1: If a code is 2-erasure-correcting, has minimum update penalty, and corrects all sets of 3-erasures except bad 3-erasures, then its check disk overhead is at least $4/c$, where c is the number of check bits.

Pf: Consider any 2-erasure-correcting code with c check bits and minimum update penalty that corrects all sets of 3-erasures except bad 3-erasures. We will prove that the maximum number of information bits in such a code is $c^2/4$. Therefore, the check disk overhead is at least $c/(c^2/4) = 4/c$.

The parity check matrix $H = [P \mid I]$ of this code can be represented as a graph. The graph contains c vertices, which 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, for the purpose of contradiction, that the graph contains a clique of size three. The vertices of the clique correspond to three rows of P. The edges correspond to three columns of P. Let the three rows of P be i_1, i_2, and i_3, and the corresponding vertices be v_{i_1}, v_{i_2}, and v_{i_3}. Let the three columns of P be j_1, j_2, and j_3, and the corresponding edges be e_{j_1}, e_{j_2}, and e_{j_3}. Without loss of generality, assume $e_{j_1} = (v_{i_1}, v_{i_2})$, $e_{j_2} = (v_{i_1}, v_{i_3})$, and $e_{j_3} = (v_{i_2}, v_{i_3})$.

Consider column j_1. Since j_1 corresponds to the edge e_{j_1} between v_{i_1} and v_{i_2}, it follows from the definition of the graph that the only 1's in column j_1 appear in rows i_1 and i_2. Similarly, the only 1's in column j_2 appear in rows i_1 and i_3, and the only 1's in column j_3 appear in rows i_2 and i_3. It follows that the sum of columns j_1, j_2, and j_3 (mod 2) is the zero vector. Therefore, columns j_1, j_2, and j_3 constitute an uncorrectable 3-erasure that is not bad, which contradicts the fact that the code corrects all sets of 3-erasures that are not bad. It follows that the graph corresponding to a code's parity check matrix cannot contain a clique of size three.

By Turán's Theorem (see, for example, [Bollobás86]), a graph with c vertices that does not contain a clique of size three has at most $c^2/4$ edges. Therefore, the graph corresponding to the parity check can contain at most $c^2/4$ edges. Since each edge corresponds to a column of P, and each column of P corresponds to an information bit, the code has at most $c^2/4$ information bits.

It follows that the check disk overhead of the code is at least $4/c$. □

The 2d-parity code is 2-erasure correcting and has minimum update penalty. We now prove that it corrects all sets of 3-erasures except bad 3-erasures.

Theorem 2: The 2d-parity code corrects all sets of 3-erasures except bad 3-erasures.

Pf: Recall the description of the 2d-parity code. 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.’’ To show that 2d-parity corrects all sets of 3-erasures except bad 3-erasures, we must show that any set of three columns of $H_{2dparity} = [P_{2dparity} \mid I]$ which does not correspond to an information bit and its two check bits, is linearly independent.

For simplicity, assume that the top $c/2$ rows of $P_{2dparity}$ correspond to the row check disks, and the bottom $c/2$ rows of $P_{2dparity}$ correspond to the column check disks (as in Figure 4). 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 two or 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. □

The check disk overhead of the 2d-parity code is $c/4$, where c is the number of check bits. Thus the 2d-parity code achieves the optimal check disk overhead possible for 2-erasure-correcting codes with minimum update penalty that correct all sets of 3-erasures except bad 3-erasures. In fact, Turán's theorem also states that the only graph with c vertices that has $c^2/4$

edges and no clique of size 3 is the bipartite graph with $c/2$ vertices on each side of the bipartition. This graph corresponds to the 2d-parity code – vertices on one side of the bipartition are row checks and vertices on the other side are column checks. Thus, if a code is 2-erasure-correcting, has minimum update cost and corrects all sets of 3-erasures except bad 3-erasures, then it must be 2d-parity.

6.2.2 Highly Reliable Triple-Erasure-Correcting Codes

We present two 3-erasure-correcting codes with minimum update penalty that correct all sets of 4-erasures except bad 4-erasures: the *steiner* code and the *additive-3* code. We also prove (Theorem 3) that any such code has a check disk overhead that is at least $6/(c-1)$. The steiner code achieves this minimum possible check disk overhead. However, the steiner code is defined only when the number of check disks is a power of 3. The additive-3 code, in contrast, is defined whenever the number of check disks is an odd multiple of 3. Moreover, the additive-3 code achieves a check disk overhead, $6/(c-3)$, which is vanishingly larger than optimal as disk arrays get larger. Therefore, the steiner code seems to be of theoretical interest only and will not be considered for the more practical considerations in Section 8.

We first prove Theorem 3. We then present the steiner code and the additive-3 code. For each code, we prove that it corrects all sets of 4-erasures except bad 4-erasures.

Theorem 3: A 3-erasure-correcting code with minimum update penalty that corrects all sets of 4-erasures except bad 4-erasures has check disk overhead at least $6/(c-1)$.

Pf: Let $H = [P \mid I]$ be the parity check matrix of a code that has the properties stated in the theorem. Let us say that a pair of rows r and s of P, and a column j of P, are *incident* with each other if column j contains 1's in rows r and s. We show that every pair of rows of P is incident with at most one column of P.

Suppose that there is a pair of rows r and s of P, such that there are two columns of P incident with these rows. Assume the first column contains 1's in rows a, r, and s, and the second contains 1's in rows r, s, and b. Then the sum of these two columns is a vector with 1's in rows a and b. If we add to this vector the column of I containing a 1 in row a, and the column of I with a 1 in row b, then the result is the zero vector. Thus there is a set of columns of H consisting of two columns of P and two columns of I that is not linearly independent. It follows that there is a 4-erasure that is not bad and that cannot be corrected, which is a contradiction.

Each column of P has weight three, and so is incident with exactly three pairs of rows. We have just proved that each pair of rows of P is incident with at most one column of P. It follows that if the number of rows of P is c, then the number of columns of P is at most $\binom{c}{2}/3$. Thus the check disk overhead of the code is at least $6/(c-1)$. $\square$

Definition: The *Steiner code* is based on a *Steiner triple system* (see, for example, [Bollobás86]). 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 (a *triple*), and every pair of elements x, y in X appears in exactly one subset X_i $(i \in \{0,1, \ldots, (c(c-1)/6)-1\})$.

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 when c is 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 $c > 3$, $\mathit{Steiner}(c) = \mathit{Steiner}(c/3) \cup c/3 + \mathit{Steiner}(c/3) \cup 2c/3 + \mathit{Steiner}(c/3) \cup$

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

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

For example, $Steiner(9)$ is

$$\{ \{0,1,2\}, \{3,4,5\}, \{6,7,8\}, \{0,3,6\}, \{1,4,7\}, \{2,5,8\},$$

$$\{0,4,8\}, \{1,5,6\}, \{2,3,7\}, \{0,5,7\}, \{1,3,8\}, \{2,4,6\} \}$$

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

The steiner code is defined by the parity check matrix $H_{steiner} = [P_{steiner} \mid I]$, which has the following form. Number the rows of $P_{steiner}$ from 0 to $c-1$. $P_{steiner}$ consists of all distinct columns that contain 1's in rows q, r, and s, where $\{q, r, s\}$ is a subset in $Steiner(c)$.

The columns of $P_{steiner}$ are a subset of the columns of P_{full3}. Since the full-3 code is 3-erasure-correcting, so is the steiner code. By the definition of a Steiner triple system, for every pair of rows of $H_{steiner}$, there is exactly one column of $P_{steiner}$ that contains 1's in that pair of rows. Since each column of $P_{steiner}$ contains three 1's, each will be selected by a pair of rows in $H_{steiner}$ $\binom{3}{2}$ times. Therefore, there are $\binom{c}{2}/\binom{3}{2}$ columns in $P_{steiner}$, and it follows that the check disk overhead of the code is $6/(c-1)$. Figure 6 shows a Steiner parity check matrix with 9 check disks constructed from our example, $Steiner(9)$.

Theorem 4: The steiner code corrects all sets of 4-erasures except bad 4-erasures.

Pf: We know that the steiner code is 3-erasure correcting; thus, any set of three columns of $H_{steiner} = [P_{steiner} \mid I]$ is linearly independent. It is therefore sufficient to prove the following 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.

The proof of our 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 of weight 3 corresponding to an information bit, and the others corresponding to the three associated check bits. Assume that $c > 3$, that c is a power of 3, and that the claim holds for all smaller powers of 3.

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 $k \in \{0, c/3, 2c/3\}$, and those that contain one element from each of $\{0, 1, 2, \ldots, c/3-1\}$, $\{c/3, c/3+1, .., 2c/3-1\}$, 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 two of them as examples.

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

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 $\{0, c/3, 2c/3\}$. 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 $\{k, k+1, \ldots, k+c/3-1\}$ $(k \in \{0, c/3, 2c/3\})$. 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. Because $Steiner(c/3)$ is 3-erasure-correcting, the columns corresponding to these (at most three) triples cannot sum to zero. The remaining columns do not contain 1's in rows $\{k_1, k_1+1, \ldots, k_1+c/3-1\}$ and, 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

$$\{r_1, s_1, t_1\}, \ \{r_2, s_2, t_2\}, \ \{r_3, s_3, t_3\}, \text{ and } \{r_4, s_4, t_4\},$$

where

$$r_k \in \{0, 1, \ldots, c/3-1\},\ s_k \in \{c/3, c/3+1, \ldots, 2c/3-1\},\ \text{and}\ t_k \in \{2c/3, 2c/3+1, \ldots, c-1\}$$

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

$$\{i, c/3 + ((i+j) \bmod c/3), 2c/3 + ((i+2j) \bmod c/3)\}$$

for some $i, j \in \{0, 1, \ldots, c/3-1\}$. Note that $i = 2(i+j) - (i+2j)$, which implies that for $k \in \{1, 2, 3, 4\}$,

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

Therefore,

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

and

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

As a consequence, $2t_1 \equiv (2t_2) \bmod c/3$. 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. $\square$

Definition: The *additive-3 code* is defined by the parity check matrix $H = [P_{additive\,3} \,|\, I]$, which has the following form. Choose c, the number of check bits, to be an odd multiple of 3. Number the rows of $P_{additive\,3}$ from 0 to $c-1$. $P_{additive\,3}$ 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 columns of $P_{additive-3}$ are a subset of the columns of P_{full-3}, so the additive-3 code is 3-erasure-correcting. To calculate the number of information bits of the additive-3 code, note that if the variables q and r are assigned values from $[0...c-1]$, then there is precisely one assignment to s from $[0...c-1]$ such that $q + r + s \equiv 1 \bmod c$. Thus there are c^2 ways of assigning not necessarily distinct values to q, r, and s such that $q + r + s \equiv 1 \bmod c$. In any such assignment, at most two of the variables can have the same assignment, because if all three had the same assignment, x, then $3x \equiv 0 \bmod c$. It follows that there are $c^2 - 3c$ assignments satisfying $q + r + s \equiv 1 \bmod c$ such that the assignments to q, r, and s are distinct. Dividing by the number of permutations of q, r, and s, we find that the number of information bits of the additive-3 code is $(c^2 - 3c)/6$; hence, the check disk overhead is $6/(c-3)$.

Theorem 5: The additive-3 code corrects all sets of 4-erasures except bad 4-erasures.

Pf: The additive-3 code is 3-erasure-correcting. 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 $P_{additive\,3}$ is 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

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

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:

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

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

These equations imply that

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

Subtracting the two equations yields $b-q \equiv (q-b) \mod c$ which implies that $2(b-q) \equiv 0 \mod c$. Because c is odd, the only number between 0 and $c-1$ which satisfies the equation $2x \equiv 0 \mod c$ 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 four columns of $P_{additive\,3}$ cannot sum to zero.

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

7. 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 check disk overhead. Any *t*-erasure-correcting code with parity check matrix $H = [P \mid I]$ and minimum possible update penalty 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, so its average group size is $(tk + c)/c$. This can be expressed as:

$$\textit{check disk overhead} \times (\textit{average group size} - 1) = t \ .$$

Therefore, in any *t*-erasure-correcting code with minimum possible update penalty, if k is large relative to c, the check disk overhead (c/k) will be small, but the average group size will be large. Fortunately, it is possible to derive new codes from the full-2 and full-3 codes which will allow us to trade check 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 be 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. We will do this by seeking particular orderings for the columns of a code's parity matrix, which we call balanced orderings, that insure that all codes derived from this parity matrix by deleting information disk columns from the right will have average and maximum group size close together.

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 five codes discussed in the previous section with examples in Figures 5 and 6 all have balanced group size. Let $H = [P \mid I]$ be the parity check matrix of a code with balanced group size. We say the the columns of H are arranged in a *balanced ordering* if for any i (less than the number of columns in P), a new code with information columns identical to the first i columns of P will have balanced group size. In Theorem 6, we prove that it is possible to achieve a balanced ordering for the full-2, full-3, 2d-parity, steiner, and 3d-parity codes. Thus codes with balanced group size can also be derived from these codes by deleting, from the right, the columns of each code's P matrix.

Theorem 6: There exist balanced orderings of the full-2, full-3, 2d-parity, 3d-parity, and steiner codes.

Pf: Suppose c is divisible by t. Consider a set of distinct columns of length c and weight t. Number the positions in the columns from 0 to $c-1$. A 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.*

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.

A theorem of Baranyai states that if t divides c, then there exists a factorization of the set consisting of all columns of length c and weight t [Brouwer79a, Bollobás86]. This theorem implies that it is possible to achieve a balanced ordering of the full-2 code when c is even, and of 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 by Bollobás [Bollobás86]. This construction can be modified slightly to produce a balanced ordering of the full-2 code when c is odd. A more general theorem of Baranyai [Brouwer79a] (which essentially implies the existence of an "approximate factorization") can be used to show that the full-3 code also has a balanced ordering when c is not divisible by 3.

In the 2d-parity code, factorizing the columns of P is trivial. We omit the proofs that the columns of P are also factorizable in the steiner code and the 3d-parity code. □

We show balanced orderings for the full-2, 2d-parity, full-3, and steiner codes in Figures 5 and 6. In Theorem 7 we prove there is no balanced ordering of the additive-3 code. However, it may be possible to produce orderings that are almost balanced, and so this theorem does not in itself indicate a significant deficiency of the code.

Theorem 7: There is no balanced ordering of the additive-3 code.

Pf: Suppose there is a balanced ordering. Consider the submatrix P' made up of the first $c/3$ columns of the ordering. Since each has weight 3, the total weight of the columns is c and the weight of every row is 1. The rows of P' are numbered 0 through $c-1$. Each column corresponds to a triple r, s, t such that the column contains 1's in rows r, s, and t and $q + r + s \equiv 1 \bmod c$. Let x be the sum of the entries in the triples associated with the columns of P'. Since the weight of each row in P' is exactly 1, x is the sum of the integers from 0 to $c-1$. That is, $x = c(c-1)/2$. The sum of the entries in each triple is congruent to 1 mod c. Since there are $c/3$ triples, it follows that $x \equiv c/3 \bmod c$. Therefore $c(c-1)/2 \equiv c/3 \bmod c$. But $c(c-1)/2 \equiv 0 \bmod c$ because by assumption, c is odd. Contradiction. □

In addition to reducing group size, balanced orderings are useful for designing extensible I/O systems. Recall, from Section 4, that if we have chosen a code with more columns than we need, as disks are added to the system the extra columns are put to use. If the original code has a balanced ordering and disks are associated with columns according to this ordering, then at all times the group sizes are balanced.

8. Evaluation of Double-Erasure-Correcting and Triple-Erasure-Correcting Codes

As we mentioned in Figure 1, a simple implementation for repair is periodic visits (for example, daily or weekly) by maintenance personnel. 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 independent, exponential disk lifetimes with mean M, we can calculate the probability of y failures (erasures) in a repair period, T,

$$\binom{N}{y}(1-e^{-T/M})^{y}\,(e^{-T/M})^{(N-y)}$$

where N is the total number of disks. We have used Monte Carlo simulation on the columns of a code's parity check matrix to estimate the fraction of y failures (y-erasures) 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 in this period times the fraction of y failures that are unrecoverable. The mean number of repair periods until data loss is then 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 inexpensive disks; that is, 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 3d-parity and steiner codes share this very high reliability.

Table 1 reports higher MTTDL for the 2d-parity code than for the full-3 code. This may seem counter-intuitive, because the latter corrects all triple erasures and the former does not. We can prove, however, that the 2d-parity code has a smaller fraction of unrecoverable 4-erasures than the full-3 code (for the disk arrays treated in Table 1). Moreover, based on Monte Carlo simulations, we conjecture that for all m larger than three, the 2d-parity code has a smaller fraction of unrecoverable m-erasures than the full-3 code. Note that in longer repair periods the likelihood of having greater than three erasures will increase, and thus the fraction of unrecoverable m-erasures (for $m > 3$) will become more significant to determining MTTDL. We believe that these larger erasure sets explain the higher MTTDL of the 2d-parity code reported in Table 1.

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. This support hardware is likely to be shared among a subset of the disks. If parity groups are organized orthogonally to support hardware groups, data redundancy provides protection against support hardware failures as well as catastrophic disk failures [Gibson92, Gibson93]. This technique is easily extended for the 2d-parity code by organizing support hardware groups on the diagonals of the 2d array [Newberg93].

As this section has shown, 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. In this case a disk array’s reliability depends on the spare pool replenishing process [Gibson92, Gibson93].

9. Codes for Correcting more than Three Erasures

As we have mentioned, we do not envision the use in disk arrays of t-erasure-correcting codes for $t>3$. From a theoretical standpoint, however, we are interested in trying to generalize our work on 2-erasure-correcting and 3-erasure-correcting codes to apply to t-erasure-correcting codes. Designing t-erasure-correcting codes is more difficult than designing 2-erasure-correcting and 3-erasure-correcting codes. It is not always possible to extend the 2-erasure-correcting and 3-erasure-correcting codes for arbitrary t. For example, we might try and define the full-t code with parity check matrix $H=[P \mid I]$, where P consists of all distinct columns of weight t. However, such a code is not t-erasure correcting when $t>3$, because there are sets of four columns whose sum is zero (namely sets consisting of two columns of P whose sum has weight two, and the two appropriate columns of I).

The t d-parity code discussed in Section 3 is t-erasure-correcting and has minimum update penalty, but its check disk overhead is high, $(t \cdot (t/c)^{\frac{1}{t-1}})$.

One approach to designing t-erasure-correcting codes is suggested by the following theorem.

Theorem 8: Let $H = [P \mid I]$ be the parity check matrix of a code such that all columns of P have weight t, and for each pair of rows of P, there is at most one column of P containing 1's in that pair of rows. Then the code defined by H is t-erasure-correcting.

Pf: Consider a set S consisting of j columns of H, where $j \le t$. We show that the sum of the columns of S is a column with nonzero weight, which will prove that the code defined by H is t-erasure-correcting. If S contains only columns from I, then the sum of the columns in S is a column of weight j. Suppose, therefore, that S contains at least one column q from P. The weight of this column is t. By assumption, for each of the other $j-1$ columns in S, there is at most one row in which both q and that other column contain a 1. So there are at most $j-1$ rows in which both q and some other column in S contain a 1. Therefore, there is at least one row in which q contains a 1, and no other column in S contains a 1. It follows that the sum of the columns in S is a column of weight at least 1. $\square$

The number of information disks in a code of the type treated in Theorem 8 is at most $\binom{c}{2}/\binom{t}{2}$, and thus the check disk overhead is at least $t(t-1)/(c-1)$. For a given number c of check disks, the problem of designing a code of the type treated in Theorem 8 that achieves this check disk overhead is equivalent to the following block design problem – given a c element set X, find a collection of subsets of X of size t such that each pair of elements in X occurs in exactly one subset. Two necessary conditions for the existence of such a design are that $t(t-1)$ divides $c(c-1)$, and that $(t-1)$ divides $(c-1)$. These conditions are known to be sufficient for $t = 4$ and $t = 5$ [Hanani61, Hanani75]. Thus, there exist 4-erasure-correcting and 5-erasure-correcting codes that achieve a check disk overhead of $t(t-1)/(c-1)$ (whenever t and c satisfy the two conditions above). These codes have much lower check disk overhead than the 4d-parity and 5d-parity codes. For example, there is a 4-erasure-correcting code with $c = 109$, $k = 981$ and check disk overhead $1/9$, found using block designs. The nearest 4d-parity code with at least as many data disks has $c = 864$, $k = 1296$ and check disk overhead $2/3$.

Unfortunately, it is not known whether the above two conditions are always sufficient for $t > 5$. It is only known that they are sufficient provided that c is *sufficiently large* by a theorem of Wilson, a proof of which can be found in [Brouwer79b].

For $t > 3$, it is an open question to determine the minimum possible check disk overhead of a t-erasure-correcting code with minimum update penalty. We know from the above results that for $t = 4$ or $t = 5$, a check disk overhead of $t(t-1)/(c-1)$ can be achieved, and so the minimum possible check disk overhead is no more than this quantity.

The td-parity code can be shown to correct all sets of $t+1$-erasures except bad $t+1$-erasures. For $t > 3$, it is an open question to design t-erasure-correcting codes that correct all sets of $t+1$-erasures except bad $t+1$ erasures, and achieve low check disk overhead.

10. Nonbinary Symbol Codes

In this paper we have restricted our attention to coding schemes based on binary codes. Schemes based on binary codes treat a disk array as a stack of codewords, each codeword composed of one bit from each disk. This allows modifications or reconstructions to be done on units as small as one bit on a disk. The minimum unit of disk access, called a sector, is usually at least 128 bytes, so one need not require that a codeword include only one bit per disk as long as the bits it includes are in the same sector. Therefore, it is also possible to consider schemes based on nonbinary codes. Such schemes have advantages and disadvantages relative to schemes based on binary codes. We discuss some of these briefly.

As we mentioned in Section 2, an important advantage of schemes based on binary codes is that check data can be manipulated efficiently, because computations are done in the field GF[2]. In schemes based on nonbinary codes, computations are done in the field GF[2^b], where $b \neq 1$ is the number of bits in a symbol, rather than in GF[2]. Computation in GF[2^b] is not difficult to design, but it is much more expensive than computation in GF[2], and its cost grows with b.

Nonbinary schemes do have the advantage that it is possible to reduce check disk overhead by increasing b. If the symbol size is permitted to be unbounded, then any number of information disks can be protected from all double erasures with only two check disks using, for example, a two-check-symbol nonbinary Hamming code, or a two-check-symbol Reed-Solomon code [MacWilliams77]. The availability of compact encoder/decoder chip sets for Reed-Solomon codes and the wide range of industrial experience with these codes has led to the use of variations of the two-check-symbol Reed-Solomon code in disk array products under the name of *P+Q parity* [ATC90].

A problem in attaining very low check disk overhead through nonbinary codes with only two check symbols is the reconstruction group size; any single disk failure will involve all disks in its reconstruction. Reducing reconstruction group size by increasing the number of check disks introduces another problem; with more than two check symbols, these codes will generally not have the minimal update penalty of two. However, there is a nonbinary analogue to the binary full-2 code. This nonbinary full-2 code protects up to $c(c-1)(2^b-1)/2$ data symbols against double erasure with c check symbols [Gibson92, Peterson72].

In using nonbinary analogues of our binary codes to achieve low disk overhead, we may decrease reliability by introducing new sets of $t+1$-erasures that are not bad, and not correctable. For example, a nonbinary 2-erasure-correcting code achieves low disk overhead by allowing more than one data disk to share the same pair of check disks. The loss of two data disks that share the same pair of check disks, and the loss of one check disk from this pair, constitute a 3-

erasure that is uncorrectable, but is not bad.

Existing nonbinary implementations correct two erasures with two check disks and are intended for very highly reliable, but small, disk arrays. Further research on nonbinary schemes for large disk arrays is needed.

11. 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. In Table 2, we summarize the characteristics of the main codes discussed in this paper. Our codes all minimize the number of check disks that must be updated whenever an information disk is updated. Beyond this requirement we 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.

12. 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. This paper was prepared while L. Hellerstein was at U. C. Berkeley and at the Massachusetts Institute of Technology and while G. Gibson was at U. C. Berkeley and at Carnegie Mellon University. The work described here was supported in part by the National Science Foundation under grant no. MIP-8715235 and grant no. CCR-8411954, as well as an AT&T Bell Labs GRPW grant, a Siemens Corporation grant and an IBM graduate fellowship.

13. References

[ATC90] *Product Description, RAID+ Series Model RX*, Revision 1.0, Array Technology Corporation, Boulder CO 80301, February 1990.

[Bates89] Bates, K. H. ‘‘Performance aspects of the HSC controller,’’ *Digital Technical Journal,* Vol. 8, February 1989.

[Bitton88] Bitton, D., J. Gray, ‘‘Disk shadowing,’’ *Proc. of the 14th Int. Conf. on Very Large Data Bases (VLDB),* 1988.

[Bell85] Bell, C. G.,‘‘Multis: a new class of multiprocessor computers,’’ *Science*, Vol. 228, April 1985.

[Bell89] Bell, C. G., ‘‘The future of high performance computers in science and engineering,’’ *Comm. of the ACM,* Vol. 32, No. 9, September 1989.

[Bollobás86] Bollobás, 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.

[Brouwer79a] Brouwer, A. E., Schrijver, A., ‘‘Uniform Hypergraphs,’’ *Packing and Covering in Combinatorics*, Schrijver, A., ed., Mathematical Centre Tracts 106, Mathematisch Centrum, Amsterdam 1979.

[Brouwer79b] Brouwer, A. E., ‘‘Wilson’s theory,’’ *Packing and Covering in Combinatorics,* Schrijver, A., ed., Mathematical Centre Tracts 106, Mathematisch Centrum, Amsterdam 1979.

[Chen90] Chen, Peter M., David A. Patterson, ‘‘Maximizing performance in a striped disk array,’’ *Proc. of the 1990 ACM SIGARCH 17th Ann. Int. Symp. of Computer Architecture,* Seattle WA, May 1990.

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

[Gelsinger89] Gelsinger, Patrick P., Paola A. Gargini, Gerhard H. Parker, Albert Y. C. Yu, ‘‘Microprocessors circa 2000,’’ *IEEE Spectrum,* October 1989.

[Gibson89] Gibson, Garth A., Lisa Hellerstein, Richard M. Karp, Randy H. Katz, David A. Patterson, ‘‘Coding techniques for handling failures in large disk arrays,’’ *Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* Boston MA, April 1989.

[Gibson92] Gibson, Garth A., *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, M.I.T. Press, 1992.

[Gibson93] Gibson, Garth A., David A. Patterson, ‘‘Designing Disk Arrays for High Data Reliability,’’ *J. of Parallel and Distributed Computing,* Vol. 17, No. 1, January 1993.

[Hanani61] Hanani, H., ‘‘The existence and construction of balanced incomplete block designs,’’ *Ann. of Math. Stat.,* Vol. 32, 1961, pp 361-386.

[Hanani75] Hanani, H., ‘‘Balanced incomplete block designs and related designs,’’ *Discrete Mathematics,* Vol. 11, 1975, pp 255-369.

[Hoagland89] Hoagland, Albert, ‘‘Information storage technology: a look at the future,’’ *IEEE Computer*, Vol. 18, July, 1985.

[Holland92] Holland, Mark, Garth A. Gibson, ‘‘Parity Declustering for Continuous Operation in Redundant Disk Arrays,’’ *Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V),* Boston MA, October 1992.

[Jilke86] Jilke, W., ‘‘Disk array mass storage systems: the new opportunity,’’ Amperif Corp., September 1986.

[Kim87] Kim, Michelle Y., *Synchronously Interleaved Disk Systems with their Application to the Very Large FFT,* PhD Dissertation, Polytechnic University, Janurary 1987.

[Klietz88] Klietz, A., J. Turner, T. C. Jacobson, ‘‘TurboNFS: fast shared access for Cray disk storage,’’ *Proc. of Cray User Group Convention*, April 1988.

[Kryder89] Kryder, Mark H., ‘‘Data storage in 2000 – trends in data storage technologies,’’ *IEEE Trans. on Magnetics,* Vol. 25, No. 6, November 1989.

[Lin88] Lin, Ting-Ting Yao, *Design and Evaluation of an On-Line Predictive Diagnostic System,* PhD Dissertation, Carnegie Mellon University, April 1988.

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

[MacWilliams77] Florence Jessie MacWilliams, Neil James Alexander Sloane, *The Theory of Error-Correcting Codes,* North-Holland Mathematical Library, Vol. 16, Elsevier Science Publishing Company, New York NY, 1977.

[Muntz90] Muntz, Richard R., John C. S. Lui, ‘‘Performance analysis of disk arrays under failure,’’ *Proc. of the 16th Int. Conf. on Very Large Data Bases (VLDB),* Dennis McLeod, Ron Sacks-Davis, Hans Schek (Eds.), Morgan Kaufmann Publishers, August 1990, pp 162-173.

[Myers86] Myers, Glenford J., Albert Y. C. Yu, David L. House, ‘‘Microprocessor technology trends,’’ *Proc. of the IEEE,* Vol. 74, No. 12, December 1986.

[Newberg93] Newberg, Lee, David Wolfe, ‘‘String Layouts for a Redundant Array of Inexpensive Disks,’’ *Algorithmica,* this issue, 1993.

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

[Peterson72] Peterson, W. Wesley, E. J. Weldon, Jr., *Error-Correcting Codes, Second Edition,* M.I.T. Press, 1972, pp 131-136.

[Rabin89] Rabin, Michael O., ‘‘Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance,’’ *J. of the Assoc. for Comp. Mach.,* Vol. 36, 1989.

[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.

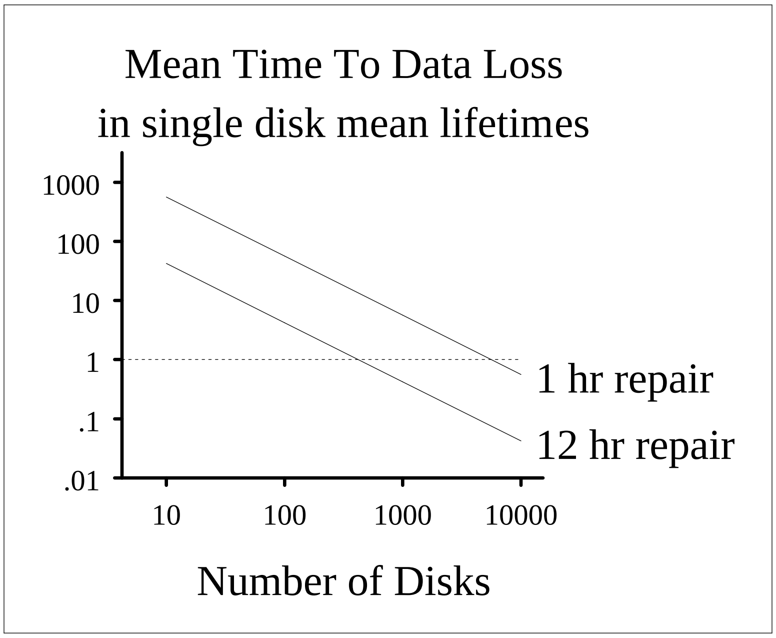

Figure 1. *The mean time to data loss in a single-erasure-correcting array is MTTDL =* $(MTTF_{disk})^2 / (\#Disks \times (Groupsize - 1) \times MTTR)$*, where* $MTTF_{disk}$ *is the mean lifetime of an individual disk, #Disks is the total number of disks, Groupsize is 1 + the number of information (user 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 inexpensive disks whose reliability (*$MTTF_{disk}$ *= 50,000 hours) is less than the best available today, 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 on-line ''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.*

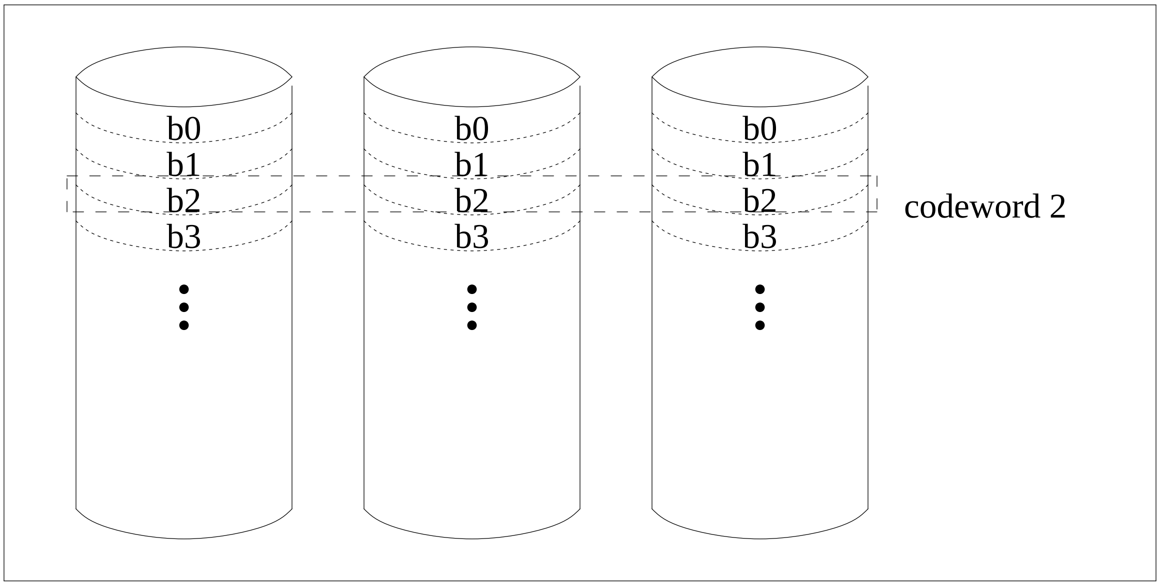

Figure 2. *A codeword consists of the set of bits at the same position on each disk. A code determines which bits in a codeword are check bits and how to calculate these from the other information bits.*

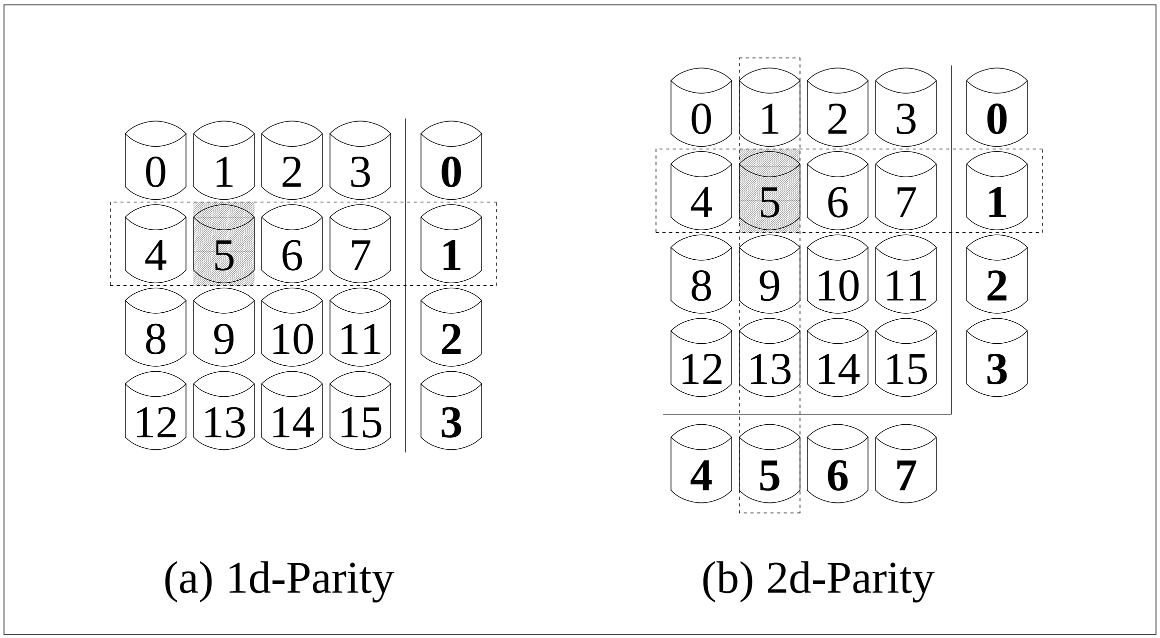

Figure 3. *With parity computed only along rows of disks we have a standard single-erasure-correcting coding scheme called 1d-parity, and with parity computed both along rows and along columns we have a standard double-erasure-correcting coding scheme called 2d-parity. In this example we have used groups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

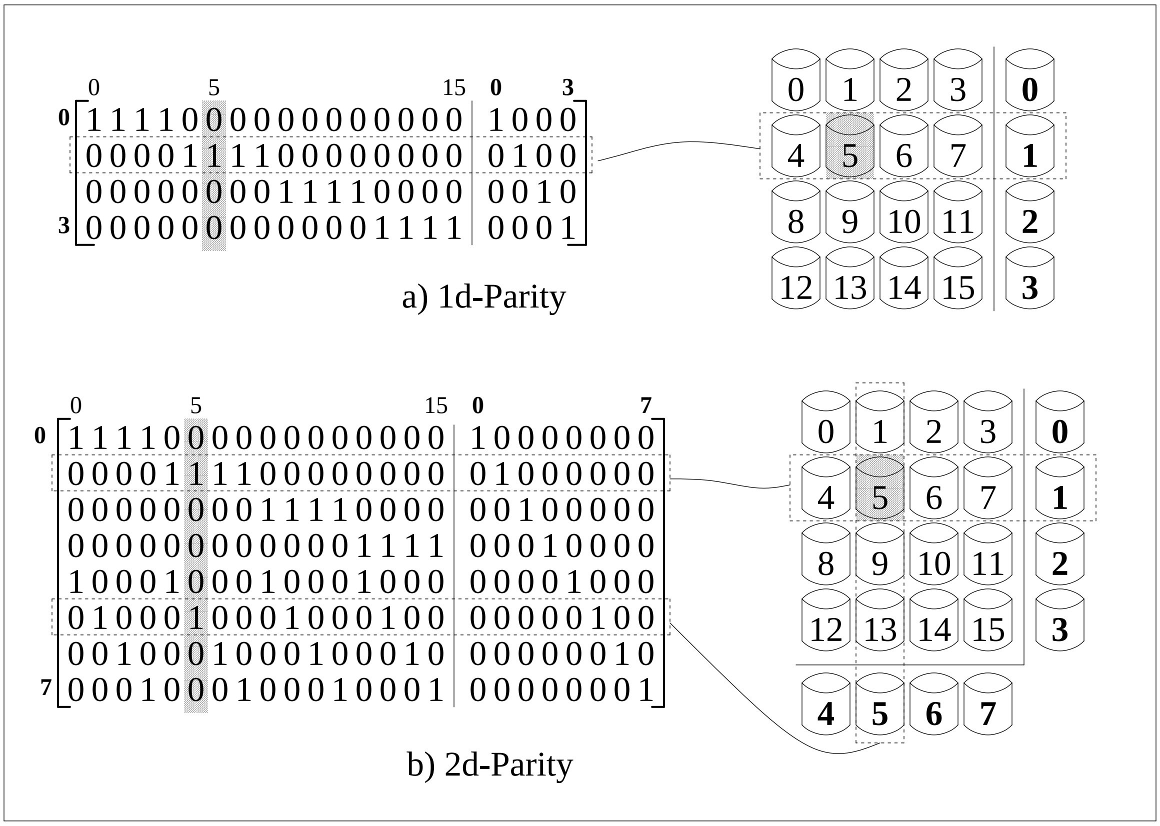

Figure 4. *We contrast the parity check matrices for 1d-parity and 2d-parity to the physical models given in Section 3 for groups of 4 information disks. In both cases disk 5 is shaded so you can see the column that represents it and the rows that represent the groups that contain it.*

$$\begin{array}{c}
\begin{array}{l}
0 \qquad\qquad\qquad\qquad\qquad\qquad 14 \quad \mathbf{0} \qquad\qquad \mathbf{5}
\end{array} \\
\left[\begin{array}{ccccccccccccccc|cccccc}
1&0&0&1&0&0&1&0&0&1&0&0&1&0&0&1&0&0&0&0&0\\
0&1&0&1&0&0&0&1&0&0&1&0&0&1&0&0&1&0&0&0&0\\
0&0&1&0&1&0&0&1&0&0&0&1&1&0&0&0&0&1&0&0&0\\
0&1&0&0&0&1&1&0&0&0&0&1&0&0&1&0&0&0&1&0&0\\
1&0&0&0&1&0&0&0&1&0&1&0&0&0&1&0&0&0&0&1&0\\
0&0&1&0&0&1&0&0&1&1&0&0&0&1&0&0&0&0&0&0&1
\end{array}\right] \\
P_{full2} \qquad\qquad\qquad\qquad I_6
\end{array}$$

Full-2 (k=15,c=6)

$$\begin{array}{c}
\begin{array}{l}
0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad 15 \quad \mathbf{0} \qquad\qquad\quad \mathbf{7}
\end{array} \\
\left[\begin{array}{cccccccccccccccc|cccccccc}
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0\\
0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0\\
1&0&0&0&0&0&0&1&0&0&1&0&0&1&0&0&0&0&0&0&1&0&0&0\\
0&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&0&0\\
0&0&1&0&0&1&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0\\
0&0&0&1&0&0&1&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&1
\end{array}\right] \\
P_{2dparity} \qquad\qquad\qquad\qquad I_8
\end{array}$$

2d-Parity (k=16,c=8)

Figure 5. *Example parity check matrices for the two 2-erasure-correcting codes discussed in Section 6 are shown in this figure. Notice that although the full-2 code has 1 fewer information disk, it also has 2 fewer check disks, 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 Figure 4; the order of columns in these examples is explained in Section 7.*

$$\begin{array}{cc} 0 \hfill 19 & \mathbf{0} \hfill \mathbf{5} \\ \multicolumn{2}{c}{\left[\begin{array}{c|c} 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0 & 1\,0\,0\,0\,0\,0 \\ 1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1 & 0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\,0 & 0\,0\,1\,0\,0\,0 \\ 0\,1\,1\,0\,1\,0\,0\,1\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,0\,1 & 0\,0\,0\,1\,0\,0 \\ 0\,1\,1\,0\,0\,1\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,0\,1 & 0\,0\,0\,0\,1\,0 \\ 0\,1\,0\,1\,1\,0\,1\,0\,0\,1\,0\,1\,1\,0\,0\,1\,0\,1\,1\,0 & 0\,0\,0\,0\,0\,1 \end{array}\right]} \\ P_{full\,3} & I_6 \end{array}$$

Full-3 (k=20,c=6)

$$\begin{array}{cc} 0 \hfill 8 & \mathbf{0} \hfill \mathbf{8} \\ \multicolumn{2}{c}{\left[\begin{array}{c|c} 0\,0\,0\,0\,0\,1\,1\,1\,0 & 1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,0\,0\,0\,0 & 0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,1\,0\,0\,0 & 0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,1\,0\,1\,0\,0 & 0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,1\,1 & 0\,0\,0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,1\,1\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 1\,0\,1\,0\,0\,0\,0\,1\,0 & 0\,0\,0\,0\,0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1\,0\,1 & 0\,0\,0\,0\,0\,0\,0\,1\,0 \\ 1\,0\,0\,0\,0\,1\,0\,0\,1 & 0\,0\,0\,0\,0\,0\,0\,0\,1 \end{array}\right]} \\ P_{additive\,3} & I_9 \end{array}$$

Additive-3 (k=9,c=9)

$$\begin{array}{cc} 0 \hfill 11 & \mathbf{0} \hfill \mathbf{8} \\ \multicolumn{2}{c}{\left[\begin{array}{c|c} 1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0 & 1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0 & 0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,1\,0\,0\,1\,0\,0\,1 & 0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0\,0\,0\,1\,0\,1\,0 & 0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,0\,1 & 0\,0\,0\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,1\,0\,1\,0\,1\,0\,0 & 0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,0\,1 & 0\,0\,0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,1\,0\,1\,0\,0\,0\,1\,1\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1\,1\,0\,0\,0\,1\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,1 \end{array}\right]} \\ P_{steiner} & I_9 \end{array}$$

Steiner (k=12,c=9)

Figure 6. *Example parity check matrices for the three 3-erasure-correcting codes discussed in Section 6 are shown in this figure.*

Code	Update Penalty	Check Disk Overhead	Group Size	MTTDL (years) Daily Repair	Weekly Repair
nonredundant	0	0%	–	0.006	0.006
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
full-3	3	1.8%	172	15,000	38
3d-parity	3	30.0%	10	$> 10^6$	$> 10^4$
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 3 and 6 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. A single disk has MTTF of 50,000 hours, about 5.7 years.

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

Table 2: Characteristics of Erasure Correcting Codes Discussed.