Ordering Disks for Double Erasure Codes

Myra B. Cohen
Computer Science Department
University of Auckland
Private Bag 92019
Auckland, New Zealand
myra@cs.auckland.ac.nz

Charles J. Colbourn
Computer Science Department
University of Vermont
Burlington, VT 05405
U.S.A.
colbourn@cs.uvm.edu

ABSTRACT

Disk arrays have been designed with two competing goals in mind, the ability to reconstruct erased disks (reliability), and the speed with which information can be read, written, and reconstructed (performance). The substantial loss in performance of write operations as reliability requirements increase has resulted in an emphasis on performance at the expense of reliability. This has proved acceptable given the relatively small numbers of disks in current disk arrays. We develop a method for improving the performance of write operations in disk arrays capable of correcting any double erasure, by ordering the columns of the erasure code to minimize the amount of parity information that requires updating. For large disk arrays, this affords a method to support the reliability needed without the generally accepted loss of performance.

1. INTRODUCTION

There has been a sustained exponential improvement in the density and performance of semiconductor technology. This has brought faster microprocessors along with larger and faster primary memory devices. Improvements in secondary storage systems have not kept pace. The performance of RISC microprocessors has been increasing by more than 50% per year [18]; disk transfer rates have only improved by about 20% each year [5]. This has transformed many CPU-bound applications to I/O-bound. Amdahl [2] predicted three decades ago that, unless accompanied by corresponding increases in secondary storage performance, substantial increases in microprocessor performance only bring about marginal improvements in overall system performance. Solutions for this problem exploit parallelism. The most successful of these is the *disk array architecture*. This organizes many independent small disks into one large logical disk. Disk arrays improve performance by employing *data striping* [19], which spreads data to multiple disks. A single I/O operation at the user level is mapped by software into a number of independent operations on the stripe units. Then I/O requests can be processed in parallel by separate disks, improving effective transfer rates. Disk striping also enhances uniform load balance.

Many applications, such as database and transaction processing systems, require both high throughput and high data availability of their storage systems. The most demanding of these applications require continuous operation. In terms of a storage system, this requires that we satisfy all requests for data even in the presence of disk failures. We must also reconstruct the content of a failed disk onto a replacement disk, restoring it to a fault-free state. The more disks we have in a disk array, the higher the performance we obtain. Unfortunately, large disk arrays have low reliability. These requirements underpin the introduction of redundancy to tolerate disk failures. Disk arrays which incorporate redundancy are known as *Redundant Arrays of Independent Disks* (RAID). Redundancy does not come without cost, however. Typically, as we increase the reliability of our system, we reduce the performance. This paper examines one method of minimizing this cost when adding redundancy.

1.1 Reliability vs. Performance

When a disk suffers a catastrophic failure, its data is rendered unreadable, and it is effectively erased. We therefore call such a disk failure *an erasure*. For convenience, we also call a set of k disk failures *a k-erasure*. Components in disk arrays allow us to determine exactly where erasures have occurred.

The performance of basic disk operations, reads and writes, can be assessed by examining the *user response time*, the largest component of which is the number and size of disk accesses required. Arrays are typically designed under the assumption that read operations in the absence of any disk failures are the most common, while write operations in fault-free mode are the next most common. Disk failures do occur, however. *Reliability* assesses the ability of the disk array to restore the contents of one or more erased disks, but the performance issues now involve more than reads and writes. When disks fail, performance is impacted by the cost of reconstructing these disks, either online or offline.

Multiple erasure disk arrays are often designed with the primary goals of minimizing costs for reads and writes in

fault-free mode, and the secondary goal of minimizing reconstruction costs for erased disks. However, we argue that an additional secondary goal should be to improve performance of read and write operations further, by a judicious choice of the erasure code used in the disk array. In this way, reliability and performance objectives do not need to be in disagreement.

Hellerstein et al. [11] pioneered the study of erasure-resilient codes for large disk arrays. They examined structural conditions which guarantee high erasure correction capability, a theme which is pursued in [4, 7]. They also undertook an analysis which reveals the need for multiple erasure correction when the number of disks is 'large'. They showed that an array of one thousand disks which protects against one error, even with periodic daily or weekly repairs, has a lower reliability than an individual disk. Their analysis assumes that erasures are detected when they occur; the truth for disk arrays is worsened if there are latent sector faults, which may remain undetected for long periods; see [1]. As arrays grow in size, the need for greater redundancy without a reduction in performance becomes apparent.

Reliability is evidently a major concern. However, the tradeoff between reliability and performance has typically been resolved in favor of performance, and consequently multiple erasure systems are often dismissed as not commercially viable. Most currently available systems handle only one disk failure [16]. Improvements in the performance of reconstruction have been the subject of an excellent analysis [20], but it is the 'poor' performance of basic read and write operations in a fault-free environment which have limited the practical use of multiple erasure systems. We therefore focus on how the choice of erasure code, and the manner in which it is used, affects performance of basic operations, especially in fault-free mode.

2. PRELIMINARIES

First we introduce a general framework for erasure codes. Let $\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$. The *weight* of $\mathbf{x}$, denoted $\mathrm{wt}(\mathbf{x})$, is the number $\sum_{i=1}^{n} x_i$. The *support* of $\mathbf{x}$, denoted $\mathrm{supp}(\mathbf{x})$, is the set $\{i \mid x_i = 1\}$. A *stripe unit* is the minimum amount of contiguous user data allocated to one disk before any data is allocated to any other disk. The size of a stripe unit must be an integral number of sectors, and is often the minimum unit of update used by system software. Because of this, we can view each disk as a collection of (disjoint) stripe units. A *parity stripe*, or simply *stripe*, is a selection of one stripe unit from each disk, usually at the 'same' physical address; see [10]. An $[n, c, k]$-*erasure-resilient code*, or briefly an $[n, c, k]$-*ERC*, consists of an encoding algorithm $\mathcal{E}$ and a decoding algorithm $\mathcal{D}$ with the following properties. Given an n-tuple S, $\mathcal{E}$ produces an $(n + c)$-tuple $\mathcal{E}(S) = (\mathcal{E}_1(S), \ldots, \mathcal{E}_{n+c}(S))$ (a stripe), called a *codeword*, such that for any $I \subseteq \{1, \ldots, n\}$, where $|I| = n+c-k$, the decoding algorithm $\mathcal{D}$ is able to recover S from $(I, \{\mathcal{E}_i(S) \mid i \in I\})$. We call an $[n, c, k]$-ERC a k-ERC when the parameters n and c are not needed.

To see the relevance of $[n, c, k]$-ERCs to the protection of data loss in a RAID, suppose that we have data which is partitioned into an n-tuple S of stripe units. We encode S into a codeword $(\mathcal{E}_1(S), \ldots, \mathcal{E}_{n+c}(S))$ using an $[n.c, k]$-ERC, Then for $1 \leq i \leq n+c$, we store $\mathcal{E}_i(S)$ on disk i of a disk array with $n + c$ disks. The definition of an $[n, c, k]$-ERC ensures that we can reconstruct the original data in the presence of up to k erasures.

For performance reasons, the codes that we study satisfy two conditions, as in [11]. First, we restrict our attention to systematic codes. An $[n, c, k]$-ERC is *systematic* if $\mathcal{E}_i(S) = S_i$, for $1 \leq i \leq n$, where $S = (S_1, \ldots, S_n)$. The stripe units $\mathcal{E}_i(S)$, for $n < i \leq n + c$, are *checks*. Using a systematic code, the encoding function leaves the data unmodified on some disks, to avoid read penalties when there are no disk failures. Secondly, we restrict ourselves to *linear* codes. Encoding a stripe employs component-wise modulo two arithmetic (exclusive-or, or parity) $\oplus$, so that it can be performed efficiently.

The first restriction allows us to separate disks into *information disks*, which contain the original data, and *check disks*, which contain the checks. In fact, the two restrictions together ensure that an $[n, c, k]$-ERC can be described in terms of a $c \times (n + c)$ matrix $H = [C \mid I]$ over $\mathbb{F}_2$, where I is the $c \times c$ identity matrix and C is a $c \times n$ matrix that determines the equations for the checks. This is a well-known result in the theory of error-correcting codes [15, 22]. The matrix H is the *parity-check matrix* of the code; see Figure 1 for an example, in which two different orderings of the columns of the same parity check matrix are shown. Given the parity-check matrix $H = [C \mid I]$ of a k-ERC, we can think of the rows of C (as well as the rows and columns of I) as being indexed by the check disks of a disk array, and the columns of C as being indexed by the information disks. The content of check disk i is the modulo two sum of the content of the information disks whose corresponding columns in C have a one in row i.

2.1 Metrics

One metric of an erasure-resilient code is particularly important: The *update penalty* is the number of check disks whose content must be changed when an update is made in the content of a given information disk. If m check disks are involved in a write, then the parallelism of the disk array is reduced by a factor of $m + 1$. So update penalties must be kept as small as possible. The update penalties of an erasure-resilient code with parity-check matrix $H = [C \mid I]$ are the column sums of C. Since updates of data are usually much more frequent than the reconstruction of data due to erasures, the update penalties are typically of most concern.

If an erasure-resilient code is able to tolerate all k-erasures, then every update must affect the content of at least $k + 1$ disks (one information disk and k check disks). Thus, the update penalties of a k-ERC are at least k. (A more general result is proved in [20].) In view of the importance of minimizing update penalties, we consider only those k-ERCs for which the update penalties are all equal to k, the minimum possible. We speak, therefore, of the update *penalty*, instead of the update *penalties* of an erasure-resilient code. The corresponding parity-check matrix $H = [C \mid I]$ has column sums for C all equal to k.

Various multiple erasure systems employ an underlying erasure code, but aggregate multiple logical disks onto a single

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	0	0	1	0	0	1	0	1	1	0	1	0	0	0	0
11	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0
12	0	1	1	0	0	0	1	0	0	1	0	0	1	0	0
13	0	1	0	1	1	0	0	0	1	0	0	0	0	1	0
14	1	0	0	1	0	1	1	0	0	0	0	0	0	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	1	1	0	0	0	0	0	1	1	0	1	0	0	0	0
11	0	0	0	0	0	1	1	0	1	1	0	1	0	0	0
12	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0
13	0	1	1	0	1	1	0	0	0	0	0	0	0	1	0
14	1	0	1	1	0	0	0	0	0	1	0	0	0	0	1

Figure 1: Parity Check Matrices for the Full 2-Code, $c = 5$

physical disk in order to reduce the cost of I/O accesses [3, 23]; in this paper, we assume that the logical disks in the erasure code are in one-to-one correspondence with the physical disks of the array in assessing numbers of disk accesses, however.

To set the stage, we note that this update penalty for which we have optimized the code is the true penalty when *one stripe unit* is written in fault-free mode. Yet, in actual operation of a disk array, striping data means that we are typically reading from (and writing to) a selection of logically consecutive disks. The fundamental question, which seems not to have been addressed, is whether we can reduce the update cost *per disk*, by taking advantage of our knowledge that many consecutive disks are being written. (For information on sizes of typical disk writes, see [17], for example.)

3. DISK OPERATIONS

Our goal is to examine performance, particularly user response time, in a disk array employing a k-erasure code. To do this, it is essential to determine how the basic I/O operations are completed. Each I/O operation maps the logical data to a location spanning one or more disks. This begins by determining the stripe or stripes employed.

3.1 Read operations

For each stripe involved, the relevant information disks in that stripe are also computed. A disk read is issued for each information disk needed. If none is erased or otherwise unavailable, these reads provide all of the needed data and no translation is required. In the event that disks have been lost or erased, the erasure code is then used to determine a set of further disks whose contents are sufficient to reconstruct the contents of all lost disks on that stripe, if such a set exists. Then reading these further disks, and employing their contents to reconstruct the lost disks, we have completed the read operation. Normally, the check or parity disks are only examined for read operations when an information disk has failed and the equation for reconstruction requires the parity disk. This can lead to a severe problem with latent errors on parity disks, since a parity disk failure may lie undetected until an information disk failure requires access to that parity disk. Rotation of the placement of parity information alleviates this problem when a whole disk is failed, but does not address the latent failure of individual sectors. From a performance viewpoint, the critical issue in the absence of erasures is the determination of a suitable block size within the stripe; when erasures are present, reconstruction costs are dominated by the *group* size (number of disks required in a reconstruction), although multiple erasure correction requires the determination of the smallest set of further disks sufficient to permit reconstruction; see, for example, [6, 9].

3.2 Writes in single erasure systems

Write operations are substantially more complicated, as a result of the fundamental need to keep the parity information current. In systems that correct a single erasure, two methods of writing have been extensively studied, based on an observed difference between 'small' and 'large' writes; see [12], for example. Again, we consider only the portion of a write which lies within a single stripe. In the simplest case, suppose that we are writing new contents to an information disk f_0, and there is a check disk c_0 whose contents are defined by $c_0 = \bigoplus_{i=0}^{\ell-1} f_i$. Let $\widehat{c_0}$ and $\widehat{f_0}$ denote the disk contents prior to the write operation, and $\overline{c_0}$ and $\overline{f_0}$ be the desired contents afterwards. We must calculate $\overline{c_0}$, and are provided with $\overline{f_0}$.

A *stripe write*

1. reads all information disks $f_1, \ldots, f_{\ell-1}$;
2. writes $\overline{f_0}$;
3. calculates $\overline{c_0} = \overline{f_0} \oplus \bigoplus_{i=1}^{\ell-1} f_i$; and
4. writes $\overline{c_0}$.

This requires $\ell - 1$ disk reads, and two disk writes.

A *read-modify-write*

1. reads $\widehat{c_0}$ and $\widehat{f_0}$;
2. writes $\overline{f_0}$;
3. calculates $\overline{c_0} = \widehat{f_0} \oplus \widehat{c_0} \oplus \overline{f_0}$; and
4. writes $\overline{c_0}$.

This requires two disk reads and two disk writes. Evidently, when writing a single block, read-modify-write appears to have a substantial performance advantage. However, when multiple disks are to be written, this advantage is less clear. If, for example, the disks $f_0, \ldots, f_{s-1}$ are all being written, and c is the only parity disk involved, then stripe write requires the same number of disk accesses, replacing $s-1$ disk reads by disk writes. However, read-modify-write requires $s-1$ additional reads and $s-1$ additional writes, for a total of $2s+2$ disk accesses. Thus when $2s + 2 > \ell + 1$, we can expect read-modify-write to involve more disk accesses than does the stripe write. See, for example, [12, 14].

For the stripe write and the read-modify-write, it is crucial that a method be available for caching or buffering the

contents of the disks accessed within the stripe. We assume throughout the presence of a buffer which is capable of holding the contents of the stripe across all disks (see [18]); otherwise, writes of each disk within the stripe must be treated as independent, and in general check disks are written and rewritten for each information disk written (see [7, 21]).

3.3 Writes in multiple erasure systems

When multiple erasure codes are used, analogues of the stripe write and read-modify-write appear not to have been analyzed. We look at these next. We assume throughout that a k-erasure code is in use, and that the update penalty is k for each information disk (i.e., that each information disk participates in the equations of exactly k parity disks). We suppose that, within a certain stripe, information disks $f_i, \ldots, f_{i+s-1}$ are being written, and that there are no failed disks at present. Each of the information disks to be written participates in k equations in the erasure code, and so the number u of check disks to be updated must be at least k and at most sk. A read-modify-write need only read the current contents of the u check disks to be updated and the current contents of the s information disks being updated, compute the new contents of the parity disks (including, in the exclusive or computation, the old and new contents of *all* information disks checked by this check disk), and then write back to the s information and u check disks. See Figure 2 for a depiction of the read-modify-write operation in accessing information disks 4 and 5, using the full 2-code shown at left in Figure 1. In this case, disks 4, 5, 10, 11, 13, and 14 are read, new parities are calculated, and the same disks written. A total of $2(s+u)$ disk accesses is made in general.

A stripe write (more properly, an *equation write*) initially determines, for all equations involving at least one of $f_i, \ldots, f_{i+s-1}$, the u check disks and all t information disks other than $f_i, \ldots, f_{i+s-1}$ which participate in these equations. Figure 3 shows a stripe write, again writing information disks 4 and 5 using the full 2-code shown at left in Figure 1. In this case, disks 0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 13, and 14 are read; then disks 4, 5, 10, 11, 13, and 14 are written. A stripe write in general requires t disk reads, the computation of the u new parity disk contents, and disk writes to s information and u parity disks. Hence a total of $t+s+u$ disk accesses are employed. It follows that, when $t < s+u$, a stripe write is preferable (assuming that disk accesses dominate performance) to a read-modify-write.

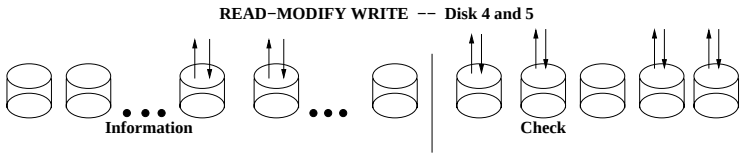
Optimizing performance involves selecting the appropriate occasions to do stripe write or read-modify-write, and this can be calculated 'on the fly' by determining the parameters t and u for making the comparison above. However, it involves more. While s is fixed as a function of the user write request, the parameters t and u are properties of the specific erasure code chosen.

Minimizing t appears not to have been studied. However, in an erasure code with group size ℓ, when writing s consecutive disks involving u check disks, an obvious upper bound is $t \leq u\ell$. Since ℓ is fixed by the code, the minimization of t is, in some sense, related to the minimization of the number u of check disks. The number u of parity disks to be updated affects the performance both of stripe writes and of read-modify-writes, although the effect in the latter appears to be more pronounced. It is therefore sensible to ask for erasure codes to minimize this parameter u. Naturally, u is a function of the number of (consecutive) disks to be read, and so we write u_s to be the *average* number of check disks updated when s consecutive information disks are updated. Evidently, $u_1 = k$, the update penalty. Thus $\frac{u_s}{s}$ can be thought of as the *observed update penalty* per disk when s consecutive disks are updated. Now $u_s \geq u_{s-1}$ for all $s > 1$, and so $u_s \geq k$.

In [7, 8], the following strategy is followed: Select an erasure code with the desired erasure correction properties, and then *order* the columns of the parity check matrix to minimize u_s for small values of s. This can be expected to improve small writes, and hence the read-modify-write, primarily. See Figure 1 for two column orderings. To make the discussion more concrete, consider 2-erasure codes. As Hellerstein *et al.* [11] note, a code with minimum update penalty two is obtained by having each information disk checked by two check disks, with no two information disks sharing the same two check disks. Treating check disks as vertices in a graph, each information disk can be viewed as an edge connecting the two check disks that check the information disk. Thus a 2-erasure code with minimum update penalty can be represented as a graph. They further observe that, to minimize check disk overhead, the best graph is the complete graph (containing all possible edges) since repeated edges violate the requirement that a double erasure be recoverable. They call the erasure code arising from the complete graph the *full 2-code*. Figure 1 is the full 2-code from the complete graph on five vertices, K_5.

For the full 2-code, we can determine t precisely. Suppose that there are c check disks in total. Then there are $\binom{c}{2}$ information disks. Now suppose that s consecutive information disks are to be written, and that these s information disks participate in u of the equations in the code. The essential property of the full 2-code is that every two check disks share exactly one information disk (i.e., that one which forms the edge between the vertices corresponding to the two check disks). Hence since the group sizes are all equal to $c-1$ (the degree of a vertex in the complete graph on c vertices), we can calculate $t + s = u(c-1) - \binom{u}{2}$. Using Figure 1, it can be verified that every two rows have exactly one column in which both have a 1 entry, and that this column is *different* if a different pair of rows is selected. Hence if u rows are selected, and *all columns containing a 1 entry in these rows counted*, we can compute the total number of 1's ($u(c-1)$) and subtract one for each of the $\binom{u}{2}$ columns containing two 1 entries within the u chosen rows.

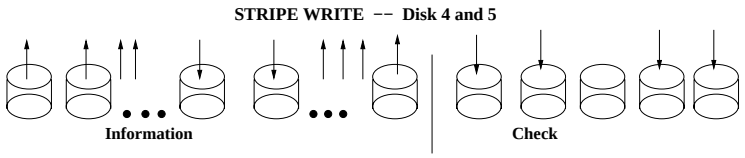
Thus t is an increasing function of u in the range $2 \leq u \leq c$; minimizing u therefore also minimizes t. As with single erasure correcting codes, the appropriate point at which to transition from read-modify-write to stripe or equation write can be determined precisely. To prefer stripe write, we require that $u(c-1) - \binom{u}{2} - s < s + u$; i.e., that $u(2c-u-3) < 4s$. When all check disks are involved ($u = c$), this requires that $s > \frac{c(c-3)}{4}$, so stripe write is preferred when (slightly less than) half the information disks, or more, are being written.



1. Pre−read information disks and check disks
2. Compute change in parity by the xor of new data with old data and old parity.
3. Write new data and check disks.

Total I/O operations: 6 reads, 6 writes = 12

Figure 2: Read-Modify-Write



Equations that need to be calculated before a write to the check disks can occur:

Check Disk 10 = 2⊕ 5 ⊕ 7⊕ 8
Check Disk 11 = 0⊕ 4 ⊕ 7⊕ 9
Check Disk 13 = 1⊕ 3 ⊕ 4⊕ 8
Check Disk 14 = 0⊕ 3 ⊕ 5⊕ 6

Total I/O operations: 8 reads, 6 writes = 14

Figure 3: Stripe write

We have found that, both for stripe write and for read-modify-write, the number of disk accesses for the full 2-code is minimized by the smallest choice for u, the number of check disks to be updated. Hence it is natural to ask how the code can be ordered so as to minimize this observed update penalty. We treat this next.

4. THE ORDERING PROBLEM

We consider the scenario in which all update penalties are two, and double erasure correction is supported. In order to minimize the fraction of storage assigned to redundant information, the appropriate erasure code is a full 2-code. However, we are free to reorder the columns (either physically or logically) of the parity check matrix. Further, in order to minimize write cost using either stripe writes or read-modify-writes of s consecutive disks, our basic task is to minimize u_s. Let us translate this into graph-theoretic vernacular.

Let $G = (V, E)$ be a graph on $n = |V|$ vertices and $m = |E|$ edges. Let $d \leq m$ be a positive integer, called the *window*. The window represents the number of (consecutive) stripe units involved in a typical write. Let $E = \{e_0, e_1, \ldots, e_{m-1}\}$. An *edge ordering* of G is a permutation π of the edge indices $\{0, 1, \ldots, m-1\}$. For the graph G with edge ordering π, and window d, define the $m+1-d$ graphs $\{G_i^{\pi,d} : 0 \leq i \leq m-d\}$ by setting $G_i^{\pi,d}$ to be the graph containing edges $\{e_{\pi(i)}, e_{\pi(i+1)}, \ldots, e_{\pi(i+d-1)}\}$. Then, when sets of d edges that are consecutive under the ordering π are accessed, the graphs $\{G_i^{\pi,d}\}$ represent the possible subgraphs accessed.

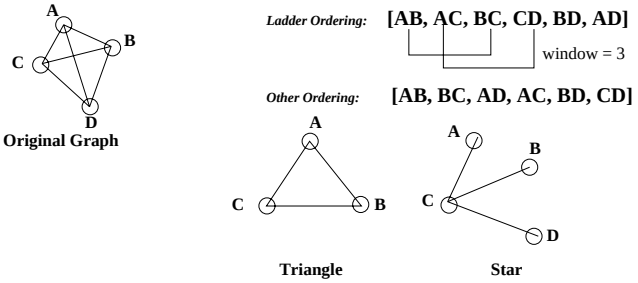


Figure 4: Edge ordering

Figure 4 shows a complete graph for $c = 4$ on the left and two different edge permutations, π_1 and π_2. The 3-access cost for the first ordering is $\frac{3+4+3+4}{4} = \frac{14}{4}$, while the 3-access cost for the second ordering is $\frac{4+4+4+4}{4} = \frac{16}{4}$. The cost of accessing a subgraph of d consecutive edges is measured by the sum of the number of edges and the number of vertices of nonzero degree in the subgraph. Since each has d edges, any reduction in access cost results from varying the numbers of vertices. Hence we define $n_i^{\pi,d}$ to be the number of vertices of nonzero degree in $G_i^{\pi,d}$. The *d-access cost* of graph G under ordering π is defined to be $\frac{\left(\sum_{i=0}^{m-d} n_i^{\pi,d}\right)}{(m+1-d)}$. The d-access cost of K_n is precisely the parameter u_d for the full 2-code with $\binom{n}{2}$ information disks, under the ordering π on the columns.

Hence we can transform the column ordering problem for codes to an edge ordering problem for graphs.

We have examined the existence of optimal edge orderings for small values of d [8]. When $d = 3$, the access cost for a specific subgraph is at least 3, and at most 6. Access cost 3 occurs when three consecutive edges induce a triangle; only triangles yield this minimum. However, when $\{e_i, e_{i+1}, e_{i+2}\}$ forms a triangle, $\{e_{i-1}, e_i, e_{i+1}\}$ cannot also form a triangle unless $n = 3$, since $e_{i-1} = e_{i+2}$ is necessary if both are triangles. Now when three consecutive edges do *not* form a triangle, the fewest vertices induced is four, which can be realized by a path or a star on three edges. Figure 4 shows the first triangle and the first star of the ladder ordering for K_4. These correspond to the first two windows. If you continue to the next window, beginning with the edge BC, you find another triangle. The minimum 3-access cost of K_n is at least 3.5 when $\binom{n}{2} \equiv 0 \pmod 2$, and is at least $3.5 - \frac{1}{n^2-n-4}$ when $\binom{n}{2} \equiv 1 \pmod 2$.

The goal is to produce an edge ordering for K_n which realizes this minimum when $d = 3$. The specific question addressed is: When can the $\binom{n}{2}$ edges of the complete graph K_n be ordered by a permutation π, so that among the $\binom{n}{2}-2$ subgraphs $\{G_i^{\pi,3}\}$, at least $\frac{1}{4}(n^2-n-6)$ subgraphs form triangles? Such an ordering of the edges of K_n is a *ladder ordering of pairs*. When $\binom{n}{2}$ is even, this requires that for any three consecutive edges e_i, e_{i+1}, e_{i+2} with $0 \le i \le \binom{n}{2} - 3$, the subgraph induced by these three edges contains three vertices when i is even, and four vertices when i is odd.

In [8], we have proved:

THEOREM 1. *A ladder ordering of pairs for K_n exists except possibly when $n \in \{15, 18, 22\}$.*

This has an immediate application to the determination of maximum access cost:

LEMMA 1. *There is an edge ordering of K_n with $(2f-4)$-maximum access cost less than or equal to f.*

In general, this does not lead to a low maximum access cost for large values of d. Nevertheless, in order to assess the value of edge orderings in improving the performance of fault-free writes, it provides an optimal method for ordering the full 2-code for small values of d. In [8], it is established that the 3-access cost can range from 3.5 to 6, and so one ought to anticipate a genuine reduction in disk accesses if the better ordering is chosen.

This theory supports our statement that an erasure code need not incur an update penalty in writing d disks that is d times the update penalty on a single disk. However, it is certainly within reason to ask whether the theory is reflected in a more practical setting.

5. SIMULATION RESULTS

RaidSim [12, 13, 14] is a simulation program written at the University of California at Berkeley [13]. Holland [12] extends it to include declustered parity and online reconstruction. The raidSim program models disk reads and writes over time. The modified version described in [12] is used as the starting point for our experiments. The original RAID and declustered parity code is left intact. RaidSim is extended to include mappings for the full 2-code, to tolerate multiple disk failures, and to detect the existence of unrecoverable erasures of three or more disks.

The preliminary experiments reported here use a full-2 code with $c = 9$, having 36 information disks and 9 check disks. The group size is 9. Since writes across a small number of disks are simulated, they are implemented only as read-modify writes. A physical rotation scheme similar to RAID 5 is used to ensure an even balance on all disks.

The performance experiments are run with a (simulated) user concurrency level of 500. This is chosen to guarantee a sufficiently heavy load on the system. We employ workloads that are pure reads, pure writes, and a mixture of 82% reads and 18% writes. The access sizes are aligned on 24K blocks. We experiment with I/O sizes that span 3, 4 and 5 disks at a time. Approximately 15 tests are run for each of these configurations. Each of these workloads is tested in fault free mode and then with one, two, three and four simultaneous failures. All disk failures occur at the start of the experiment and remain for the entire test. Information is reconstructed (if possible) each time that a read is issued on an erased disk. Runs which include an uncorrectable erasure with at least three failures are excluded from the statistics. Once we move beyond two failures the results should be interpreted with caution, since unrecoverable situations do arise.

We experimented with three separate orderings of the parity check matrix. The first is a ladder ordering which provides the greatest overlap of check disks. The second is a disjoint or pessimal ordering that provides us with a maximum of disjoint consecutive edges. The last ordering was the full 2-code in lexicographic order. By the nature of its generation, this ordesring provide some overlap of consecutive disks. Figure 5 shows graphs of two sample experiments for straight write workloads, the first writing three consecutive information disks, the second five.

These experiments bear out the conclusion predicted by the theory; the improvement in user response time of the best ordering over the worst is not only dramatic, but is consistent with our assumption that numbers of disk accesses dominate the user response time performance. We see similar, but not as dramatic results, with mixed workloads. For the read workloads, as expected we see no difference in performance in fault-free mode, but begin to see a similar but less pronounced separation when we introduce faults.

6. CONCLUDING REMARKS

We have shown that by buffering disk blocks within a stripe, we can obtain a substantial reduction in the number of disk accesses needed in both small and large writes for double erasure codes. Indeed, the general technique applies to k-erasure codes which are linear and systematic, provided that suitable column orderings to maximize overlap among the check disks accessed can be produced. For the full 2-code, we have established that writes to three consecutive disks

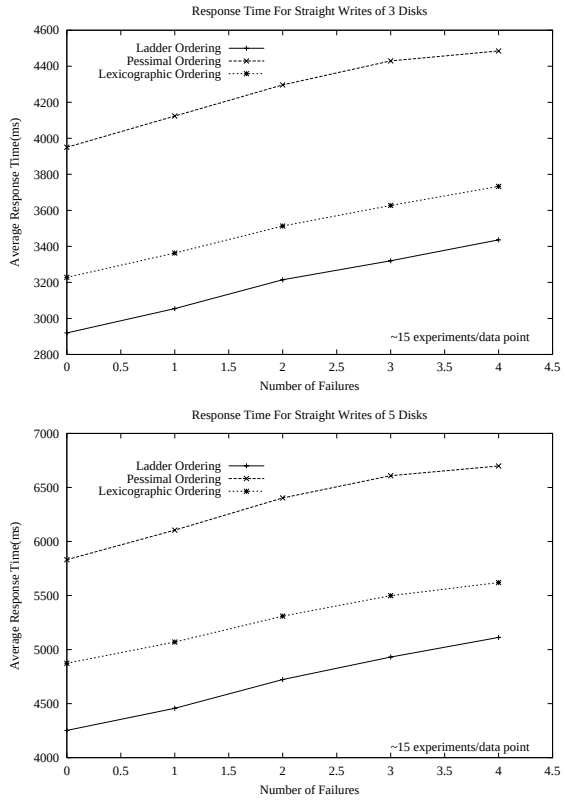


Figure 5: User response times

can be optimized to produce an effective update penalty of $1.1\overline{6}$ per disk, while writing to each disk stripe individually must incur an update penalty of 2. We have also established similar improvements for writes across larger numbers of disks, but do not report these results here.

The technique developed demonstrates that multiple erasure disk arrays need not suffer the performance loss over single erasure systems in the basic read and write operations, and hence shows promise for supporting disk arrays of one thousand or more disks while ensuring acceptable reliability.

7. ACKNOWLEDGMENTS

Research of the authors is supported by the Army Research Office (U.S.A.) under grant DAAG55-98-1-0272 (Colbourn).

8. REFERENCES

[1] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Computer Architecture*, pages 62–72. IEEE, June 1997.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS 1967 Spring Joint Computer Conference, volume 30*, pages 483–485. AFIPS, April 1967.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in raid architectures. *IEEE Trans. Computers*, 44(2):192–202, February 1995.

[4] Y. M. Chee, C. J. Colbourn, and A. C. H. Ling. Asymptotically optimal erasure-resilient codes for large disk arrays. *Discrete Applied Mathematics*, 102(1-2):3–36, May 2000.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–188, June 1994.

[6] M. B. Cohen. Performance analysis of triple erasure codes in large disk arrays. Master's thesis, University of Vermont, 1999.

[7] M. B. Cohen and C. J. Colbourn. Optimal and pessimal orderings of Steiner triple systems in disk arrays. *Theoretical Computer Science*, To appear.

[8] M. B. Cohen and C. J. Colbourn. Ladder orderings of pairs and raid performance. Submitted for publication 2000.

[9] M. B. Cohen and C. J. Colbourn. Steiner triple systems as multiple erasure codes in large disk arrays. In *Proceedings of IPCCC 2000 (19th IEEE International Conference on Performance, Computing and Communications)*, pages 288–294. IEEE, February 2000.

[10] G. A. Gibson. *Redundant Disk Arrays, Reliable Parallel Secondary Storage*. MIT Press, 1992.

[11] L. Hellerstein, G. A. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2-3):182–208, Aug-Sept 1994.

[12] M. C. Holland. *On-Line Data Reconstruction in Redundant Disk Arrays*. PhD thesis, Carnegie Mellon University, 1994.

[13] E. K. Lee. Software and performance issues in the implementation of a RAID prototype. Technical Report Technical Report CSD-90-573, University of California at Berkeley, 1990.

[14] E. K. Lee. *Performance Modeling and Analysis of Disk Arrays*. PhD thesis, University of California at Berkeley, 1993.

[15] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, 1997.

[16] P. Massiglia. *The RAID Book, A Storage System Technology Handbook, 6th Edition*. The RAID Advisory Board, 1997.

[17] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Systems*, 7(10):1075–1089, Oct 1996.

[18] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[19] K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 336–342. IEEE, February 1986.

[20] E. J. Schwabe and I. M. Sutherland. Flexible use of redundancy in disk arrays. *Theory Comput. Systems*, 32(5):561–587, Sept-Oct 1999.

[21] D. Stodolsky, G. Gibson, and M. Holland. Parity logging: Overcoming the small write problem in redundant disk arrays. *Computer Architecture News*, 21(2):64–75, May 1993.

[22] S. A. Vanstone and P. C. van Oorschot. *An Introduction to Error Correcting Codes with Applications*. Kluwer Academic Publishers, 1989.

[23] L. Xu, V. Bohossian, J. Bruck, and D. G. Wagner. Low density MDS codes and factors of complete graphs. *IEEE Trans. Information Theory*, 45(6):1817–1826, Sept 1999.