

Almost Complete Address Translation (ACATS) Disk Array Declustering

Thomas J.E. Schwarz, S.J.
Saint Louis University
Department of Mathematics and
Computer Science
221 North Grand Boulevard
Saint Louis, MO 63103-2007

Walter A. Burkhard*
Gemini Storage Systems Laboratory
Department of Computer Science and
Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114

Abstract

We present a novel declustering scheme (ACATS) for reliability stripes in an orthogonal disk array. Our scheme is deterministic, run-time efficient, and provides frequently the best possible and always an almost best possible distribution of failure-induced incremental rebuild workloads. Our scheme provides protection against single disk as well as single string failures within the disk array. Our approach presents a framework in which the Level 5 RAID organization logically appears as a Level 4 RAID; it facilitates the provision of distributed sparing in exactly the same manner. ACATS does not require the existence of a suitably configured block design or of a run-time efficient pseudo-random number generator; it is applicable to arbitrarily configured orthogonal disk arrays. Our scheme is faster than declustering schemes that use pseudo-random permutations and achieves better uniformity of disk loads during a disk rebuild; it is simpler than schemes based on block designs. ACATS provides a rich spectrum of declustering schemes.

1. Introduction

Arrays of disks combine parallel accessibility with failure tolerance and have found wide-spread commercial application. This technology achieves failure tolerance by placing the disks in *reliability groups*; a reliability group contains both client data objects and check data objects. The term *parity stripe* designates a similar logical notion for parity-based failure tolerance schemes. Our approach is not limited to parity-based schemes and we use the terminology reliability stripe. A *reliability stripe* contains data disks as well as one or more additional check and spare disks.

*Walter Burkhard was supported in part by grants from the University of California MICRO program as well as Symbios Logic, Wichita, Kansas.

The check disks contain redundant information, often parity stored on a single disk, of the data on the other disks in the reliability stripe. The redundant data in conjunction with the client data on all the other disks in the strip allow data reconstruction for a failed disk. RAID Level 4 & 5 designs [6] are briefly described. Level 4 consists of a parity group (stripe) containing a dedicated parity disk with the remaining disks containing client data objects. Level 5 is similar with the parity data distributed in round robin fashion throughout the disks of the group. Both designs provide fault tolerance for single disk failures. The inclusion of a spare disk [2] within the stripe provides space for the reconstructed data. Figure 1 presents a disk array consisting of three reliability stripes.

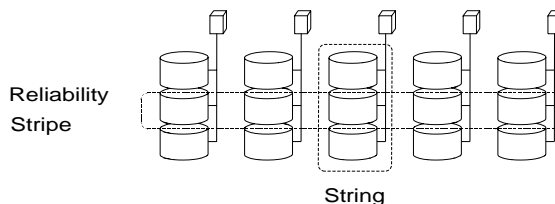


Figure 1. Disk array configuration with five strings of three disks.

Typically, disks in an array share essential components such as cabling, cooling and power. Some components such as buffer memory are shared by all disks; failure of such a component implies loss of all data. Other components such as power supplies, controllers, cabling, cooling, are shared only among a small group of disks on a *string*. If a component shared by a string of disks fails, we lose access to all data stored on disks in the string. The orthogonal arrangement of reliability stripes and strings, presented within figure 1, allows reconstruction of all data after a string failure.

We distinguish three failure modes in an orthogonal disk array configured as in figure 1: (1) Failure of an essential component destroys accessibility of all data in the disk ar-

ray, (2) failure of an essential string component renders all data on disks of the string inaccessible, and (3) failure of an individual disk renders data on the disk inaccessible. Failures of type (2) and (3) do not constitute data loss; we can reconstruct the inaccessible data from the redundant data stored on other disks within the array. We can improve the reliability of the array configuration by component duplication; this decreases the dataloss rate for modes (1) and (2).

The impact of disk and string failure can be limited by providing “hot spare disks.” In the case of failure, we reconstruct the data on the afflicted disk(s) and store them on the spare disks. Then a single failure only reduces the failure tolerance of the disk array. The speed of the data reconstruction significantly impacts the survival rate for individual file blocks, so low reconstruction times become a principal goal. Even more importantly, during the reconstruction process we need to access those disks within the reliability stripe of the inaccessible disk. The increment due to the reconstruction load lowers the throughput disk array clients experience. The array configuration must have enough unused resources to process the reconstruction as well as the client load; consequently the reconstruction increment limits the entire disk array performance during fault-free operation as well. *Declustering* spreads the client data, check data, and spare space of a reliability stripe over possibly all disks of the configuration. The resulting reduction in the reconstruction load increment improves performance in comparison with non-declustering systems. Since heavy demand for a particular disk is distributed over the disk array, we observe performance improvements during fault-free operation as well.

Several declustering schemes have been proposed. We note, however, that some authors choose to use the phrase *clustering schemes* for exactly the same notion. Reddy and Bannerjee [?], Holland and Gibson [1], and Ng and Mattson [5] propose data layout schemes based on balanced incomplete block designs (BIBD). Schwabe and Sutherland [7] propose new BIBD constructions as well as approximately-balanced layouts for parity-declustering within disk arrays. Muntz and Liu [3] mention the BIBD approach. However balanced incomplete block designs are not available for all configurations of disk arrays. Merchant and Yu [?] as well as Schwarz and Burkhard [8] describe schemes based upon pseudo-random permutations. Newberg and Wolfe [4] discuss string layouts in the context of determining the reliability of 2-dimensional parity organization disk arrays.

We present a *deterministic* disk array declustering scheme which we call the *almost complete address translation scheme*(ACATS). ACATS achieves optimal declustering for a disk array organized in strings. The scheme is readily and efficiently implementable. It does not require table storage, as is the case for block design schemes, or relatively large amounts of calculation per access, as is nec-

essary in schemes using pseudo-random permutations. We present an easily accessible version of ACATS in section 2, then we improve on it in section 3 to achieve full generality and finally upgrade to a version in section 4 which meets additional requirements. This iterative approach is to improve the readability of our paper. Section 4 contains a discussion of the rebuilding scheme. We provide some performance results for ACATS as well as other declustering schemes in section 5. The Appendix contains the details of our mathematical results.

2. Almost Complete Address Translation

Figure 1 presents the basic RAID Level 4 design [6]. Logical disks are arranged in an orthogonal array of columns and rows; each column depicts disks comprising a string and each row depicts disks comprising a reliability stripe. The rightmost string consists of spare disks and the second-rightmost string contains check disks, which store the check data for the reliability stripe. The other disks store client data. Our ACATS scheme presents to the user the picture of such a Level 4 RAID, while it implements a higher performing Level 5 RAID with distributed sparing. Our scheme works for an arbitrary number of spare disks, but for ease of exposition we only discuss schemes with a single logical string of spare disks. In the actual implementation, each disk stores client, check and spare data.

Our address scheme maps the data objects of the logical disks of a reliability stripe row to all disks within the array. There are four requirements our mapping must satisfy:

1. The objects comprising a reliability stripe must reside on distinct physical strings.
2. The check, spare, and client data objects must be evenly distributed over all physical disks within the array.
3. The workload during string reconstruction must be evenly spread over the disks of the remaining strings. Similarly, the workload during a single disk rebuild must be evenly spread over the disks on strings other than the string containing the failed disk.
4. The mapping from logical to physical addresses must be reasonably efficient and the space to store the algorithm and data must be small.

These requirements elaborate on the conditions mentioned by Muntz and Liu [3] as well as Holland and Gibson [1] by including strings.

2.1. Definition

Logical data objects are stored on logical disks; the data objects are mapped to physical disks. Typically, data ob-

jects are disk tracks or sector blocks. We have n strings each containing m disks. Our mapping is described in terms of two cyclic-shift permutations. We present our scheme under the assumption that the number of disks per string, m , is a prime number. In Section 3, we discuss an extension generalizing the scheme to all m .

We give the logical address of a data object (sector block or disk track) as a triple (s, d, t) where s is the *logical string number*, d is the *disk-in-string number* (DIS) and t is the data object number. For example, the data object with logical address $(2, 1, 5)$ is the 5th data object on logical disk 1 of logical string 2.

We begin by expressing data object numbers t in m -ary notation $(\dots t_2 t_1 t_0)_m$, so that $t = \sum_{\nu \geq 0} t_\nu \cdot m^\nu$. Both binary and decimal representations of numbers are examples of such notation. Define level l to be $1 + \lfloor \log_m(n-1) \rfloor$. Each physical string i , $0 \leq i < n$, has an associated *permutation vector* \vec{a}_i , consisting of l coordinates, defined as

$$\vec{a}_i = (a_{l-1}, a_{l-2}, \dots, a_1, a_0) \quad (1)$$

such that i expressed in m -ary notation is $\sum_{\eta=0}^{l-1} a_\eta \cdot m^\eta$. We also use the notation

$$\vec{t} = (t_{l-1}, t_{l-2}, \dots, t_1, t_0)$$

in the next definition. Our mapping is described by $(s, d, t) \mapsto$

$$((s+t) \bmod n, (\vec{a}_{(s+t) \bmod n} * \vec{t} + d) \bmod m, t) \quad (2)$$

where $*$ denotes the scalar product; logical string s is mapped to physical string $(s+t) \bmod n$, logical disk d (of logical string s) is mapped to physical disk $(\vec{a}_{(s+t) \bmod n} * \vec{t} + d) \bmod m$ and the data object number t is unchanged. Thus our mapping uses only the l least significant m -ary digits of t .

As an example of our ACATS mapping, we present in figure 2 the mapping for $n = 5$ strings and $m = 3$ disks per string as in figure 1. The level l is 2 and the permutation vectors $\vec{a}_0, \vec{a}_1, \vec{a}_2, \vec{a}_3, \vec{a}_4$ respectively are given by

$$(0, 0), \quad (0, 1), \quad (0, 2), \quad (1, 0), \quad (1, 1)$$

The mapping in figure 2 presents the first few objects on each of the fifteen disks. For object 0, the logical and physical locations are the same; this object demonstrates the logical layout within the figure. The rows designate reliability stripes and the columns designate strings. The pairs show the physical locations for each of the logical objects; since an object does not change its number under the ACATS mapping, we list only the physical string \mathcal{S} and disk in string \mathcal{D} values $(\mathcal{S}, \mathcal{D})$. For example, object 2 stored on logical disk 1 of logical string 1 is mapped to disk 1 of string 3; this is designated in figure 2 by $(3, 1)$ located at logical disk 1 of

string 1 for object 2. Similarly, object 4 logically stored on disk 2 of string 4 is mapped to disk 0 of string 3. The period of the layout is 45 objects; for larger object numbers, the layout of figure 2 is repeated cyclically. Take for example logical object 47 on disk 1 of string 1; it is mapped to string 3 and, since $47 = (1202)_3$ in ternary notation and since we only use the two least significant digits, to the same disk as object 2 of logical disk 1 of string 1.

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	object 0
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	
(1,1)	(2,2)	(3,0)	(4,1)	(0,0)	object 1
(1,2)	(2,0)	(3,1)	(4,2)	(0,1)	
(1,0)	(2,1)	(3,2)	(4,0)	(0,2)	
(2,1)	(3,0)	(4,2)	(0,0)	(1,2)	object 2
(2,2)	(3,1)	(4,0)	(0,1)	(1,0)	
(2,0)	(3,2)	(4,1)	(0,2)	(1,1)	
(3,1)	(4,1)	(0,0)	(1,0)	(2,0)	object 3
(3,2)	(4,2)	(0,1)	(1,1)	(2,1)	
(3,0)	(4,0)	(0,2)	(1,2)	(2,2)	
(4,2)	(0,0)	(1,1)	(2,2)	(3,1)	object 4
(4,0)	(0,1)	(1,2)	(2,0)	(3,2)	
(4,1)	(0,2)	(1,0)	(2,1)	(3,0)	
(0,0)	(1,2)	(2,1)	(3,1)	(4,0)	object 5
(0,1)	(1,0)	(2,2)	(3,2)	(4,1)	
(0,2)	(1,1)	(2,0)	(3,0)	(4,2)	
	\vdots		\vdots		
	\vdots		\vdots		
	\vdots		\vdots		

Figure 2. ACATS mapping implicit logical addresses to physical addresses.

2.2. Immediate Consequences

Our mapping permutes the strings cyclically and then the disk in string addresses within a string. It is therefore a bijection which respects the orthogonal arrangement of the reliability stripes and strings; the objects comprising a stripe reside physically on n different strings.

There are several immediate consequences following from our ACATS definition. First, we note that the objects comprising a reliability stripe physically reside on n distinct strings. Thus we satisfy our first mapping requirement. Accordingly, our layout scheme will tolerate single string failure as well as single disk failure.

Second, the layout scheme is cyclic with period $\text{lcm}(m^l, n)$. This follows since the disk in string permutations depend upon the l least-significant m -ary digits of the object number and there are n strings.

We demonstrate in the Appendix that each pair of disks, on differing strings, is associated with identical reliability

stripes exactly m^{l-1} times within a sequence of m^l consecutive object addresses. Accordingly, we have the following uniformity result for our ACATS mapping: During the rebuild of a single disk, each of the disks on strings other than the string containing the failed disk, will be accessed m^{l-1} times during reconstruction of m^l consecutive addresses. The rebuild process provides uniform distribution of the incremental work load over all disks on strings not containing the failed disk. So while the data, check and spare space is evenly distributed throughout the array, the incremental work load is distributed over all disks on the strings not containing the failed disk, uniformly when the disks contain m^k for $k \geq l$ data objects and almost uniformly otherwise. Our third point regarding string reconstruction follows from these remarks.

3. Generalization

We treat the case of m being a prime power and the case of arbitrary m separately.

Assuming that m is a prime power, we can represent the DIS numbers with values from the field of m elements. We then calculate the middle component of our definition in equation 2 using field operations instead of calculations modulo m .

Otherwise, we modify definition in equation 2 for our ACATS mapping for general m ; the permutation vectors \vec{a}_i consist of the binary representation of i and have only 0 and 1 as coordinate values. Each physical string i , $0 \leq i < n$, has an associated l -dimensional \vec{a}_i , defined as

$$\vec{a}_i = (a_{l-1}, a_{l-2}, \dots, a_1, a_0) \quad (3)$$

such that i expressed in binary notation is $\sum_{\eta=0}^{l-1} a_\eta \cdot 2^\eta$. In this situation, we calculate the middle component of our definition in equation 2 using operations of the quotient ring – the integers module m . We choose the level l to be $1 + \lfloor \log_2(n-1) \rfloor$ which is larger than our previous level $1 + \lfloor \log_m(n-1) \rfloor$, with the consequence that the layout period is now larger.

4. Disk and String Rebuilding

Rebuilding either a disk or a string of disks means reconstructing and storing objects on appropriate spare space. We first discuss disk rebuilding and then string rebuilding.

Rebuilding the objects of a failed disk involves accessing the accessible data and check objects of the reliability group containing the failed disk. The reconstructed data is stored within the distributed spare space of the reliability group. Suppose that physical disk \vec{d} in string \vec{s} has failed. Further assume that for a given object number t , the data

was mapped from logical disk d of string s . Then we have

$$\tilde{s} = (s + t) \bmod n \quad \text{and} \quad \tilde{d} = (\vec{a}_{\tilde{s}} * \vec{t} + d) \bmod m \quad (4)$$

Conversely, we can express the address of the logical disk in terms of the physical disk:

$$s = (\tilde{s} - t) \bmod n \quad \text{and} \quad d = (\tilde{d} - \vec{a}_{\tilde{s}} * \vec{t}) \bmod m \quad (5)$$

for object t . The other logical disks placed in the same reliability stripe d are the disks located on strings other than string \vec{s} . The corresponding physical disks are the ones that will be accessed to reconstruct data object t on the failed disk: These are disks $(\vec{a}_i * \vec{t} + d) \bmod m$ on strings $i \neq \vec{s}$. The set of disks in the reconstruction set RS for object t on physical disk \vec{d} of string \vec{s} is given by $RC(\vec{s}, \vec{d}, t) =$

$$\left\{ \left(j, \left((\vec{a}_j - \vec{a}_{\vec{s}}) * \vec{t} + \vec{d} \right) \bmod m \right) \mid j \neq \vec{s} \right\}$$

A given physical disk on string i will now lie in the reconstruction set for m^{l-1} values of the m^l possible values for object number t . We prove this uniformity result within the appendix.

Continuing our example in figure 2 with five strings each containing three disks, suppose that disk 1 of string 3 has failed. Figure 3 presents the reconstruction sets for the first few objects; the organization has period 45 objects. The rightmost column contains the logical addresses of the failed objects. In this configuration, each disk not on phys-

(0,1)	(1,1)	(2,1)	(4,1)	object 0	(3,1)
(1,2)	(2,0)	(4,2)	(0,1)	object 1	(2,1)
(2,2)	(4,0)	(0,1)	(1,0)	object 2	(1,1)
(4,1)	(0,0)	(1,0)	(2,0)	object 3	(0,0)
(4,2)	(0,0)	(1,1)	(2,2)	object 4	(4,0)
(0,0)	(1,2)	(2,1)	(4,0)	object 5	(3,0)
	⋮				⋮

Figure 3. Rebuilding failed disk 1 of string 3; logical addresses of failed objects

ical string 3 will be accessed 15 times while rebuilding the data for a contiguous sequence of 45 objects. In general, each disk not on the physical string containing the reconstructed disk will be accessed P/m times during the rebuilding of P consecutive data objects; P is the layout period.

We have assumed that we use the spare space to store the reconstructed value. Accordingly, if we are not currently storing reconstructed data within the spare space, there is no need to reconstruct this space of the failed disk. In this case,

(0,1) (1,1) (2,1) (4,1)	object 0
(1,2) (2,0) (4,2) (0,1)	object 1
(2,2) (4,0) (0,1) (1,0)	object 2
(4,1) (0,0) (1,0) (2,0)	object 3
(3,1) is spare space	object 4
(0,0) (1,2) (2,1) (4,0)	object 5
⋮	⋮

Figure 4. Efficient rebuilding failed disk 1 of string 3

we can save $1/n$ of the reconstruction effort and shorten the disk rebuild time. Unfortunately, if n and m contain a non-trivial common factor, our scheme does not distribute these savings uniformly.

We can easily identify the data objects that map spare space onto the failed disk or string. If \vec{s} equals $(q + t) \bmod n$, where q designates the address of a string of disks that contains logical spare space, we need not reconstruct this space for t . We show an example of efficient rebuilding within figure 4.

ACATS maps spare space to a given physical disk every n^{th} object number. If $\gcd(m^l, n)$ is not 1, then the rebuild workload decrease accrued from avoiding reconstruction of the spare space will not be equally distributed throughout the array. Therefore, we modify our scheme slightly so the period will be $m^l \cdot n$ to better distribute the savings throughout the array. The modified mapping is $(s, d, t) \mapsto$ (6)

$$((s + t + t/T) \bmod n, (\vec{d}_{(s+t+t/T) \bmod n} * \vec{t} + d) \bmod m, t)$$

where T is $\text{lcm}(m^l, n)$ and l, m , and n are as before and t/T designates integer division. Our performance reports in the next section will use this improved mapping scheme.

String rebuilding involves rebuilding each of the disks of the string as discussed above. ACATS maps all data of the string, without track number changes, to other strings so that each the objects of each reliability group will appear within distinct strings. This provides a spectrum of string rebuild strategies. Depending on the available resources, it is possible to concurrently rebuild r disks for $1 \leq r \leq m$.

5. Disk and String Rebuilding Performance

The single disk rebuild load is uniformly distributed over the array of reliability stripes if the number of objects per disk is a multiple of the layout period. In this section, we present results indicating how well ACATS distributes the

single disk rebuild workload for other numbers of objects per disk. In our scheme, the workload for a string rebuild is uniformly distributed over all disks not on the string containing the failed disk. We also compare the rebuild performance of the deterministic ACATS introduced here with that of pseudo-random ACATS [8].

We have derived previously as an immediate consequence that the layout period of our ACATS scheme is $\text{lcm}(m^l, n)$. Table 1 contains the periods for several small arrays.

	n=4	n=5	n=6	n=7	n=8	n=9	n=10
m=3	36	45	18	63	72	27	270
m=4	4	80	48	112	16	144	80
m=5	20	5	150	175	200	225	50
m=6	36	1080	216	252	1728	216	1080
m=7	28	35	42	7	392	441	490

Table 1. ACATS periods for layouts of n strings and m disks per string.

We utilize two performance measures, *opt* and *max*. These are defined as follows: For a given failed disk, we determine the number of times each disk within the array participates within the rebuild process *for the entire disk*. This participation will not necessarily be uniformly distributed due to spare space, the layout period and the number of data objects actually rebuilt. For each failed disk, we get such a collection of values but we interested in the largest number of participations independent of the failed disk – this value is designated *max*. The optimal value, designated *opt*, for a perfectly distributed workload, is calculated for a given number of data objects T as follows

$$\text{opt}(T) = \lceil T/m \rceil - \lfloor (T/m) \cdot (1/n) \rfloor.$$

The subtrahend reflects the savings accrued by not rebuilding the spare space.

Our empirical results show that the difference between $\text{opt}(T)$ and $\text{max}(T)$ for m a prime or a prime power are negligible, and are small for other values of m . Our results are for disk arrays logically configured as RAID Level 4 with a string of spare disks; we have used the improved ACATS mapping of equation 6. We tabulate some sample values within Table 2 as well as presenting graphs in figures 5 and 6.

The results displayed within the graph of figure 6 indicate the optimal values increase linearly with the number of objects rebuilt. The slope of these curves is proportional to $(n-1)/(m \cdot n)$ where n is the number of disks per reliability stripe and m is the number of disks per string; the $(n-1)/n$ factor is present since the spare space is not rebuilt. The maximum value curve has the same general behavior and is almost indistinguishable from the optimal values except for the 6 disks per string with 10 disks per reliability stripe

curve. Even in this case, we observe the regular oscillations of the maximum value curve, which touches the optimum curve at multiples of the period, where our layout scheme is optimally uniform. Since the absolute difference between the two curves stays the same from oscillation to oscillation, the relative difference approaches zero for large number of objects. The maximum curves are not perceptible for the other three configurations. The graph in figure 5 elaborates this information.

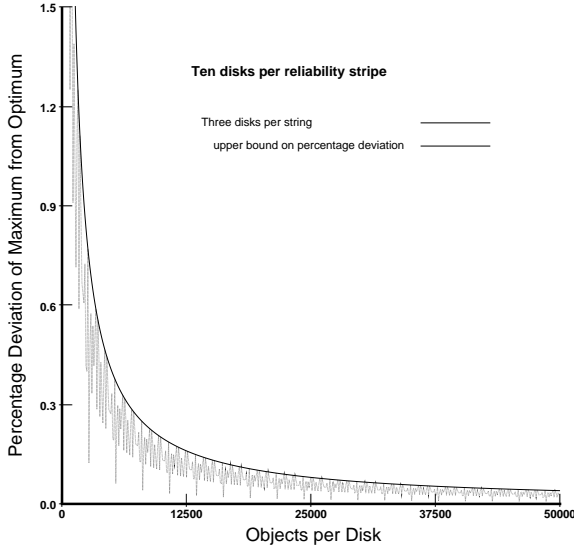


Figure 5. ACATS Performance – Ten Disk Reliability Stripe Configurations

This graph, which is typical of a wide range of configurations, shows the percentage deviation of max from opt

$$\frac{(max(T) - opt(T)) \cdot 100}{opt(T)}$$

for a given number of data objects T . This graph clearly shows hyperbolic decrease with increasing numbers of data objects. The darker bounding hyperbolic curve is drawn to fit the data. The curve shows the percentage deviation to be less than 0.3% for T no smaller than approximately 8000. For $m = 6$ disks per string, the ring based scheme entails larger deviation; nevertheless, the bound is still hyperbolically decreasing. For $m = 6$ and $n = 10$ disks per reliability stripe, the percentage deviation is less than 10% for T at least approximately 12000. This deviation is large enough to be visible in the graph of figure 6.

We can give an upper bound for the ratio $r(T)$ of the difference of the optimum and maximum values over the optimum value given the number of objects T :

$$r(T) = \frac{max(T) - opt(T)}{opt(T)}$$

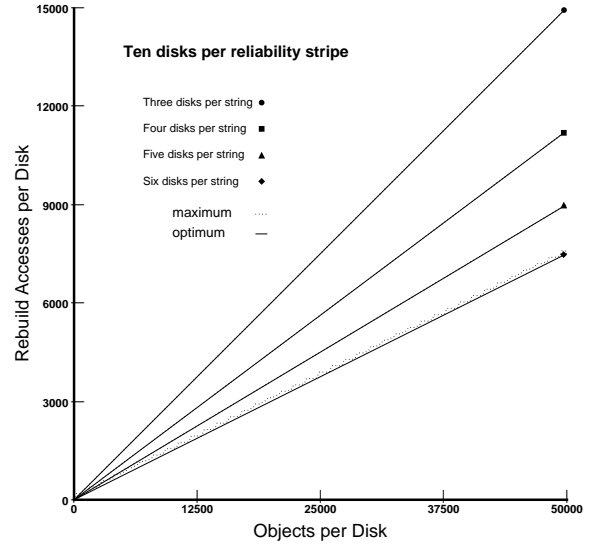


Figure 6. ACATS Performance – Ten Disk Reliability Stripe Configurations

Since the opt and the max functions either stay constant or are incremented by one if argument T is incremented by one, the maximum of the difference between these two is at worst attained for $T = P/2$ where P is the period of our scheme and is $P/2 - P/(2m)$. Thus,

$$r(T) \leq \frac{P/2 - P/(2m)}{T(n-1)/(nm)} = \frac{(m-1)nP}{2T(n-1)}$$

which converges to zero quickly as $T \rightarrow \infty$.

	n=5, T=2000	n=5, T=20000	n=10, T=2000	n=10, T=20000
m=3	534,536	5334,5336	601,604	6001,6003
m=4	400,401	4000,4001	450,453	4500,4501
m=5	320,321	3200,3201	360,361	3600,3601
m=6	268,288	2668,2679	301,389	3001,3111

Table 2. opt, max for various configurations with $T = 2000$ and $T = 20000$.

String rebuilding presents a variety of workload increments depending on the amount of concurrency. The spectrum ranges from the least increment which is entailed by a sequential approach (rebuilding one disk at a time) to the greatest increment entailed by maximal parallelism. The greatest increment would result in no performance benefit from declustering.

5.1 Comparison with other Declustering Methods

We can compare our ACATS scheme with pseudo-random declustering schemes in two ways, in the speed of the as-

segment and in the uniformity of the distribution of the incremental rebuild workload. We use *max*, which gives the maximum number of accesses to a disk in order to recover data on a failed disk, to measure this uniformity.

The expected maximum value is calculated by first calculating the distribution of the maximum under the multinomial distribution, which gives the probability that we need to access a certain disk in a string a given number of times. By taking the $n - 1^{th}$ -power of the distribution, we obtain the distribution for the maximum number of accesses over all strings, and with this distribution we finally calculate the expected maximum value.

In table 3, we give the expected value of this measure for a variety of disk arrays. The derivation of exact values for larger numbers of objects per disk is computationally intricate; instead we derive an extrapolation using the normal approximation of the binomial probability distribution. The difference between the expected maximum and optimal values is approximately proportional to the variance, which increases with the square root of the total number of objects per disk. Therefore random permutation based schemes show a convergence of order $o(c/\sqrt{T})$ for $T \rightarrow \infty$ which is slower than that for our deterministic ACATS approach. As an example, for a disk array with $n = 10$ disks per reliability stripe and $m = 5$ disks per string, the expected maximum should then exceed the optimal value for 10000 objects per disk by approximately 1.5% instead of the 14.3% observed for 1000 objects. For comparison, in table 4 we give *max* values for deterministic ACATS; the percentage deviation in these cases is very small except for the ring-based scheme for $m = 6$.

n	m=2	m=3	m=4	m=5	m=6
optimal:	500.000	333.333	250.000	200.000	166.667
2	512.613	348.829	266.366	216.577	183.192
3	517.839	353.667	270.733	220.555	186.857
4	520.969	356.451	273.217	222.805	188.923
5	523.155	358.380	274.933	224.358	190.348
6	524.816	359.845	276.235	225.536	191.429
7	526.146	361.019	277.280	226.481	192.296
8	527.250	361.996	278.149	227.268	193.019
9	528.190	362.830	278.892	227.940	193.636
10	529.007	363.557	279.539	228.526	194.175
11	529.728	364.199	280.112	229.045	194.652
12	530.373	364.774	280.625	229.510	195.079
13	530.954	365.294	281.089	229.931	195.466

Table 3. Expected maximum values for pseudo-random disk array schemes.

The implementation of pseudo-random declustering leads to slower calculation of object placement, which could necessitate the storage of a table. The run-time performance of our scheme is better than that of schemes based

n	m=2	m=3	m=4	m=5	m=6
optimal:	500.000	333.333	250.000	200.000	166.667
2	500	334	252	200	167
3	500	334	252	200	167
4	500	334	252	200	167
5	500	334	252	200	167
6	500	334	252	200	167
7	500	334	252	200	167
8	500	334	252	200	167
9	500	334	252	200	192
10	500	334	252	200	195
11	500	334	252	200	197
12	500	334	252	200	198
13	500	334	252	200	200

Table 4. Maximum values for the deterministic ACATS scheme.

on run-time efficient utility pseudo-random number generators. Similarly, schemes based on block design need to be well implemented for good run-time performance and can require table lookup too. In contrast to our ACATS scheme, which is universally applicable, a change in the number of objects per disk forces the adaptation of another declustering scheme.

6. Conclusions

We have presented the deterministic ACATS family of declustering schemes that provides general applicability as well as ease of implementation and efficient run time performance. We have summarized ACATS as well as other schemes in table 5.

declustering scheme	algorithm size	pseudo-random numbers required	string fault tolerant
ACATS	small	no	yes
BIBD [1,4,6,8]	layout description table may be needed	no	no
pseudo-random [3]	small	yes	no
pseudo-random ACATS [10]	small	yes	no

Table 5. Declustering Schemes.

For small disks, with at most a few tens of thousands of data objects, our deterministic ACATS provides much better and predictable performance compared with the pseudo-random based declustering schemes. Only for larger numbers of data objects does the difference between these two approaches become insignificant.

The implementation of a pseudo-random declustering scheme leads to slower calculation of object placement, and could necessitate the storage of large tables. Similarly, schemes based on block designs must be well implemented to obtain good run-time performance; these implementa-

tions may require table lookup. A change in the number of objects per disk, caused for example by installing new disks, forces the adaptation of another declustering scheme.

Our deterministic ACATS scheme is universally and efficiently applicable. Moreover, its performance is predictable unlike that of pseudo-random based schemes.

References

- [1] M. C. Holland and G. A. Gibson.
- [2] J. Menon and R. L. Mattson. Distributed Sparing in Disk Arrays. In *Proceedings of the COMPCOM Conference*, pages 410–416, San Francisco, 1992.
- [3] R. R. Muntz and J. C. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th VLDB Conference*, pages 162–173, Brisbane, June 1990.
- [4] L. Newberg and D. Wolfe. String Layout for a Redundant Array of Inexpensive Disks. *Algorithmica*, 12:209–224, 1994.
- [5] S. W. Ng and R. L. Mattson. Maintaining Good Performance in Disk Arrays During Failure Via Uniform Parity Group Distribution. In *Proceedings of the First International Symposium on High-Performance Distributed Computing*, pages 260–269, 1992.
- [6] D. A. Patterson, G. A. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings SIGMOD International Conference on Data Management*, pages 109–116, Chicago, May 1988.
- [7] E. J. Schwabe and I. M. Sutherland. Improved parity-declustered layouts for disk arrays. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 76–84, Cape May, N.J., June 1994.
- [8] T. J. Schwarz and W. A. Burkhard. Reliability and Performance of RAIDs. *IEEE Transactions on Magnetics*, 31:1161–1166, March 1995.

Appendix

Our goal here is to demonstrate that our ACATS organization provides “almost” uniform access during a disk reconstruction operation. We consider the number of solutions to a set of linear homogeneous equations.

Recall that there are n strings each containing m disks. We base our uniformity result on the following lemma, which is well known with either m is prime or m is a prime power. In these cases, our arithmetic regarding the d portion of the mapping is performed in a field of m elements. The following lemma holds in general when the mapping calculations are within a module because of the structure of our permutation vectors.

Lemma 1 *For distinct i and j such that $0 \leq i, j < n$ and for arbitrary x and y such that $0 \leq x, y < m$, there are m^{l-1} solutions \vec{t} such that*

$$(\vec{a}_i - \vec{a}_j) * \vec{t} = y - x.$$

We employ the lemma with DIS addresses x on physical string j and y on physical string i to demonstrate they participate within the same reliability group exactly m^{l-1} times within a consecutive sequence of m^l data objects.

Proof: We consider only the case when the finite field solution is not possible; that is, m is neither a prime nor a prime power. We instead work in the quotient ring – the integers modulo m . In this case, the permutation vectors \vec{a}_i for $0 \leq i < n$ consist of integers 0 and 1. Consequently, the vector $(\vec{a}_i - \vec{a}_j)$ is non-zero consisting of integers 0, 1, and -1 modulo m . Assume that its first non-zero coefficient is located in position p . An arbitrary choice of \vec{t} in all but the p^{th} coordinate t_p transforms our equation into one with one indeterminate, namely t_p , which has exactly one solution:

$$\sum_{0 \leq \nu < l, \nu \neq p} (a_{i,\nu} - a_{j,\nu})t_\nu + (a_{i,p} - a_{j,p})t_p = y - x.$$

This follows since the difference $(a_{i,p} - a_{j,p})$ is either a 1 or -1 so that we avoid division in solving this equation. This observation proves our lemma.

Accordingly, since there were m^{l-1} choices for the free values, we have m^{l-1} distinct solutions as claimed. \square