Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering

Guillermo A. Alvarez Walter A. Burkhard Flaviu Cristian

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093−0114, USA
{galvarez, burkhard, flaviu}@cs.ucsd.edu

Abstract

We present DATUM, a novel method for tolerating multiple disk failures in disk arrays. DATUM is the first known method that can mask any given number of failures, requires an optimal amount of redundant storage space, and spreads reconstruction accesses uniformly over disks in the presence of failures without needing large layout tables in controller memory. Our approach is based on information dispersal, a coding technique that admits an efficient hardware implementation. As the method does not restrict the configuration parameters of the disk array, many existing RAID organizations are particular cases of DATUM. A detailed performance comparison with two other approaches shows that DATUM's response times are similar to those of the best competitor when two or less disks fail, and that the performance degrades gracefully when more than two disks fail.

1 Introduction

Disk arrays [15] offer significant advantages over conventional disks. Fragmentation of the total storage space into multiple inexpensive disks allows cost-effective solutions that benefit from the aggregate bandwidth of the component disks, and from the smaller seek latencies associated with shorter arms. The mean time between failures (*MTBF*) for the array as a whole decreases linearly with the number of independent component disks, possibly resulting in mere days before catastrophic failure and loss of data.

It is therefore necessary to store, along with the data, redundant information in order to recover from disk failures. Most existing approaches like RAID-5 [15] and parity declustering [10] rely on parity calculations to provide tolerance against a single disk failure. However, there are several reasons for considering architectures capable of masking multiple simultaneous disks failures [3, 4, 7, 9]:

- Applications are placing stronger demands on storage subsystems as a result of manipulating large audio and video objects in real-time. Existing analyses [4] show that increasing the number of check sectors per stripe is a better (i.e. more cost-effective) approach to improving reliability than increasing the number of reliability groups or relying on the likely improvements of each disk's MTBF.
- Disks can exhibit latent sector failures, in which an unavailable group of sectors is detected only when trying to access these sectors. Data are irretrievably lost if the array tolerates a single failure and a latent sector failure is detected when trying to reconstruct a crashed disk.
- Safety-critical applications can require large MTTDLs that can not be achieved by tolerating single failures.
- Some approaches [5, 6] distribute the controller's functionality among loosely coupled nodes (to avoid the centralized controller as a single point of failure). In such scenarios, the contents of one or more correct disks may become inaccessible because some other component has failed, e.g. the communication network. Communication failures often are much more likely to happen than several actual disk crashes. Storage systems must continue to provide correct service in these situations.

This work was partially supported by grants from the Air Force Office of Scientific Research, California MICRO, Oakland, and Symbios Logic, Wichita.

In this paper we present DATUM (Disk Arrays with opTimal storage, Uniform declustering and Multiple-failure tolerance), a novel declustering method that tolerates multiple disk failures in a disk array. DATUM allows the user ample freedom in configuring the disk array, without constraining the values of array parameters; up to $n-1$ failures can be tolerated in an array of n disks. DATUM uses the theoretical minimum amount of storage space for storing redundant data in the array. Sectors containing redundancy are uniformly distributed over all disks, thereby eliminating bottlenecks when failures occur and easing the burden on surviving disks during on-line reconstruction [10]. Our approach can accommodate distributed sparing as well, with similar benefits. Datum also performs the minimal number of disk accesses to implement small writes. Thus, DATUM is the first known method for tolerating an arbitrary number of failures that is optimal with respect to both storage space and write overhead, and that distributes redundant data uniformly by using a set of layout functions that can be evaluated efficiently with very low memory requirements.

Our approach is based on the information dispersal algorithm (*IDA*) [17], a coding technique for which a fast VLSI implementation exists [2]. We tested an implementation of DATUM against implementations of parity declustering [10] and EVENODD [3], by utilizing a very accurate disk array simulator [8]. Our results show that despite its generality, DATUM has the same performance as its best competitor method for the failure scenarios they have in common, and that response times continue to degrade gracefully beyond that point: the scenarios tolerated only by DATUM having more than two failures.

Section 2 describes the system model, terminology, and failure assumptions for the rest of the paper. In Section 3, we describe the basics of the IDA and its application to DATUM. The location functions that result in the uniform, declustered layout of data and redundancy are presented in Section 4. Section 5 evaluates DATUM's performance for diverse workloads and failure scenarios. Related studies are described in Section 6, and a conclusion is offered in Section 7.

2 System Model

A disk array has a total of n identical disks. Each disk has an independent controller. Different disks can service independent requests in parallel. Each controller provides the abstraction of a linear address space of sector numbers within the disk it controls, thus hiding bad media portions and the position of each sector in terms of surface, track and sector within track. For simplicity, we assume that each disk has the same number of usable sectors.

There is also an *array controller*, that translates user requests into individual operations on the disks of the array, supervises their completion, and reconfigures the array in the presence of failures. It can be implemented in software, in hardware, or as a hybrid. It also provides a linear address space to the disk array *clients*. Disks can fail during their operation; the failure of a disk is detected when a read or write operation on it fails. We assume the array controller can identify a failed disk. We do not discuss how to tolerate failures in other components of the system, such as interconnection buses or array controllers [6, 12].

Disk space is logically structured into *stripe units*; each stripe unit has a fixed number of sectors, and contains either user data or redundant data. DATUM services accesses involving an integral number of stripe units at the client level. A *stripe* consists of a fixed number of user data stripe units, plus a number of redundant stripe units computed from the user data units. The existence of redundant units is transparent to the clients, that have access to a linear address space containing only user data.

3 Information Dispersal

The information dispersal algorithm (*IDA*) [17] encodes a sequence $F = (d_1, d_2, \ldots, d_m)$ of m integers into a sequence of $m+f$ integers in such a way that any m of the $m+f$ integers suffice to recover F. The IDA transformation is linear and of the form

$$(d_1, d_2, \ldots, d_m) \cdot T = (e_1, e_2, \ldots, e_m, e_{m+1}, \ldots, e_{m+f}) \tag{1}$$

where T is an $m \times (m+f)$ matrix having the property that any m columns are linearly independent. IDA can recover the user data F from any m of the $m+f$ values as follows: Suppose we have access to the sequence $(e_{i_1}, e_{i_2}, \ldots, e_{i_m})$. We know that

$$(d_1, d_2, \ldots, d_m) \cdot (t_{i_1}\ t_{i_2}\ t_{i_3} \cdots t_{i_m}) = (e_{i_1}, e_{i_2}, \ldots, e_{i_m})$$

where t_i denotes the i^{th} column of transformation T and the appropriate columns of T are shown. Since these m columns are linearly independent, we can invert this matrix and obtain F from the m integers.

In our application of IDA, the sequence F represents m equal-sized portions of user data (disk array stripe units) and the $m+f$ values represent encoded data including the redundant data (again stripe units.)

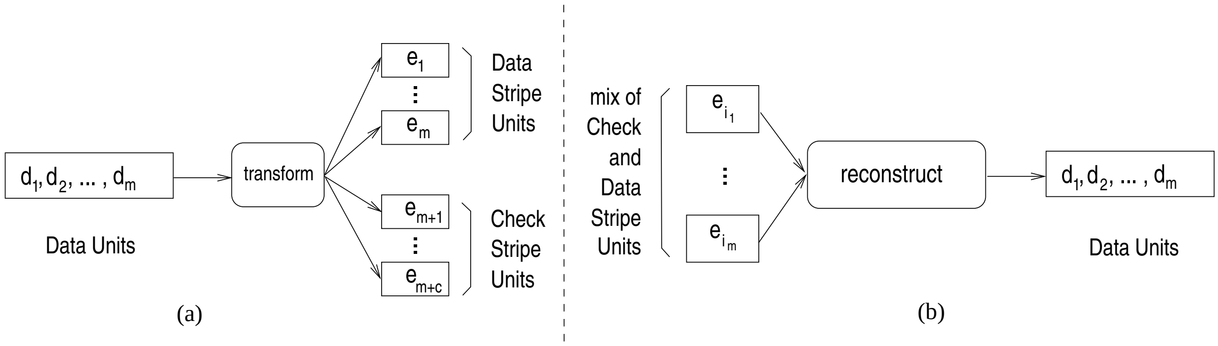

Figure 1: Information Dispersal Algorithm: (a) Encoding; (b) Decoding

Each stripe unit is stored on a different disk in the array. Any data stripe can be reconstructed if m or more disks are correct; that is, if f or fewer disks have failed. Figure 1(a) illustrates how redundancy is calculated when writing a whole stripe of client data to the array; part (b) shows how the whole stripe is reconstructed from any m encoded stripe units. We require an additional property for DATUM to obtain RAID Level 5-like behavior: m of the encoded stripe units must be identical to the m client data stripe units. This can be done only if the whole user data portion of the stripe is also recorded in the array in the clear; that is, the encoding must be *systematic* [11]. The matrix T, shown in Equation (2), consists of the identity matrix and f more columns so that the set of $m+f$ columns satisfies the linear independence condition.

$$T = \begin{pmatrix} 1 & 0 & \ldots & 0 & t_{1,m+1} & t_{1,m+2} & \cdots & t_{1,m+f} \\ 0 & 1 & \ldots & 0 & t_{2,m+1} & t_{2,m+2} & \cdots & t_{2,m+f} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 & t_{m,m+1} & t_{m+2} & \ldots & t_{m,m+f} \end{pmatrix} \quad (2)$$

Rabin [17] and Preparata [16] describe ways of constructing T that result in low computational costs; however, none achieves the systematic property yielding m clear stripe units. We can always obtain a matrix T with the desired structure using elementary row operations on a candidate matrix obtained by any method.

IDA makes no assumption regarding the amount of space required to represent the encoded values. Maximum distance separable (*MDS*) error-correcting encoding [11], which is similar to IDA, ensures that the representation of m stripe units, each of size s bytes, as $m+f$ stripe units will require exactly $(m+f)\times s$ bytes. This is accomplished by using finite fields to represent the values. In fact, with this additional requirement, the IDA transformation matrix becomes the codeword generator matrix for an MDS code; thus IDA generalizes MDS encoding. DATUM uses only the theoretical minimum amount of disk space to store each stripe in such a way that its contents are recoverable even if f stripe units are missing.

DATUM's application of IDA is based on the $GF(256)$ finite field. Elements of our transformation matrices range over $GF(256)$, so we can represent any element in a single byte without wasting space. The encoding and decoding operations are done one byte at a time in each stripe unit. When we denote client data stripe unit i in terms of byte values as follows $(d_{1,i}, d_{2,i}, \ldots, d_{s,i})$, Equation (1) yields the j^{th} byte for each of the encoded stripe units.

$$(d_{j,1}, d_{j,2}, \ldots, d_{j,m}) \cdot T = (e_{j,1}, e_{j,2}, \ldots, e_{j,m}, e_{j,m+1}, \ldots, e_{j,m+f}) \quad (3)$$

The two field operations can be implemented very efficiently: addition is the bitwise exclusive-or operation, and multiplication can be implemented using fast table lookups using a discrete logarithm [11] table of size 256 bytes. In general, for a finite field of η elements we can store the logarithm table with η elements rather than the structure with $\Theta(\eta^2)$ elements that the full multiplication table requires.

A *small write* occurs when a single stripe unit is written by a client application. Small writes are typically expensive in RAIDs as the redundant data must be updated as well; our byte-wise interleaving scheme of Equation (3) allows us to process them in the following manner. We denote components of redundant stripe unit j as $(e_{1,m+j}, \ldots, e_{s,m+j})$ (where s is the size of the stripe unit, and $1 \le j \le f$) and the elements of T by $t_{i,j}$. Assume, without loss of generality, that data stripe 1 has been updated yielding $(d'_{1,1}, \ldots, d'_{s,1})$ as the result of a small write. Each of the f check stripe units must be updated as well: for $i = 1, \ldots, s$ and $j = 1, \ldots, f$ we have

$$e_{i,m+j} = d_{i,1}t_{1,m+j} \oplus d_{i,2}t_{2,m+j} \oplus \cdots \oplus d_{i,m}t_{m,m+j}$$
(previous values), and
$$e'_{i,m+j} = d'_{i,1}t_{1,m+j} \oplus d_{i,2}t_{2,m+j} \oplus \cdots \oplus d_{i,m}t_{m,m+j}$$
(only data stripe 1 changed),

thus we obtain the update expression:

$$e'_{i,m+j} = d'_{i,1}t_{1,m+j} \oplus d_{i,1}t_{1,m+j} \oplus e_{i,m+j}. \quad (4)$$

Therefore, to implement a small write, it is necessary to: *i)* read the old values of the data unit being written and the f redundant units, *ii)* recompute the check stripe unit values as in Equation (4), and finally *iii)* write the new data stripe value as well as the f check stripe units. As a consequence of the Singleton bound [11], it is not possible to write less than $f+1$ stripe units if the array is to tolerate up to f failures. Accordingly, DATUM performs the optimal number $f+1$ disk writes per small write operation. The literature discusses system-level optimizations that do not require the user to wait until the $f+1$ writes complete successfully. For instance, updates to redundant stripes can be deferred while logging them to disk [22], while writing them to non-volatile cache [12], or by doing nothing at all [18]. These optimizations are in principle applicable to any disk array architecture, including DATUM, and we do not consider them here.

4 Layout Functions

One of the contributions of this work is a suite of functions that map each client address to the set of disks and sectors (i.e. offsets within each participating disk) where the corresponding user data and redundant data are stored in the array. In general, performance under failure is better when the units of each stripe are written in a subset of the n disks of the array. Our layout functions meet the first five layout goals set forth by Holland and Gibson [10]. When a disk fails and each stripe is distributed over all the disks of the array (like in RAID-5 [15]), an access to a random stripe unit has probability $1/n$ of hitting the failed disk, thus resulting in $n-1$ accesses to all the other stripe units. On the other hand, when each stripe is laid out over $k \leq n$ disks, the probability of hitting the failed disk remains the same, but the overhead is only $k-1$ accesses when a request to the failed disk is served. Similar arguments show that client applications also experience a much less substantial performance degradation during on-line reconstruction of a failed disk, as only some of the stripes being reconstructed will have units in each of the surviving disks.

In the remaining part of this section we present the functions that convert between client-level addresses and disks and sectors in DATUM. After that, we discuss some of the properties of our functions.

4.1 Set of Disks for a Stripe

Given an address for a stripe unit, it is necessary to compute the set of disks that have fragments of the corresponding stripe (in case the access being serviced involves on-the-fly reconstruction). Space has to be allocated uniformly over all disks to avoid bottlenecks. We represent the disks that store units for a given stripe with a *k-combination*: a tuple $< X_1, \ldots, X_k >$ where each $X_i \in [0, n-1]$ (a disk identifier). Furthermore we require that in each k-combination: $X_i < X_{i+1}, i = 1, \ldots, k$. This is to avoid ambiguity as we intend to use k-combinations as sets. There are $num\_combs = \binom{n}{k}$ possible k-combinations.

We define the total order $\succ$ over the set of combinations as: $< X_1, \ldots, X_k > \succ < Y_1, \ldots, Y_k >$ if and only if either $X_k > Y_k$, or there exists a $j < k$ such that $X_i = Y_i$ for $k \geq i > j$ and $X_j > Y_j$. Given a k-combination $c =< X_1, \ldots, X_k >$, its position in the total order is given by (the reader interested in the proofs of the results mentioned in this paper is directed to [1]):

$$loc(< X_1, \ldots, X_k >) = \binom{X_k}{k} + \binom{X_{k-1}}{k-1} + \cdots + \binom{X_1}{1}$$

Since the *loc* mapping is a bijection, we can compute its inverse *invloc*. The following algorithm computes the k-combination given its position in the order $\succ$; the answer resides in integer array `c`.

```
invloc(position)
   for (int i=k; i≥1; i--)
      l = i;
      while (choose(l,i) ≤ j)
         l = l+1;
      c[i] = l-1;
      position = position - choose(l-1,i);

   return(c)
```

Stripe unit *addr* as seen by the user is contained in stripe number $str = \lfloor addr/m \rfloor$ of the DATUM array, since each stripe has m data units—the existence of redundant units is transparent to users. Given that in general there will be more stripes than k-combinations, we map *str* into the disks of the k-combination $invloc(str \bmod num\_combs)$. That is, the mapping from stripes into disks repeats cyclically every *num_combs* stripes.

4.2 Allocation of Redundant and Data Stripe Units

Given the disks $c =< X_1, \ldots, X_k >$ corresponding to a stripe *str*, we must choose m of these disks to hold the data units, and the remaining $k-m$ will hold the redundant units. This choice must be made in such a way that the redundant units are evenly distributed among disks to maximize available disk bandwidth in the failure-free case (when redundant units are never accessed for reconstruction, so a disk containing more redundant data than data units would be underutilized).

We take disks in positions
$\lfloor str/num\_combs \rfloor f, (\lfloor str/num\_combs \rfloor f{+}1) \bmod k, \ldots,$
$(\lfloor str/num\_combs \rfloor f + f - 1) \bmod k$
for the redundant units, and the remaining disks for data units. That is, we take the first f disks of each tuple for the first *num_combs* stripes, then move on to the next f positions of c, and so on. It is easy to see that the same positions will be used for redundant units after $lcm(k,f)/f$ iterations of this procedure, where $lcm(k,f)$ denotes the least common multiple of k and f. k/f iterations are enough in the best case, and at most k in the worst case. This generalizes the function of [10] for $f=1$, that takes in turn each of the disks of the tuple to hold the parity redundant unit.

4.3 Offset of Stripe Units within Disks

Once we have found the set of disks $c =< X_1, \ldots, X_k >$ corresponding to a stripe *str*, we have to determine the offsets at which each stripe unit of *str* is stored in each X_i. We start using each disk at offset 0 in its address space, and from then on we allocate each unit to the lowest free offset in the disk where it must be stored. Therefore, the sector offset in disk X_i where *str*'s unit is allocated equals the number of times X_i appeared in k-combinations smaller than $< X_1, \ldots, X_k >$, times the number of sectors per stripe unit (that is a constant of the array).

We need to determine the number of times $\#X_i$ that the value X_i has appeared in k-combinations c' such that $c \succ c'$. It turns out that $\#X_i$ can be computed in the following way:

$$\begin{aligned} \#X_k &= loc(c) - \binom{X_k}{k}, \\ \#X_j &= \sum_{i=1}^{j-1} \binom{X_i}{i} + \sum_{i=j+1}^{k} \binom{X_i - 1}{i-1}, \\ &\quad \text{for } j = 1, \ldots, k-1. \end{aligned}$$

Figure 2 shows the first iteration of our layout functions for $n=4, k=3, m=2$ (i.e. $f=1$, a single failure is tolerated). The figure displays the assignment of stripe units to disks. For this example, there are four 3-combinations, so each of the three cycles in Figure 2 contains four stripes. In the first cycle we map each stripe's redundant unit to the first element of each 3-combination, in the second cycle we take the second, and in the third cycle we take the last. The figure shows that each disk contains the same number of redundancy units in each iteration.

To illustrate the application of our layout functions, consider how the stripe unit with $addr = 9$ in the user's address space is located in the array. This unit forms part of stripe $\lfloor addr/m \rfloor = 4$ in the array's address space. Stripe 4 is mapped according to the 3-combination $invloc(4 \bmod \binom{n}{k}) = invloc(0) =< 0,1,2 >$. Of the three disks of the 3-combination, redundant information will be in the position $\lfloor 4/\binom{n}{k} \rfloor f = 1$, that corresponds to disk number 1 in the 3-combination. Since each stripe contains 2 data units and $9 \bmod 2 = 1$, the desired unit is the second data unit of the stripe—therefore, it is mapped to disk 2 (the last component of the 3-combination). It remains to determine the offset of disk 2 that must be accessed: this equals $\#(2, < 0,1,2 >)$ $+\binom{n}{k} \lfloor stripe\_num/\binom{n}{k} \rfloor$. The first term equals 0, the second term equals 3; thus the stripe unit is located at disk 2, offset 3 as shown in the figure.

4.4 Properties of the Layout Functions

The set of layout functions used in DATUM can tolerate multiple failures: each stripe is split into k stripe units, and units of the same stripe are never stored in the same disk. The data in every stripe is recoverable even if up to f disks of the array become unavailable.

In order to reconstruct the L bytes of a stripe, DATUM reads L/m bytes from each of any m surviving disks. The bytes needed from any disk are contiguous (because sectors of the same stripe unit are contiguous on disks), so the seek delays are minimal. Therefore, reconstruction accesses (either to service a client request or to reconstruct a failed disk) in DATUM are minimal and contiguous.

Redundant units are uniformly distributed on all the disks. Since these must be updated during every user write, the aggregate load is shared equally among all disks. During degraded-mode operation, the additional load of reconstructing stripe units on failed disks is evenly distributed among the surviving disks. Similarly when a failed disk is replaced by a spare, it must be reconstructed to leave the array in a failure-free state.

Offset	Disk 0	Disk 1	Disk 2	Disk 3
0	S0,R	S0,U0	S0,U1	S1,U3
1	S1,R	S1,U2	S2,U4	S2,U5
2	S2,R	S3,R	S3,U6	S3,U7
3	S4,U8	S4,R	S4,U9	S5,U11
4	S5,U10	S5,R	S6,R	S6,U13
5	S6,U12	S7,U14	S7,R	S7,U15
6	S8,U16	S8,U17	S8,R	S9,R
7	S9,U18	S9,U19	S10,U21	S10,R
8	S10,U20	S11,U22	S11,U23	S11,R
	...	...	...	...

Sn,Um: Data unit with address m, contained in stripe n
Sn,R: Redundant unit of stripe n

Figure 2: Sample Layout for $f = 1$

Reconstruction is often performed on-line, i.e. while the array is servicing user requests. In order to avoid the presence of bottlenecks and to minimize the degradation of the service, reconstruction accesses are also evenly spread—because every pair of disks appears together in the same number of k-combinations. Also, *large writes* (when the access updates all the data portion of a stripe) are serviced without reading the previous contents of any disk sector.

The layout functions have low computational and memory requirements. It is convenient to store a few binomial coefficient values in memory for improving efficiency, as one or more of these functions are in the critical path of every request sent to the array.

5 Performance

We report the results of a performance comparison made by running DATUM, EVENODD [3], and parity declustering [10] under a highly-concurrent workload of small reads and writes. Our goal is to compare how these different methods perform under a spectrum of failure scenarios, from failure-free operation to three unavailable disks. We chose parity declustering as a basis for comparison because it distributes data onto disks using balanced block designs, and it uses the same encoding (parity) as the RAID family [15]. EVENODD was considered because it is the only existing scheme that can tolerate two failures while using the same minimal amount of redundant storage, regardless of array size. The redundant units are rotated cyclically over the different disks of the array in our implementation of EVENODD, as suggested in [3].

The experiments were run on RaidFrame [8], a tool where implementations of RAID architectures can be tested and evaluated. RaidFrame's synthetic workload generator generates concurrent sequences of requests for disk accesses, from multiple client threads. Each RAID architecture converts each individual user requests into sets of read and write accesses on specific disks of the array; our implementations of DATUM, parity declustering, and EVENODD are at this level in the hierarchy. Individual disk accesses are serviced by an event-driven disk simulator, calibrated to several choices of real disks. The current release of RaidFrame consists of approximately 50,000 lines of C code.

In our case, array parameters were configured as follows. The array had a total of $n = 15$ disks in every case, so all measurements are normalized to cost. Each stripe unit consisted of eight 512-byte sectors, for a total of 4Kbytes. The individual disks of the array were HP2247s, with a total storage of approximately 900 Mbytes, average seek time 10 milliseconds, and revolution time 11 milliseconds. Each disk's controller used the shortest-seek-time-first queueing policy in a queue that can hold 10 outstanding requests. Regarding the synthetic workload, it consisted of random, independent 4-Kbyte accesses (the size of one stripe unit) uniformly distributed over the user's address space. There was no caching or read-ahead used in our simulations; these optimizations are up to the client applications (e.g. a file system manager). Accesses were aligned to a stripe unit boundary. We chose these workloads because small accesses (particularly writes) are typically most demand-

	DATUM	EVENODD	Parity decl.
Failure-free	$m = 9$ $k = 10$	$m = 13$ $k = 15$	$m = 9$ $k = 10$
$f = 1$	$m = 9$ $k = 10$	$m = 13$ $k = 15$	$m = 9$ $k = 10$
$f = 2$	$m = 9$ $k = 11$	$m = 13$ $k = 15$	N/A
$f = 3$	$m = 9$ $k = 12$	N/A	N/A

Table 1: Experiment Configurations

ing for RAID architectures. The same disk array simulator and similar synthetic workloads have been used a number of times in the literature [8, 10, 22]. For larger accesses, there are a number of possible optimizations that can be implemented for any of the three schemes discussed in this section. Our implementation of the DATUM array architecture took about 2,400 lines of code.

For each failure scenario, we report the average response times for each request under workloads of pure reads and pure writes, as a function of the number of array access requests per second. Table 1 shows the configurations used for each method under each failure scenario. Each simulation ran until the size of the 95% confidence interval was within a 2% of the reported values; therefore, 95% confidence intervals are fairly small and we do not include them in the graphs. DATUM's configuration depends on the number of failures we want to tolerate.

Bestavros reports in [2] times of approximately 100 nanoseconds/byte for the VLSI implementation of IDA. Since his design does not allow pipelining, this amounts to less than 0.5 milliseconds for a 4096-byte stripe. This has been added wherever necessary to the times spent by DATUM in disk accesses. According to [2], it is possible to design a more sophisticated chip that can support pipelining and be driven by a clock ten times faster.

5.1 Failure-free Operation

For these experiments, the DATUM array was configured with $k = 10$ and $m = 9$, i.e. a single redundant unit per stripe. Figure 3 shows the performance of read and write operations for the three RAID architectures under consideration. The read operations have roughly the same performance, as each of them implements small reads in the failure-free case by simply reading the stripe unit requested by the user. The cost of writes is similar for DATUM and parity declustering; in both cases the array reads the old value of the data unit being written and the redundant unit of its stripe, recalculates the redundant unit, and writes both units back to the disks. The cost of small writes in EVENODD is substantially larger since the array needs to read, recompute and write *two* redundant units for each small write; this makes the array reach its point of saturation at about 180 reqs/sec, compared to 280 reqs/sec for the other two architectures.

5.2 Single Disk Failure

The DATUM array has the same configuration as in the previous section, as it must tolerate a single failure. Figure 4 shows the performance of read and write operations for the three RAID architectures.

The execution times for read operations are larger than in the failure-free. When a disk is unavailable, the reads directed to it must be implemented by reading all the remaining units of the stripe and reconstructing the missing data. EVENODD would have a better response time if stripe units were three or four times larger in this same configuration because each stripe spans the 15 disks of the array (as opposed to a subset of 10 disks for DATUM and parity declustering), so reads in parallel from different disks would make a difference. As in the failure-free case, EVENODD writes are slower because two redundant units must be updated. The performance difference is due to the fact that EVENODD can always tolerate two failures, while DATUM can be configured to the reliability requirements of each installation For every architecture, the writes are slower than in the failure-free case because a reconstruction must be made when writing an unavailable data unit.

5.3 Double Disk Failure

We compare DATUM with EVENODD only, as parity declustering does not tolerate more than one failure. The DATUM array is configured as $m = 9, k = 11$—with two redundant units per stripe. Figure 5 depicts the response times for DATUM and EVENODD.

A larger fraction of the small reads require a reconstruction here, increasing the average response time. The read performance of the two schemes is quite similar, even though EVENODD seems to be able to sustain a slightly larger arrival rate before saturation. We attribute this fact to the larger stripe size that places less load on each individual disk. Regarding writes, the performance of the two schemes is also very similar. EVENODD's performance is almost the same in the single- and double-failure scenarios; we attribute this to the fact that each stripe spans all the disks, so the reconstructions that must be performed when an unavailable data

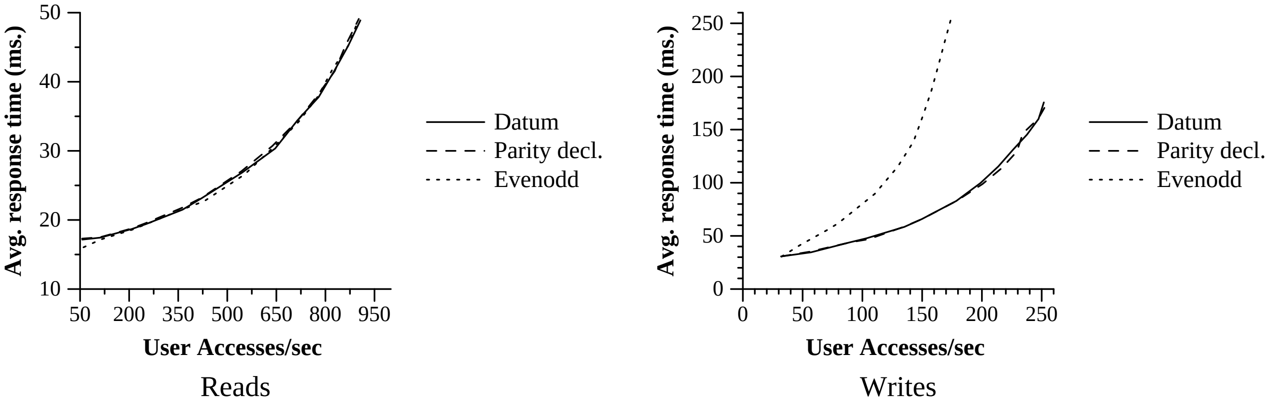

Figure 3: Response times, failure-free case

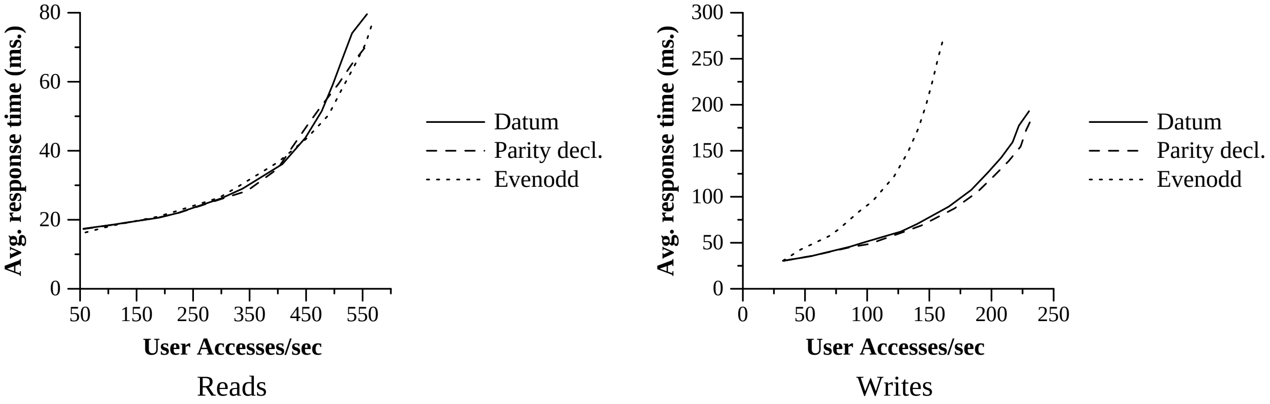

Figure 4: Response times, one failure

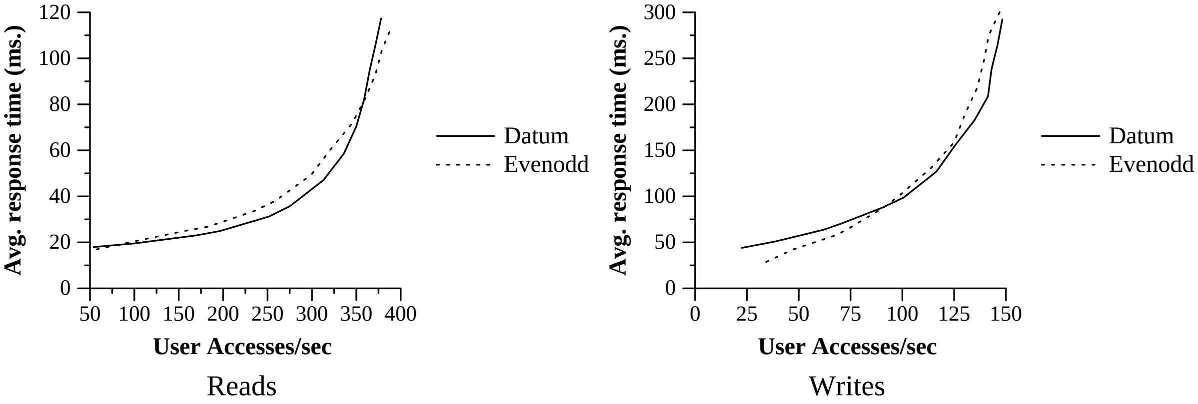

Figure 5: Response times, two failures

unit is written place additional demands on every surviving disk.

5.4 Triple Disk Failure

DATUM is the only method among the three considered that can tolerate three or more failures. The DATUM array is configured as $m = 9, k = 12$—with three redundant units per stripe.

The response times are shown in Figure 6. They are naturally larger than in the previous failure scenarios; but the response curves have similar characteristics. Moreover, for each failure scenario, the DATUM curves have a first derivative that is roughly within a constant factor of the first derivative of the previous scenario. This constant rate of degradation suggests that DATUM degrades gracefully as more failures occur; we plan to make additional measurements to test the accuracy of this conjecture.

6 Related Work

We discuss three main groups of relevant studies, although some of them belong to more than one category: coding methods, parity declustering, and RAID architectures. Coding methods for tolerating multiple failures have been proposed by Gibson *et al* [9]; their f-dimensional parity method can tolerate multiple failures, but it achieves the optimal redundancy only when $k = i^f$ for some integer i, and in that case it needs fi^{f-1} separate disks for storing redundant information. The other codes also require an increasing number of redundant disks as a function of the reliability group size, unlike DATUM where the size of a redundant unit is independent of the number of units per stripe. EVENODD [3] tolerates two disk failures, and can be efficiently implemented on existing controller hardware. Even though EVENODD does not inherently prevent declustering, the layout proposed in [3] makes every disk a member of every stripe; this limits the array to a maximum of 259 disks, and is likely to result in performance degradation during on-line reconstruction [10]. Burkhard and Menon [4] analyze the additional dependability gained by using MDS codes to tolerate multiple failures; that paper concentrates on reliability calculations, and does not address the issues of run-time performance or array layout. They conclude that increasing the fault-tolerance within each reliability group is more cost effective than partitioning the array into more groups, or relying on improvements to the reliability of each disk. Their conclusions are applicable to DATUM, as our IDA is an MDS code as well.

Several existing studies investigate parity declustering, i.e. the uniform distribution of data and parity redundancy over the array when each stripe uses a subset of the disks. Muntz and Lui [14] suggest balanced incomplete block designs (BIBDs) to achieve uniform distribution; Holland and Gibson [10] implemented the scheme and evaluated its performance. BIBDs have two main drawbacks: they do not exist for every array configuration (thus only certain values of the array parameters can be implemented), and they can require substantial amounts of controller memory for table storage, of the order of several Mbytes. To address the first problem, Schwabe and Sutherland [19] have extended the BIBD generation techniques, and bounded the degradation in uniformity incurred when using approximate block designs. Unlike these approaches, DATUM does not place restrictions on the configuration parameters of the array, and does not need to store a block design in memory as the location functions implement the mappings (our mappings are equivalent to a complete block design that is not stored in memory). Pseudo-random permutation declustering has been proposed by Schwarz and Burkhard [21] as well as Merchant and Yu [13]. ACATS declustering, proposed by Schwarz and Burkhard [20], works deterministically for any array configuration thus avoiding the need for a pseudo-random generator and explicit BIBD configuration storage.

RAID architectures were originally proposed by Patterson *et al* in [15]. From the solutions presented there, RAID-5 provides tolerance against a single failure and does not decluster the parity information. Multiple-erasure SID [7] is a RAID scheme for multimedia workloads that can tolerate an arbitrary number of failures while performing contiguous accesses to the disks. Solutions for the layout constraints exist only for some array configurations, and the fraction of the storage devoted to redundant information is at least $f/(\sqrt{m}+f)$, compared to the optimal $f/(m+f)$.

7 Conclusions

We presented DATUM, an architecture for redundant declustered disk arrays that can tolerate an arbitrary number of simultaneous disk failures. Since the trend for the coming years dictates larger number of smaller disks at each installation and/or distributed striping approaches, there is a pervasive need for going beyond the reliability provided by schemes that can tolerate single or even double failures. DATUM relies on information dispersal as a coding method. The layout functions pre-

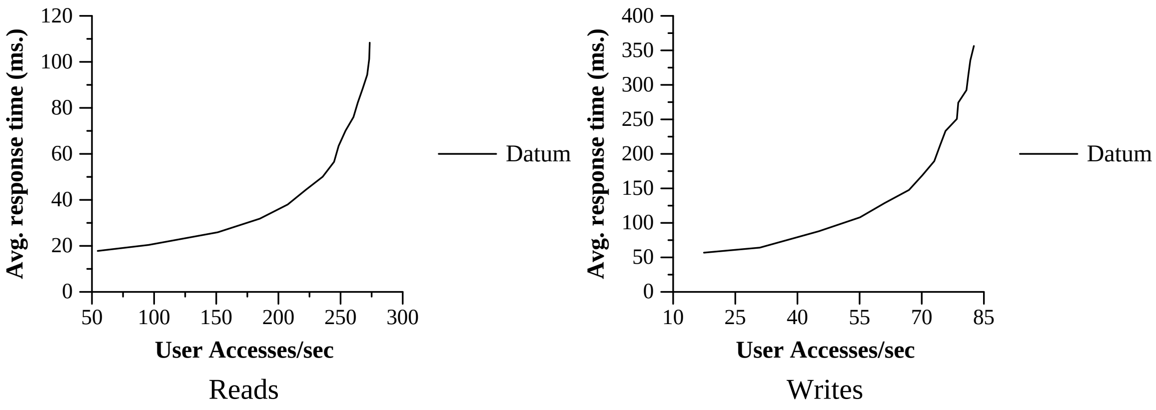

Figure 6: Response times, three failures

sented achieve uniform declustering of the user data and redundant data over the disks, and they place no restrictions on the configuration parameters of the array. The functions can be evaluated efficiently, and have very low memory requirements. Several existing layout policies are special cases of DATUM.

DATUM is the only existing method that requires the optimal amount of redundant storage, generalizes to any number of failures and any array configuration, and declusters uniformly user data and redundant data over the disk array. It also accommodates distributed sparing. The number of disk accesses performed on writes are also minimal. A VLSI implementation of the coder and decoder already exist, and its speed is adequate for this purpose. We compared the performance of DATUM with two other RAID architectures, and found that it performs as well as the best of its competitors for the common failure scenarios they can tolerate. Moreover, DATUM degrades gracefully as the number of failures increases beyond the number of failures tolerated by the other two architectures.

Acknowledgements

We want to thank Jim Zelenka and William Courtright for supporting RaidFrame, and the anonymous reviewers for contributing to improve the presentation.

References

[1] G. Alvarez, W. Burkhard, and F. Cristian. Tolerating multiple failures in raid architectures with optimal storage and uniform declustering. Technical report CS96-500, UCSD–CSE, November 1996. Also available by anonymous `ftp` from `elm.ucsd.edu/pub/galvarez/datum.ps.Z`.

[2] A. Bestavros. SETH: A VLSI chip for the real-time information dispersal and retrieval for security and fault-tolerance. In *Proc. Intl. Conf. on Parallel Processing*, 1990.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. Even-odd: An efficient scheme for tolerating double disk failures in RAID architectures. In *Proc. of the Annual International Symposium on Computer Architecture*, pages 245–54, 1994.

[4] W. Burkhard and J. Menon. Disk array storage system reliability. In *Proc. of the International Symposium on Fault-tolerant Computing*, pages 432–41, 1993.

[5] L. Cabrera and D. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–39, 1991.

[6] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *Proc. of the Annual International Symposium on Computer Architecture*, pages 52–63, 1993.

[7] A. Cohen and W. Burkhard. SID for efficient reconstruction in fault-tolerant video servers. In *Proc. of ACM Multimedia*, pages 277–86, November 1996.

[8] W. Courtright, G. Gibson, M. Holland, and J. Zelenka. A structured approach to redundant disk array implementation. In *Proc. of the International Symposium on Performance and Dependability*, 1996.

[9] G. Gibson, L. Hellerstein, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. In *Proc. of the International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems, pages 123–32, 1989.

[10] M. Holland and G. Gibson. Parity declustering for continuous operation on redundant disk arrays. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 23–35, 1992.

[11] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, The Netherlands, 1977.

[12] J. Menon and J. Cortney. The architecture of a fault-tolerant cached RAID controller. In *Proc. of the Annual International Symposium on Computer Architecture*, pages 76–86, 1993.

[13] A. Merchant and P. Yu. Performance analysis of a dual striping strategy for replicated disk arrays. In *Proc. of the Second International Conference on Parallel and Distributed Information Systems*, pages 148–157, San Diego, January 1993.

[14] R. Muntz and J. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th VLDB Conference*, pages 162–173, Brisbane, June 1990.

[15] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[16] F. Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–4, September 1989.

[17] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–48, April 1989.

[18] S. Savage and J. Wilkes. AFRAID— a frequently redundant array of independent disks. In *Proc. of the USENIX 1996 Annual Technical Conference*, January 1996.

[19] E. Schwabe and I. Sutherland. Improved parity-declustered layouts for disk arrays. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 76–84, Cape May, N.J., June 1994.

[20] T. Schwarz and W. Burkhard. Almost complete address translation (ACATS) disk array declustering. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 324–331, New Orleans, June 1996.

[21] T. Schwarz and W. Burkhard. Reliability and performance of RAIDs. *IEEE Transactions on Magnetics*, 31:1161–1166, March 1995.

[22] D. Stodolsky, G. Gibson, W. Courtright, and M. Holland. A redundant disk array architecture for efficient small writes. Technical report CMU-CS94-170, Carnegie Mellon Univ., July 1994.