Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays

Draft copy submitted to the Journal of Distributed and Parallel Databases.
A revised copy is published in this journal, vol. 2 no. 3, July 1994..

Mark Holland
Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3890
(412) 268-5237
mark.holland@ece.cmu.edu

Garth A. Gibson
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3890
(412) 268-5890
garth.gibson@cs.cmu.edu

Daniel P. Siewiorek
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3890
(412) 268-2570
dan.siewiorek@cs.cmu.edu

Architectures and Algorithms for On-Line Failure Recovery In Redundant Disk Arrays[1]

Abstract

The performance of traditional RAID Level 5 arrays is, for many applications, unacceptably poor while one of its constituent disks is non-functional. This paper describes and evaluates mechanisms by which this disk array failure-recovery performance can be improved. The two key issues addressed are the data layout*, the mapping by which data and parity blocks are assigned to physical disk blocks in an array, and the* reconstruction algorithm*, which is the technique used to recover data that is lost when a component disk fails.*

The data layout techniques this paper investigates are variations on the declustered parity *organization, a derivative of RAID Level 5 that allows a system to trade some of its data capacity for improved failure-recovery performance. Parity declustering improves the failure-mode performance of an array significantly, and a parity-declustered architecture is preferable to an equivalent-size multiple-group RAID Level 5 organization in environments where failure-recovery performance is important. The presented analyses also include comparisons to a RAID Level 1 (mirrored disks) approach.*

With respect to reconstruction algorithms, this paper describes and briefly evaluates two alternatives termed stripe-oriented reconstruction *and* disk-oriented reconstruction*, and establishes that the latter is preferable as it provides faster reconstruction. The paper then revisits a set of previously-proposed reconstruction optimizations, evaluating their efficacy when used in conjunction with the disk-oriented algorithm. The paper concludes with a section on the reliability vs. capacity trade-off that must be addressed when designing large arrays.*

1. Introduction

The performance of a storage subsystem during its recovery from a disk failure is crucial to applications such as on-line transaction processing (OLTP) that mandate both high I/O performance and high data reliability. Such systems demand not only the ability to recover from a disk failure without losing data, but also that the recovery process (1) function without taking the system off-line, (2) rapidly restore the system to its fault-free state, and (3) have minimal impact on system performance as observed by users. Condition (2) ensures that the system's vulnerability to data loss is minimal, while conditions (1) and (3) provide for on-line failure recovery. A good example is an airline reservation system, where inadequate recovery from a disk crash can cause an interruption in the availability of booking information and thus lead to flight delays and/or revenue loss. Furthermore, because fault-tolerant storage systems exhibit degraded performance while recovering from the failure of a component disk, the fault-free system load must be kept light enough for performance during recov-

1. Portions of this material are drawn from papers at the 5th Conference on Architectural Support for Programming Languages and Operating Systems, 1992, and at the 23rd Symposium on Fault-Tolerant Computing, 1993.

ery to be acceptable. For this reason, a decrease in performance degradation during failure recovery can translate directly into improved fault-free performance. With this in mind, the twin goals of the techniques discussed in this paper are to minimize the time taken to recover the content of a failed disk onto a replacement, that is, to restore the system to the fault-free state, and to simultaneously minimize the impact of failure recovery on the performance of the array (throughput and response time) as observed by users.

Fault-tolerance in a data storage subsystem is generally achieved either by *disk mirroring* [Bitton88, Copeland89, Hsiao91], or by *parity encoding* [Arulpragasam80, Gibson93, Kim86, Park86, Patterson88]. In the former, one or more duplicate copies of each user data unit are stored on separate disks. In the latter, commonly known as Redundant Arrays of Inexpensive[2] Disks (RAID) Levels 3, 4, and 5 [Patterson88], a small portion (as large a 25%, but often much smaller) of the array's physical storage is used to store an error correcting code computed over the file system's data. Studies [Chen90a, Gray90] have shown that, due to superior performance on small read and write operations, a mirrored array, also known as RAID level 1, can deliver higher performance to OLTP workloads than can a parity-based array. Unfortunately, mirroring is substantially more expensive -- its storage overhead for redundancy is 100%; that is, four or more times larger than that of typical parity encoded arrays. Furthermore, recent studies [Stodolsky93, Menon92a, Rosenblum91] have demonstrated techniques that allow the small-write performance of parity-based arrays to approach that of mirroring. This paper, therefore, focuses on parity-based arrays, but includes a comparison to mirroring where meaningful.

Section 2 of this paper provides background on redundant disk arrays. Section 3 introduces parity declustering and Section 4 describes data layout schemes for implementing parity declustering. Section 5 describes our performance evaluation environment. Section 6 describes alternative *reconstruction algorithms*, techniques used to recover data lost when a disk fails. Section 7 then presents performance evaluation. The first part of this section compares the performance of a declustered-parity array to that of an equivalent-sized multiple-group RAID level 5 array, and the second part investigates the trade-off between disk capacity overhead and failure-recovery performance in a declustered-parity array. Section 8 describes and evaluates a set of modifications that can be applied to the reconstruction algorithm. Section 9 discusses techniques for selecting a system configuration based on the requirements of the environment and application. Section 10 summarizes the contribu-

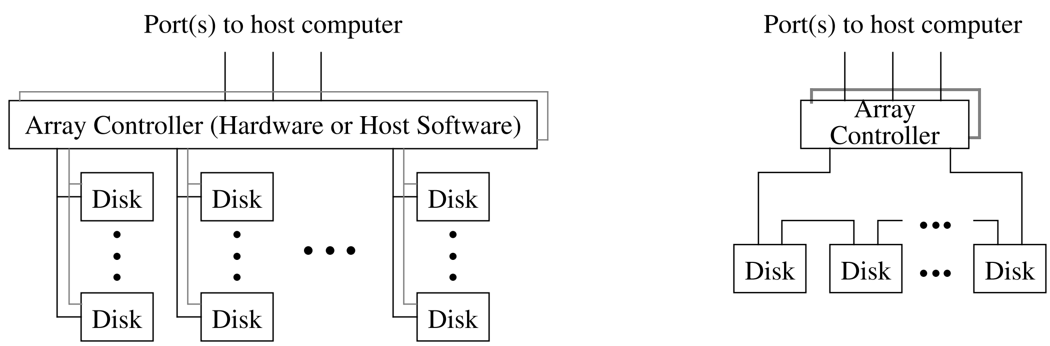
2. Because of industrial interest in using the RAID acronym and because of their concerns about the restrictiveness of its "Inexpensive" component, RAID is sometimes reported as an acronym for Redundant Arrays of Independent Disks.

tions of this paper and outlines interesting issues for future work.

2. Redundant disk arrays

Patterson, Gibson, and Katz [Patterson88] present a taxonomy of redundant disk array architectures, RAID levels 1 through 5. Of these, RAID levels 3 is best at providing large amounts of data to a single requestor with high bandwidth, while RAID levels 1 and 5 are most appropriate for highly concurrent access to shared files. The latter are preferable for OLTP-class applications, since OLTP is often characterized by a large number of independent processes concurrently requesting access to relatively small units of data [TPCA89]. For this reason and because of the relatively high cost of redundancy in RAID level 1 arrays, this paper focuses on architectures derived from the RAID level 5 organization.

Figure 1a and Figure 1b illustrate two possible disk array subsystem architectures. Today's systems use the architecture of Figure 1a, in which the disks are connected via inexpensive, low-bandwidth (e.g. SCSI [ANSI86]) links to an array controller, which is connected via one or more high-bandwidth parallel buses (e.g. HIPPI [ANSI91]) to one or more host computers. Array controllers and disk busses are often duplicated (indicated by the dotted lines in Figure 1) so that they do not represent a single point of failure [Menon93]. The controller functionality can also be distributed amongst the disks of the array [Cao93].



(a) Bus-connected (b) High bandwidth serial connected

Figure 1: Disk array architectures.

As disks get smaller [Gibson92], the large cables used by SCSI and other bus interfaces become increasingly unattractive. The system sketched in Figure 1b offers an alternative. It uses high-bandwidth serial links for disk interconnection. This architecture scales to large arrays more easily because it eliminates the need for the array controller to incorporate a large number of string controllers. While serial-interface disks are not yet common, standards for them are emerging (P1394 [IEEE93], Fibre Channel [Fibre91], DQDB [IEEE89]). As the cost of high-bandwidth serial connectivity is reduced, architectures similar to that of Figure 1b may supplant today's short, parallel bus-based arrays.

In both organizations, the array controller is responsible for all system-related activity: controlling individual disks, maintaining redundant information, executing requested transfers, and recovering from disk or link failures. The functionality of an array controller can also be implemented in software executing on the subsystem's host or hosts. The algorithms and analyses presented in this paper apply to all array control implementations.

Figure 2 shows an arrangement of data and parity on the disks of an array using the "left-symmetric" variant of the RAID level 5 architectures [Chen90b, Lee91]. Logically contiguous user data is broken down into blocks and striped across the disks to allow for concurrent access by independent processes [Livny87]. The shaded blocks, labelled *Pi*, store the parity (cumulative exclusive-or) computed over corresponding data blocks, labelled *Di.0* through *Di.3*. An individual block is called a *data unit* if it contains user data, a *parity unit* if it contains parity, and simply a *unit* when the data/parity distinction is not pertinent. A set of data units and their corresponding parity unit is referred to as a *parity stripe*. The assignment of parity blocks to disks rotates across the array in order to avoid hot-spot contention; since every update to a data unit implies that a parity unit must also be updated, the distribution of parity across disks should be balanced.

	DISK0	DISK1	DISK2	DISK3	DISK4	
0	D0.0	D0.1	D0.2	D0.3	P0	Parity Unit
1	D1.1	D1.2	D1.3	P1	D1.0	
2	D2.2	D2.3	P2	D2.0	D2.1	Data Unit
3	D3.3	P3	D3.0	D3.1	D3.2	
4	P4	D4.0	D4.1	D4.2	D4.3	Parity Stripe

Figure 2: Data layout in a 5-disk array employing the left-symmetric RAID level 5 organization.

Because disk failures are detectable [Patterson88, Gibson93], arrays of disks constitute an erasure channel [Peterson72], and so a parity code can correct any single disk failure. To see this, assume that disk number two has failed and simply note that

$$(Pi = Di.0 \oplus Di.1 \oplus Di.2 \oplus Di.3) \Rightarrow (Di.2 = Di.0 \oplus Di.1 \oplus Pi \oplus Di.2)$$

An array containing a failed disk can be restored to its fault-free state by successively reconstructing each block of the failed disk and storing it on a replacement drive. This is generally performed by a background process in either the host or the array controller. Note that an array, degraded by a failed disk, need not be taken off-line to implement reconstruction, because reconstruction accesses can be interleaved with user accesses to data on non-failed disks, and because user accesses to data on the failed disk can be serviced "on-the-fly" by immediate reconstruction. Once reconstruction is complete, the array can again tolerate the loss of any single disk, and so is again fault-free, albeit with a diminished number of on-line spare disks until the faulty drives can be physically replaced. Gibson and Patterson [Gibson93] show that a small number of spare disks suffice to provide a high degree of protection against data loss in relatively large arrays (>70 disks). Although the above organization can be easily extended to tolerate multiple disk failures, this paper focuses on single-failure toleration.

3. Parity declustering

The RAID Level 5 organization presents a problem for continuous-operation systems like OLTP. Because on-line reconstruction issues a reconstruction access to each surviving disk for each access to user data on a failed disk, the load on each surviving disk during failure recovery is its fault-free load plus the fault-free load of the failed disk. If a spare disk is available for a reconstruction process to rebuild lost data onto, then surviving disks must also bear this additional load. This per-disk load increase (approximately 100%) experienced during reconstruction necessitates that each disk's fault-free load be less than 50% of its capacity so that the surviving disks will not saturate when a failure occurs. Disk saturation may be unacceptable for OLTP applications because they mandate a minimum level of responsiveness; the TPC-A benchmark [TPCA89], for example, requires that 90% of all transactions complete in under two seconds. Long queueing delays caused by disk saturation can violate these requirements.

The *declustered parity* [Muntz90, Holland92, Merchant92] disk array organization addresses this problem. This scheme reduces a surviving disk's load increase from over 100% to an arbitrarily small percentage by increasing the amount of error correcting information that is stored in the array. This can be thought of as trading some of an array's data capacity for improved performance in the presence of disk failure.

Referring again to Figure 2, note that each parity unit protects C-1 data units, where C is the number of disks in the array. If instead the array were organized such that each parity unit protected some smaller number of data units, say G-1, then more of the array's capacity would be consumed by parity, but the reconstruction of a single data unit would require that the host or controller read only G-1 units instead of C-1. As illustrated in Figure 3, parity declustering can also be viewed as the distribution of the parity stripes comprising a logical RAID level 5 array on G disks over a set of C phys-

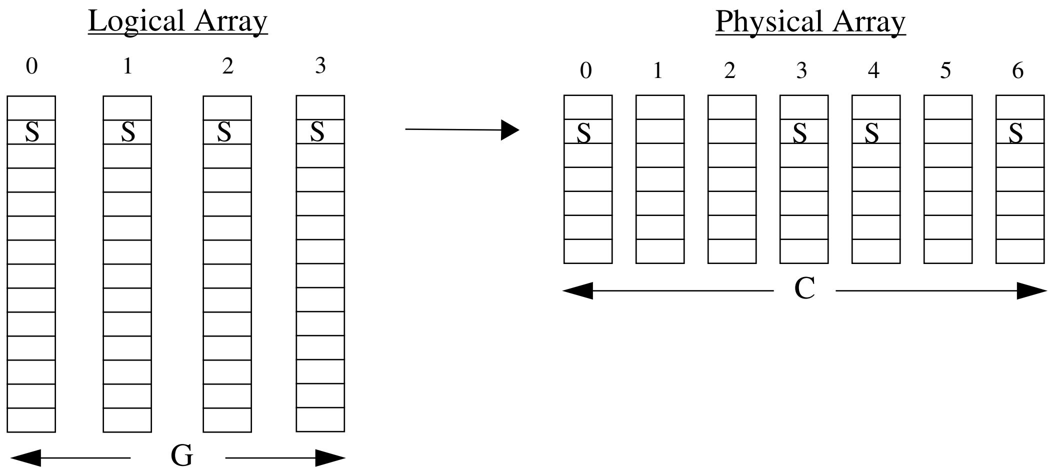


Figure 3: Declustering a parity stripe of size four over an array of seven disks.

ical disks. The advantage of this rearrangement is that not every surviving disk is involved in the reconstruction of a particular data unit; *C*-*G* disks are left free to do other work. Thus each surviving disk sees a user-invoked on-the-fly reconstruction load increase of $(G\text{-}1)/(C\text{-}1)$ instead of $(C\text{-}1)/(C\text{-}1)$ = 100%. The fraction $(G\text{-}1)/(C\text{-}1)$, which is referred to as the *declustering ratio* and denoted by α, can be made arbitrarily small either by increasing *C* for a fixed *G* as shown in Figure 3, or by decreasing *G* for a fixed *C*. As α is made smaller, performance during failure recovery improves since the load increase on each surviving disk diminishes, but more of the array's capacity is consumed by parity. For $G=2$ the declustered parity scheme is similar to separately-developed declustered mirroring approaches in that two copies of each data unit are maintained, with the mirror data being distributed over the other disks in the array [Copeland89, Hsiao91]. At the other extreme, $G=C$ (α = 1.0), parity declustering is equivalent to RAID level 5. Many of the performance plots in subsequent sections are presented with α on the x-axis.

4. Disk array layouts for parity declustering

In most disk array systems, the array controller (whether implemented in hardware or as a device driver in the host operating system) implements an abstraction of the array as a linear address space. A disk-managing application such as a file system views the disk array's data units as a linear sequence of disk sectors that can be read or written with application data. Parity units typically do not appear in this address space, that is, they are not addressable by the application program. The array controller translates addresses in this user space into physical disk locations (disk identifiers and disk offsets) as it performs requested accesses. It is also responsible for performing the redundancy-maintaining accesses implied by application write accesses. This mapping of an application's logical unit of stored data to physical disk locations and associated parity locations is referred to as the disk array's *layout*. In this section we discuss goals for a disk array layout, present our layout for declustered parity based on balanced incomplete block designs, and contrast it to a layout proposed by Merchant and Yu [Merchant92] which supports more configurations of large arrays, at the cost of higher complexity.

4.1. Layout goodness criteria

Extending from non-declustered disk array layout research [Lee90, Dibble90], we have identified six criteria for a good disk array layout.

1. *Single failure correcting*. No two stripe units in the same parity stripe may reside on the same physical disk. This is the basic characteristic of any single-failure-tolerating redundancy organization. In arrays in which groups of disks have a common failure mode, such as power or data cabling, this criteria should be extended to prohibit the allocation of stripe units from one parity stripe to two or more disks sharing that common failure mode [Schulze89, Gibson93].

2. *Distributed recovery workload.* When any disk fails, its user workload should be evenly distributed across all other disks in the array. When replaced or repaired, its reconstruction workload should also be evenly distributed.

3. *Distributed parity.* Parity information should be evenly distributed across the array to balance parity update load.

4. *Efficient mapping*. The functions mapping a file system's logical block address to physical disk addresses for the corresponding data unit and parity stripe, and the appropriate inverse mappings, must be efficiently implementable; they should consume neither excessive computation nor memory resources.

5. *Large write optimization*. The allocation of contiguous user data to disk data units should correspond to the allocation of disk data units to parity stripes. This insures that whenever a user performs a write that is the size of the data portion of a parity stripe and starts on a parity stripe boundary, it is possible to execute the write without pre-reading the prior contents of any disk data, since the new parity unit depends only on the new data.

6. *Maximal parallelism*. A read of contiguous user data with size equal to a data unit times the number of disks in the array should induce a single data unit read on all disks in the array (while

requiring alignment only to a data unit boundary). This insures that maximum parallelism can be obtained.

The first four of these deal exclusively with relationships between stripe units and parity stripe membership, while the last two make recommendations for the relationship between user data allocation and parity stripe organization. A file system is, of course, not required to allocate contiguous user data contiguously in the array's address space. In this sense the array controller has no direct control over whether or not the last two criteria are always met, even if it is implemented as a device driver in the host. The best that can be done is to meet these last two criteria for data units that are contiguous in the address space of the array.

4.2. Layouts based on balanced incomplete block designs

Our primary goal in designing a layout strategy for parity declustering was to meet our second goodness criterion: every surviving disk in the array should absorb an equivalent fraction of the total extra workload induced by a failure, including both accesses invoked by users and reconstruction accesses. An equivalent formulation is that the same number of units be read from each surviving disk during the reconstruction of a failed disk. This will be achieved if the total number of parity stripes that include a given pair of disks is constant across all pairs of disks, that is, if disks number i and j appear together in a parity stripe exactly n times for any i and j, where n is some fixed constant. As suggested by Muntz and Lui, a layout with this property can be derived from a *balanced incomplete block design* [Hall86]. This section shows how such a layout may be implemented.

A block design is an arrangement of v distinct objects into b tuples[3], each containing k elements, such that each object appears in exactly r tuples, and each pair of objects appears in exactly λ_p tuples. For example, using non-negative integers as objects, a block design with $b = 5$, $v = 5$, $k = 4$, $r = 4$, and $\lambda_p = 3$ is given in Table 1.

Tuple Number	Tuple
0	0, 1, 2, 3
1	0, 1, 2, 4
2	0, 1, 3, 4
3	0, 2, 3, 4

Table 1: A sample block design.

3. These tuples are called *block*s in the block design literature. We avoid this name as it conflicts with the commonly held definition of a block as a contiguous chunk of data. Similarly we use λ_p instead of the usual λ for the number of tuples containing each pair of objects to avoid conflict with the common usage of λ as the rate of arrival of user accesses at the array.

Tuple Number	Tuple
4	1, 2, 3, 4

Table 1: A sample block design.

This example demonstrates a simple form of block design, called a *complete block design*, which includes all combinations of exactly k distinct elements selected from the set of v objects. The number of these combinations is $\binom{v}{k}$. Note that only three of v, k, b, r, and λ_p are free variables since the following two relations are always true: $bk = vr$, and $r(k\text{-}1) = \lambda_p(v\text{-}1)$. The first of these relations counts the objects in the block design in two ways, and the second counts the pairs in two ways.

The layout we use associates disks with objects and parity stripes with tuples. For clarity, the following discussion is illustrated by the construction of the layout in Figure 4 from the block design in Table 1. To build a parity layout, we find a block design with $v = C$, $k = G$, and the minimum possible value for b. Our mapping identifies the elements of a tuple in a block design with the disk numbers on which each successive stripe unit of a parity stripe is allocated. In Figure 4, the first tuple in the design of Table 1 is used to lay out parity stripe 0: the three data blocks in parity stripe 0 are on disks 0, 1, and 2, and the parity block is on disk 3. Based on the second tuple, stripe 1 is on disks 0, 1, and 2, with parity on disk 4. In general, stripe unit j of parity stripe i is assigned to the lowest available offset on the disk identified by the j^{th} element of tuple $i \bmod b$ in the block design.

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	P1
1	D1.0	D1.1	D1.2	D2.2	P2
2	D2.0	D2.1	D3.1	D3.2	P3
3	D3.0	D4.0	D4.1	D4.2	P4

Figure 4: Example data layout in a declustered parity organization.

It is apparent from Figure 4 that this approach produces a layout that violates our distributed parity criterion (3). To resolve this violation, we duplicate the above layout G times (four times for the example in Figure 4), assigning parity to a different element of each tuple in each duplication, as shown in Figure 5. This layout, the entire contents of Figure 5, is further duplicated until all stripe units on each disk are mapped to parity stripes. We refer to one iteration of this layout (the first four blocks on each disk in Figure 5) as a *block design table*, and one complete cycle (all blocks in Figure 5) as a *full block design table*.

Of course, if the block design has a very large number of tuples, then the size of one full table can exceed the size of the array. This results in violations of criteria two and three. Hence, it is necessary

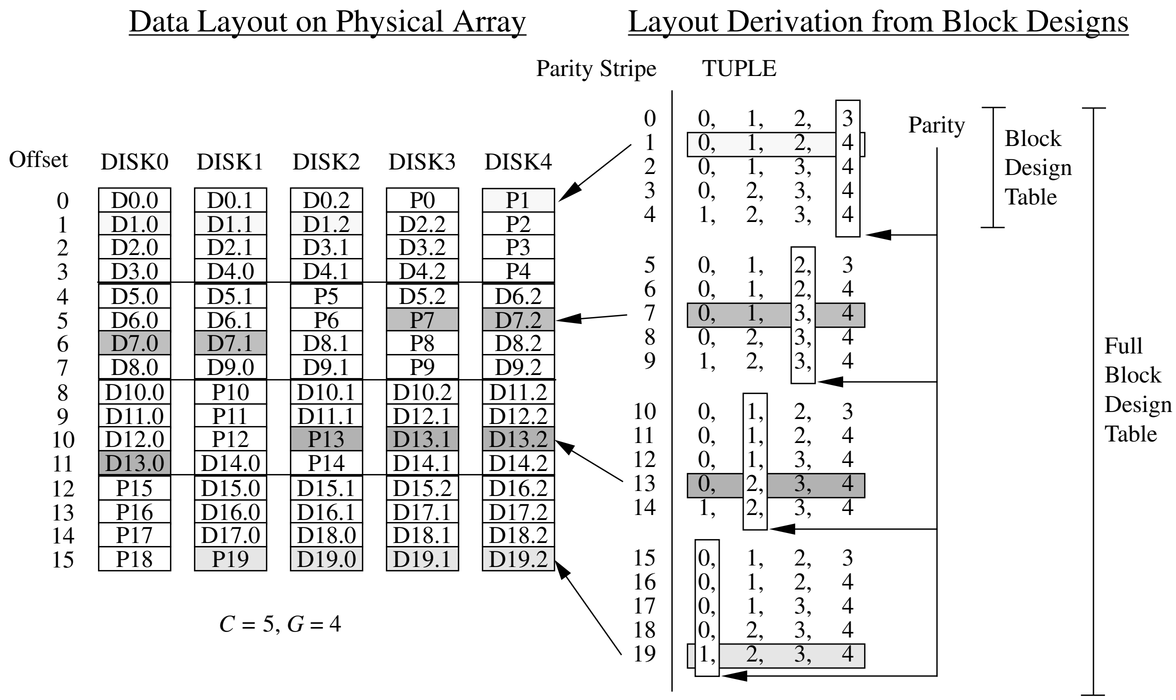


Figure 5: Full block design table for a parity declustering organization.

to find an appropriately small design for each combination of C and G.

It is easy to verify that the layout of Figure 5 meets the first four of our criteria: (1) No two stripe units from the same parity stripe will be assigned to the same disk because no tuple in the block design contains the same element more than once. (2) The failure-induced workload is evenly balanced because each disk appears together with each other disk in exactly λ_p parity stripes in one block design table. This property implies that when any disk fails, exactly λ_p stripe units must be read from each other disk in order to reconstruct the missing data for that table. Since the failure-induced workload is balanced in each table, it is balanced over the entire array. (3) Parity is balanced because over the course of one full table, parity is assigned to each element of each tuple in the block design exactly once (refer to the boxes labelled "parity" in Figure 5), and since each element appears exactly Gr times in the full table, each disk is assigned a parity unit exactly Gr times over the course of the full table. Again, since parity is balanced in every full table, it is balanced over the entire array. (4) While it is not guaranteed that a block design will exist for every possible combination of C and G, nor that the number of blocks will be sufficiently small that the size of a full table will not exceed the size of the array, we have identified acceptable block designs for all combinations of C and G up to 40 disks, and for many of the possible combinations beyond 40 disks. We are in the process of constructing a database of these designs. Section 9 discusses the problem of designing larger arrays.

As previously mentioned, criteria five and six are dependent on the assignment of user data units to units in the address space of the array, and so a data layout mechanism can not guarantee that they will be met. Assuming that this user data mapping is sequential, that is, that successive blocks of user data are mapped to the successive data units of successive parity stripes, the above layout meets criterion 5 (the large write optimization), but fails to meet criterion 6 (maximum parallelism). To see this, note that since consecutive user data is always consecutive within a parity stripe, a write of G-1 user data units aligned on a G-1 unit boundary will always map to the complete set of data units in some parity stripe, and so the large write optimization can be applied. However, Figure 4 or Figure 5 shows that reading C (5, in this case) successive user data units starting at the unit marked *D0.0* results in disks 0 and 1 being used twice, and disks 3 and 4 not at all, and hence criterion six is violated.

As illustrated in Figure 6, it is possible to meet criterion six by employing a user-data mapping similar to Lee's left-symmetric layout for non-declustered arrays [Lee91], but this causes the layout to violate criterion 5. This mapping works by assigning each successive user data block to the first available data unit on each successive disk, thereby guaranteeing that criterion 6 is met. It causes criterion 5 to be violated because successive user data blocks are assigned to differing parity stripes.

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0	D1	D2	P	P
1	D5	D6	D7	D3	P
2	D10	D11	D12	D8	P
3	D15	D16	D17	D13	P
4	D20	D21	P	D18	D4
5	D25	D26	P	P	D9
6	D30	D31	D22	P	D14
7	D35	D36	D27	P	D19

Figure 6: Meeting criterion six via left-symmetric parity-declustered layout.

The figure shows the parity stripes that are allocated by the first two iterations of the block design table, and where data units are mapped in the style of Lee's left-symmetric layout.

Since typical OLTP transactions access data in small units [TPCA89], large accesses account for a small fraction of the total workload, typically deriving from decision-support or array-maintenance functions rather than actual transactions. Thus, for OLTP environments, only a small minority of the user accesses touch more than one data unit, and the fraction of reads that access more than G-1 units is even smaller [Ramakrishnan92]. Therefore the benefits of achieving criterion six in our layout would be marginal at best in OLTP workloads. We will not considered further the problem of meeting this criterion[4].

4.3. Layouts based on random permutations

Merchant and Yu [Merchant92] have independently developed an array layout strategy for declustered parity disk arrays. This section briefly describes their layout strategy and compares it to the block-design based approach developed above.

Their approach distributes failure-induced workload (criterion two) and parity (criterion three) over the disks in the array by randomizing the assignment of data and parity units to disks. The layout defines a linear address space consisting of units numbered 0 through BC-1, where B is the number of units on a disk and C is the number of disks in the array. Every G^{th} unit in this address space (units number G-1, $2G$-1, $3G$-1, etc.) contains parity for the previous G-1 units. If the assignment of these units to disks were truly random, then there would be no guarantee that the units comprising a parity stripe all reside on different disks (criterion one). Instead, their layout uses a set of *random permutations* on the disk identifiers to assign units to disks.

Define a set of random permutations of the integers from 0 to C-1 as follows: P_n, the nth permutation in the set, maps the integer a to $P_{n,a}$, where $0 \leq a < C$ and $0 \leq P_{n,a} < C$, as illustrated:

$$P_n: (0, 1, \ldots, C-1) \rightarrow (P_{n,0}, P_{n,1}, \ldots, P_{n,C-1})$$

To map the location of the i^{th} data unit, let $n = \lfloor i / C \rfloor$ and $j = i \bmod C$. The physical location of unit i is offset n into the disk with identifier $P_{n,j}$. Thus the permutation P_n is used to identify the disks on which units number nC through $(n+1)C$-1 reside.

When C is a multiple of G, no parity stripe will span more than one permutation. Since the elements of each permutation are distinct, the units comprising a parity stripe will all reside on different disks, and so criterion one is met. If C is not a multiple of G, then using each permutation $R = \mathrm{LCM}(C,G)/G$ times sequentially, where *LCM*() is the least-common-multiple function, ensures that no parity stripe spans a permutation, again meeting the needs of criterion 1. The fact that the set of permutations used to map an array is selected randomly implies both that parity blocks are randomly distributed, and that each parity stripe is mapped to a set of disks chosen randomly from the $\binom{C}{G}$ possible combinations, ensuring that criteria two and three are also met. Criterion four is met as long as the permutation P_n can be computed efficiently. Merchant and Yu present an algorithm for this that operates by controlling the exchange phase of a series of applications of a shuffle-exchange network with random bits derived from a linear-congruential random number generator. While cer-

4. The large-read performance of declustered parity organizations can be improved by optimizing the ordering of tuples in the block design and elements in each tuple, but the effect is not dramatic.

tainly requiring substantial computation, this algorithm's asymptotic computation needs grow slowly with respect to *C* and *G*. Criteria five and six are met or failed under the same conditions as in the block-design based layout.

We have verified by simulation that this layout yields array performance essentially identical to that of our block-design layout. The advantage of this algorithm, then, is that it is able to generate a layout for arbitrary *C* and *G*, whereas the block design approach is limited to those combinations of *C* and *G* for which a design can be found. The disadvantage is the relatively large amount of computation a host or controller must do to compute a physical disk address every time a unit of data is accessed. By way of contrast, the block-design based algorithm computes physical disk addresses by a table lookup and a few simple arithmetic operations.

4.4. Choosing between layouts

Complete block designs such as the one in Table 1 are easily generated, but in most cases they are too large for our purposes. The number of blocks in a complete design, $\binom{C}{G}$, is in general so large that our block-design-based layout fails to have an efficient mapping. For example, a 41 disk array with 10% parity overhead (*G*=10) mapped by a complete block design will have over one billion tuples in its block design table. In addition to the ridiculous amount of memory required to store this table, the layout generated from it will meet neither the distributed parity nor distributed reconstruction criteria because even large disks rarely have more than a few million sectors. Fortunately, there exists an extensive literature on the theory of *balanced incomplete block designs* (BIBDs), which are simply designs having fewer than $\binom{C}{G}$ tuples.

The construction of BIBDs is an active area of research in combinatorial theory, and there exists no technique that allows the direct construction of a design with an arbitrarily-specified set of parameters. Instead, designs are generated on a case-by-case basis, and tables of known designs [Hanani75, Hall86, Chee90, Mathon90] are published and periodically updated. These tables are dense when *v* is small (less than about 45), but become gradually sparser as *v* increases. Recalling that the layout equates *v* with *C* and *k* with *G,* Hanani [Hanani75], for example, gives a table of designs that can be used to generate a layout for any value of *G* given *C* not larger than 43, and for many, but not all, combinations with larger *C*.

Since the block design approach is computationally more efficient than the random-permutation approach, we recommend that it be used if the array can be configured using values of *C* and *G* for which an acceptably small block design is known. When a system's goals cannot be met using any such configuration, then, of course use the random-permutation algorithm. Section 9 discusses the problem of configuring very large arrays.

5. Evaluation methodology

All analyses in this paper were done using an event-driven disk-array simulator called *raidSim* [Chen90b, Lee91], originally developed for the RAID project at U.C. Berkeley [Katz89]. It consists of four primary components, illustrated in Figure 7. The top level of abstraction contains a *synthetic*

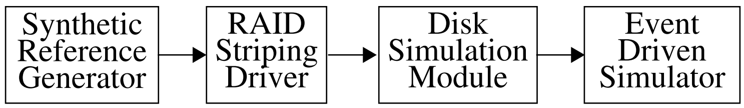


Figure 7: The structure of raidSim.

reference generator. Table 2a shows the workload generated for our simulations. This workload is based on access statistics measured on an airline-reservation OLTP system [Ramakrishnan92]. The requests produced by this workload generator are sent to a *RAID striping drive*r, whose function is to translate each user request into the corresponding set of disk accesses. Table 2b shows the configuration of our extended version of this striping driver. Low-level disk operations generated by the striping driver are sent to a *disk simulation module*, which accurately models significant aspects of each specific disk access (seek time, rotation time, cylinder layout, etc.). Table 2c shows the characteristics of the 314 MB, 3 1/2 inch diameter IBM 0661 Model 370 (Lightning) disks on which our simulations are based [IBM0661]. At the lowest level of abstraction in raidSim is an *event-driven simulator*, which is invoked to cause simulated time to pass.

All simulation results reported represent averages over five independently seeded simulation runs. In all cases, this resulted in very small confidence intervals (a few percent of the mean) and so the performance plots in subsequent sections do not report these actual intervals. For simulations of fault-free and degraded-mode arrays (refer to Section 7), the simulation was not terminated until the 95% confidence interval on the user response time had fallen to less than 2% of the mean. For reconstruction-mode runs, the simulation was terminated at the completion of reconstruction. All simulation were "warmed up" by running a few accesses before initiating the collection of statistics for that run.

6. Algorithms for lost data reconstruction

A *reconstruction algorithm* is a strategy used by a background reconstruction process to regenerate data resident on the failed disk and store in on a replacement. In this section we evaluate two such

Table 2a: Workload Parameters

Access type	% of workload	Operation	Size (KB)	Alignment (KB)	Distribution
1	80%	Read	4	4	Uniform
2	16%	Write	4	4	Uniform
3	2%	Read	24	24	Uniform
4	2%	Write	24	24	Uniform

Number of requesting processes:	3 x (number of disks)
Think time distribution:	Exponential, with mean varied to adjust offered load

Table 2b: Array Parameters

Stripe unit size:	24KB
Reconstruction unit:	24KB
Head scheduling:	FIFO
User data layout:	Sequential user data -> sequential units of sequential parity stripes
Parity layout:	Block-design based
Disk spindles:	Synchronized

Table 2c: Disk Parameters

Geometry:	949 cylinders, 14 heads, 48 sectors/track
Sector size:	512 bytes
Revolution time:	13.9 ms
Seek time model:	$2.0 + 0.01 \cdot cyls + 0.46 \cdot \sqrt{cyls}$ (ms, *cyls* = seek distance in cylinders-1) 2.0 ms min, 12.5 ms average, 25 ms max
Track skew:	4 sectors
Cylinder skew:	17 sectors
MTTF:	150,000 hours

algorithms, and then report on a study investigating the effects of modifying the size of the *reconstruction unit*, which is the amount of data read or written in each reconstruction access.

6.1. Comparing reconstruction algorithms

The most straightforward approach, which we term the *stripe-oriented* algorithm, is as follows:

for each unit on the failed disk
1. Identify the parity stripe to which the unit belongs.
2. Issue low-priority read requests for all other units in stripe, including the parity unit.
3. Wait until all reads have completed.
4. Compute the exclusive-or over all units read.
5. Issue a low-priority write request to the replacement disk.
6. Wait for the write to complete.

end

This algorithm uses low-priority requests in order to minimize the impact of reconstruction on user response time, since commodity disk drives do not generally support any form of preemptive access. A low-priority request is used even for the write to the replacement disk, since this disk services writes in the user request stream as well as reconstruction writes [Holland92].

The problem with this algorithm is that it is unable to consistently utilize all disk bandwidth not absorbed by user accesses. First, it does not overlap reads of surviving disks with writes to the replacement, so the surviving disks are idle with respect to reconstruction during the write to the replacement, and vice versa. Second, the algorithm simultaneously issues all the reconstruction reads associated with a particular parity stripe, and then waits for all to complete. Some of these read requests will take longer to complete than others, since the depth of the disk queues and disk head locations will not be identical for all disks. Therefore, during the read phase of the reconstruction loop, each involved disk may be idle from the time that it completes its own reconstruction read until the time that the slowest read completes. Third, in the declustered parity architecture, not every disk is involved in the reconstruction of every parity stripe, and so some disks remain idle during every iteration of the algorithm.

These deficiencies can be partially overcome by parallelizing this algorithm, that is, by simultaneously reconstructing a set of P parity stripes instead of just one [Holland92, Holland93], but this does not guarantee that the reconstruction process will absorb all the available disk bandwidth. Disks may still idle with respect to reconstruction because the set of P parity stripes under reconstruction at any point in time is not guaranteed to use all the disks in the array. Furthermore, the number of outstanding disk requests each independent reconstruction process maintains varies as accesses are issued and complete, and so the number of such processes must be large if the array is to be consistently utilized. Finally, a large number of reconstruction processes require a large amount of memory resources for buffering.

A better approach is to restructure the reconstruction algorithm as a *disk-oriented*, instead of *stripe-oriented*, process. Instead of creating one reconstruction process, the host or array controller creates C processes, each associated with one disk. Each of the C-1 processes associated with a surviving disk execute the following loop:

repeat
1. Find the lowest-numbered unit on this disk that is needed for reconstruction.
2. Issue a low-priority request to read the indicated unit into a buffer.
3. Wait for the read to complete.
4. Submit the unit's data to a centralized buffer manager for subsequent XOR.

until (all necessary units have been read)

The process associated with the replacement disk executes:

repeat

1. Request a buffer of fully reconstructed data from the buffer manager, blocking if none available.
2. Issue a low-priority write of the buffer to the replacement disk.
3. Wait for the write to complete.

until (the failed disk has been reconstructed)

In this way the buffer manager provides a central repository for data from parity stripes that are currently "under reconstruction." When a new buffer arrives from a surviving-disk process, the buffer manager XORs the data into an accumulating "sum" for that parity stripe, and notes the arrival of a unit for the indicated parity stripe from the indicated disk. When it receives a request from the replacement-disk process it searches its data structures for a parity stripe for which all units have arrived, deletes the corresponding buffer from its active list, and returns this buffer to the replacement-disk process.

The advantage of the disk-oriented approach is that it is able to maintain one low-priority request in each disk's queue at all times, which means that it will absorb all of the array's bandwidth not absorbed by user accesses. This is demonstrated in the simulation results of Figure 8, which plots the reconstruction time and user response time versus the declustering ratio (α) for 1, 8, and 16-way parallel stripe-oriented reconstruction, and for disk-oriented reconstruction. This figure shows that the disk-oriented algorithm makes more efficient use of the system resources: reconstruction time is reduced by up to 40% over the 16-way parallel stripe-oriented version, while the average and 90th percentile response times remain essentially the same, independent of the value of α. Low-parallelism versions of the stripe-oriented algorithm yield slightly better user response time because they cause disks to idle fairly frequently, allowing user requests to more often arrive to find an empty disk queue. This does not happen in the disk-oriented algorithm because reconstruction accesses are always initiated as soon as any disk becomes idle. Because of its superior reconstruction time characteristics, the disk-oriented algorithm is used for all the following performance analyses.

6.2. Unit of reconstruction selection

In the algorithms presented so far, the reconstruction processes read or write one unit per reconstruction access. Since the rate at which a disk drive is able to read or write data increases with the size of an access, it is worthwhile to investigate the benefits of using reconstruction accesses that are different in size from one data unit, that is, to decouple the size of the reconstruction unit from that of the data unit. The block-design based layout described above requires a simple modification to support this decoupling. This section describes this modification and then investigates the sensitivity of failure-mode performance to the size of the reconstruction unit.

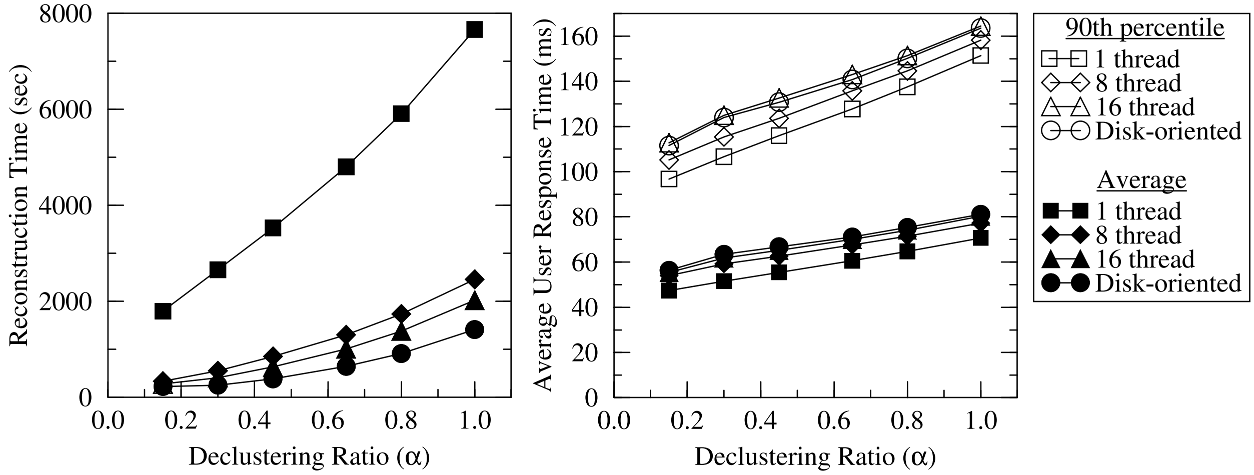


Figure 8: Comparing reconstruction algorithms.

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	P1
1	D5.0	D5.1	D5.2	P5	P6
2	D1.0	D1.1	D1.2	D2.2	P2
3	D6.0	D6.0	D6.2	D7.2	P7
4	D2.0	D2.1	D3.1	D3.2	P3
5	D7.0	D7.1	D8.1	D8.2	P8
6	D3.0	D4.0	D4.1	D4.2	P4
7	D8.0	D9.0	D9.1	D9.2	P9

Data Unit Size

Reconstruction Unit Size

Figure 9: Doubling the size of the reconstruction unit.

Referring back to Figure 4, assume that the reconstruction unit is four times as large as the data unit, and that disk number 1 has failed. If the reconstruction process at some point reads four consecutive units starting at offset zero on disk 2, the data that is read contains data unit *D3.1*, which is not needed to reconstruct disk 2. In general, since the units necessary to reconstruct a particular drive are interspersed on the disks with units that are not, the reconstruction process must either waste time and resources reading unnecessary data, or it must break up its accesses into sizes smaller than one reconstruction unit, which results in substantially less efficient data transfer from the disks.

This problem can be eliminated by repeating the tuple assignment pattern enough times to pack multiple data stripe units into a single reconstruction unit. This modified layout is illustrated in Figure 9, where the reconstruction unit size is twice the data unit size. While Figure 4 advances to the next tuple in the block design after each parity stripe, the modified layout advances after every *n* parity stripes, where *n* is the reconstruction unit size divided by the data unit size. Note that the layout stripes data units across reconstruction units, instead of filling each reconstruction unit with data units

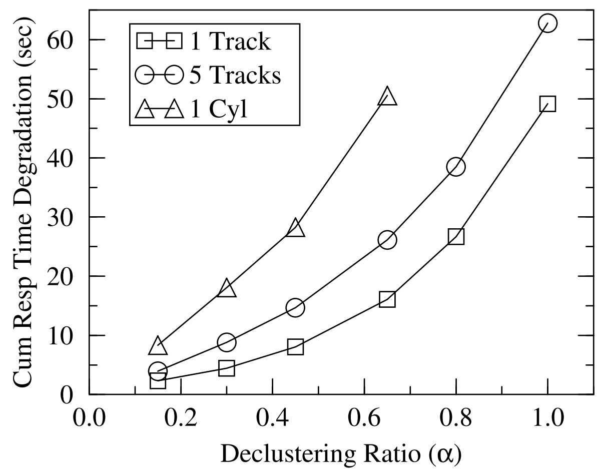


Figure 10: Cumulative response time degradation during reconstruction.
$CumDeg = (RespTime_{recon} - RespTime_{fault\text{-}free}) * ReconTime$

before switching to the next. This avoids excessive clustering of consecutive user data units onto small sets of disks.

The above modification can of course be extended to pack an arbitrary number of data units into each reconstruction unit. With this modified layout, each reconstruction unit occupies a contiguous region on each disk, and so can be read in a single access without transferring extraneous data.

Using a large reconstruction units speeds reconstruction because disks are more efficient for large transfers than for small ones, but it lengthens user response time because a large access monopolizes a disk for a long period of time. To quantify this trade-off, Figure 10 plots the cumulative response time degradation during disk-oriented reconstruction versus the declustering ratio for a 20-disk array using the workload described in Table 2a. The cumulative degradation is the product of the reconstruction time and the increase in average user response time during reconstruction over the fault-free response time. By this "total extra wait time" metric, the increase in efficiency obtained by increasing the size of the reconstruction unit above one track does not compensate for the elongation in response time it causes. This demonstrates that for many arrays reconstruction units should not be selected strictly for maximum disk bandwidth.

7. Performance evaluation

This section examines the performance, in terms of throughput and response time, of the declustered parity organization under three operating conditions: when the array is fault-free, when it is in degraded mode, i.e. when a disk has failed but no replacement is available, and during the reconstruction of a disk. Declustering is intended to improve degraded- and reconstruction-mode performance without affecting fault-free performance. This section also examines the implications of declustering on the reliability of the array. Declustering exposes more disks to second failure during reconstruction, but it also makes reconstruction time much faster.

In this section we will answer two questions. First, how does a parity declustered array compare to an equivalent-size non-declustered array that uses the left-symmetric RAID level 5 layout in multiple groups of disks? In this comparison, the two systems have the same number of disks and contain the same amount of user data. Second, once we understand when to use declustering at all, what benefits can be obtained by reducing the value of G for a fixed number of disks in the array? Reducing G results in less available user data space, but improves the failure-recovery performance substantially. In this latter exploration we include the case where G=2, which corresponds to mirrored disks with the backup copy distributed over the array. For completeness, we also include the case where the mirror copy of each drive resides on exactly one other drive rather than being distributed.

Our results show that parity declustering is a better solution to the failure-recovery problem than the traditional approach of breaking up an array into multiple independent groups. They also show that parity declustering can reduce reconstruction time by up to almost an order of magnitude over RAID level 5 for low values of the declustering ratio, while simultaneously reducing user response time by a factor of about two.

7.1. Comparison to RAID level 5

One way to handle the problem of very long user response time during failure recovery in a RAID level 5 disk array is to stripe user data across multiple groups[5]. The overall average performance degradation experienced when a drive fails in a multi-group array is less than that of a single group array because the load doubles on only the drives in the affected group. This means that on average only one access in N_{groups} experiences degraded performance, where N_{groups} is the number of groups in the array.

This section compares a multi-group RAID level 5 organization to a single-group declustered-parity array. Our comparison examines arrays with a total of 20 or 40 disks. We use this range of sizes because arrays of these sizes are not much larger than many current disk array products available with

5. Following the terminology of Patterson, Gibson, and Katz [Patterson88], a group in a single-failure tolerating array is a set of disks that participate in a redundancy encoding to tolerate at most one concurrent failure. In this sense an array with parity declustered over all disks is a single group.

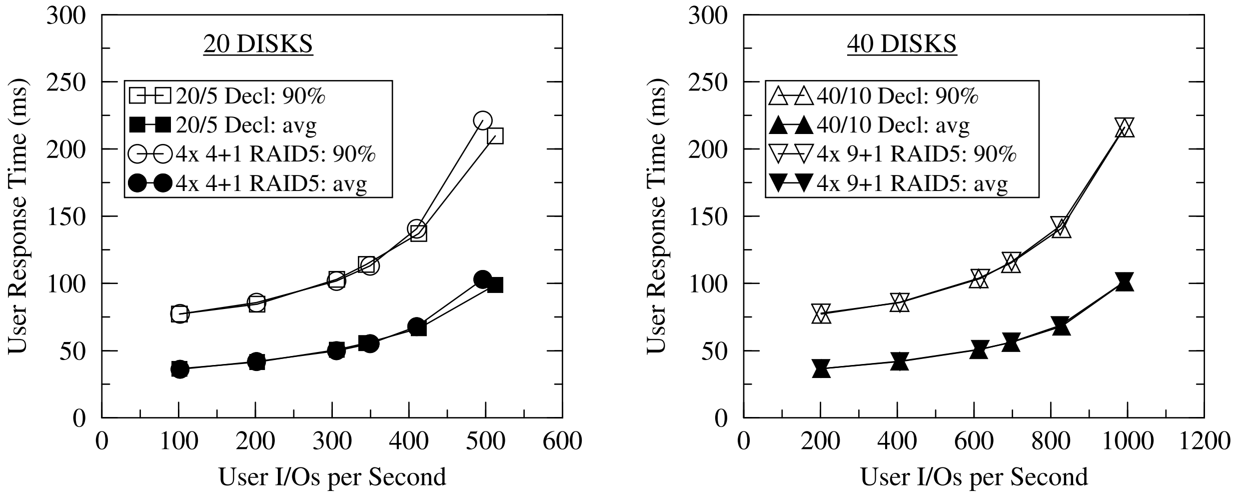


Figure 11: Comparing RAID level 5 to parity declustering: fault-free performance.

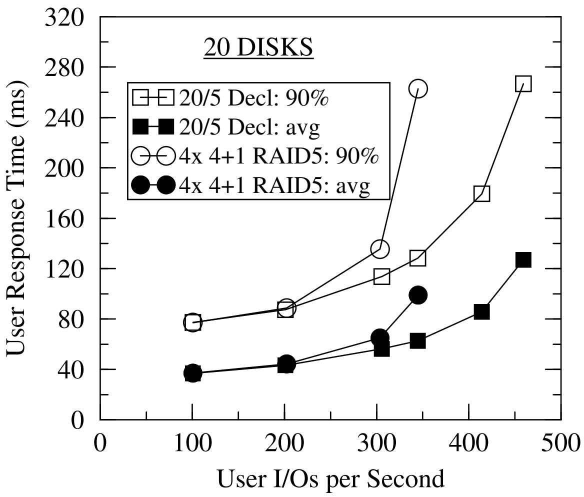
multi-group RAID level 5. We keep constant the fraction of the array's capacity consumed by parity. Specifically, we fix the size of a parity stripe at 5 units for the 20 disk array, and 10 units for the 40 disk array. This means we compare a 4-group 4+1 RAID level 5 (α=1.0) to a C=20, G=5 declustered array (α=0.21), and a 4-group 9+1 RAID level 5 (α=1.0) to a C=40, G=10 declustered array (α=0.23). The performance plots in this section bear out the intuitive conclusion that because our OLTP-based workload is uniform across the data in these disk arrays, an array with more disks is able to support more work without changing the shape of the response time curves. In Section 9 we revisit the implications of larger array sizes by partitioning very large arrays into multiple groups without varying the declustering ratio.

7.1.1. No effect on fault-free performance

Figure 11 plots the average and ninetieth-percentile user response time vs. the achieved user I/O operations per second when the declustered parity and RAID level 5 arrays are fault-free. The left-hand plot makes the comparison for a 20-disk array, and the right-hand plot for a 40-disk array. This figure shows that for OLTP workloads, declustering parity causes no fault-free performance degradation with respect to RAID level 5. Because each user access in a fault-free workload invokes the same number of disk I/Os in each array, we are not surprised to see that the 20-disk and 40-disk graphs are images of each other with the x-axis scaled by a factor of two. The small deviation in the 20-disk graph at high user access rate is within the error margins of a fault-free simulation estimate for response time.

7.1.2. Declustering greatly benefits degraded-mode performance

Figure 12 plots the respective disk arrays' user response-time against achieved user I/Os per sec-



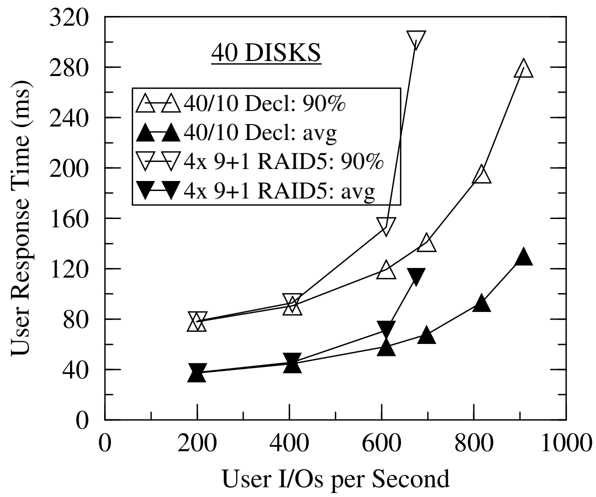


Figure 12: Comparing RAID level 5 to parity declustering: degraded-mode performance.

ond when each array contain one failed disk, but reconstruction has not yet been started. At low workloads the two organizations perform identically, since the extra I/Os caused by accesses to the failed disk's data can easily be accommodated when disk utilization is low. As the workload climbs, the failure-recovery problem in RAID level 5 arrays becomes evident: in both 20 and 40 disk systems, the RAID level 5 group containing the failure saturates at about 15 user I/Os per second per disk, and forms a system performance bottleneck. Because the declustered-parity arrays distribute failure-induced work across all disks in the array, they are able to deliver about 25% more I/Os per second while still delivering a 90th percentile user response time of about 15% over the fault-free case.

7.1.3. Declustering benefits persist during reconstruction

Figure 13 shows average and 90th percentile user response times in *reconstruction mode*; that is, while reconstruction is ongoing. In contrast to the degraded-mode performance shown in Figure 12, Figure 13 shows that parity declustered arrays deliver a slightly worse response time in reconstruction mode. A multiple group RAID level 5 array suffers less penalty for reconstruction at low loads than do parity declustered arrays because many disks experience no load increase and those that do see an increase have plenty of available bandwidth. But, because all reconstruction work is endured by only one group of a RAID level 5 multi-group array, this group quickly becomes saturated as the on-line user load increases. Once a group in the RAID level 5 array is saturated, its long response times dramatically increase average and 90th percentile response times.

Turning to the issue of time until reconstruction completes, Figure 14 illustrates the heart of the failure recovery problem in RAID level 5 arrays. Since the workload doubles on surviving disks in the group containing a failed disk, and since these are the only disks that participate in recovering the

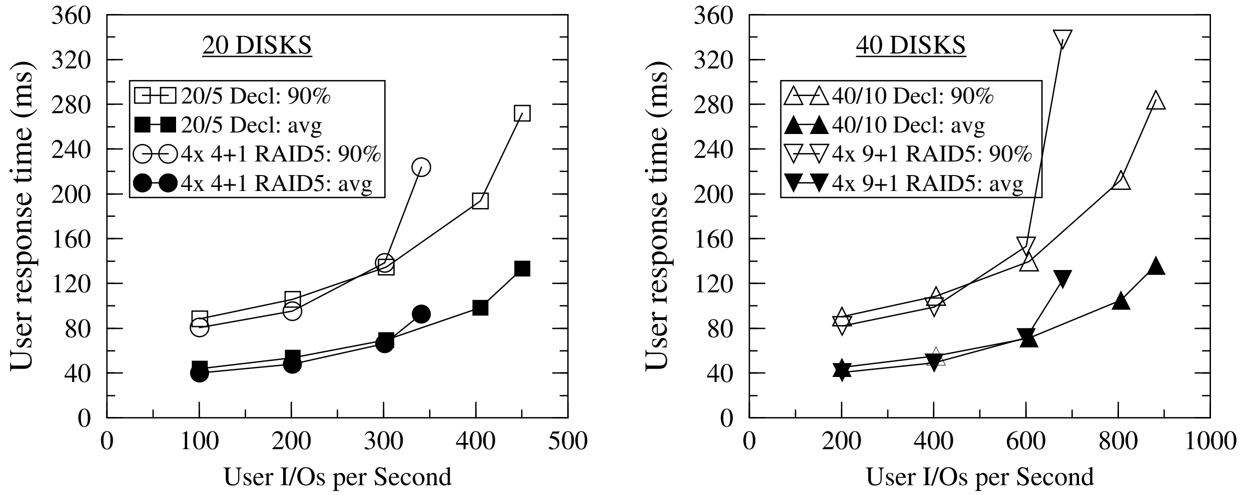


Figure 13: Comparing RAID level 5 to parity declustering: response time during reconstruction.

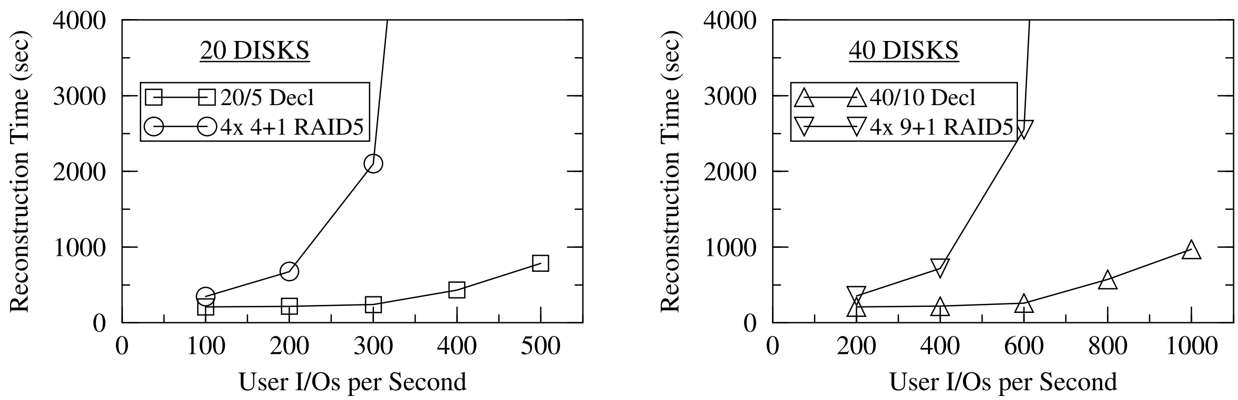


Figure 14: Comparing RAID level 5 to parity declustering: reconstruction time.

contents of this failed disk, reconstruction time is very sensitive to fault-free user workload. The declustered parity organization was designed to overcome this problem by both reducing the per-disk load increase in reconstruction and utilizing all disks in the array to participate in this reconstruction. In other words, a RAID level 5 array has reconstruction bandwidth equal only to the unused bandwidth on the disks in one group, but a declustered parity array provides the full unused bandwidth of the array to effect reconstruction.

The minimum possible reconstruction time is the time required to write the entire contents of the replacement disk at the maximum bandwidth of the drive. Our simulated 320 megabyte drives support a maximum write rate of approximately 1.6 MB/sec, and so the minimum possible reconstruction time is approximately 200 seconds. In Figure 14, reconstruction time in the declustered parity organization at 15 user I/Os per second per disk (this load, utilizing the array's disks about 50% of the time in fault-free mode, is 300 total I/Os per second for 20 disks and 600 for 40 disks) is approximately

260 seconds, indicating that near optimal reconstruction performance is obtained. Contrast this with the RAID level 5 organization, where reconstruction time is essentially unbounded at this user access rate. To emphasize, Figure 13 and Figure 14 show response time and reconstruction time in the same on-line reconstruction event – they show that parity declustering provides huge savings in reconstruction time as well as savings in response time for moderately and heavily loaded disk systems.

7.1.4. Declustering also benefits data reliability

Our final figure of merit is the probability of losing data because a disk failure occurring while another disk is under reconstruction. Assuming that the likelihood of failure of each disk is independent of that of each other disk; that is, that there are no dependent disk failure modes in the system, Gibson [Gibson93] models the mean time to data loss as

$$MTTDL = \frac{MTTF_{disk}^2}{N_{groups} N_{diskspergroup} (N_{diskspergroup} - 1) MTTR_{disk}}$$

where $MTTF_{disk}$ is the mean time to failure for each disk, N_{groups} is the number of groups in the array, $N_{diskspergroup}$ is the number of disks in one group ($N_{diskspergroup}=G$ in RAID level 5 arrays and $N_{diskspergroup}=C$ in parity-declustered arrays), and $MTTR_{disk}$ is the mean time to repair (reconstruct) a failed disk[6]. From this, the probability of data loss in a time period T can be modeled as

$$P(\text{data loss in time } T) = 1.0 - e^{-T/MTTDL}$$

Figure 15 shows the probability of losing data within 10 years (an optimistic estimate of a disk array's useful lifetime) due to a double-failure condition in each of the two organizations, using $MTTF_{disk}$ = 150,000 hours. The RAID level 5 array is more reliable at low user access rates because a multiple-group RAID level 5 array can tolerate multiple simultaneous disk failures without losing data as long as each failure occurs in a different group. In contrast, there are no double-failure conditions that do not cause data loss in a declustered parity array. However, as the user access rate rises, the reconstruction time, and the resulting probability of data loss, rises much more rapidly in the RAID level 5 array. For our example arrays and workload, the declustered parity array becomes more reliable at about 10 user accesses per second per disk (a fault-free utilization of about 40%). This is significantly less than the user workload required to saturate the RAID level 5 array during reconstruction (about 14 accesses/second/disk).

6. Gibson treats dependent failure modes and the effects of on-line spare disks in depth. As nearly all of that work applies here directly, we will only describe the simple and illustrative case of independent disk failures.

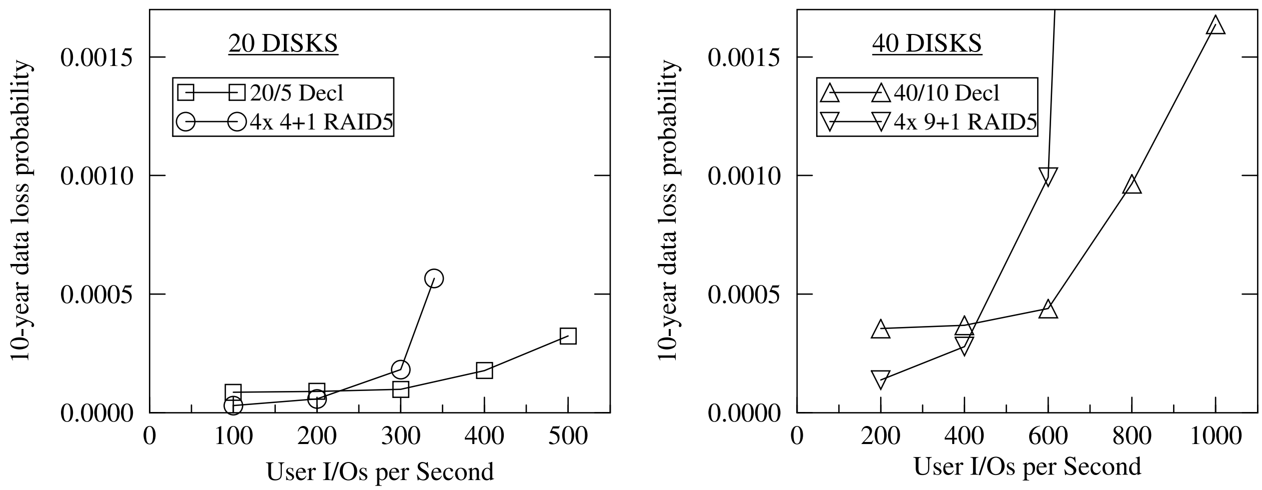


Figure 15: Comparing RAID level 5 to parity declustering: probability of data loss within 10 years.

7.1.5. Summary: declustered parity allows higher normal loads in on-line systems

In this section we have considered the effects of replacing a multi-group RAID level 5 array with a declustered parity array of the same cost and the same user capacity. Essential for its viability, declustered parity achieves the same fault-free performance as an equivalent RAID level 5 array. Its advantage is that it also supports higher user workloads with lower response time in both degraded and reconstruction mode, has dramatically shorter reconstruction time, and at moderate and high user workloads, has superior data reliability. We believe that this makes a compelling case for the use of parity declustering in on-line systems that cannot tolerate substantial degradation during failure recovery.

7.2. Varying the declustering ratio

In contrast to the prior section which showed that a single group array with a declustering ratio (α) between 0.20 and 0.25 has substantial advantages over a multi-group array with a declustering ratio of 1.0 (RAID level 5), this section examines the effect on failure recovery performance of varying the declustering ratio (α) in a fixed-size single-group array. Because the size of the array, C, is fixed, varying the declustering ratio ($\alpha=(G-1)/(C-1)$) is achieved by varying the size of each parity stripe, G, which determines the parity overhead, $1/G$, and correspondingly, the fraction of storage available to store user data, $(G-1)/G$. As α is decreased from 1.0, the user data capacity of the array decreases but the failure-recovery performance improves since the total failure-induced workload decreases. We shall show that declustering ratios larger than 0.25, which provide lower parity overhead, yield much of the performance benefits of the example in the last section. We shall also show that in systems very sensitive to performance during failure recovery, declustered mirroring $(G=2)$ is a special case with minimal declustering ratio, high parity overhead, and failure-recovery performance advantages unavailable in most other declustered organizations.

We consider the same two array sizes, 20 disks and 40 disks, and report on the performance of the arrays on the workload described in Table 2a, using a fixed user access rate of 14 user I/Os per second per disk. This rate was selected because it is approximately the maximum that the arrays can support using a RAID level 5 layout (α=1.0). It causes the disks to be utilized at slightly less that 50% in the fault-free case.

The arrays are evaluated at α=1.0, α=0.75, α=0.5, α=0.25, and two special cases G=3 and G=2. The case G=3 is significant because when a parity stripe contains only two data units and one parity unit, it is possible to improve small-write performance by replacing the normal 4-access update (data read-modify-write followed by parity read-modify-write) by a 3-access update. In this case, the controller reads the data unit that is *not* being updated, computes the new parity from this unit and the unit to be written, and then writes the new data and new parity.

The case G=2 is important because it is equivalent to disk mirroring, except that the backup copy of each disk is distributed across the other disks in the array instead of being located on a single drive. For comparison, the graphs also include the case where the backup copy is located on a single drive. To distinguish between these two, we refer to the case where the backup copy is on a single disk as "mirroring", and the case where it is declustered as the "G=2" case.

In both mirroring and parity declustering with G=2, the four accesses associated with a small-write operation are replaced by two: one write to each copy of the data. Another optimization also applies: since there are two copies of every data unit, it is possible to improve the performance of the array on read accesses by selecting the "closer" of the two copies at the time the access is initiated [Bitton88]. Our simulator contains an accurate disk model, and so we implement this as follows: when a read access is initiated, the simulator locates the two copies that can be read and then computes the completion time of the request for each of the two possible accesses. This computation takes into account all components of the access time (queueing, seeking, rotational latency, and data transfer). The simulator selects and issues the access that will complete sooner. We refer to this as the *shortest access* optimization. We will see that these optimizations can be significant for performance, but they only apply in the G=2 and G=3 cases, which are expensive in terms of capacity overhead.

7.2.1. Fault-free performance: benefits of high overhead optimizations

Figure 16 shows that the response time performance of a fault-free array is independent of α in

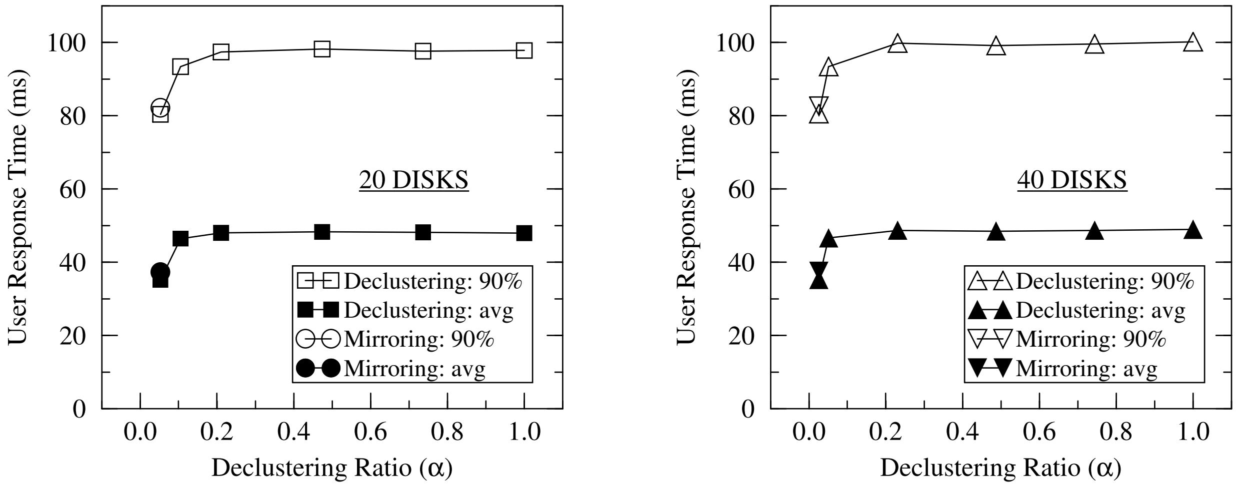


Figure 16: Varying the declustering ratio: user response time in fault-free mode.

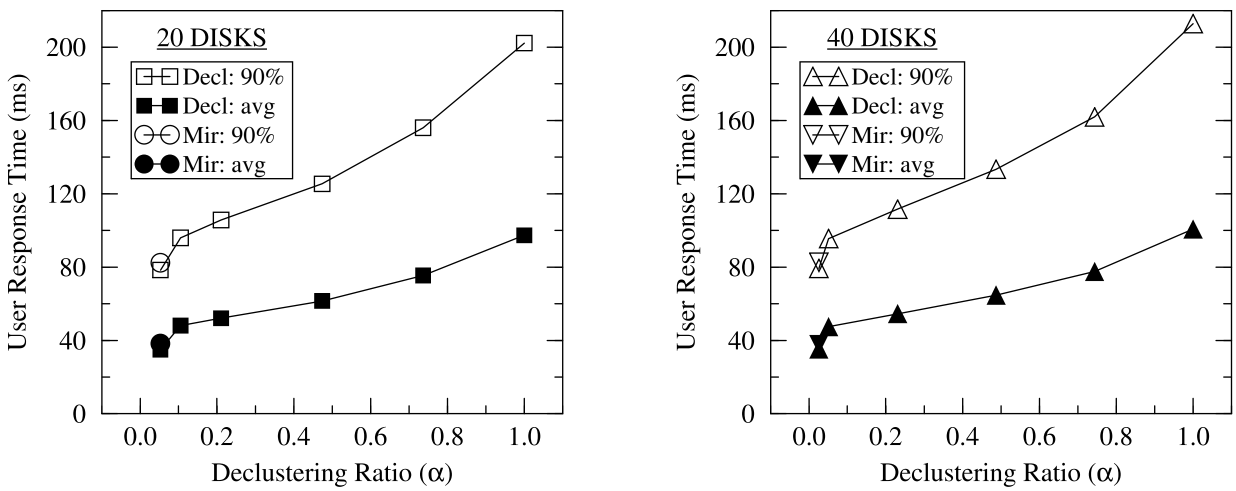


Figure 17: Varying the declustering ratio: user response time in degraded mode.

all cases except G=2 and G=3, where the above-described optimizations can be applied. This figure confirms the result of Section 7.1.1 that declustering parity does not negatively effect fault-free performance. Similarly, declustered parity with G=2 performs essentially identically to mirroring. Figure 16 does not show the full benefit of a three-access update when G=3 because our OLTP workload is dominated by read rather than write operations. However, for G=2, the combined response-time benefit of a two-access update and the shortest access optimization is close to 40% for average response time, and 20% for 90th-percentile response time. Thus for workloads such as OLTP that are

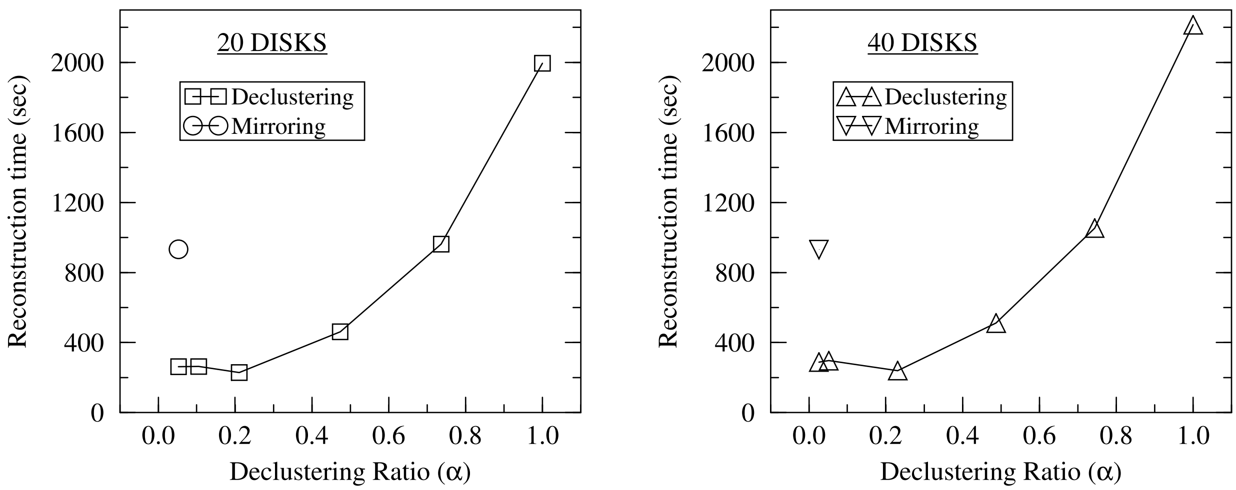


Figure 18: Varying the declustering ratio: reconstruction time.

dominated by small accesses, the main consideration for fault-free performance is whether or not the value of the optimizations available in the G=2 case warrants the large capacity overhead it incurs.

7.2.2. Degraded-mode and reconstruction-mode performance: declustering at its best

Figure 17 demonstrates the declustering ratio's direct effect on degraded-mode performance of an array. As the declustering ratio, α, ranges down from 1.0 the array's response time decreases almost linearly to a minimum that is about half of its maximum at α=1.0. In contrast to Figure 16, the minimum response times that occur with small declustering ratios are little degraded from their fault-free counterparts. This lack of degradation at low α occurs because reconstructing data on-the-fly is adding very little to each surviving disk's utilization. However, when α=1.0 the degraded-mode utilization is close to 100% because our user workload induces in fault-free arrays an utilization of slightly less than 50%. Hence, response time is dramatically longer when degraded than when fault-free.

User response time during reconstruction shows essentially the same characteristics as user response time in degraded mode because user accesses are given strict priority over reconstruction accesses, and so reconstruction is just a little more load on each surviving disk. However, Figure 18 shows that reconstruction time decreases by an order of magnitude as α drops from 1.0 to 0.2. The shape of this curve is determined by the interaction of two separate bottlenecks: at high α the rate at which data can be read from surviving disks limits reconstruction rate, but, at low α the replacement disk is the bottleneck[7]. Since a high declustering ratio causes surviving disks to be saturated with work, reconstruction time falls off steeply with decreasing α, flattening out at the point where the

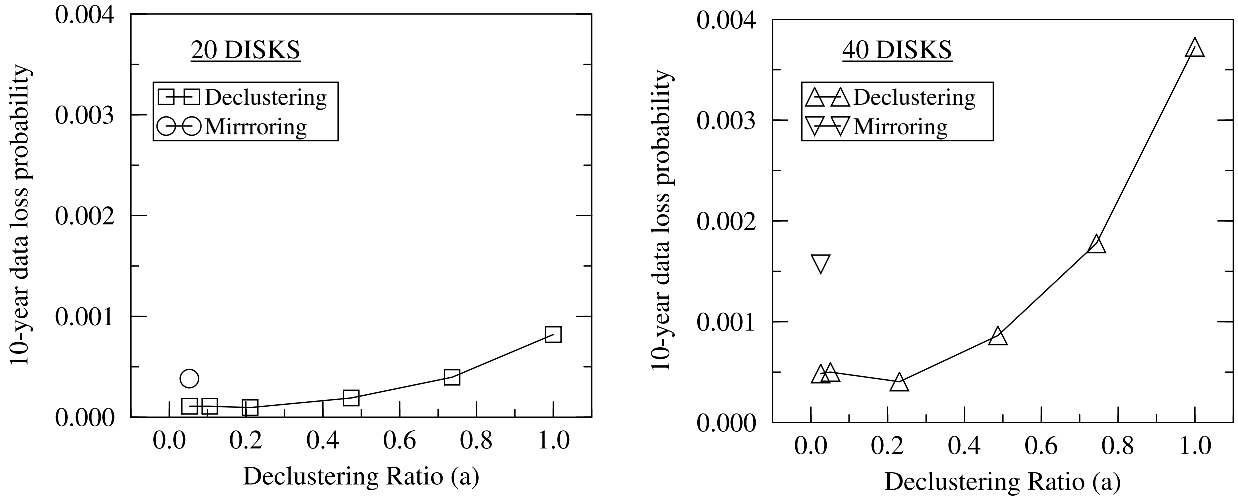


Figure 19: Varying the declustering ratio: probability of data loss within 10 years.

replacement disk becomes saturated with reconstruction writes.

Finally, reconstruction time is much longer in the case of mirroring than for declustered parity with G=2 because a declustered array has the aggregate unused bandwidth of the entire array available to read blocks of the backup copy, while a mirrored array has only the bandwidth of a single disk. The reconstruction time is not as long as in the case of α=1.0 (RAID level 5) because mirroring handles user accesses more efficiently.

7.2.3. High data reliability: another advantage for declustered parity

Figure 19 shows the probability of losing data within 10 years due to a disk failure occurring while the reconstruction of another disk is ongoing (refer to Section 7.1.5). Decreasing reconstruction time by decreasing the declustering ratio in an array directly decreases the probability of data loss in any time period. This figure, then, is largely determined by the data in Figure 18.

7.2.4. Summary

In contrast to parity declustered arrays with fixed declustering ratios determined by a fair cost comparison to multi-group RAID level 5 arrays in Section 7.1, this section examined the choices available if an array's declustering ratio is varied. As this ratio is lowered, and hence cost or overhead rises, declustered mirroring offers special benefits over declustered parity layouts with slightly higher declustering ratios. Alternatively, if lowering cost or overhead is of prime interest, then a declustering ratio of 0.5 is of particular interest. It provides half the benefit for improving degraded- and recon-

7. This bottleneck may be eliminated allowing reconstruction time to be further reduced by distributing the capacity of spare disks throughout the array [Menon92b].

struction-mode performance and nearly all the benefit for reducing reconstruction time and data reliability while costing only twice the parity overhead of a single group RAID level 5 array.

8. Work reducing variations to reconstruction algorithms

Muntz and Lui [Muntz90] identified two simple modifications to a reconstruction algorithm, each intended to improve reconstruction performance or duration by reducing the total work required of surviving disks. In the first, called *redirection of reads*, user read requests for data units belonging to a failed disk that have already been reconstructed are serviced by the replacement disk instead of invoking on-the-fly reconstruction as is done in degraded mode. This reduces the number of disk accesses needed to service the read from G-1 to one. Although this seems to be an obvious thing to do, we shall see that it can lengthen reconstruction time. In the second modification, *piggybacking of writes*, when a user read request causes a data unit to be reconstructed on-the-fly, that data unit is written to the replacement drive as well as being delivered to the requesting process. This is intended to speed reconstruction by reducing the total number of data units that need to be recovered, but in our evaluation it will turn out to have little effect.

Additionally, there are two ways to service a user write to a data unit whose contents have not yet been reconstructed. In the first, the new data is written directly to the replacement disk, and the parity is updated by reading all the other units in the parity stripe, XORing them together with the new data, and writing the result to the parity unit. In the second method, the parity is updated by reconstructing the missing data on-the-fly, and then XORing together the new data, the old data, and the old parity and writing the result to the parity unit. In the latter case, the data unit being updated remains invalid until recovered by the background reconstruction process. The difference between these two approaches is that the former writes the replacement disk and does not read the old parity. We view sending user writes to the replacement disk (the former approach) as a third modification that can be applied, and refer to it as the *user writes* option.

These three options affect the distribution of work between surviving disks and the replacement disk. When all three options are off, the replacement disk sees only reconstruction writes and user writes to data that has been previously reconstructed, while the remainder of the workload is serviced by surviving drives. Enabling an option shifts workload from the surviving disks to the replacement disk: redirecting reads shifts user-read workload, piggybacking writes shifts reconstruction workload, and enabling user writes to the replacement shifts user-write workload.

In a previous paper [Holland92] we analyzed the performance of these options using the stripe-

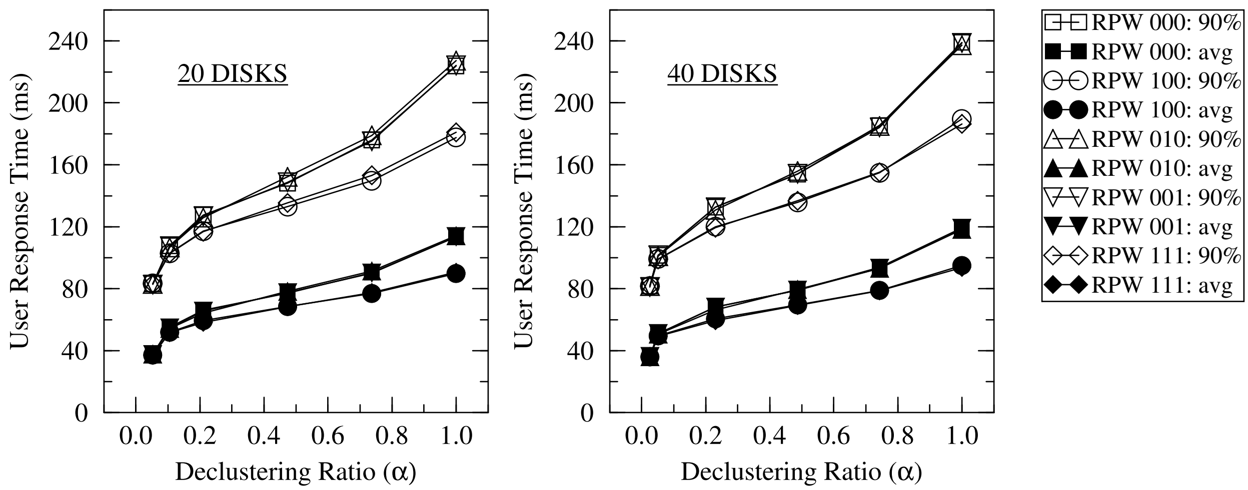


Figure 20: User response time for five combinations of reconstruction options.

In this figure's legend, R *indicates redirection of reads,* P *indicates piggybacking of writes,* W *indicates user-writes to the replacement drive,* 0 *indicates that an option is off, and* 1 *indicates that an option is on. The figure is difficult to read because of the overlapping lines; in all plots, the* 000*,* 010*, and* 001 *curves are essentially coincident, as are the* 100 *and* 111 *curves.*

oriented reconstruction algorithm, a 50% write workload, and a smaller striping unit (4 KB). This section revises this analysis using the disk-oriented reconstruction algorithm, the more realistic and less write-intensive workload described in Table 2a, and track-sized stripe units. Larger stripe units have been recommended for varied workloads because they reduce the probability that small requests require service from multiple disks arms while still allowing parallel transfer for requests large enough to benefit substantially [Chen90b]. Our prior study showed that the piggybacking and user-writes options had a measurable effect on reconstruction time. Because of the lower write fraction and the larger reconstruction unit in the new study, these effects have essentially disappeared, and so we find that redirection of reads is the only option that significantly influences failure-mode performance. As expected, the effects of redirection are more pronounced in the new study because of the read-dominated workload.

In the following we show at most five of the possible eight combinations of these three reconstruction algorithm options: all options off, each option on with the other two off, and all options on. As we shall see, only one option, the redirection of reads option, is effective for our workload.

8.1. The effects of the reconstruction options

Figure 20 shows the average and 90th percentile user response time during reconstruction for five combinations of the reconstruction options. This figure shows that the piggybacking of writes and

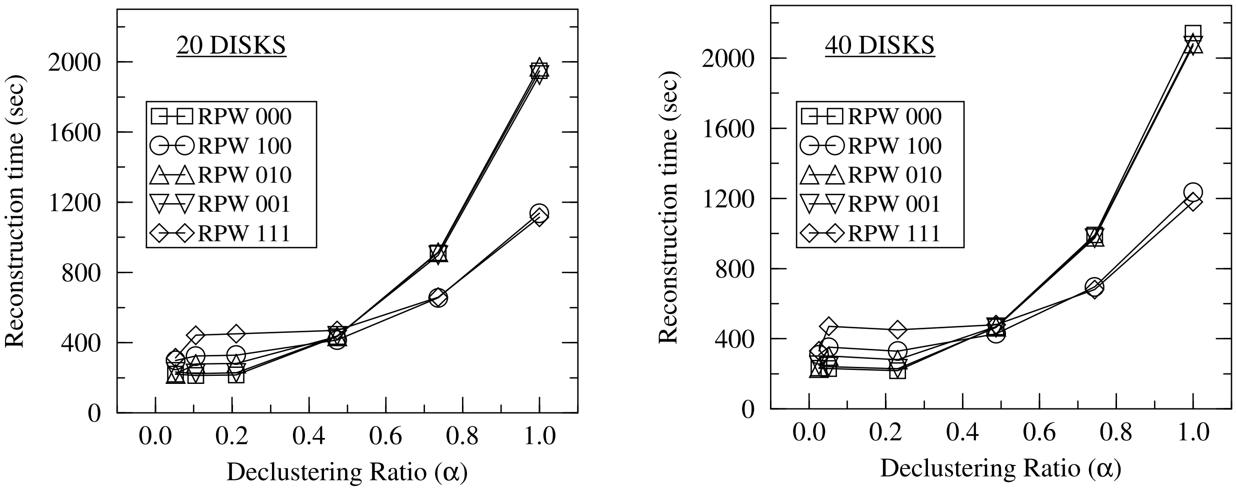


Figure 21: Reconstruction time for five combinations of reconstruction options.

Refer to Figure 20 for a description of the legend.

user-writes options have little effect on user response time. To understand this, first note that updating a particular unit on the replacement drive may improve response time only if that unit is re-accessed prior to the completion of reconstruction. However, for a random workload, the probability of re-accessing the same data unit before reconstruction completes is fairly small, and so these two reconstruction options have little effect.

Redirection of reads, in contrast to the other options, can be effective for our OLTP workload. It improves user response time by 10-20% when the declustering ratio is near 1.0, with its benefit diminishing to zero as this ratio decreases. It is most effective when this ratio is large because the surviving disks are heavily loaded by reconstruction. Off-loading work from these drives by redirecting reads to the underutilized replacement disk improves response time by both reducing the number of I/Os necessary to service a user read and by servicing such a read on a lightly-utilized drive. As α is reduced, however, both these effects diminish: it takes fewer disk reads to service a user read to the failed drive and the replacement disk utilization increases because these more lightly loaded surviving disks reconstruct units more quickly.

Figure 21 shows the reconstruction time for five combinations of options. The piggybacking of writes and user-writes options again make little difference in the failure-mode performance. In this case, it is because the workload is dominated by accesses that are smaller than one reconstruction unit. When a user- or piggybacked-write operation occurs on the replacement disk, only a fraction of a reconstruction unit is updated and marked as reconstructed. When a reconstruction process examines this unit to decide if it needs to be reconstructed, it will find that some portion of the unit is still unrecovered. The reconstruction process then has the option of reconstructing only the unrecovered portion of the unit, or of reconstructing the entire unit. Because there is little difference between the time taken to read an entire track and the time taken to read a track less one unit, and because many disks cannot read two blocks on one track as quickly as they read the whole track, our implementation always chooses the latter option. Hence, most of the potential benefits to reconstruction time from user- and piggybacked-write options are lost, since these writes do not update entire reconstruction units. Moreover, at low α, these two options actually have a negative effect on reconstruction time since they cause more work to be sent to the over-utilized replacement disk.

While redirection of reads reduces user response time during recovery at all values of α, it does not have the same effect on reconstruction time. Figure 21 shows that this option halves reconstruction time at α=1.0, but doubles it at α=0.1. This is partly because the replacement disk is over-utilized at low α, but there is also another reason. In the absence of user workload, the replacement disk services only writes from the reconstruction process and writes to previously-reconstructed data. Because the reconstruction writes are purely sequential, the replacement drive experiences a very low average positioning overhead, and operates at high efficiency. When any of the reconstruction options are enabled, the replacement disk incurs a significant reduction in its efficiency because it must service far more randomly located accesses. This accounts for the significant increase in reconstruction time at low α when the reconstruction options are enabled.

8.2. Dynamic use of reconstruction options

As Figure 21 shows, the value of a reconstruction algorithm option depends on which part of the array, replacement or surviving disk, is limiting the rate of reconstruction. In addition to being dependent on an array's declustering ratio, this effect is dependent on the amount of the failed disk's data so far reconstructed. Recognizing this dependence, Muntz and Lui suggested that the reconstruction algorithm should monitor disk utilizations and enable or disable each option dynamically, depending on whether surviving disks or the replacement disk constitutes a bottleneck.

Figure 22 and Figure 23 show, respectively, user response time during reconstruction and reconstruction time using a monitored application of redirection of reads instead of a constant (always enabled) application or no (always disabled) application. We have chosen to dynamically apply only the redirection of reads option because it is the only option that significantly affects recovery mode performance for our OLTP workload. We refer to this dynamic reconstruction algorithm as the *monitored redirection* option. We employ a simple monitoring scheme: the duration of disk busy and idle

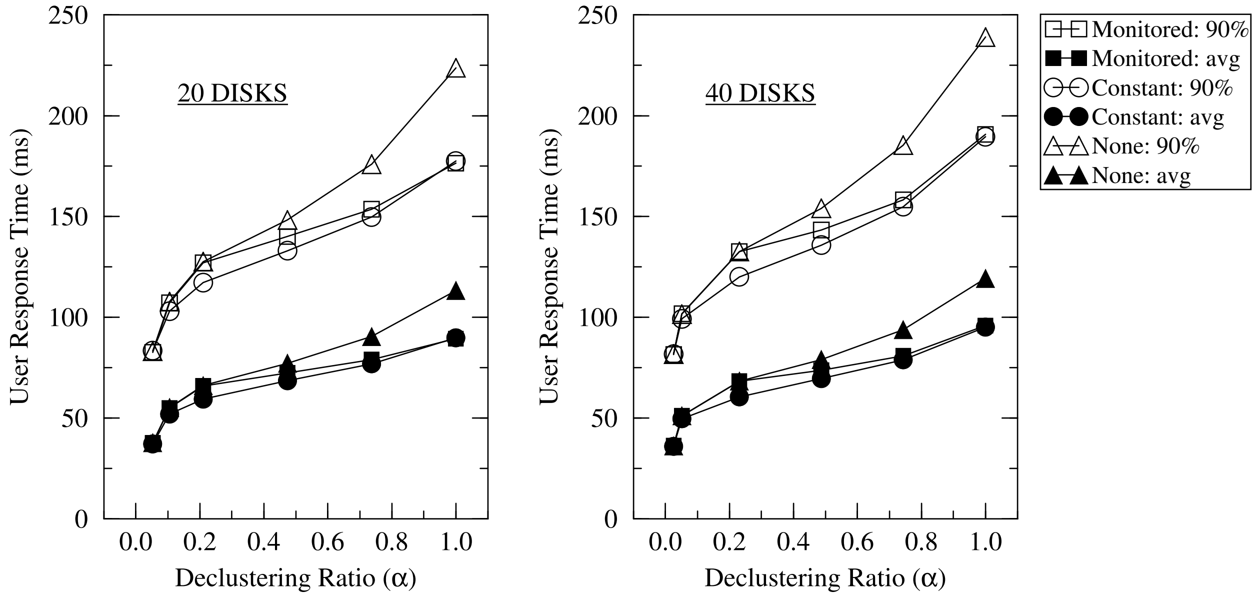


Figure 22: Evaluating monitored redirection of reads: response time.

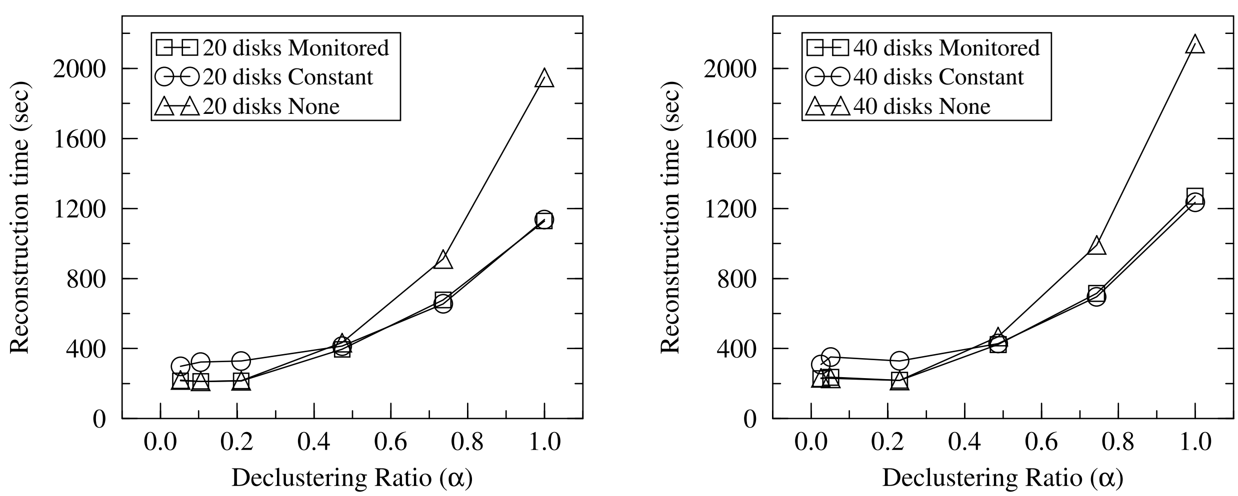


Figure 23: Evaluating monitored redirection of reads: reconstruction time.

periods is recorded, and every 300 accesses a new estimate for the utilization of each disk is generated. If the replacement disk utilization is higher than the average surviving disk utilization, the replacement is declared the bottleneck, and redirection of reads is disabled until the next time the estimates are updated. If the opposite is true, the surviving disks are declared the bottleneck, and redirection of reads is enabled until the next utilization estimate update.

As Figure 22 shows, the response-time performance of monitored redirection is actually worse at moderate and low declustering ratios than the no-redirection case shown in Figure 20 because redirection of reads, uniformly beneficial to response time when enabled, is largely disabled. Figure 23, however, shows that reconstruction time is minimized because the reconstruction rate is at all times limited by whichever disks are the reconstruction bottleneck.

To summarize, for our OLTP workload, the only effective work-reducing variation to our disk-oriented reconstruction algorithm is the redirection of reads. This option improves user response time by as much as 10% - 20% when the declustering ratio is large while reducing reconstruction time by as much as 40%. However, with low declustering ratios, redirection of reads benefits response time not at all and lengthens reconstruction time. A dynamic application of this option based on monitoring disk utilizations achieves much of its benefits without its costs independent of the declustering ratio.

9. Array configuration: single versus multiple groups revisited

Section 7.1 shows that for arrays of up to about 40 disks, a single declustered group organization yields better failure-mode performance than an organization that separates disks into a set of independent RAID level 5 groups. In this section we revisit the question of when to configure a set of disks as a single group or multiple groups, where the data reliability of each group is independent of failures in other groups. In particular, we are interested in how to configure arrays that have more than 40 disks. In this context an array configuration is a set of values for the number of disks in a group, C, the number of units in one parity stripe, G, and the number of groups, denoted N_{groups}. we shall see that it is not always desirable, and sometimes not viable, to structure a large array as a single declustered group.

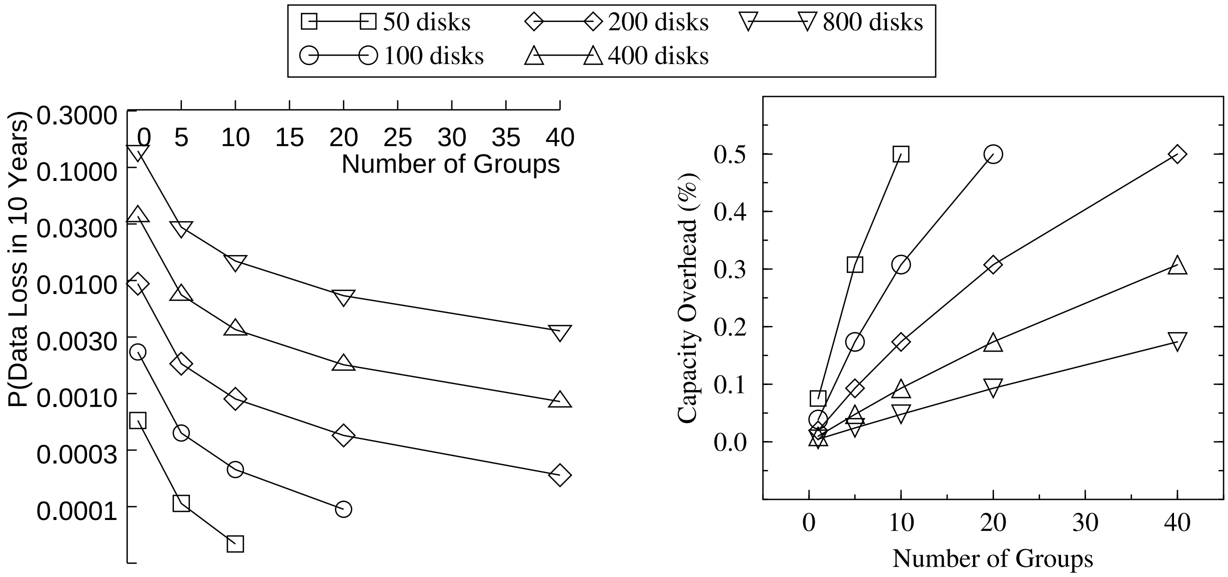
A primary consideration in the construction of large single-group arrays is their susceptibility to data loss arising from failures in equipment other than the disks [Gibson93]. For example, if the bus-connected disk array architecture shown in Figure 1a provides only one path to each disk but shares this path over multiple disks, the failure of a path renders multiple disks unavailable, if not damaged, for long periods of time. We say that such a path failure constitutes a dependent failure mode for the set of disks on that path. To make such an array tolerant of all single failures according to criteria 1 in Section 4.1, these disks may not reside in the same redundancy group. A cost effective way to do this is to organize each rank of drives as an independent parity group. It follow then that the size of each declustered group (C) can be no larger than the number of cable paths in the array. With today's technology, board area and cable connector size limit the number of paths operating in a single array to a relatively small number, usually much less than 40. In this case, layouts based on block designs and the results of Section 7.1 are directly applicable.

In disk arrays with sufficient redundancy in non-disk components such as the fully duplicated versions of Figure 1, the number of disks managed as a single parity group could be much larger than 40. In the process of configuring such large arrays, the fundamental trade-off is between cost, data reliability, parity overhead, fault-free performance, and on-line failure recovery performance. Remaining with our OLTP-like model of such an array's workload, we assume that the goal of an configuration is to achieve the lowest cost array which meets specific I/O throughput and response time requirements and that component disk capacity can be manipulated to meet data capacity targets. In particular, to maximize throughput for a target number of disks, we seek fault-free disk utilizations as high as possible while insuring that response time requirements are met during on-line reconstruction. The most effective method of doing this is to minimize the increase in disk utilization during on-line reconstruction, which is measured by the declustering ratio, $\alpha=(G-1)/(C-1)$, because this is the increase in load on surviving disks during on-the-fly reconstruction in degraded-mode. Left to determine are the size of each group in the array, C, and the number of these groups, N_{groups}, and the impact of these two parameters on data reliability and parity overhead.

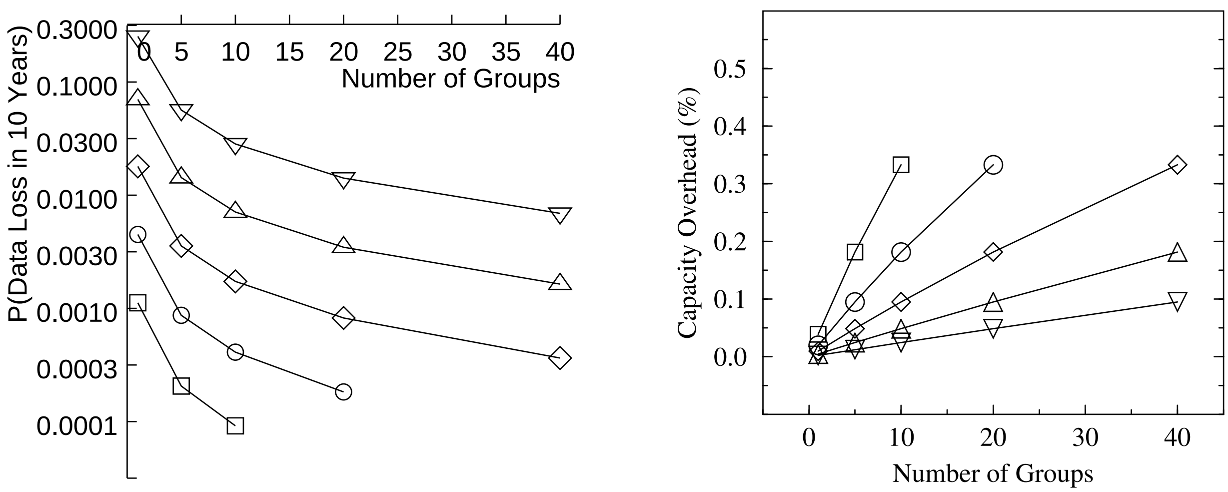
The data reliability equations in Section 7.1.5 show that mean time until data loss is inversely proportional to group size, C, and failure recovery time, *MTTR*, for a fixed array size. But given a fault-free user workload and a declustering ratio, failure recovery time is a largely a function of a single disk's capacity and performance as shown in Figure 14 and Figure 18. This implies that data reliability increases with decreasing group size (which means increasing the number of groups). However, with a fixed declustering ratio, decreasing the group size reduces the parity stripe size, G, which increases the parity overhead of the array, $1/G$. Increasing parity overhead, in turn, increases the amount of storage space each disk must provide, increasing overall array cost. This is the final trade-off: data reliability against cost.

Figure 24 quantifies this reliability versus overhead trade-off for various array sizes, using α=0.25 and α=0.5, the IBM Lightning drives described in Table 2, and reconstruction times given in Figure 18. In general, reconstruction time may be estimated by simulation, as in this paper, or by using an analytical model such as that of Merchant and Yu [Merchant92] or Muntz and Lui [Muntz90].

Figure 24 shows that large arrays will have a 3% to 30% chance of losing data within 10 years. Where this is too large a risk, the array must be partitioned into multiple independent groups. When this is done, data reliability can be increased by an order of magnitude while parity overhead remains beneath 20%, when α=0.25, and beneath 10%, when α=0.5.



(a) α = 0.25



(b) α = 0.50

Figure 24: The tradeoff between reliability and capacity for α = 0.25 and α = 0.50.

The data loss probabilities are plotted on a log scale, while the capacity overhead scale is linear.

This figure also allows us to revisit the question presented in Section 4.4. In this section we discussed selecting between a declustered parity layout based on balanced incomplete block designs or based on random permutations. Pessimistically, if a declustered parity group size exceeds 40 we cannot guarantee a small block design for arbitrary declustering ratio; for such a guarantee, Merchant and Yu's random permutations layout can be used. In terms of Figure 24, points in the lower right of the data loss probability charts correspond to multiple group configurations where individual groups are not larger than 40 disks. If block designs are used, this figure also shows that the parity overhead can be as low as 10%, when α=0.25, or 5%, when α=0.5.

10. Conclusions

In this paper we have attempted to identify and evaluate the primary techniques that can be used to address the failure recovery problem in redundant disk arrays, discussing both the organization of data and parity in the array and the algorithms used to recover from failure. The traditional organization used to address this problem, dividing a RAID level 5 array into multiple groups, is not adequate in high-performance high-availability systems because it forces the array to be utilized at less than 50% in the fault-free case. To solve this, we presented an implementation of *parity declustering*. For the configurations we investigated, this organization supports 40-50% higher failure-mode workload than an equivalent-size RAID level 5 array, and thus is preferable in environments where performance during failure recovery is important. We then investigated the benefits of trading off array capacity for failure-mode performance, and concluded that parity declustering can reduce average and 90th percentile user response time by a factor of two in both degraded mode and reconstruction mode, and can reduce reconstruction time by up to an order of magnitude.

Once the data layout organization is decided upon, the second primary technique for improving the failure-mode performance of a disk array is to tune the reconstruction algorithm. We presented *disk-oriented* reconstruction, and demonstrated that it yields up to 40% faster reconstruction than the traditional *stripe-oriented* approach, while maintaining identical user responsiveness. We investigated the benefits and drawbacks of three modifications that can be applied to the reconstruction algorithm, concluding that for read-dominated workloads, the only option that had significant impact on failure-mode performance was whether user reads to previously-reconstructed data were serviced by the replacement disk or by the surviving disks (the *redirection of reads* option). Since the benefit of redirection is configuration-dependent, we analyzed a proposed technique for optimally controlling its application based on observed disk utilizations. We concluded that the strategy does yield optimal reconstruction time, but that the simpler strategy of applying redirection at for all applicable accesses allows the system to achieve about 10% better user response time for certain configurations.

In the final section of the paper we quantified the trade-offs involved in determining the configuration of large arrays, concluding that it is in most cases necessary to partition large arrays into multiple independent groups to achieve acceptable data reliability. We showed that this partitioning can increase the data reliability by an order of magnitude, while still keeping parity overhead in the range of 10-20%.

There remain several areas to explore on the topic of failure recovery. First, parity-based redundant disk arrays exhibit small-write performance that is up to a factor of four worse than non-redundant arrays, and a factor of two worse than mirrored arrays, and so it would be highly desirable to combine parity declustering with *parity logging* [Stodolsky93], which is a technique for improving this small-write performance in disk arrays. Second, the block-design based layout could be made much more general by relaxing the requirements on the tuples used for layout. For example, it might be possible to derive a balanced layout from a *packing* or *covering* [Mills92] instead of an actual block design, or a layout might be derived from a design in which the number of objects per tuple is not constant. Each of these approaches would expand the range of configurations that can be implemented using the block-design-based layout presented in this paper. Finally, implementing distributed sparing [Menon92b] in a declustered array could eliminate the replacement disk as the reconstruction bottleneck for low values of the declustering ratio (α), and perhaps yield extremely fast reconstruction (on the order of tens of seconds).

References

[ANSI86] *American National Standard for Information Systems -- Small Computer System Interface (SCSI)*, ANSI X3.132-1986, New York NY, 1986.

[ANSI91] *American National Standard for Information Systems -- High Performance Parallel Interface -- Mechanical, Electrical, and Signalling Protocol Specification*, ANSI X3.183-1991, New York NY, 1991.

[Arulpragasam80] J. Arulpragasam and R. Swarz, "A Design for State Preservation on Storage Unit Failure," *Proceedings of the International Symposium on Fault Tolerant Computing*, 1980, pp. 47-52.

[Bitton88] D. Bitton and J. Gray, "Disk Shadowing," *Proceedings of the 14th Conference on Very Large Data Bases*, 1988, pp. 331-338.

[Cao93] P. Cao, S.B. Lim, S. Venkataraman, and J. Wilkes, "The TickerTAIP parallel RAID architecture," *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 52-63.

[Chee90] Y.M. Chee, C. Colbourn, D. Kreher, "Simple t-designs with $v \leq 30$," *Ars Combinatoria*, v. 29, 1990.

[Chen90a] P. Chen, et. al., "An Evaluation of Redundant Arrays of Disks using an Amdahl 5890," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 74-85.

[Chen90b] P. Chen and D. Patterson, "Maximizing Performance in a Striped Disk Array," *Proceedings of International Symposium on Computer Architecture*, 1990.

[Copeland89] G. Copeland and T. Keller, "A Comparison of High-Availability Media Recovery Techniques," *Proceedings of the ACM Conference on Management of Data*, 1989, pp. 98-109.

[Dibble90] P. Dibble, "A Parallel Interleaved File System," University of Rochester Technical Report 334, 1990.

[Fibre91] *Fibre Channel -- Physical Layer*, ANSI X3T9.3 Working Document, Revision 2.1, May 1991.

[Gibson92] G. Gibson, "*Redundant Disk Arrays: Reliable, Parallel Secondary Storage*," MIT Press, 1992.

[Gibson93] G. Gibson and D. Patterson, "Designing Disk Arrays for High Data Reliability," *Journal of Parallel and Distributed Computing*, January, 1993.

[Gray90] G. Gray, B. Horst, and M. Walker, "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," *Proceedings of the Conference on Very Large Data Bases*, 1990, pp. 148-160.

[Hall86] M. Hall, *Combinatorial Theory (2nd Edition)*, Wiley-Interscience, 1986.

[Hanani75] H. Hanani, "Balanced Incomplete Block Designs and Related Designs," *Discrete Mathematics*, v. 11, 1975.

[Holland92] M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 23-25.

[Holland93] M. Holland, G. Gibson, and D. Siewiorek, "Fast, On-Line Failure Recovery in Redundant Disk Arrays," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993.

[Hou93] R. Hou, J. Menon, and Y. Patt, "Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array," *Proceedings of the Hawaii International Conference on Systems Sciences*, 1993, pp. 70-79.

[Hsiao91] H.-I. Hsiao and D.J. DeWitt, "A Performance Study of Three High-Availability Data Replication Strategies," *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1991, pp. 18-28.

[IBM0661] IBM Corporation, IBM 0661 Disk Drive Product Description, Model 370, First Edition, Low End Storage Products, 504/114-2, 1989.

[IEEE89] *Proposed IEEE Standard 802.6 -- Distributed Queue Dual Bus (DQDB) -- Metropolitan Area Network*, Draft D7, IEEE 802.6 Working Group, 1989.

[IEEE93] IEEE High Performance Serial Bus Specification, P1394/Draft 6.2v0, New York, NY, June, 1993.

[Katz89] R. Katz, et. al., "A Project on High Performance I/O Subsystems," *ACM Computer Architecture News*, Vol. 17(5), 1989, pp. 24-31.

[Kim86] M. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. 35 (11), 1986, pp. 978-988.

[Lee90] E. Lee, "Software and Performance Issues in the Implementation of a RAID Prototype," University of California Technical Report UCB/CSD 90/573, 1990.

[Lee91] E. Lee and R. Katz, "Performance Consequences of Parity Placement in Disk Arrays," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 190-199.

[Livny87] M. Livny, S. Khoshafian, H. Boral, "Multi-disk Management Algorithms," *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, 1987, pp. 69-77.

[Mathon90] R. Mathon and A. Rosa, "Tables of Parameters of BIBDs with $r \leq 41$ Including Existence, Enumeration and Resolvability Results: An Update," *Ars Combinatoria*, v. 30, 1990.

[Menon92a] J. Menon and J. Kasson, "Methods for Improved Update Performance of Disk Arrays," *Proceedings of the Hawaii International Conference on System Sciences*, 1992, pp. 74-83.

[Menon92b] J. Menon and D. Mattson, "Comparison of Sparing Alternative for Disk Arrays," *Proceedings of the International Symposium on Computer Architecture*, 1992.

[Menon93] J. Menon and J. Cortney, "The Architecture of a Fault-Tolerant Cached RAID Controller," *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 76-86.

[Merchant92] A. Merchant and P. Yu, "Design and Modeling of Clustered RAID," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1992, pp. 140-149.

[Mills92] W.H. Mills and R.C. Mullin, "Coverings and Packings," Chapter 9 in *Contemporary Design Theory: A Collection of Surveys*, John Wiley & Sons, Inc., 1992, pp. 371-399.

[Muntz90] R. Muntz and J. Lui, "Performance Analysis of Disk Arrays Under Failure," *Proceedings of the Conference on Very Large Data Bases*, 1990, pp. 162-173.

[Park86] A. Park and K. Balasubramanian, "Providing Fault Tolerance in Parallel Secondary Storage Systems," Princeton University Technical Report CS-TR-057-86, 1986.

[Patterson88] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the Conference on Management of Data*, 1988, pp. 109-116.

[Peterson72] W. Peterson and E. Weldon Jr., *Error-Correcting Codes*, second edition, MIT Press, 1972.

[Ramakrishnan92] K. Ramakrishnan, P. Biswas, and R. Karedla, "Analysis of File I/O Traces in Commercial Computing Environments," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1992, pp. 78-90.

[Rosenblum91] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proceedings of the Symposium on Operating System Principles*, 1991, pp. 1-15.

[Schulze89] M. Schulze, G. Gibson, R. Katz, and D. Patterson, "How Reliable is a RAID?," *Proceedings of COMPCON,* 1989, pp. 118-123.

[Stodolsky93] D. Stodolsky, G. Gibson, and M. Holland, "Parity Logging: Overcoming the Small-Write Problem in Redundant Disk Arrays," *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 64-75.

[TPCA89] *The TPC-A Benchmark: A Standard Specification*, Transaction Processing Performance Council, 1989.