Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems, 1992.

Parity Declustering for Continuous Operation in Redundant Disk Arrays

Mark Holland
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890
holland@ece.cmu.edu

Garth A. Gibson
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
garth@cs.cmu.edu

Abstract

We describe and evaluate a strategy for declustering the parity encoding in a redundant disk array. This declustered parity organization balances cost against data reliability and performance during failure recovery. It is targeted at highly-available parity-based arrays for use in continuous-operation systems. It improves on standard parity organizations by reducing the additional load on surviving disks during the reconstruction of a failed disk's contents. This yields higher user throughput during recovery, and/or shorter recovery time.

We first address the generalized parity layout problem, basing our solution on balanced incomplete and complete block designs. A software implementation of declustering is then evaluated using a disk array simulator under a highly concurrent workload comprised of small user accesses. We show that declustered parity penalizes user response time while a disk is being repaired (before and during its recovery) less than comparable non-declustered (RAID5) organizations without any penalty to user response time in the fault-free state.

We then show that previously proposed modifications to a simple, single-sweep reconstruction algorithm further decrease user response times during recovery, but, contrary to previous suggestions, the inclusion of these modifications may, for many configurations, also slow the reconstruction process. This result arises from the simple model of disk access performance used in previous work, which did not consider throughput variations due to positioning delays.

1. Introduction

Many applications, notably database and transaction processing, require both high throughput and high data availability from their storage subsystems. The most demanding of these applications require continuous operation, which in terms of a storage subsystem requires (1) the ability to satisfy all user requests for data even in the presence of a disk failure, and (2) the ability to reconstruct the contents of a failed disk onto a replacement disk, thereby restoring itself to a fault-free state. It is not enough to fulfill these two requirements with arbitrarily degraded performance; it is not unusual for an organization that requires continuously-available data to incur financial losses substantially larger than the organization's total investment in computing equipment if service is severely degraded for a prolonged period of time. Since the time necessary to reconstruct the contents of a failed disk is certainly minutes and possibly hours, we focus this paper on the performance of a continuous-operation storage subsystem during on-line failure recovery. We do not recommend on-line failure recovery in an environment that can tolerate off-line recovery because the latter restores high performance and high data reliability more quickly.

Redundant disk arrays, proposed for increasing input/-output performance and for reducing the cost of high data reliability [Kim86, Livny87, Patterson88, Salem86], also offer an opportunity to achieve high data availability without sacrificing throughput goals. A single-failure-correcting redundant disk array consists of a set of disks, a mapping of user data to these disks that yields high throughput [Chen90b], and a mapping of a parity encoding for the array's data such that data lost when a disk fails can be recovered without taking the system off-line [Lee91].

Most single-failure-correcting disk arrays employ either *mirrored* or *parity-encoded* redundancy. In mirroring [Bitton88, Copeland89, Hsiao90], one or more duplicate copies of all data are stored on separate disks. In parity encoding [Kim86, Patterson88, Gibson91], popularized as Redundant Arrays of Inexpensive Disks (RAID), some subset of the physical blocks in the array are used to store a single-error-correction code (usually parity) computed over subsets of the data. Mirrored systems, while potentially able to deliver higher throughput than parity-based systems for some workloads [Chen90a, Gray90], increase cost by consuming much more disk capacity for redundancy. In this

This work was supported by the National Science Foundation under grant number ECD-8907068, by the Defense Advanced Research Projects Agency monitored by DARPA/CMO under contract MDA972-90-C-0035, and by an IBM Graduate Fellowship.

paper, we examine a parity-based redundancy scheme called *parity declustering*, which provides better performance during on-line failure recovery than more common RAID schemes, without the high capacity overhead of mirroring [Muntz90][1].

Our primary figures of merit in this paper are reconstruction time, which is the wallclock time taken to reconstruct the contents of a failed disk after replacement, and user response time during reconstruction. Reconstruction time is important because it determines the length of time that the system operates at degraded performance, and because it is a significant contributor to the length of time that the system is vulnerable to data loss caused by a second failure. Given a fixed user throughput, contrasting user fault-free response time to the response time after failure (both before and during reconstruction) gives us the measure of our system's performance degradation during failure recovery.

Section 2 of this paper describes our terminology and presents the declustered parity organization. Section 3 describes related studies, notably the introduction of declustering by Muntz and Lui [Muntz90], and explains the motivations behind our investigation. Section 4 presents our parity mapping, which was left as an open problem by Muntz and Lui. Section 5 gives a brief overview of our simulation environment and Sections 6 and 7 present brief analyses of the performance of a declustered array when it is fault-free, and when there is a failed disk but no replacement. Section 8 then covers reconstruction performance, contrasting single-thread and parallel reconstruction, and evaluating alternative reconstruction algorithms. Section 9 concludes the paper with a look at interesting topics for future work.

2. The declustered parity layout policy

Figure 2-1 illustrates the parity and data layout for a left-symmetric RAID5 redundant disk array [Lee91]. *A data stripe unit*, or simply a *data unit* is defined as the minimum amount of contiguous user data allocated to one disk before any data is allocated to any other disk. A *parity stripe unit*, or simply a *parity unit*, is a block of parity information that is the size of a data stripe unit. We use the term *stripe unit* instead of *data unit* or *parity unit* when the distinction between data and parity is not pertinent to the point being made. The size of a stripe unit, called a *unit* for convenience, must be an integral number of sectors, and is often the minimum unit of update used by system software. A *parity stripe*[2] is the set of data units over which a parity unit is computed, plus the parity unit itself. In Figure 2-1, *Di.j* represents one of the four data units in parity stripe number *i*, and *Pi* represents the parity unit for parity stripe *i*. Parity units are distributed across the disks of the array to avoid the write bottleneck that would occur if a single disk contained all parity units.

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

Figure 2-1: Parity and data layout in a left-symmetric RAID5 organization.

The disk array's data layout provides the abstraction of a linear ("logical block") address space to the file system. In addition to mapping data units to parity stripes, a left-symmetric RAID5 organization also specifies the data layout: data is mapped to stripe units *Di.j* according to ascending *j* within ascending *i*. In Figure 2-1, this means that user data is logically D0.0, D0.1, D0.2, D0.3, D1.0, D1.1, etc. This is a good choice because any set of five contiguous data units map to five different disks, maximizing read parallelism. However, a file system may or may not allocate user blocks contiguously in this address space, so a disk array data mapping can not alone insure maximal parallelism.

In Figure 2-1, parity is computed over the entire width of the array; that is, P0 is the cumulative parity (exclusive-or) of data units D0.0 through D0.3. When a disk is identified as failed, any data unit can be reconstructed by reading the corresponding units in the parity stripe, including the parity unit, and computing the cumulative exclusive-or of this data. Note, however, that all the disks in the array are needed by every access that requires reconstruction. We shall see that declustered parity relaxes this constraint.

Following Muntz and Lui, let G be the number of units in a parity stripe, including the parity unit, and consider the problem of decoupling G from the number of disks in the array. This reduces to a problem of finding a parity mapping that will allow parity stripes of size G units to be distributed over some larger number of disks, C. For our purposes, this larger set of C disks is the whole array. For comparison purposes, the RAID5 example in Figure 2-1 has $G = C = 5$. This property, that $G = C$, defines RAID5 mappings in the context of this paper.

One perspective on the concept of parity declustering in redundant disk arrays is demonstrated in Figure 2-2; a logical RAID5 array with $G = 4$ is distributed over $C = 7 > G$ disks, each containing fewer units. (The way this

1. Muntz and Lui use the term *clustered* where we use the term *declustered*. Their use may be derived from "clustering" independent RAIDs into a single array with the same parity overhead. Our use follows the earlier work of Copeland and Keller [Copeland89] where redundancy information is "declustered" over more than the minimal collection of disks.

2. Muntz and Lui [Muntz90] use the term *group* to denote what we call a parity stripe, but we avoid this usage as it conflicts with the Patterson, et. al. definition [Patterson88] as a set of disks, rather than a set of disk blocks.

mapping is generated is the topic of Section 4.) The advantage of this approach is that it reduces the reconstruction workload applied to each disk during failure recovery. To see this, note that for any given stripe unit on a failed (physical) disk, the parity stripe to which it belongs includes units on only a subset of the total number of disks in the array. In Figure 2-2, for example, disks 2, 3, and 6 do not participate in the reconstruction of the parity stripe marked ‘S’. Hence, these disks are called on less often in the reconstruction of one of the other disks. In contrast, a RAID5 array has $C = G$, and so all disks participate in the reconstruction of all units of the failed disk.

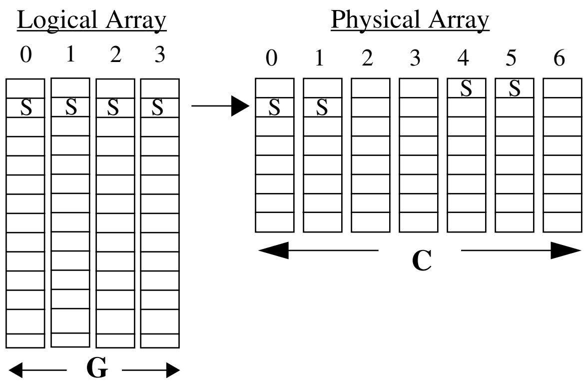

Figure 2-2: One possible declustering of a parity stripe of size four over an array of seven disks.

Figure 2-3 presents a declustered parity layout for $G = 4$ and $C = 5$ that is described in Section 4. What is important at this point is that fifteen data units are mapped onto five parity stripes in the array’s first twenty disk units, while in the RAID5 organization of Figure 2-1, sixteen data units are mapped onto four parity stripes in the same number of disk units. More disk units are consumed by parity, but not every parity stripe is represented on each disk, so a smaller fraction of each surviving disk is read during reconstruction. For example, if, in Figure 2-3, disk zero fails, parity stripe four will not have to be read during reconstruction. Note that the successive stripe units in a parity stripe occur at varying disk offsets: Section 6 shows that this has no significant performance impact. Although Figure 2-3 does not spread parity units evenly over all disks, the constructions we present in Section 4 all possess this property.

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 2-3: Example data layout in a declustered parity organization.

Muntz and Lui define the ratio $(G\text{-}1)/(C\text{-}1)$ as α. This parameter, which we call the *declustering ratio*, indicates the fraction of each surviving disk that must be read during the reconstruction of a failed disk. Note that $\alpha = 1$ for the RAID5 organization, indicating that all of every surviving disk participates in a reconstruction. All of our performance graphs in Sections 6, 7, and 8 are parameterized by α.

The parameters C and G and the ratio α together determine the reconstruction performance, the data reliability, and the cost-effectiveness of the array. C determines the cost of the array because it specifies the number of disks. It also determines the array’s data reliability because our mappings require some data from all surviving disks to reconstruct a failed disk. Thus any two failures in the C disks constituting the array will cause data loss. G, on the other hand, determines the percentage of total disk space consumed by parity, $1/G$. Finally, the declustering ratio α determines the reconstruction performance of the system; a smaller value should yield better reconstruction performance since a failed disk’s reconstruction workload is spread over a larger number of disks. In general, system administrators need to be able to specify C and G at installation time according to their cost, performance, capacity, and data reliability needs. This paper provides analyses upon which these decisions can be based.

3. Related work

The idea of improving failure-mode performance by declustering redundancy information originated with mirrored systems [Copeland89, Hsiao90]. Copeland and Keller describe a scheme called *interleaved declustering* which treats *primary* and *secondary* data copies differently. Traditionally, mirrored systems allocate one disk as a primary and another as a secondary. Copeland and Keller instead allocate only half of each disk for primary copies. The other half of each disk contains a portion of the secondary copy data from each of the primaries on all other disks. This insures that a failure can be recovered since the primary and secondary copies of any data are on different disks. It also distributes the workload associated with reconstructing a failed disk across all surviving disks in the array. Hsiao and DeWitt propose a variant called *chained declustering*, that increases the array’s data reliability.

Muntz and Lui applied ideas similar to those of Copeland and Keller to parity-based arrays. They proposed the declustered parity organization described in Section 2, and then modeled it analytically, making a number of simplifying assumptions. We attempt in this paper to identify the limits of this theoretical analysis and provide performance predictions based instead on a software implementation and array simulation. Toward this end we have two primary concerns with the Muntz and Lui analysis.

First, their study assumes that either the set of surviving disks or the replacement disk is driven at 100% utilization. Unfortunately, driving a queueing system such as a magnetic disk at full utilization leads to arbitrarily long response times. Response time is important to all customers and critical in database and on-line transaction-processing (OLTP) systems. The specifications for OLTP systems often require some minimal level of responsiveness, for example, one benchmark requires that 90% of the transactions must complete in under two seconds [TPCA89]. In a continuous-operation system that requires minutes to hours for the recovery of a failed disk, this rule will apply even during these relatively rare recovery intervals. Our analysis reports on user response time during recovery and presents a simple scheme trading off reconstruction time for user response time.

Our second concern with the Muntz and Lui analysis is that their modeling technique assumes that all disk accesses have the same service time distribution. Real disk accesses are, however, subject to positioning delays that are dependent on the current head position and the position of target data. As an example, suppose that a given track on a replacement disk is being reconstructed, and that a few widely scattered stripe units on that track are already valid because they were written as a result of user (not reconstruction) accesses during reconstruction. These units may be skipped over by reconstruction, or they may simply be reconstructed along with the rest of the track and over-written with the data that they already hold. Whichever option is selected, the previously-reconstructed sectors will still have to rotate under the disk heads, and hence the time needed to reconstruct the track will not decrease. The Muntz and Lui model assumes that the track reconstruction time is reduced by a factor equal to the size of the units not needing reconstruction divided by the size of the track, which is not the case. This idea that disk drives are not "work-preserving" due to head positioning and spindle rotation delays is an effect that is difficult to model analytically, but relatively straightforward to address in a simulation-based study.

In this paper we will use balanced incomplete and complete block designs (described in Section 4.2) to achieve better performance during reconstruction. Reddy [Reddy91] has also used block designs to improve the recovery-mode performance of an array. His approach generates a layout with properties similar to ours, but is restricted to the case where $G = C/2$.

4. Data layout strategy

In this section, we describe our layout goals, and the technique used to achieve them. We then comment on the generality of our approach.

4.1. Layout goals

Previous work on declustered parity has left open the problem of allocating parity stripes in an array. Extending from non-declustered parity layout research [Lee90, Dibble90], we have identified six criteria for a good parity layout. 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. Because file systems are free to and often do allocate user data arbitrarily into whatever logical space a storage subsystem presents, our parity layout procedures have no direct control over these latter two criteria.

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 redundancy organization that recovers the data of failed disks. 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 reconstruction.* 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. Every data update causes a parity update, and so an uneven parity distribution would lead to imbalanced utilization (hot spots), since the disks with more parity would experience more load.
4. *Efficient mapping*. The functions mapping a file system's logical block address to physical disk addresses for the corresponding data units and parity stripes, 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 data units should correspond to the allocation of 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.

As shown in Figure 2-1, the left-symmetric mapping for RAID5 arrays meets all of these criteria.

4.2. Layout strategy

Our declustered parity layouts are specifically designed to meet our criterion for distributed reconstruction while also lowering the amount of reconstruction work done by each surviving disk. The distributed reconstruction criterion requires 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 number of times that a pair of disks contain stripe units from the same parity stripe is constant across all pairs of disks. Muntz and Lui recognized and suggested that such layouts might be found in the literature for *balanced incomplete block designs* [Hall86]. This paper demonstrates that this can be done and one way to do it.

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 Figure 4-1.

Tuple 0: 0, 1, 2, 3	Tuple 3: 0, 2, 3, 4
Tuple 1: 0, 1, 2, 4	Tuple 4: 1, 2, 3, 4
Tuple 2: 0, 1, 3, 4	

Figure 4-1: A sample complete 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 $\binom{v}{k}$. It is useful to note that only three of v, k, b, r, and λ_p are free variables because the following two relations are always true: $bk = vr$, and $r(k-1) = \lambda_p(v-1)$. 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 4-2 from the block design in Figure 4-1. To build a parity layout, we find a block design with $v = C$, $k = G$, and the minimum possible value for b (as explained in Section 4.3). 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 are allocated. In Figure 4-2, the first tuple in the design of Figure 4-1 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. The layout shown in the top quarter of Figure 4-2, and in Figure 2-3, is derived via this process from the block design in Figure 4-1.

It is apparent from Figure 2-3 that this approach produces a layout that violates our distributed parity criterion (3). To resolve this violation, we derive a layout as above and duplicate it G times (four times in Figure 4-2), assigning parity to a different element of each tuple in each duplication, as shown in the right side of Figure 4-2. This layout, the entire contents of Figure 4-2, 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 4-2) as the *block design table*, and one complete cycle (all blocks in Figure 4-2) as the *full block design table*.

We now show how well this layout procedure meets the first four of our layout criteria. No pair of stripe units in the same parity stripe will be assigned to the same disk because the G elements of each tuple are distinct. This insures our single failure correcting criterion. The second criterion, which is that reconstruction be distributed evenly, is guaranteed because each pair of objects appears in exactly λ_p tuples. This means that in each copy of the block design table, disk i occurs in exactly λ_p parity stripes with each other disk. Hence, when disk i fails, every other disk reads exactly λ_p stripe units while reconstructing the stripe units associated with each block design table. Note that the actual value of λ_p is not significant. It is only necessary that it be constant across all pairs of disks, and this is guaranteed by the definition of a block design.

Our distributed parity criterion is achieved because a full block design table is composed of G block design tables each assigning parity to a different element of each tuple. This results in the assignment of parity to each disk in the cluster exactly r times over the course of a full block design table. To see this, refer to Figure 4-2: if we group together the vertical boxes in the right half of the figure we see that we have reconstituted one block design table; that is, the parity assignment function touches each element of the block design table exactly once over the course of one full block design table. Since each object appears in exactly r tuples in a block design table, each disk will be assigned parity in exactly r tuples in the full block design table.

Unfortunately it is not guaranteed that our layout will have an efficient mapping, our fourth criterion, because the size of a block design table is not guaranteed to be small. However, Section 4.3 demonstrates that small block design tables are available for a wide range of parity stripe and array sizes.

Finally, our fifth and sixth criteria depend on the data mapping function used by higher levels of software. Unfortunately, the simple mapping of data to successive data units within successive parity stripes (Figure 4-2) that we use in our simulations, while meeting our large-write optimization criterion, does not meet our maximal parallelism criterion; that is, not all sets of five adjacent data units from the mapping, D0.0, D0.1, D0.2, D1.0, D1.1, D1.2, D2.0, etc., in Figure 4-2 are allocated on five different disks. Instead, reading five adjacent data units starting at data unit zero causes disk 0 and 1 to be used twice, and disks 3 and 4 not at all. On the other hand, if we were to employ a data mapping similar to Lee's left-symmetric parity (for non-declustered RAID5 arrays), we may fail to satisfy our large-write optimization

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.

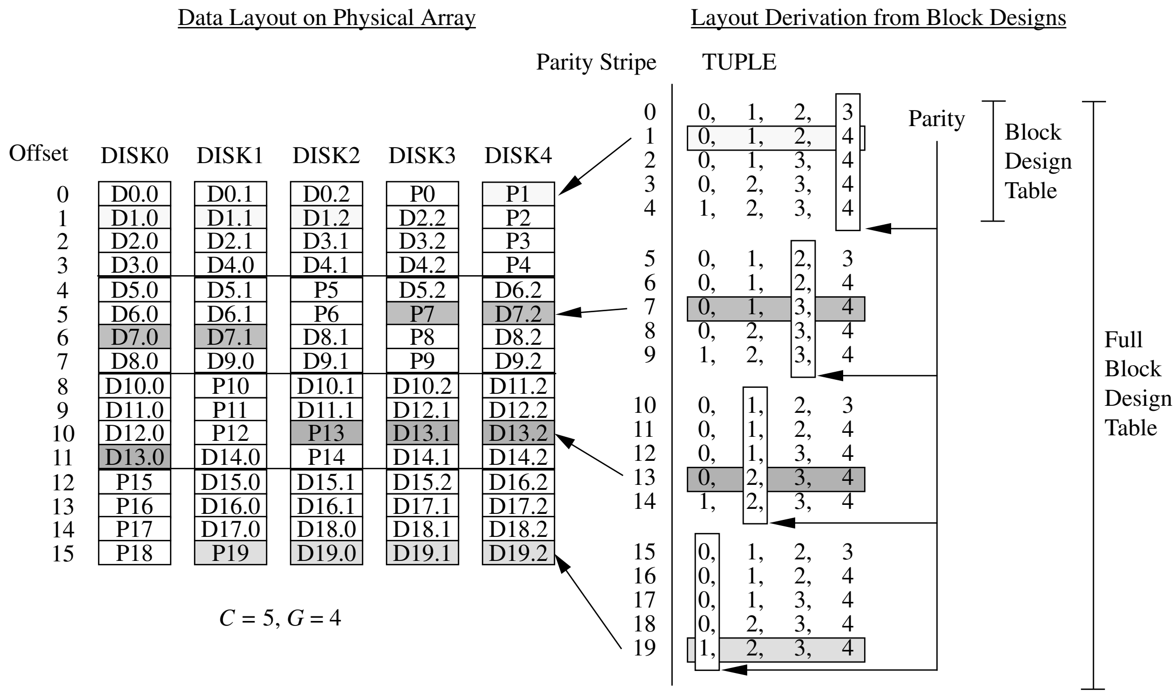

Figure 4-2: Full block design table for a parity declustering organization.

criterion. We leave for future work the development of a declustered parity scheme that satisfies both of these criteria.

4.3. On the generation of block designs

Complete block designs such as the one in Figure 4-1 are easily generated, but in many cases they are insufficient for our purposes. When the number of disks in an array (C) is large relative to the number of stripe units in a parity stripe (G) then the size of the block design table becomes unacceptably large and the layout fails our efficient mapping criterion. For example, a 41 disk array with 20% parity overhead (G=5) allocated by a complete block design will have about 3,750,000 tuples in its full block design table. In addition to the exorbitant memory requirement for this table, the layout will not meet our distributed parity or distributed reconstruction criteria because even large disks rarely have more than a few million sectors. For this reason we turn to the theory of *balanced incomplete block designs* [Hall86].

Our goal, then, is to find a small block design on C objects with a tuple size of G. This is a difficult problem for general C and G. Hall presents a number of techniques, but these are of more theoretical interest than practical value since they do not provide sufficiently general techniques for the direct construction of the necessary designs. Fortunately, Hall also presents a list containing a large number of known block designs, and states that, within the bounds of this list, a solution is given in every case where one is known to exist. Figure 4-3[4] presents a scatter plot of a subset of Hall's list of designs. Whenever possible, our parity declustering implementation uses one of Hall's designs.

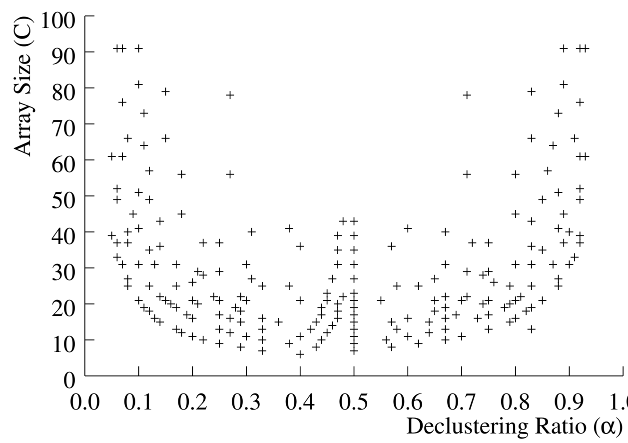

Figure 4-3: Known block designs.

Sometimes a balanced incomplete block design with the required parameters may not be known. In cases where we cannot find a balanced incomplete block design, we attempt to use a complete block design on the indicated parameters. When this method produces an unacceptably large design, we resort to choosing the closest feasible design point; that is, the point which yields a value of α closest to what is desired. All of our results indicate that the performance of an array is not highly sensitive to such small variations in α. The block designs we used in our simulations are given in [Holland92].

4. These block designs (and others) are available via anonymous ftp from niagara.nectar.cs.cmu.edu (128.2.250.200) in the file /usr0/anon/pub/Declustering/BD_database.tar.Z.

5. The simulation environment

We acquired an event-driven disk-array simulator called *raidSim* [Chen90b, Lee91] for our analyses. The simulator was developed for the RAID project at U.C. Berkeley [Katz89]. It consists of four primary components, which are illustrated in Figure 5-1. Each component is described below.

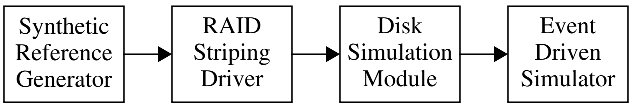

Figure 5-1: The structure of raidSim.

At the top level of abstraction is a *synthetic reference generator*, which is capable of producing user request streams drawn from a variety of distributions. Table 5-1 (a) shows the configuration of the workload generator used in our simulations. We have restricted our attention to random accesses of size 4 KB to model an OLTP system with an effective buffer cache [Ramakrishnan92]. Each request produced by this generator is sent to a *RAID striping drive*r, which was originally the actual code used by the Sprite operating system [Ousterhout88] to implement a RAID device on a set of independent disks. Table 5-1 (b) shows the configuration of our extended version of this striping driver. These upper two levels of *raidSim* should actually run on a Sprite machine. 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 5-1 (c) 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.

The striping driver code was originally taken directly from the Sprite source code, with essentially zero modification to accommodate simulation, and all our modifications conform to Sprite constraints. This assures that reference streams generated by this driver are identical to those that would be observed in an actual disk array running the same synthetic workload generator. It also forces us to actually implement our layout strategy and reconstruction optimizations, since we have extended the code in such a way that it could be re-incorporated into an operating system at any time. This minimizes the possibility that un-considered implementation details will lead to erroneous conclusions. All reconstruction algorithms discussed in Section 8 have been fully implemented and tested under simulation in our version of the RAID striping driver.

It is evident that our simulator models a software implementation of a striped disk array, but we expect a hardware implementation to deliver similar performance. This is because (1) we do not model the time taken to compute the XOR functions in the CPU, (2) our simulator does not incur rotation slips due to interrupt overhead and command processing time in the CPU, and (3) the code path length for the mapping functions is not significantly longer in our declustered layouts than in the RAID5 case.

(a) Workload Parameters		
Access size:	Fixed at 4 KB	
User access rate:	105, 210, and 378 accesses/second	
Alignment:	Fixed at 4 KB	
Distribution:	Uniform over all data	
Write Ratio:	0% and 100%	(Sections 6, 7)
	50%	(Section 8)
(b) Array Parameters		
Stripe Unit:	Fixed at 4KB	
Number of Disks:	Fixed at 21 spindle-synchronized disks.	
Head Scheduling:	CVSCAN [Geist87]	
Parity Stripe Size:	3, 4, 5, 6, 10, 14, 18, and 21 stripe units	
Parity Overhead:	33, 25, 20, 17, 10, 7, 6, and 5%, resp.	
Data Layout:	RAID5:	Left Symmetric
	Declustered:	By parity stripe index
Parity Layout:	RAID5:	Left Symmetric
	Declustered:	Block Design Based
Power/Cabling:	Disks independently powered/cabled	
(c) Disk Parameters		
Geometry:	949 cylinders, 14 heads, 48 sectors/track	
Sector Size:	512 bytes	
Revolution Time:	13.9 ms	
Seek Time Model:	$2.0 + 0.01 \cdot dist + 0.46 \cdot \sqrt{dist}$ (ms)	
	2 ms min, 12.5 ms avg, 25 ms max	
Track Skew:	4 sectors	

Table 5-1: Simulation parameters.

6. Fault-free performance

Figures 6-1 and 6-2[5] show the average response time experienced by read and write requests in a fault-free disk array as a function of the declustering ratio, α. Our simulated system has 21 disks, so the fraction of space consumed by parity units, 1/G, is 1/(20α+1). In the 100% read case we show three average response time curves corresponding to user access rates (λ) of 105, 210, and 378[6] random reads of 4 KB per second (on average, 5, 10, and 18 user reads of 4 KB per second per disk are applied to disks capable of a maximum of about 46 random 4 KB reads per second). In the 100% write case we show two curves with much longer average response time, corresponding to $\lambda = 105$ and

5. In all performance plots, each data point is an average over five independently-seeded runs of the simulator. We find that the 95% confidence intervals are too small to include on any graph. The maximum width of a confidence interval over all data points on all plots is 6.7% of the value of the data point.

6. In our simulated arrays of 21 disks, 105, 210, and 378 accesses per second correspond to 5, 10, and 18 accesses per second per disk. These user accesses expand into even more physical accesses due to four-cycle writes and on-the-fly reconstruction. Five accesses/second/disk was chosen as a low load, ten was chosen as a value close to the maximum supportable workload (50% writes) in reconstruction mode, and eighteen was determined to be the maximum supportable user per-disk access write (50% writes) in the fault-free mode.

λ = 210 random user writes of 4 KB per second. User writes are much slower than user reads because writes must update parity units as well as data units. Our striping driver's fault-free behavior is to execute four separate disk accesses for each user write instead of the single disk access needed by a user read. Because of this high cost for user writes, our system is not able to sustain 378 user writes of 4 KB per second (this would be 72 4 KB accesses per second per disk).

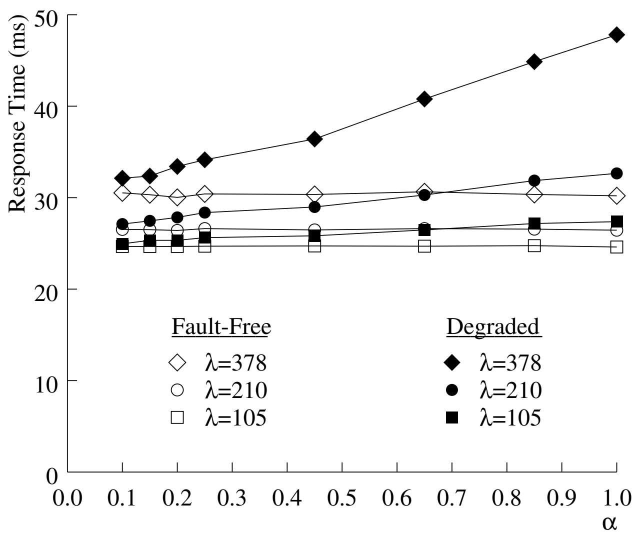

Figure 6-1: Response time: 100% reads.

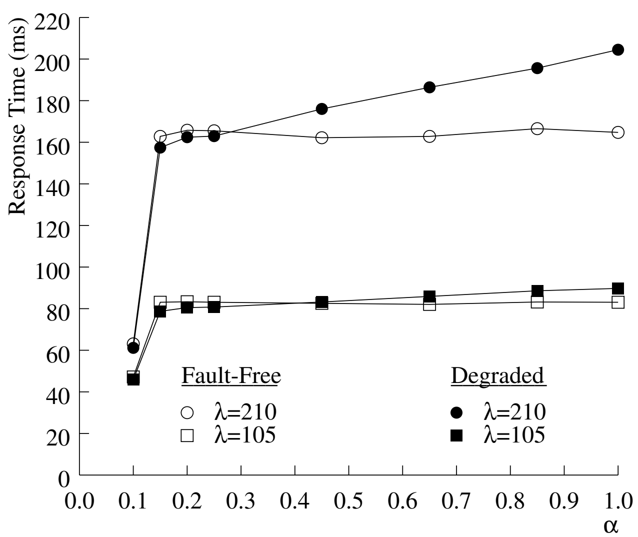

Figure 6-2: Response time: 100% writes.

Although using four separate disk accesses for every user write is pessimistic [Patterson88, Menon92, Rosenblum91], it gives our results wider generality than would result from simulating specifically optimized systems.

Figures 6-1 and 6-2 show that, except for writes with $\alpha = 0.1$, fault-free performance is essentially independent of parity declustering. This exception is the result of an optimization that the RAID striping driver employs when requests for one data unit are applied to parity stripes containing only three stripe units [Chen90a]. In this case the driver can choose to write the specified data unit, read the other data unit in this parity stripe, then directly write the corresponding parity unit in three disk accesses, instead of pre-reading and overwriting both the specified data unit and corresponding parity unit in four disk accesses. In the rest of this paper we will neglect the case of $\alpha = 0.1$ ($G = 3$) to avoid repeating this optimization discussion.

We note that our parity declustering implementation may not perform equivalently to a left-symmetric RAID5 mapping when user requests are larger than one data unit. Declustered parity has the advantage of exploiting our large-write optimization with smaller user writes because it has smaller parity stripes. On the other hand, because our implementation does not currently meet our maximal parallelism criterion, it will not be as able to exploit the full parallelism of the array on large user accesses. Overall performance for workloads not modeled by our simulations will be dictated by the balancing of these two effects, and will depend on the access size distribution.

7. Degraded-mode performance

Figures 6-1 and 6-2 also show average response time curves for our array when it is degraded by a failed disk that has not yet been replaced. In this case, an on-the-fly reconstruction takes place on every access that requires data from the failed disk. When such an access is a read, the failed disk's contents are reconstructed by reading and computing an exclusive-or over all surviving units of the requested data's parity stripe. Because the amount of work this entails depends on the size of each parity stripe, G, these degraded-mode average response time curves suffer less degradation with a lower parity declustering ratio (smaller α) when the array size is fixed ($C = 21$). This is the only effect shown in Figure 6-1 (100% reads), but for writes there is another consideration. When a user write specifies data on a surviving disk whose associated parity unit is on the failed disk, there is no value in trying to update this lost parity. In this case a user write induces only one, rather than four, disk accesses. This effect decreases the workload of the array relative to the fault-free case, and, at low declustering ratios, it may lead to slightly better average response time in the degraded rather than fault-free mode!

8. Reconstruction performance

The primary purpose for employing parity declustering over RAID5 organizations has been given as the desire to support higher user performance during recovery and shorter recovery periods [Muntz90]. In this section we show this to be effective, although we also show that previously proposed optimizations do not always improve reconstruction performance. We also demonstrate that there remains an important trade-off between higher performance during recovery and shorter recovery periods.

Reconstruction, in its simplest form, involves a single sweep through the contents of a failed disk. For each stripe unit on a replacement disk, the reconstruction process reads all other stripe units in the corresponding parity stripe and computes an exclusive-or on these units. The resulting unit is then written to the replacement disk. The time needed to entirely repair a failed disk is equal to the time needed to replace it in the array plus the time needed to reconstruct its entire contents and store them on the replacement. This latter time is termed the *reconstruction time*. In an array that maintains a pool of on-line spare disks, the replacement time can be kept sufficiently small that repair time is essentially reconstruction time. Highly available disk arrays require short repair times to assure high data reliability because the mean time until data loss is inversely proportional to mean repair time [Patterson88, Gibson93]. However, minimal reconstruction time occurs when user access is denied during reconstruction. Because this cannot take less than the three minutes it takes to read all sectors on our disks, and usually takes much longer, continuous-operation systems require data availability during reconstruction.

Muntz and Lui identify two optimizations to a simple sweep reconstruction: *redirection of reads* and *piggybacking of writes*. In the first, user accesses[7] to data that has already been reconstructed are serviced by (redirected to) the replacement disk, rather than invoking on-the-fly reconstruction as they would if the data were not yet available. This reduces the number of disk accesses invoked by a typical request during reconstruction. In the second optimization, user reads that cause on-the-fly reconstruction also cause the reconstructed data to be written to the replacement disk. This is targeted at speeding reconstruction, since those units need not be subsequently reconstructed.

Muntz and Lui also mention that in servicing a user's write to a data unit whose contents have not yet been reconstructed, the device driver has a choice of writing the new data directly to the replacement disk or updating only the parity unit(s) to reflect the modification. In the latter case, the data unit(s) remains invalid until the reconstruction process reaches it. Their model assumes that a user-write targeted at a disk under reconstruction is always sent to the replacement, but for reasons explained below, we question whether this is always a good idea. Therefore, we investigate four algorithms, distinguished by the amount and type of non-reconstruction workload sent to the replacement disk. In our *minimal-update* algorithm, no extra work is sent; whenever possible user writes are folded into the parity unit, and neither reconstruction optimization is enabled. In our *user-writes* algorithm (which corresponds to Muntz and Lui's *baseline* case), all user writes explicitly targeted at the replacement disk are sent directly to the replacement. The *redirection* algorithm adds the redirection of reads optimization to the user-writes case. Finally, the *redirect-plus-piggyback* algorithm adds the piggybacking of writes optimization to the redirection algorithm.

7. In what follows, we distinguish between *user accesses*, which are those generated by applications as part of the normal workload, and *reconstruction accesses*, which are those generated by a background reconstruction process to regenerate lost data and store it on a replacement disk.

8.1 Single thread vs. parallel reconstruction

Figures 8-1 and 8-2 present the reconstruction time and average user response time for our four reconstruction algorithms under a user workload that is 50% 4 KB random reads and 50% 4 KB random writes. These figures show the substantial effectiveness of parity declustering for lowering both the reconstruction time and average user response time relative to a RAID5 organization ($\alpha = 1.0$). For example, at 105 user accesses per second, an array with declustering ratio 0.15 reconstructs a failed disk about twice as fast as the RAID5 array while delivering an average user response time that is about 33% lower.

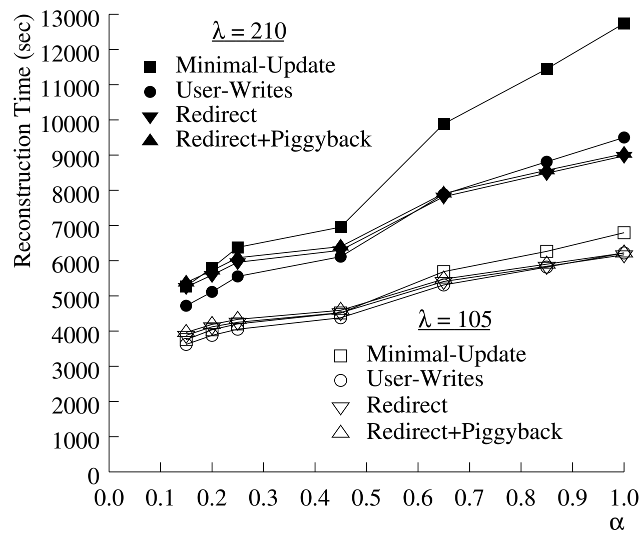

Figure 8-1: Single-thread reconstruction time: 50% reads, 50% writes.

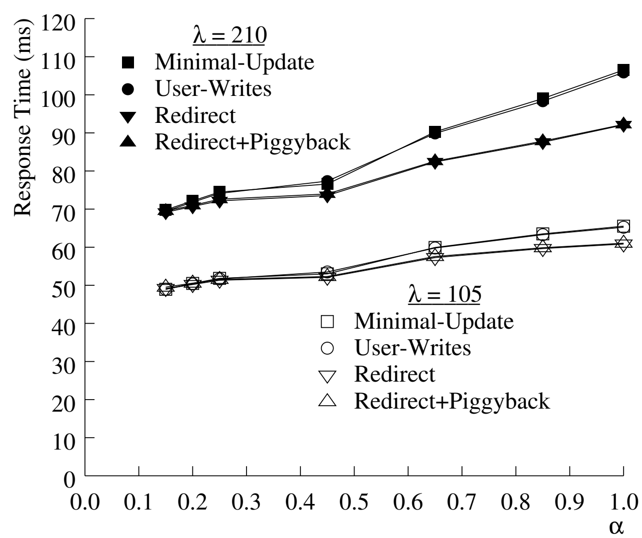

Figure 8-2: Single-thread average user response time: 50% reads, 50% writes.

While Figures 8-1 and 8-2 make a strong case for the advantages of declustering parity, even the fastest reconstruction shown, about 60 minutes[8], is 20 times longer than the physical minimum time these disks take to read or write their entire contents. The problem here is that a single reconstruction process, which uses blocking I/O operations, is not able to highly utilize the array's disks, particularly at low declustering ratios. To further reduce reconstruction time we introduce parallel reconstruction processes. This should speed reconstruction substantially, although it will also further degrade the response time of concurrent user accesses.

Figures 8-3 and 8-4 show the reconstruction time and average user response time when eight processes are concurrently reconstructing a single replacement disk[9]. For workloads of 105 and 210 user accesses per second, reconstruction time is reduced by a factor of four to six relative to Figure 8-1. This gives reconstruction times between 10 and 40 minutes. However, response times suffer increases of 35% to 75%. Still, even the worst of these average response times is less than 200 ms, so a simple transaction such as the one used in the TPC-A benchmark, which should require less than three disk accesses per transaction, should have a good chance of meeting its required transaction response time of two seconds.

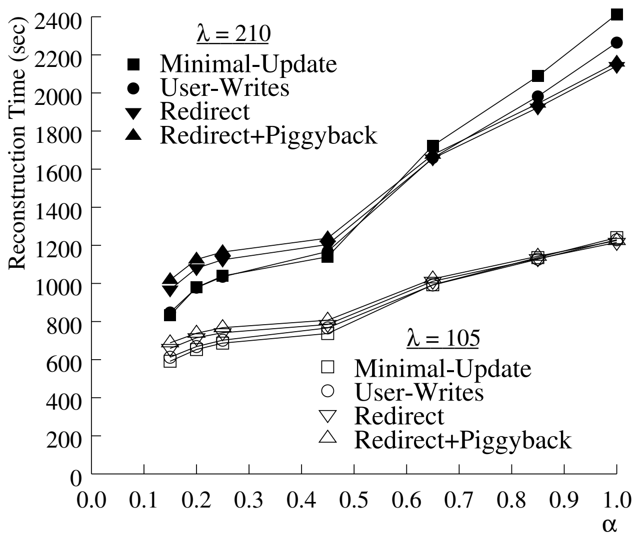

Figure 8-3: Eight-way parallel reconstruction time: 50% reads, 50% writes.

8.2 Comparing reconstruction algorithms

The response time curves of Figures 8-2 and 8-4 show that Muntz and Lui's redirection of reads optimization has little benefit in lightly-loaded arrays with a low declustering ratio, but can benefit heavily-loaded RAID5 arrays with 10% to 15% reductions in response time. The addition of piggybacking of writes to the redirection of reads algorithm

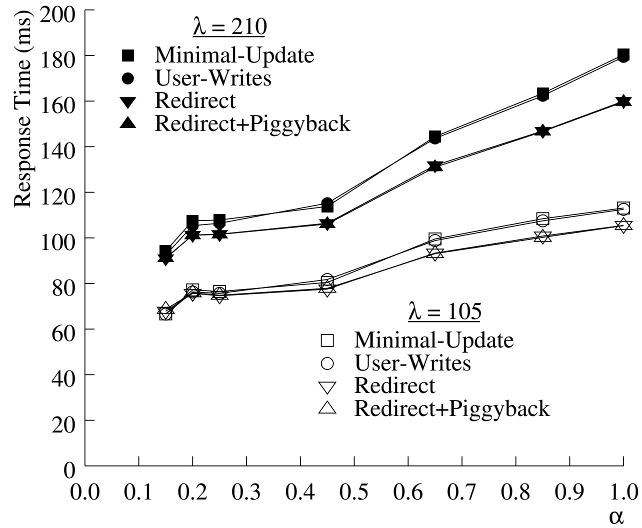

Figure 8-4: Eight-way parallel average user response time: 50% reads, 50% writes.

is intended to reduce reconstruction time without penalty to average user response time. With the user loads we employ, piggybacking of writes yields performance that differs very little from redirection of reads alone. We will not pursue piggybacking of writes further in this section.

Figures 8-1 and 8-2 also show that the two "more optimized" reconstruction algorithms do not consistently decrease reconstruction time relative to the simpler algorithms. In particular, under single-threaded reconstruction, the user-writes algorithm yields faster reconstruction times than all others for all values of α less than 0.5. Similarly, the eight-way parallel minimal-update and user-writes algorithms yields faster reconstruction times than the other two for all values of α less than 0.5. These are surprising results. We had expected the more optimized algorithms to experience improved reconstruction time due to the off-loading of work from the over-utilized surviving disks to the under-utilized replacement disk, and also because in the piggybacking case, they reduce the number of units that need to get reconstructed. The reason for this reversal is that loading the replacement disk with random work penalizes the reconstruction writes to this disk more than off-loading benefits the surviving disks unless the surviving disks are highly utilized. A more complete explanation follows.

For clarity, consider single-threaded reconstruction. Because reconstruction time is the time taken to reconstruct all parity stripes associated with the failed disk's data, one at a time, our unexpected effect can be understood by examining the number of and time taken by reconstructions of a single unit as shown in Figure 8-5. We call this single stripe unit reconstruction period a *reconstruction cycle*. It is composed of a read phase and a write phase. The length of the read phase, the time needed to collect and exclusive-or the surviving stripe units, is approximately the maximum of G-1 reads of disks that are also supporting a moderate load

8. For comparison, RAID products available today specify on-line reconstruction time to be in the range of one to four hours [Meador89, Rudeseal92].

9. Each reconstruction process acquires the identifier of the next stripe to be reconstructed off of a synchronized list, reconstructs that stripe, and then repeats the process.

of random user requests. Since these disks are not saturated, off-loading a small amount of work from them will not significantly reduce this maximum access time. But, even a small amount of random load imposed on the replacement disk may greatly increase its average access times because reconstruction writes are sequential and do not require long seeks. This effect, suggesting a preference for algorithms that minimize non-reconstruction activity in the replacement disk, must be contrasted with the reduction in number of reconstruction cycles that occurs when user activity causes writes to the portion of the replacement disk's data that has not yet been reconstructed.

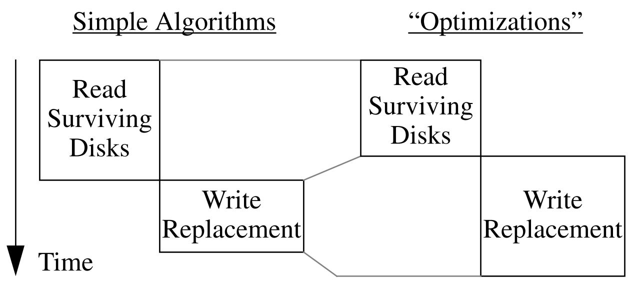

Figure 8-5: The effect of redirection and piggybacking on the reconstruction cycle.

Figure 8-6 presents a sample of the durations of the intervals illustrated in Figure 8-5. These numbers are averaged over the reconstruction of the last 300 stripe units on a replacement disk. For these final reconstruction cycles, the piggybacking of writes is not likely to occur, but the redirection of reads will be at its greatest utility. We selected this operating point to give the redirection of reads algorithm its maximum advantage. The figure shows that in the eight-way parallel case, the more complex algorithms tend to yield lower read phase times and higher write phase times. These numbers suggest that with a low declustering ratio the minimal-update algorithm has an advantage over the user-writes algorithm and both are faster than the other two algorithms.

The single-thread case of Figure 8-6 and Figures 8-1 and 8-3 do not entirely bear out this suggestion because the minimal-update algorithm gets none of its reconstruction work done for it by user requests, as is the case for the other algorithms. In the single-threaded case, reconstruction is so slow that the loss of this "free reconstruction" benefit dominates and the minimal-update algorithm reconstructs more slowly than the others for all but the smallest declustering ratio. However, in the eight-way parallel case, reconstruction is fast enough that free reconstruction does not compensate for longer services times experienced by the replacement disk because free reconstruction moves the heads around randomly.

8.3 Comparison to analytic model

Muntz and Lui also proposed an analytic expression for reconstruction time in an array employing declustered

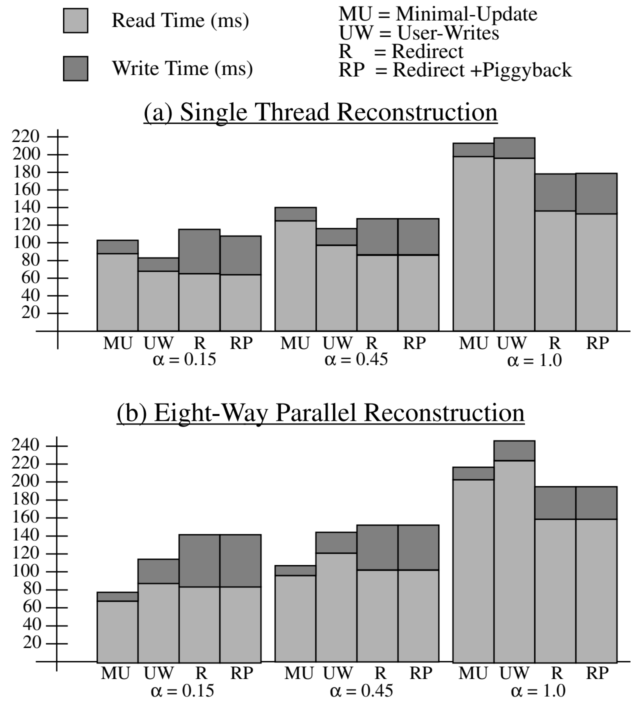

Figure 8-6: Reconstruction cycle time (ms) at 210 user accesses per second.

parity [Muntz90]. Figure 8-7 shows our best attempt to reconcile their model with our simulations. Because their model takes as input the fault-free arrival rate and read-fraction of accesses to the disk rather than of user requests, we apply the following conversions, required because each user write induces two disk reads and two disk writes. Letting the fraction of user accesses that are reads be R, Muntz and Lui's rate of arrival of disk accesses is (4-3R) times larger than our user request arrival rate, and the fraction of their disk accesses that are reads is (2-R)/(4-3R).

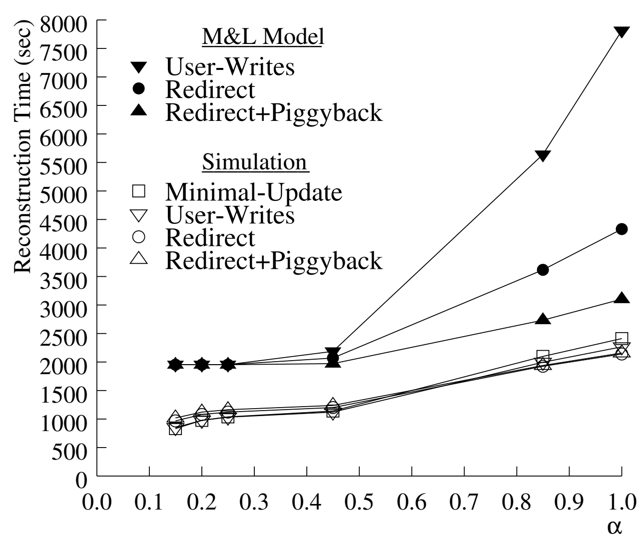

Figure 8-7: Comparing reconstruction time model to simulation: 210 user accesses/second.

Their model also requires as input the maximum rate at which each disk executes its accesses. It is this parameter that causes most of the disagreement between their model and our simulations. In Figure 8-7 we use the maximum rate at which one disk services entirely random accesses (46 accesses per second) in their model. This is consistent with the way Muntz and Lui's designed their model, but it does not account for the fact that real disks execute sequential accesses much faster than random accesses. For example, with these disks and random accesses the minimum time required to read or write an entire disk is over 1700 seconds, which is more than three times longer than our fastest simulated reconstruction time.

The other manifestation of the problem of using a single disk service rate is their predictions for the benefits of the redirection of reads and piggybacking of writes optimizations. Because redirecting work to the replacement disk in their model does not increase this disk's average access time, their predictions for the user-writes algorithm are more pessimistic than for their other algorithms.

9. Conclusions and future work

In this paper we have demonstrated that parity declustering, a strategy for allocating parity in a single-failure-correcting redundant disk array that trades increased parity overhead for reduced user-performance degradation during on-line failure recovery, can be effectively implemented in array-controlling software. We have exploited the special characteristics of balanced incomplete and complete block designs to provide array configuration flexibility while meeting most of our proposed criteria for the "goodness" of a parity allocation. In particular, using block designs to map parity stripes onto a disk array insures that both the parity update load and the on-line reconstruction load is balanced over all (surviving) disks in the array. This is achieved without performance penalty in a normal (non-recovering) user workload dominated by small (4 KB) disk accesses.

Previous work proposing declustered parity mappings for redundant disk arrays has suggested, without implementation, the use of block designs, has proposed two optimizations to a simple sweep reconstruction algorithm, and has analytically modeled the resultant expected reconstruction time [Muntz90]. In addition to extending their proposal to an implementation, our work evaluates their optimizations and reconstruction time models in the context of our software implementation running on a disk-accurate simulator. Our findings are strongly in agreement with theirs about the overall utility of parity declustering, but we disagree with their projections for expected reconstruction time and the value of their optimized reconstruction algorithms. Our disagreements largely result because of our more accurate models of magnetic-disk accesses. We find that their estimates for reconstruction time are significantly pessimistic and that their optimizations actually slow reconstruction in some cases. We also find that in practice, multiple, parallel reconstruction processes are necessary to attain fast reconstruction, although this additional load can degrade user response time substantially. Our most surprising result is that in an array with a low declustering ratio and parallel reconstruction processes, the simplest reconstruction algorithm produces the fastest reconstruction time because it best avoids disk seeks.

We feel that parity declustering and related performance/reliability tradeoffs is an area rich in research problems, a sample of which we give here. Reconstruction time could be further improved by making the parity unit size larger than the data unit size. This amounts to striping the data on a finer grain than the parity, and would allow reconstruction to proceed in larger units. We also have not spent much time on the data mapping in our declustered parity arrays; we think many of our layouts may be amenable to both our large write optimization and maximal parallelism criteria. With an eye to our performance modeling, we would like to see an analytical model of a disk array, or even of a single drive, that incorporates more of the complexity of disk accesses. We view analytical modeling as very useful for generating performance predictions in short periods of time, but this paper has demonstrated that care must be taken in their formulation. To broaden our results, we intend to explore disk arrays with different stripe unit sizes and user workload characteristics. One important concern that neither our nor Muntz and Lui's work considers is the impact of CPU overhead and architectural bottlenecks in the reconstructing system [Chervenak91]. For greater control of the reconstruction process we intend to implement throttling of reconstruction and/or user workload as well as a flexible prioritization scheme that reduces user response time degradation without starving reconstruction. Finally, we hope to install our software implementation of parity declustering on an experimental, high-performance redundant disk array and measure its performance directly.

Acknowledgments

We're indebted to Ed Lee and the Berkeley RAID project for supplying us with the raidSim simulator, and to Peter Chen, Art Rudeseal, Dan Siewiorek, Dan Stodolsky, Terri Watson, and John Wilkes for their insightful reviews of early drafts of the paper. Bob Barker, Karl Brace, Derek Feltham, Anurag Gupta, John Hagerman, Augustine Kuo, Jason Lee, Kevin Lucas, Tom Marchok, Lily Mummert, Howard Read, Roland Schaefer, and Terri Watson graciously supplied compute cycles on their workstations when time was short, and we would never have met our deadlines without them.

References

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

[Chen90a] P. Chen, et. al., "An Evaluation of Redundant Arrays of Disks using an Amdahl 5890," *Proceedings of the ACM SIGMETRICS Conference*, 1990, pp. 74-85.

[Chen90b] P. Chen and D. Patterson, "Maximizing Performance in a Striped Disk Array", *Proceedings of ACM SIGARCH Conference*, 1990, pp. 322-331.

[Chervenak91] A. Chervenak and R. Katz, "Performance of a RAID Prototype," *Proceedings of the ACM SIGMETRICS Conference*, 1991, pp. 188-197.

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

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

[Geist87] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems*, Vol. 5(1), 1987, pp. 77-92.

[Gibson91] G. Gibson, "Redundant Disk Arrays: Reliable, Parallel Secondary Storage," Ph.D. Dissertation, University of California, UCB/CSD 91/613, 1991. Also to be published by MIT Press.

[Gibson93] G. Gibson and D. Patterson, "Designing Disk Arrays for High Data Reliability," *Journal of Parallel and Distributed Computing*, to appear in 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 16th Conference on Very Large Data Bases*, 1990, pp. 148-160.

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

[Holland92] M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays", Carnegie Mellon University School of Computer Science Technical Report CMU-CS-92-130, 1992.

[Hsiao90] H.-I. Hsiao and D. DeWitt, "Chained Declustering: A New Availability Strategy for MultiProcessor Database Machines," *Proceedings of the 6th International Data Engineering Conference*, 1990, pp. 18-28.

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

[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 ASPLOS-IV*, 1991, pp. 190-199.

[Livny87] M. Livny, S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms," *Proceedings of the ACM SIGMETRICS Conference*, 1987, pp. 69-77.

[Meador89] W. Meador, "Disk Array Systems," *Proceedings of COMPCON*, 1989, pp. 143-146.

[Menon92] 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.

[Muntz90] R. Muntz and J. Lui, "Performance Analysis of Disk Arrays Under Failure," *Proceedings of the 16th Conference on Very Large Data Bases*, 1990, pp. 162-173.

[Ousterhout88] J. Ousterhout, et. al., "The Sprite Network Operating System," *IEEE Computer*, February 1988, pp. 23-36.

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

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

[Reddy91] A. Reddy and P. Bannerjee, "Gracefully Degradable Disk Arrays," *Proceedings of FTCS-21*, 1991, pp. 401-408.

[Rosenblum91] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991, pp. 1-15.

[Rudeseal92] A. Rudeseal, private communication, 1992.

[Salem86] K. Salem, H. Garcia-Molina, "Disk Striping", *Proceedings of the 2nd IEEE Conference on Data Engineering*, 1986.

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

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