On-Line Data Reconstruction In Redundant Disk Arrays

A dissertation submitted to the Department of Electrical and Computer Engineering, Carnegie Mellon University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

by

Mark Calvin Holland

Abstract

There exists a wide variety of applications in which data availability must be continuous, that is, where the system is never taken off-line and any interruption in the accessibility of stored data causes significant disruption in the service provided by the application. Examples include on-line transaction processing systems such as airline reservation systems and automated teller networks in banking systems. In addition, there exist many applications for which a high degree of data availability is important, but continuous operation is not required. An example is a research and development environment, where access to a centrally-stored CAD system is often necessary to make progress on a design project. These applications and many others mandate both high performance and high availability from their storage subsystems.

Redundant disk arrays are systems in which a high level of I/O performance is obtained by grouping together a large number of small disks, rather than building one large, expensive drive. The high component count of such systems leads to unacceptably high rates of data loss due to component failure, and so they typically incorporate redundancy to achieve fault tolerance. This redundancy takes one of two forms: replication or encoding. In replication, the system maintains one or more duplicate copies of all data. In the encoding approach, the system maintains an error-correcting code (ECC) computed over the data. The latter category of systems is very attractive because it offers both low cost per megabyte and high data reliability, but unfortunately such systems exhibit very poor performance in the presence of a disk failure. This dissertation addresses the design of ECC-based redundant disk arrays that offer dramatically higher levels of performance in the presence of failure than systems comprising the current state of the art, without significantly affecting the performance, cost, or reliability of these systems.

The first aspect of the problem considered here is the organization of data and redundant information in the array. The dissertation demonstrates techniques for distributing the workload induced by a disk failure across a large set of disks, thereby reducing the impact of the failure recovery process on the system as a whole.

Once the organization of data and redundancy has been specified, additional improvements in performance during failure recovery can be obtained through the careful design of the algorithms used to recover lost data from redundant information. The dissertation shows that structuring the recovery algorithm so as to assign one recovery process to each disk in the array, as opposed to the traditional approach of structuring it so as to assign a process to each unit of in a set of data units to be concurrently recovered, provides significant advantages.

Finally, the dissertation develops a design for a redundant disk array targeted at extremely high availability through extremely fast failure recovery. This development also demonstrates the generality of the techniques presented here.

Acknowledgments

First and foremost thanks of course go to my parents, Robert and Esther Holland. Their support has been unconditional and unwavering, but they've given me more than that. The really significant thing Mom and Dad did for me was to show me that education is the primary road to a better and more meaningful life. One is enriched by each new level of understanding that one acquires, irrespective of traditional boundaries between domains of knowledge. For this lesson, I'm more grateful to them than I can say. I love you both.

My advisor, Dan Siewiorek, gave me the freedom to pursue my own academic interests, and supported me even when my work did not exactly coincide with his own research agenda. This involved a great deal of extra effort on his part, and his willingness to make sure I succeeded didn't go unnoticed. It was he who initially suggested the topics for both my Master's and Ph.D. I'm very pleased to have had the opportunity to work with him, and my only regret is that the path my studies took did not allow us to work more closely.

This dissertation has grown out of long and fruitful discussions with Garth Gibson. Garth is one of the sharpest and most capable people I've ever worked with, and it was his encyclopedic knowledge of data storage technology, and where it is and should be going, that guided my studies from the start. This would be impressive even if it were all, but Garth and I were also able to establish a rare relationship based on confidence and communication that allowed our interaction to be pleasurable as well as productive. Thanks Garth.

Before I leave off thanking my advisors, one more point has to be made. Garth and Dan stood by me when a personal crisis caused me to shirk my studies for a while. This I appreciate more than anything else.

Bill Courtright, Hugo Patterson, and Dan Stodolsky all deserve thanks for contributing to my thesis through constant discussion, review, and technical assistance. In working with them I felt genuinely like a member of a team; like each of us made it a goal that we should all succeed. The three of you made it all a positive experience for me.

Stephanie Byram has put up with a lot from me lately, and I want her to know that I realize that nothing goes unnoticed. Thanks for tolerating, Steph. I'll be there when you need me.

Finally, I want to express my thanks to Yale Patt, now with the University of Michigan at Ann Arbor. About nine years ago, Yale took a chance and gave some responsibility to an undergraduate student who lacked confidence and was uncertain of his abilities. I firmly believe that the opportunities that arose from his support have led to my every success since then. The rare and beautiful thing Yale did for me was to trust me in the absence of any compelling reason to do so. I really hope, Yale, that you continue to extend your confidence to others at the risk of getting burned.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This dissertation provides techniques for designing data storage subsystems that are *highly available*, which we define as systems that tolerate component failures in order to maximize the probability that all stored data is available for retrieval, with maximum performance, at all times. There exists a wide variety of applications in which data availability must be continuous. These systems are never taken off-line and any interruption in the accessibility of stored data causes significant disruption in the service provided by the application. Examples include

- airline reservation systems, where the non-availability of booking information can lead to flight delays and/or revenue loss,
- database servers for point-of-sale terminal systems, where inventory, distribution, and pricing control systems all rely on the central collection of information from each sale,
- file servers that support a large number of clients with differing work schedules, and
- automated teller networks in banking systems, where the accessibility of funds relies on the accessibility of account information.

In addition, there exist many applications for which a high degree of data availability is important, but continuous operation is not required. An example is a research and development environment, where access to a centrally-stored CAD system is necessary to make progress on a design project. In all of these applications, the availability of data stored on a computer is crucial to the function of the parent organization.

The performance-related definition of availability described above differs slightly from the standard usage, which defines availability strictly as the probability that a system is operational at a particular time instant [Siewiorek92, p. 4]. However, the performance aspect of availability is critical in the data storage arena, because typical storage subsystem experience severe performance degradation in the presence of failure. Note that in the above-described applications, extended periods of unacceptable performance are tantamount to data non-availability.

A primary mechanism by which data becomes temporarily unavailable or irretrievably lost is disk failure in the data storage subsystem. Consequently, application areas such as those listed above demand not only the ability to recover from such failures 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 the users.

The necessity of condition (1) is clear; taking the system off-line to repair a failure results in an obvious breach of availability. Condition (2) is necessary for two reasons. First, the recovery process consumes input/output bandwidth in the storage subsystem, and so the system's users experience degraded performance during recovery from a component failure. It is necessary to minimize the duration of this degradation. Second, storage subsystems often tolerate only a single failure at time, and so minimizing the recovery time minimizes the probability of irretrievable data loss due to a second failure occurring before the first has been recovered. Condition (3) is necessary because performance degradation that is so severe as to render the system unusable is equivalent to data non-availability.

Performance and availability are not independent metrics. Because the failure recovery process consumes input/output bandwidth in the storage subsystem, it is necessary that the *fault-free* system load be kept low enough that the system performance will not be degraded to an unacceptable level by the extra input/output load induced to recover from a failure, should one occur. For example, if the load on a storage subsystem increases by 50% in the presence of a failure, then the fault-free system load must be kept below about 65% of its maximum in order to avoid throughput loss due to saturation should a failure occur. If, however, the system can be re-designed such that the load increase during failure recovery is only 25%, then the fault-free system load can be up to about 80% of its maximum without risking loss of throughput. For this reason, improved failure-recovery performance can translate directly into improved fault-free system performance.

Traditionally, continuous-operation systems implement fault-tolerance in the data storage subsystem by *disk mirroring*, that is, by replicating every block of user data on at least two disks and providing at least two independent access paths to each disk. The backup copy (or copies) of each data block are used to maintain availability in the presence of a failure (or set of failures) that renders a disk inaccessible. This approach is expensive because it incurs a storage capacity overhead for redundancy of at least 100%. Consequently, it is generally used only in relatively large-scale, high-cost systems where the need for availability justifies the expense. Smaller-scale applications such as file servers for local area networks often rely exclusively on periodic backup for fault-tolerance. These systems may therefore experience both irretrievable data loss and extended periods of data non-availability when component failures occur.

Over the past few years, *redundant disk arrays* have begun to supplant both mirrored and non-redundant storage in both continuous-operation and high-availability applications. These systems are commonly known as Redundant Arrays of Independent Disks (RAID) Levels 1 through 5, with each level defining a variation on the basic architecture. The different levels have different performance, reliability, and capacity overhead characteristics. Levels 2 through 5 achieve high availability at lower storage cost than mirroring, which is Level 1. Instead of duplicating every byte of file system data, Levels 2 through 5 use a portion of the physical data space of the disks comprising the array to store an error correcting code computed over the file system data. Since these arrays have lower capacity overhead than mirrored systems, there is substantial motivation to use them in all applications where data availability is important.

Unfortunately, the redundant disk arrays constituting the current state of the art exhibit poor performance during the process of failure recovery. In the disk array organization most appropriate to transaction-processing and database applications (RAID Level 5), the load on the surviving drives increases dramatically in the presence of a disk failure. This increase can be up to 100%, depending on the characteristics of the workload being serviced by the array. This requires that the fault-free system load be limited to about 50-60% of its maximum so that system throughput can be maintained in the presence of failure. In actuality the load must be even lower in order to maintain adequate system responsiveness; if a fault-free system is loaded at 60% of its maximum and it's failure-induced workload increase is 60%, the system will be saturated in the presence of failure, and hence the access response times as observed by the users will be essentially unbounded. This inability to handle component failure gracefully constitutes a major limitation in the applicability of redundant disk arrays to high-availability applications. Improving performance in the presence of failure is the topic of this dissertation.

This dissertation considers *user-observed response time* and *total failure recovery time* to be the primary figures of merit for failure-recovery performance. The former measures the time taken to complete a user's read or write request, and the analyses presented here always report both *average* and *90th percentile* values since most applications mandate a minimum level of responsiveness. The latter measures the total time taken to reconstruct and store on a replacement disk the entire contents of a failed disk. In general, this dissertation does not consider user throughput (total number of input/output operations executed per second) to be a figure of merit, because it assumes throughout that the storage subsystem must continue to service the full user-applied workload in the presence of failure. Under this assumption, no degree of throughput degradation during failure recovery is acceptable, and the systems considered here do not experience any. This is achieved by keeping the fault-free user workload sufficiently light that storage subsystem does not saturate in the presence of failure.

My thesis is that it is possible to construct redundant disk arrays that exhibit arbitrarily-small user-observed response-time degradation during failure recovery, that simultaneously minimize the duration of the recovery process, and that achieve this using a data capacity overhead for redundancy of less than 100%, which is the overhead required by a mirrored-disk system. In support of this thesis, the dissertation makes the following contributions:

- Demonstrates an implementation of *declustered parity*, a disk array architecture that allows for arbitrarily-small performance degradation during failure recovery, and develops variations on it that improve various aspects of performance during failure recovery,
- Demonstrates an efficient on-line data reconstruction algorithm, and then introduces and analyzes a number of modifications and optimizations that can be applied to it,
- Demonstrates that a system composed of the above two components delivers superior performance and availability when compared to existing disk array organizations, and
- Develops and evaluates a disk array organization that achieves extremely rapid failure recovery, on the order of 30 seconds per disk failure, by combining the above components with *distributed sparing*, which is a technique for allocating spare units

to disks in a disk array.

Chapter 2: Background Information

This chapter reviews the state of the art in redundant disk array technology, and then provides background information on the methodologies used in the rest of the dissertation. To place the work in context, Figure 2.1 loosely illustrates the storage hierarchy typical of computer systems, both general- and special-purpose. This dissertation focuses on the reliability and availability characteristics of the “Magnetic Disk” component, occasionally utilizing or making reference to “Main Memory” resources one level up.

Resource	Access Time	Size
Processor Registers	1 - 10 ns	0.1 KB
Cache Memory	10 - 100 ns	1 - 1000 KB
Main Memory	100 - 1000 ns	1 - 1000 MB
Magnetic Disk	5 - 50 ms	1 - 1000 GB
Off-Line Storage	0.5 - 15 minutes	Up to 100s of TB

Figure 2.1: The ubiquitous storage hierarchy of computer systems.

Because of the trade-off between access time and size, the storage elements in computer systems are organized hierarchically, with the fastest-but-smallest elements closest to the processor, and the largest-but-slowest elements farther away. This maximizes throughput by allowing the most commonly accessed data to be held in the fastest storage. Note the large gap between the access time of main memory and that of magnetic disk; this will be a primary motivating factor for the descriptions of storage subsystem architectures that follow.

The remainder of this chapter is organized as follows. The first section motivates the work in this dissertation by describing trends in the computer industry that are causing storage subsystems to be composed of increasing numbers of disks. The second section provides background information on disk and redundant disk array technology. The third section describes the methodology used throughout this thesis for evaluating the performance of the proposed techniques. It also identifies the workloads that are of primary interest; specifically, it describes the workload characteristics of transaction processing environments, where data availability is crucial.

2.1. The need for improved availability in the storage subsystem

There exist several trends in the computer industry that are driving the design of storage subsystems toward higher levels of parallelism. This means that current and future systems will achieve better I/O performance by increasing the number, rather than the performance, of the individual disks used [Patterson88, Gibson92]. This distinction is important in that, as will be seen, it implies directly the need for improved data availability. This section briefly describes these trends (Sections 2.1.1 through 2.1.3), and shows why they lead to the need for improved availability in the storage subsystem (Section 2.1.4).

2.1.1. The widening access gap

First and foremost, processors are increasing in performance at a much faster rate than disks. Microprocessors are increasing in computational power at between 25 and 30% per year [Myers86, Gelsinger89], and projections for future performance increases range even higher. Gelsinger et. al. [Gelsinger89] predicts that the huge transistor budgets projected for microprocessors in the 1990s will allow on-chip multiprocessing, yielding a further 20% annual growth rate for microprocessors. Bell [Bell89] projects supercomputer growth rates of about 150% per year.

Disk drives, by way of contrast, have been increasing in performance at a much slower rate. Comparing the state of the art in 1981 [Harker81] to that in 1993 [Wood93] shows that the average seek time[1] for a disk drive improved from about 16 ms to about 10 ms, rotational latency from about 8.5 ms to about 5 ms, and data transfer rate from about 3 MB/sec (which was achieved only in the largest and most expensive disks) to about 5 MB/sec. Combining these, the time taken to perform an average 8 KB access improved from 27.1 ms to 15.0 ms, or by about 45%, in the twelve-year period. This corresponds to an annual rate of improvement of less than 5%.

Increased processor performance leads directly to increased demand for I/O bandwidth [Gibson92, Kung86, Patterson88]. Since disk technology is not keeping pace with processor technology, it is necessary to use parallelism in the storage subsystem to meet the increasing demands for I/O bandwidth. This has been, and continues to be, the primary motivation behind disk array technology.

1. Seek time, rotational latency, and transfer rate are defined in Section 2.2.1.

2.1.2. The downsizing trend in disk drives

Prior to the early 1980s, storage technology was driven by the large-diameter (14 inch) drives [IBM3380, IBM3390] used by mainframes in large-scale computing environments such as banks, insurance companies, and airlines. These were the only drives that offered sufficient capacity to meet the requirements of these applications [Wood93]. This changed dramatically with the growth of the personal computer market. The enormous demand for small form-factor, relatively inexpensive disks produced an industry trend toward *downsizing*, which is defined as the technique of re-implementing existing disk drive technology in smaller form factors. This trend was enabled primarily by the rapid increase in storage density achieved during this period, which allowed the capacity of small-form factor drives to increase from a few tens of megabytes when first introduced to over 2 gigabytes today [IBM0664]. It was also facilitated by the rapid growth in VLSI integration levels during this period, which allowed increasingly sophisticated drive control electronics to be implemented in smaller packages. Further impetus for this trend derived from the fact that smaller form-factor drives have several inherent advantages over large disks:

- smaller disk platters and smaller, lighter disk arms yield faster seek operations,
- less mass on each disk platter allows faster rotation,
- smaller platters can be made smoother, allowing the heads to fly lower, which improves storage density,
- lower overall power consumption reduces noise problems.

These advantages, coupled with very aggressive development efforts necessitated by the highly competitive personal computer market, have caused the gradual demise of the larger drives. In 1994, the best price/performance ratio is achieved using 3 1/2 inch disks, and the 14 inch form factor has all but disappeared. The trend is toward even smaller form factors: 2 1/2 inch drives are common in laptop computers [ST9096], and 1.3 inch drives are available [HPC3013]. One-inch diameter disks should appear on the market by 1995, and should be common by about 1998. At a (conservative) projected recording density in excess of 1-2 GB per square inch [Wood93], one such disk should hold well over 2 GB of data.

These tiny disks will enable very large scale arrays. For example, a one-inch disk might be fabricated for surface-mount, rather than using cables for interconnection as is currently the norm, and thus a single printed circuit board could easily hold an 80-disk array. Several such boards could be mounted in a single rack to produce an array containing on the order of 250 disks. Such an array would store at least 500 GB, and even if disk performance does not improve at all between now and 1998, could service either 12,500 concurrent I/O operations or deliver 1.25 GB per second aggregate bandwidth. The entire system (disks, controller hardware, power supplies, etc.) would fit in a volume the size of a filing cabinet.

To summarize, the inherent advantages of small disks, coupled with their ability to provide very high I/O performance through disk array technology, leads to the conclusion that storage subsystems are, and will continue to be, constructed from a large number of small disks, rather than from a small number of powerful disks. Many trends in the storage industry substantiate this claim. For example, Montgomery Securities predicts that the redundant disk array market will exceed seven billion dollars by 1994 [Jones91]. Storage Technology Corporation, traditionally a maker of large-form-factor IBM-compatible disk drives, has stopped developing disks altogether, and is replacing this product line by one based on disk arrays [Rudeseal92].

2.1.3. The advent of new, I/O intensive applications

Finally, increases in on-line storage capacity and commensurate decreases in cost per megabyte enable new technologies that demand even higher levels of I/O performance. The most visible example of this is in the emergence of digital audio and video applications such as video-on-demand [Rangan93]. Others include scientific visualization, and large-object servers such as spatial databases [McKeown83, Stonebraker92]. These applications are all characterized by the fact that, if implemented on a large scale, their demands for storage and I/O bandwidth will far exceed the ability of current data storage subsystems to supply them. These applications will drive storage technologies by consuming as much capacity and bandwidth as can be supplied, and hence necessitate higher levels of parallelism in storage subsystems.

2.1.4. Why these trends necessitate higher availability

The preceding discussion demonstrated that higher degrees of I/O parallelism (an increased number of disks in a storage subsystem) are increasingly necessary to meet the storage demands of current and future systems. The discussion deliberately avoided identifying the specific organizations to be used in future storage systems, but made the case that such systems will be comprised of a relatively large number of independent disks. However, constructing a storage subsystem from a large number of disks has one significant drawback: the reliability of such a system will be worse than that of a system constructed from a small number of disks, because the disk array has a much higher component count. This subsection describes this problem, the solutions to it, and then motivates the work described in the rest of this dissertation by showing that the existing solutions do not adequately solve the problem.

As the number of disks comprising a system increases, the reliability of that system falls. Specifically, assuming the failure rates for a set of disks to be identical, independent, exponentially-distributed random variables, a simple reliability calculation shows that the mean time to data loss for a group of N disks is only $1/N$ times as long as that of a single disk [Patterson88]. Gibson analyzed a set of disk lifetime data to investigate the accuracy of the assumptions behind this calculation, and found "reasonable evidence to indicate that the lifetimes of the more mature of these products can be modeled by an exponential distribution" [Gibson92, p. 113]. Working from this assumption, a 100-disk array comprised of disks with 300,000 hour mean time to failure (typical for current disks) will experience a failure every 3000 hours, or about once every 125 days. As disks get smaller and array sizes grow, the problem gets worse: a 600-disk array experiences a failure approximately once every three weeks.

Disk arrays typically incorporate some form of redundancy in order to protect against data loss when these failure occurs. This is generally achieved either by *disk mirroring* [Katzman77, Bitton88, Copeland89, Hsiao91], or by *parity encoding* [Arulpragasam80, Kim86, Park86, Patterson88, Gibson93]. 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) [Patterson88], a portion of the array's physical capacity is used to store an error correcting code computed over the data stored in the

2. Because of industrial interest in using the RAID acronym and because of their concerns about the restrictiveness of its "Inexpensive" component, RAID is often reported as an acronym for Redundant Arrays of Independent Disks [RAID93].

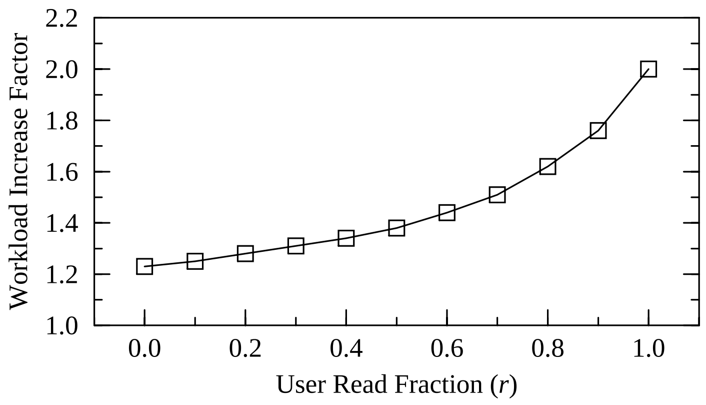


Figure 2.2: Failure-induced workload increase in RAID Level 5.

The figure shows the factor by which the surviving-disk workload increases when a disk fails in a RAID Level 5 array [Ng92a]. Note that the y-axis starts at 1.0.

array. Section 2.2.2 describes both of these approaches in detail. Studies have shown that, due to superior performance on small read and write operations, a mirrored array, also known as RAID Level 1, may deliver higher performance to many important workloads than can a parity-based array [Chen90a, Gray90]. Unfortunately, mirroring is substantially more expensive — its storage overhead for redundancy is 100%, whereas the overhead in a parity-encoded array is generally less than 25%, and may be less than 10%. Furthermore, several recent studies [Rosenblum91, Menon92a, Stodolsky93] demonstrated techniques that allow the small-write performance of parity-based arrays to approach and sometimes exceed that of mirroring.

The low redundancy overhead and potential high performance of parity-encoded arrays makes them very attractive in I/O intensive applications. However, their performance during the process of failure recovery is poor, and this limits their applicability to continuous-operation and other high-availability environments. Specifically, Ng and Mattson [Ng92a] derive the following equation for the workload increase factor on the surviving disks in a RAID Level 5 array containing a single failed disk:

$$\frac{Util_{faulty}}{Util_{faultfree}} = \frac{N}{N-1} + \frac{(N-2)\,r + (N-8)\,w}{(N-1)\,(r+4w)}$$

where r is the fraction of all user accesses that are reads, $w = 1 - r$, and N is the number of disks in the array. Figure 2.2 plots this function for a 40-disk array.

Since typical OLTP workloads are read-dominated [Ramakrishnan92], the load on the surviving disks increases by typically between 50-100% in the presence of a disk failure. This severely degrades the performance as observed by the users, and dramatically lengthens the period of time required to recover the lost data and store it on a replacement drive. This inability to achieve rapid recovery and high performance in the presence of failure constitutes the “failure recovery problem” in parity-encoded redundant disk arrays. It points directly to the need to improve upon the existing disk array organizations to achieve higher degrees of availability.

This dissertation demonstrates techniques that significantly improve the performance of parity-encoded disk arrays in the presence of disk failure. The goals are to simultaneously minimize both the duration of the failure recovery process, and its impact on the performance of the array as observed by the users. This allows the construction of low-cost arrays with high performance and high degrees of availability, for use in any application where interruptions in the accessibility of data interfere with the smooth operation of the organization.

2.2. Technology background

This section describes the structure and organization of modern disk drives and disk arrays. Subsequent chapters assume a reasonably intimate knowledge of this material. The section on disk technology has been kept to a minimum; it covers only those aspects of disk drive structure and terminology that are utilized and referred to in the remainder of the dissertation. Product manuals such as Digital Equipment Corporation’s *Mass Storage Handbook* [DEC86] provide more thorough descriptions of disk drive technology. This section describes disk array structure and functionality in more detail, because this information is essential to understanding the availability techniques to be developed.

2.2.1. Disk technology

Figure 2.3 shows the primary components of a typical disk drive. A disk consists of a stack of platters coated with magnetic media, with data stored on all surfaces. The platters rotate on a common spindle at constant velocity past the read/write heads (one per surface), each of which is fixed on the end of a disk arm. The arms are connected to a com-

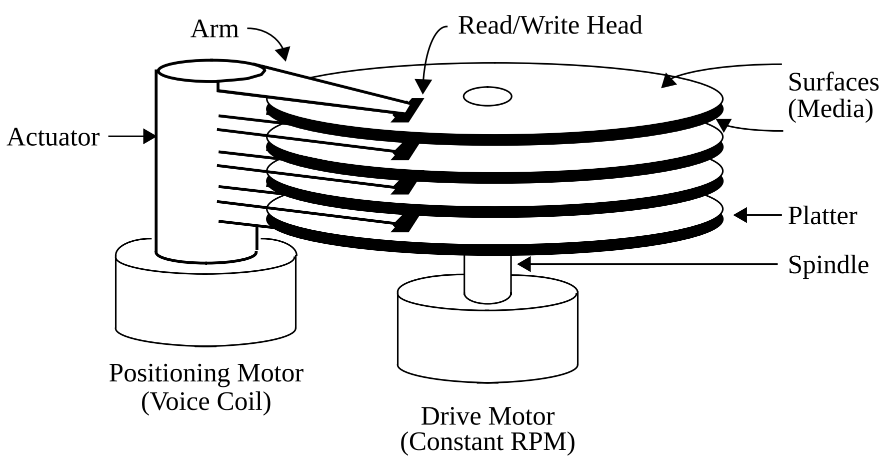


Figure 2.3: Physical components of a disk drive.

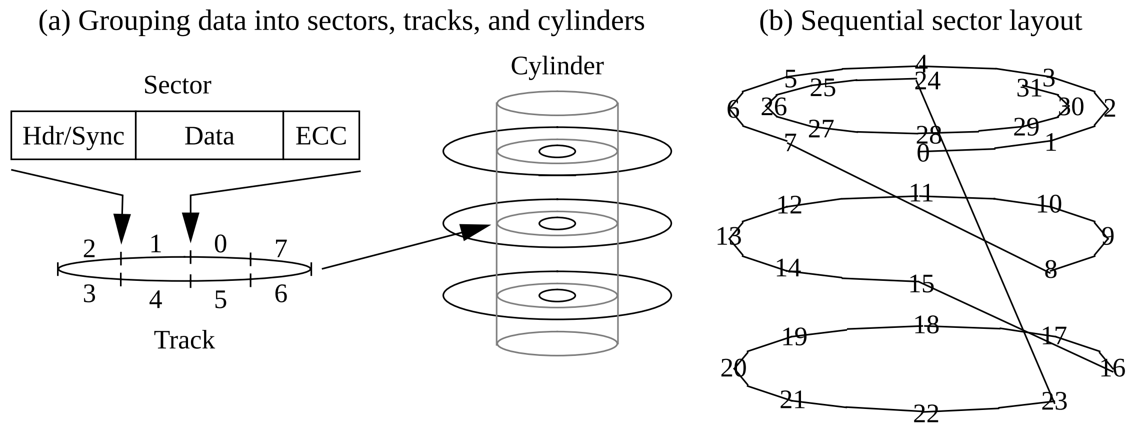


Figure 2.4: Data layout on a disk drive.

mon shaft called an actuator. Applying a directional current to a positioning motor causes the actuator to rotate small distances in either direction. Rotating the actuator causes the disk heads to move, in unison, radially along the platters, thereby allowing access to a band spanning most of the coated surface of each platter.

Figure 2.4 illustrates how data is typically organized on a disk. Part (a) shows how a block of sequential user data (almost always 512 bytes) is collected together and stored in a *sector*. A sector is the minimum-sized unit that can be read from or written to a disk drive. A header area in front of each sector contains sector identification and clock synchronization information, and a trailer area contains an error correcting code computed over the header and data. The set of sectors on a single surface at constant radial distance from the spindle is called a *track*, and the set of all tracks at constant radial offset is called a *cylinder.* At current densities, a typical 3 1/2 inch disk has 50-100 sectors per track, 1000-3000 cylinders, and 4-20 surfaces.

In order to access a block of data, the drive control electronics moves the actuator to position the disk heads over the correct cylinder, waits for the desired data to rotate under the heads, and then reads or writes the indicated sectors. Moving the actuator is called *seeking*, and takes 1-20 ms depending on the seek distance. Current disks rotate at between 3600 and 7200 RPM, making the expected rotational latency (one half of one revolution) between 4.2 and 8.3 ms. Thus for each access the disk must first *seek* to the indicated cylinder and then *rotate* to the start of the requested data. The combination of these two operations is referred to as *positioning* the disk heads.

If a user access requests a full track's worth of data, the rotational latency can be eliminated by reading or writing the data in the order that the requested sectors pass under the heads, rather than waiting until the first sector rotates under the heads to commence the operation. This is called *zero latency* operation or *full-track I/O,* and can be extended to include the case where the access spans only part of a track.

Note that the tracks near the outside of each surface have greater circumference than those near the spindle. A technique called *zoned bit recording (ZBR),* takes advantage of this and stores more sectors per track in the outer cylinders. This approach groups sets of 50-200 adjacent cylinders into zones, with the number of sectors per track being constant within each zone, but successively larger in the outer zones than the inner.

Figure 2.4b illustrates the assignment of sequential data to sectors, tracks, and cylinders. Nearly all disks read or write only one head at time, that is, they do not access multiple heads in parallel,[3] and so sequential user data is sequential in any given sector. Thus, as shown in the figure, sequential data starts at sector zero, proceeds around to the end of the track, moves to the next track (which is actually on the underside of the first platter),

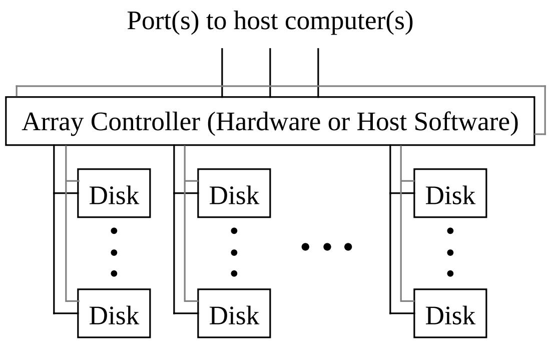
3. This is because the disk heads cannot be positioned independently, and thermal variations in the rigidity of the actuator, platters, and spindle make it difficult or impossible to keep all the disk heads simultaneously positioned over their respective tracks. There do exist a few disks that access multiple heads in parallel by careful management of head alignment [Fujitsu2360], but these are not commodity products and typically have lower density and higher cost per megabyte than standard disks.

continues this way to the end of the cylinder, and then moves to the next cylinder and starts again. Note that in this example a rotational distance equal to one sector is skipped upon crossing a track boundary (moving from sector 7 to 8), and two sectors are skipped upon crossing a cylinder boundary (moving from sector 23 to 24). These gaps are called the *track skew* and *cylinder skew.* The data is laid out in this manner to assure that the drive control electronics will have time to reposition the actuator when a user access spans a track or cylinder boundary. The track skew is shorter than the cylinder skew because only fine adjustments are necessary when switching to a new track within one cylinder, whereas switching to a new cylinder requires the actuator to be moved one full cylinder width and then fine-adjusted over the new track. Typical values for track and cylinder skew in current technology are about 0.5 and 1.5 ms, respectively.

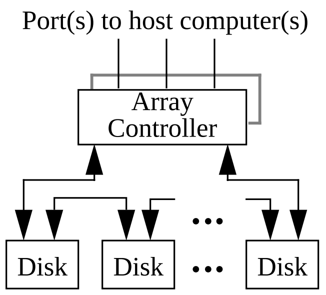
The interface electronics in a disk drive typically contain a buffer memory, varying in size from about 32 KB to about 1 MB, which serves two purposes. First, several disks may share a single path to the CPU, and the memory serves to speed-match the disks to the bus. In order to avoid holding the bus for long periods of time, a disk will typically read data into the buffer and then burst-transfer it to the CPU. The buffer serves the same purpose on a write operation: the CPU burst-transfers the data to the drive's buffer, and the drive writes it to the media at its own rate. Reading and writing to and from the buffer, instead of directly between the media and the bus, also eliminates *rotational position sensing (RPS) misses* [Buzen87], which occur in bufferless disks when the transfer path to the CPU is not available at the time the data arrives under the disk heads. The second purpose served by the buffer is as a cache memory [IBM0661, Maxtor89]. Applications typically access files sequentially, and so the disks comprising a storage subsystem typically observe a sequential access pattern as well. Thus after each read operation, the disk controller will continue to read sequential data from the media into the buffer. If the next block of requested data is sequential with respect to the previous block, the disk can often service it directly from the buffer instead of accessing the media. This yields both higher throughput and lower latency. Many disks generalize this *readahead* function so that the buffer becomes a full-fledged cache memory.

2.2.2. Disk array technology

This section describes the structure and operation of disk arrays in detail.


Port(s) to host computer(s)
Array Controller (Hardware or Host Software)
Disk
Disk
Disk
Disk
Disk
Disk

(a) Bus-connected



(b) High-bandwidth serial connected

Figure 2.5: Disk array architectures.

2.2.2.1. Disk array architecture

Figure 2.5 illustrates two possible disk array subsystem architectures. Today's systems use the architecture of Figure 2.5a, 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 the figure) so that they do not represent a single point of failure [Katzman77, Menon93]. The controller functionality can also be distributed amongst the disks of the array [Cao93].

As disks get smaller [Gibson92], the large cables used by SCSI and other bus interfaces become increasingly unattractive. The system sketched in Figure 2.5b offers an alternative. It uses high-bandwidth bidirectional 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. Further, by making each link bidirectional, it provides two paths to each disk without duplicating busses. 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 2.5b 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 thesis apply to all array controller implementations.

2.2.2.2. Defining the RAID levels: data layout and ECC

An array controller implements the abstraction of a *linear address space*. The array appears to the host as a linear sequence of data units, numbered 0 through $N \cdot B - 1$, where N is the number of disks in the array and B is the number of units of user data on a disk. Units holding ECC do not appear in the address space exported by the array controller; they are not addressable by the application program. The array controller translates addresses in this linear space into physical disk locations (disk identifiers and disk offsets) as it performs requested accesses. It is also responsible for performing the redundancy-maintenance accesses implied by application write operations. We refer to the mapping of an application's logical unit of stored data to physical disk locations and associated ECC locations as the disk array's *layout*.

Fundamental to all disk arrays is the concept of *striping* consecutive units of user data across the disks of the array [Kim86, Livny87, Patterson88, Gibson92, Merchant92b]. Striping is defined as breaking up the linear address space exported by the array controller into blocks of some size, and assigning the consecutive blocks to consecutive disks, rather than filling each disk with consecutive data before switching to the next. The *striping unit* (or *stripe unit*) [Chen90b] is the maximum amount of consecutive data assigned to a single disk. The array controller has the freedom to set the striping unit arbitrarily; the unit can be as small as a single bit or byte, or as large as an entire disk. Striping has two benefits: automatic load balancing in concurrent workloads, and high bandwidth for large sequential transfers by a single process.

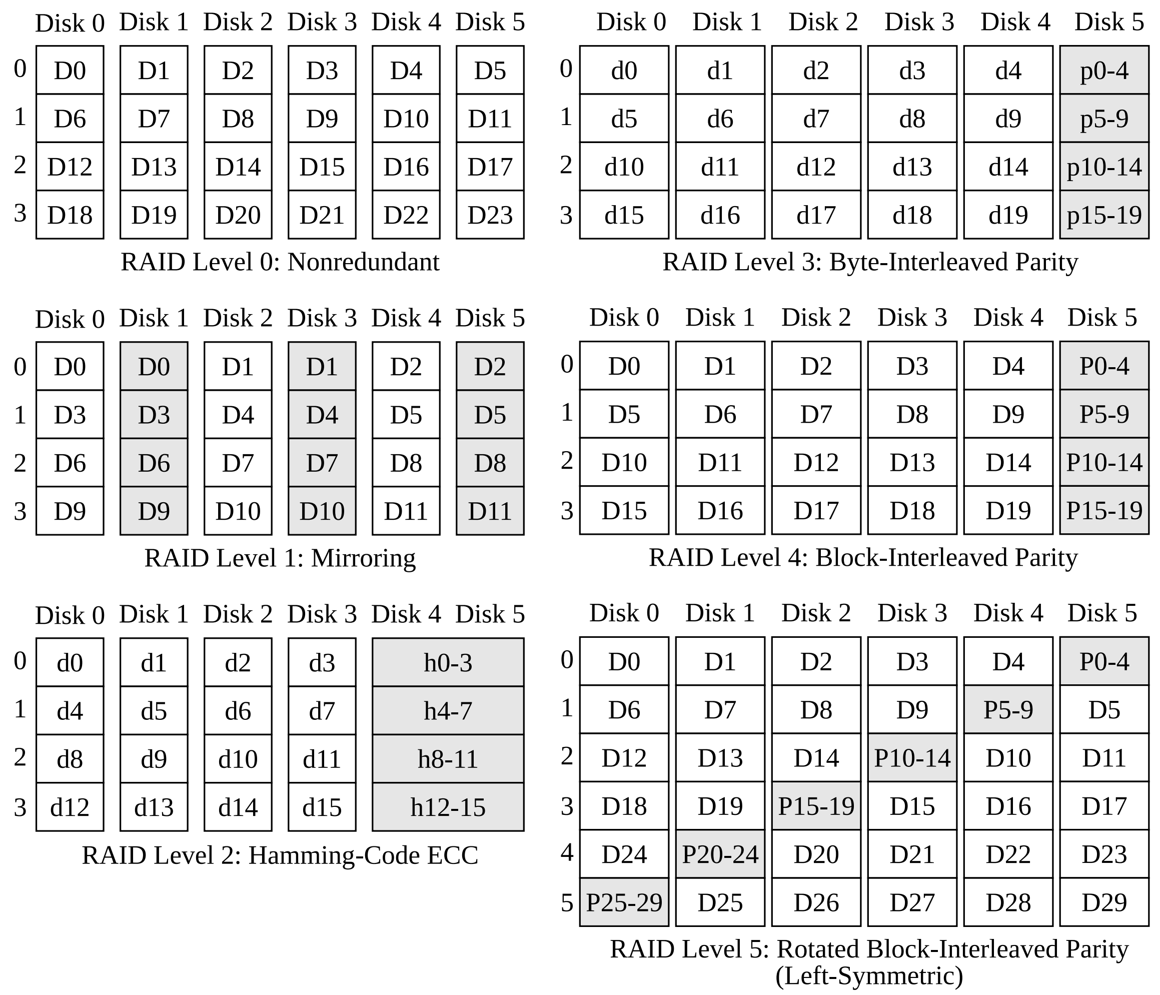
Disk arrays achieve load balance in concurrent workloads (those that have many processes concurrently accessing the stored data) by selecting the stripe unit to be large enough that most small accesses are serviced by a single disk. Such arrays typically select the striping unit to be larger than the expected access size, which causes a typical access to be serviced by only one disk. This allows the independent processes to perform small accesses concurrently in the array, and as long as the processes' access patterns are not pathologically regular with respect to the striping unit, it assures that the load will be approximately evenly balanced over the disks. Thus, an N-disk coarse-grain striped array can service *N* I/O requests in parallel, but each of them occurs at the bandwidth of a single disk.

Arrays achieve high data rates in low-concurrency workloads by striping at a finer grain, for example, one byte or one sector. Such arrays are used when the expected workload is a single process requesting data in very large blocks. Fine-grain striping assures that each access uses all the disks in the array, which maximizes performance when the workload concurrency (number of processes) is one[4]. After the initial seek and rotational delay penalties associated with each access, a fine-grain-striped array transfers data to or from the CPU at *N* times the rate of a single disk. Therefore, a fine-grain striped array can service only one I/O at any one time, but is capable of reading or writing the data at a very high rate.

Patterson, Gibson, and Katz [Patterson88] classified redundant disk arrays into five types, called RAID Levels 1 through 5, based on the organization of redundant information and the layout of user data on the disks. This terminology has gained wide acceptance [RAID93], and is used throughout this dissertation. The term "RAID Level 0" has since entered common usage to indicate a non-redundant array. Figure 2.6 illustrates the layout of data and redundant information for the six RAID levels. The remainder of this section briefly introduces each of the levels, and subsequent sections provide additional details.

RAID Level 1, also called *mirroring* or *shadowing*, is the standard technique used to achieve fault-tolerance in traditional data storage subsystems [Katzman77, Bitton88]. The disks are grouped into mirror pairs, and one copy of each data block is stored on each of the disks in the pair. To unify the taxonomy, RAID Level 1 defines the user data to be block-striped across the mirror pairs, but traditional mirrored systems instead fill each disk with consecutive user data before switching to the next. This can be though of as setting the stripe unit to the size of one disk. RAID Level 1 is a highly reliable organization since the system can tolerate multiple disk failures (up to *N*/2) without losing data, so long as no two disks in a mirror pair fail. It can be generalized to provide multiple-failure tolerance

4. Since the host views the array as one large disk, it never attempts to read or write less than one sector, and hence every user access uses all the disks in the array. Note that one sector is the minimum unit that can be read from or written to an individual disk, and so a fine-grain striped array typically disallows accesses that are smaller than *N* times the size of one sector, where *N* is the number of disks in the array. This rarely poses a problem since fine-grain striped arrays are typically used in applications where the average request size is very large.



	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	D5
1	D6	D7	D8	D9	D10	D11
2	D12	D13	D14	D15	D16	D17
3	D18	D19	D20	D21	D22	D23

RAID Level 0: Nonredundant

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	d0	d1	d2	d3	d4	p0-4
1	d5	d6	d7	d8	d9	p5-9
2	d10	d11	d12	d13	d14	p10-14
3	d15	d16	d17	d18	d19	p15-19

RAID Level 3: Byte-Interleaved Parity

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D0	D1	D1	D2	D2
1	D3	D3	D4	D4	D5	D5
2	D6	D6	D7	D7	D8	D8
3	D9	D9	D10	D10	D11	D11

RAID Level 1: Mirroring

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	P0-4
1	D5	D6	D7	D8	D9	P5-9
2	D10	D11	D12	D13	D14	P10-14
3	D15	D16	D17	D18	D19	P15-19

RAID Level 4: Block-Interleaved Parity

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	d0	d1	d2	d3	h0-3	
1	d4	d5	d6	d7	h4-7	
2	d8	d9	d10	d11	h8-11	
3	d12	d13	d14	d15	h12-15	

RAID Level 2: Hamming-Code ECC

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	P0-4
1	D6	D7	D8	D9	P5-9	D5
2	D12	D13	D14	P10-14	D10	D11
3	D18	D19	P15-19	D15	D16	D17
4	D24	P20-24	D20	D21	D22	D23
5	P25-29	D25	D26	D27	D28	D29

RAID Level 5: Rotated Block-Interleaved Parity
(Left-Symmetric)

Figure 2.6: Data and redundancy organization in RAID Levels 0 through 5.

The figure shows the first few units on each disk in each of the RAID levels. "D" represents a block of user data (of unspecified size, but some multiple of one sector), "d' a bit or byte of user data, "hx-y" a Hamming code computed over user data bits/bytes x through y, "px-y" a parity (exclusive-or) bit/byte computed over data blocks x through y, and "Px-y" a parity block over user data blocks x through y. Note from these definitions that the number of bytes represented by each individual box and label in the above diagrams varies with the RAID level. The numbers on the left indicate the offset into the disk, expressed in stripe units. Shaded blocks represent redundant information, and non-shaded blocks represent user data.

Level 0 is non-redundant, and therefore not fault-tolerant. Level 1 is simple mirroring, in which two copies of each data block are maintained. Level 2 uses a Hamming error-correction code to achieve fault-tolerance at a lower capacity overhead than Level 1. *Levels 3 through 5 exploit the fact that failed disks are self-identifying. Thus Levels 3 through 5 achieve fault tolerance using a simple parity (exclusive-or) code, lowering the capacity overhead to only one disk out of 6 in this example. Levels 3 and 4 are distinguished only by the size of the striping unit: one bit or one byte in Level 3, and one block in Level 4. In Level 5, the parity blocks rotates through the array rather than being concentrated on a single disk, to avoid throughput loss due to contention for the parity drive.*

by maintaining more than two copies of each data unit. Its drawback is that its cost per megabyte of storage is at least double that of RAID Level 0.

RAID Level 2 provides high availability at lower cost per megabyte by utilizing well-known techniques used to protect main memory against transient data loss. The disks comprising the array are divided into *data disks* and *check disks*. User data is bit- or byte-striped across the data disks, and the check disks hold a Hamming error correcting code [Peterson72, Gibson92] computed over the data in the corresponding bits or bytes on the data disks. This reduces the storage overhead for redundancy from 100% in mirroring to a value in the approximate range of 25-40% (depending on the number of data disks) in RAID Level 2, but reduces the number of failures that can be tolerated without data loss. As will be seen, the reliability and performance of such a system can still be very high. It can be extended to support multiple failure toleration by using an *n*-failure-tolerating Hamming code, which of course increases the capacity overhead for redundancy and the computational overhead for computing the codes.

Thinking Machines Corporation's Data Vault storage subsystem [TMC87] employed RAID Level 2, but this organization ignores an important fact about failure modes in disk drives. Since disks contain extensive error detection and correction functionality, and since they communicate with the outside world via complex protocols, the array controller can directly identify failed disks from their status information, or by their failure to adhere to the communications protocol. A system in which failed components are *self-identifying* is called an *erasure channel*, to distinguish it from an *error channel*, in which the locations of the errors are not known. An *n*-failure detecting code for an error channel becomes an *n*-failure correcting code when applied to an erasure channel [Gibson89, Peterson72]. RAID Level 3 takes advantage of this fact to reduce the storage overhead for redundancy still further.

In RAID Level 3, user data is bit- or byte-striped across the data disks, and a simple parity code is used to protect against data loss. A single check disk (called the *parity disk*) stores the parity (cumulative exclusive-or) over the corresponding bits on the data disks. This reduces the capacity overhead for redundancy to 1/*N*. When the controller identifies a disk as failed, it can recover any unit of lost data by reading the corresponding units from all the surviving disks, including the parity disk, and XORing them together. To see this, assume that disk 2 in the RAID Level 3 diagram within Figure 2.6 has failed, and note that

$$(p_{0-4} = d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4) \Rightarrow (d_2 = d_0 \oplus d_1 \oplus p_{0-4} \oplus d_3 \oplus d_4)$$

Multiple failure tolerance can be achieved in RAID Level 3 by using more than one check disk, and a more complex error-detecting/correcting code such as a Reed-Solomon [Peterson72] or MDS code [Burkhard93, Blaum94]. RAID Level 3 has very low storage overhead and provides very high data transfer rates. Since user data is striped on a fine grain, each user access uses all the disks in the array, and hence only one access can be serviced at any one time. Thus this organization is best suited for applications such as scientific computation, in which a single process requests a large amount of sequential data from the array.

Since all accesses use all disks in RAID Level 3, the disk heads move in unison, and so the cylinder over which the heads are currently located is always the same for all disks in the array. This assures that the seek time for an access will be the same on all disks, which avoids the condition in which some disks are idle waiting for others to finish their portion of an access. In order to assure that rotational latency is also the same for each access on each disk, systems using RAID Level 3 typically use phase-locked loop circuitry to synchronize the rotation of the spindles of the disks comprising the array. Many disks currently on the market support this spindle synchronization.

RAID Level 4 is identical to Level 3 except that the striping unit is relatively coarse-grained (perhaps 32KB or larger [Chen90b]), rather than a single bit or byte. The block of parity that protects a set of data units is called a *parity unit*. A set of data units and their corresponding parity unit is called a *parity stripe*. RAID Level 4 is targeted at applications like on-line transaction processing (OLTP), in which a large number of independent processes concurrently request relatively small units of data from the array. Since the striping unit is large, the probability that a single small access will use more than one disk is low, and hence the array can service a large number of accesses concurrently. This organization is also effective at workloads that are predominantly small accesses, but contain some fraction of larger accesses. The array services concurrent small accesses in parallel, but achieves a high data rate on the occasional large access by utilizing many disk arms.

In RAID Level 4, each disk typically services a different access, and so unless the workload applied contains a significant fraction of large accesses, the heads do not remain synchronized. Consequently, there is no compelling reason to synchronize the spindles either. However, spindle synchronization never degrades performance, and can improve it on large accesses, disks arrays typically use it whenever the component disks support it.

The problem with RAID Level 4 is that the parity disk can be a bottleneck in workloads containing a significant fraction of small write operations. Each update to a unit of user data implies that the corresponding parity unit must be updated to reflect the change. Thus the parity disk sees one update operation for every update to every data disk, and its utilization due to write operations is *N*-1 times larger than that of the data disks. This does not occur in RAID Level 3, since every access uses every disk. To solve this problem, RAID Level 5 distributes the parity across the disks of the array. This assures that the parity-update workload is as well-balanced across the disks as the data-update workload.

In RAID Level 5, there are a variety of ways to lay out data and parity such that parity is evenly distributed over the disks [Lee91]. The structure shown in Figure 2.6 is called the *left-symmetric* organization, and is formed by first placing the parity units along the diagonal, and then placing the consecutive user data units on consecutive disks, at the lowest available offset on each disk. This method for assigning data units to disks assures that, if there are any accesses in the workload large enough to span many stripe units, the maximum possible number of disks will be used to service them. Chapter 3 clarifies the importance of this property.

RAID Levels 2 and 4 are of less interest than the others, because levels 3 and 5 provide better solutions, respectively. We omit Levels 2 and 4 from the remaining discussion.

2.2.2.3. Reading and writing data in the different RAID levels

This section describes the techniques used to read and write data in the different RAID levels, both when the array is fault-free (“fault-free mode”) and when it contains a single failed disk (“degraded mode”). The focus is on the techniques used to maintain parity, and to continue operation in the presence of failure. This section uses the terms “read throughput” and “write throughput” to indicate the maximum rates at which data can be read from or written to the array.

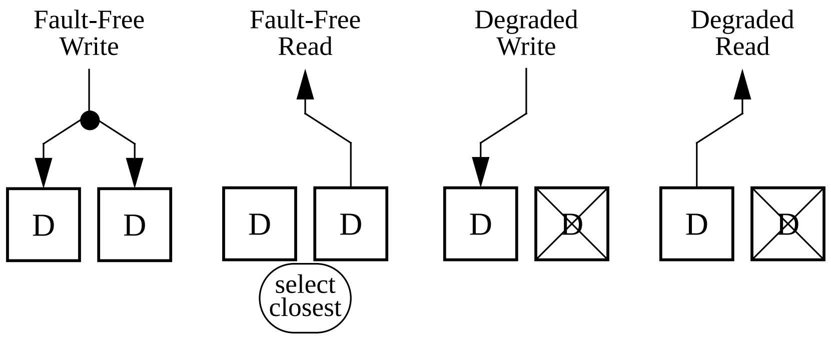


Figure 2.7: Read and write operations in RAID Level 1 (mirroring).

In all cases, the array controller maps the linear array address and access type supplied by the host (the “user” read or write) to the indicated set of operations on physical disks (the corresponding “disk” reads and/or writes). In RAID Level 0, the set of reads or writes so generated can be immediately and concurrently initiated, since there is no parity to maintain, and no possibility of continuing operation in the presence of failure. Thus the read throughput and write throughput of a RAID Level 0 array are both N times the throughput of a single disk. In Levels 1, 3, and 5, the disk operations triggered by a user read or write operation are more complex, especially in the presence of a disk failure, and often must be sequenced appropriately.

2.2.2.3.1. RAID Level 1

Figure 2.7 illustrates the different read and write operations in RAID Level 1. In fault-free mode, the controller must send user write operations to both disks. This reduces the maximum possible write throughput to 50% of that of RAID Level 0. The two write operations can, in general, occur concurrently, but some systems perform them sequentially in order to guarantee that the old data will be recoverable should the first write fail.

Typically, read requests are sent to only one of the two disks in the pair, so that the other will be free to service other read operations. The controller can service user reads in fault-free mode from either copy of the data. This flexibility allows the controller to improve throughput by selecting, for each user read operation, the disk that will incur the least positioning overhead [Bitton88, Bitton89]. This is frequently called the *shortest seek optimization*, and can improve read throughput by up to about 15% over RAID Level 0 [Chen90a].

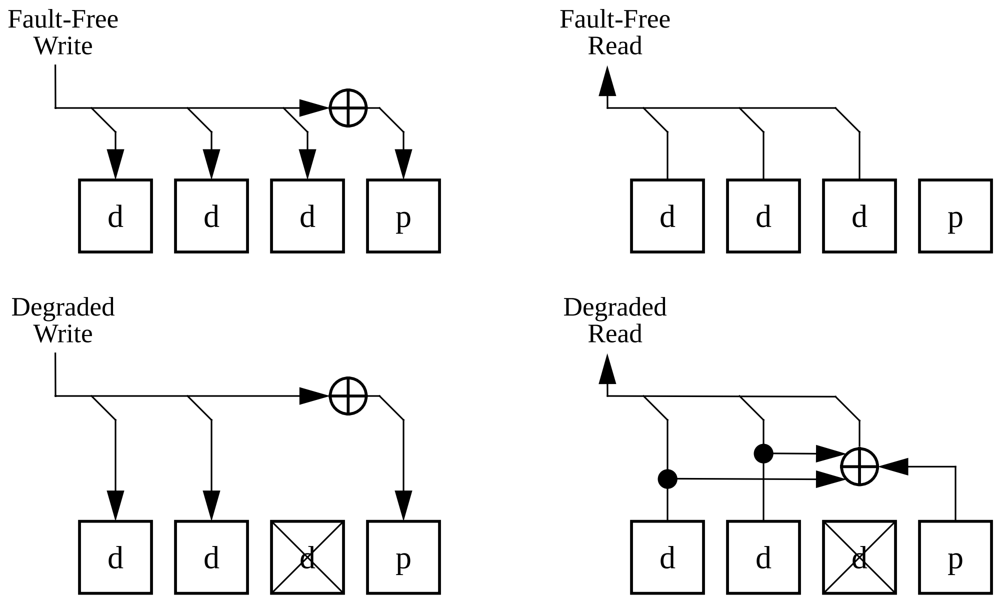


Figure 2.8: Read and write operations in RAID Level 3 (bit-interleaved parity).

The diagonal lines in the figure indicate that when the host accesses (reads or writes) a block of data consisting of bits 0 *through* n-1*, disk* 0 *services bits* 0, 3, 6, …, n-3*, disk* 1 *services bits* 1, 4, 7, …, n-2*, and disk 2 services bits* 2, 5, 8, …, n-1. *The array controller arranges for the correct bits to read from or written to the correct drive. On a write operation, the controller writes to disk* 3 *a block containing the following bits:* (0⊕1⊕2), (3⊕4⊕5), (6⊕7⊕8), …, ((n-3)⊕(n-2)⊕(n-1)). *Note that the controller implements this bit-level parity operation using only sector-sized accesses on the disks, and so* n *must be a multiple of* 8·N·S*, where* N *is the number of disks in the array, and* S *is the number of bytes in a sector. The controller typically enforces this condition, since the only alternative is to use read-modify-write operations on the individual disks, which drastically reduces efficiency.*

In degraded mode, the controller sends user write operations that target a unit with one copy on the failed disk only to the surviving disk in the pair, instead of to both. This does not affect the utilization on the surviving disk, because it does not absorb any write traffic that it would not otherwise encounter. However, in the presence of a disk failure, the surviving disk must absorb, in addition to its regular workload, all the read traffic targeted at the failed drive in fault-free mode. In read-intensive workloads, this can cause the utilization on the surviving disk to double. User reads and writes that do not target any units on the failed disk occur as if the array were fault-free.

2.2.2.3.2. RAID Level 3

Figure 2.8 illustrates reads and writes in RAID Level 3. The following discussion assumes that each user access is some multiple of *(N-1)·S* in size, where *N* is the number of disks in the array and *S* is the number of bytes in a sector (almost always 512). This is because each access uses all data disks, and the minimum sized unit that can be read from or written to a disk is one sector. If the array is to support accesses that are not a multiple of this size, the controller must handle any partial-sector updates via read-modify-write operations, which can degrade write performance.

In fault-free mode, user write operations update the old data in place. The controller updates the parity disk by computing the cumulative XOR of the data being written to each drive, and writing the result to the parity disk concurrently with the write of the user data to the data disks. The controller may perform this XOR operation before the write is initiated, or as the data flows down to the disks [Katz93]. Because the XOR happens at electronic speeds (a few microseconds per complete user access) but the disk runs at mechanical speeds (milliseconds per access), this computation typically has no measurable effect on the performance of the array. User read operations simply stream the data into the controller; the parity disk remains idle during this time.

A degraded-mode user write operation in RAID Level 3 occurs in exactly the same manner as in fault-free mode, except that the controller suppresses the write to the failed disk. A degraded-mode user read is serviced by reading the parity and the surviving data, and XORing them together to reconstruct the data on the failed drive. Disk arrays that stripe data on a fine grain (a bit or a byte) have the property that their performance in degraded mode is not significantly different than their performance in fault-free mode. This is because the controller accesses all disks during every access in any case, and so supporting degraded-mode operation simply amounts to modifying the bit streams sent to and from each drive. The XOR operations that occur in degraded mode are typically performed as the data streams into or out of the controller, and so they do not significantly increase access times.

2.2.2.3.3. RAID Level 5

Figure 2.9 illustrates the various translations of user accesses to disk accesses in RAID Level 5. User write operations in fault-free mode are handled in one of three ways, depending on the number of units being updated. In all cases, the update mechanisms are designed to guarantee the property that after the write completes, the parity unit holds the

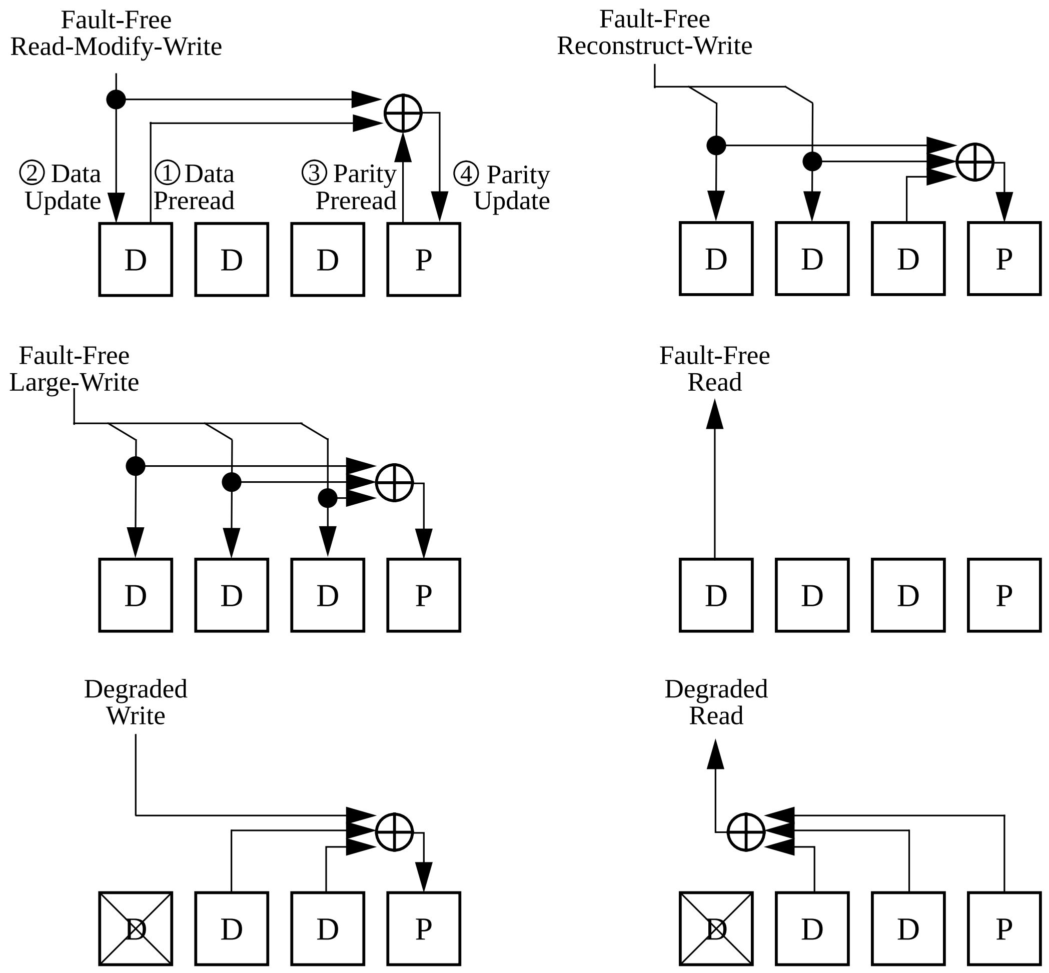


Figure 2.9: Read and write operations in RAID Level 5 (rotated parity).

cumulative XOR over the corresponding data units, or

$$P_{new} = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_{N-1}$$

If the update affects only one data unit, the prior content of that unit is read and XORed with the new data about to be written. This produces a map of the bit positions that need to be toggled in the parity unit in order that the parity unit should reflect the new data. These changes are applied to the parity unit by reading its old contents, XORing in the previously generated map, and writing the result back to the parity unit. The correctness of this transformation is shown as follows, where a new data block $D_{2,new}$ is being written to a unit on disk number 2 in an *N*-disk array:

$$P_{new} = P_{old} \oplus (D_{2,old} \oplus D_{2,new}) \Rightarrow$$
$$P_{new} = D_1 \oplus (D_{2,old} \oplus D_{2,old}) \oplus D_{2,new} \oplus D_3 \oplus \ldots \oplus D_N \Rightarrow$$
$$P_{new} = D_1 \oplus D_{2,new} \oplus D_3 \oplus \ldots \oplus D_N$$

This parity update operation is called a *read-modify-write*, and is easily generalized to the case where the user access targets more than one data unit. In this case, the controller reads the previous contents of all data units to be updated, and then XORs them together with the new data, prior to reading, XORing, and re-writing the parity unit. Read-modify-write updates are used for all fault-free user write operations in which the number of data units being updated is less than half the number of data units in a parity stripe.

The preread-and-then-write operation performed on the data unit is typically done atomically to minimize the positioning overhead incurred by the access [Stodolsky93]. This is also true for the parity unit. Since the old data must be available to perform the parity update, the data preread-and-write is typically allowed to complete (atomically) before the parity preread-and-write is started.

In applications that tend to read blocks of data shortly before writing them, the performance of the read-modify-write operation can be improved by acquiring the old contents of the data unit to be updated from the system's buffer cache, rather than reading it from disk. This reduces the number of disk operations required from four to three. This situation is very common in OLTP environments [TPCA89, Menon92c].

When the number of data units being updated exceeds half of one parity stripe, there is a more efficient mechanism for updating the parity. In this case, the controller writes the new data without pre-reading the old contents of the written unit, reads and XORs together all of the data units in the parity stripe that are *not* being updated, XORs in to this result each of the new data units to be written, and writes the result to the parity unit. The new parity that is written is therefore the cumulative XOR of the new data units and the data units not being updated, which is correct. This is called a *reconstruct-write* operation, because of its similarity to the way failed data is recovered.

The final mechanism used to update parity in a fault-free RAID Level 5 array is the degenerate case of the reconstruct-write that occurs when a user access updates all data units in a parity stripe. In this case, the controller does not need to read any old data, but instead simply updates each data unit in-place, and then XORs together all the new data units in buffer memory and writes the result to the parity unit. This is often called a *large write*, and is the most efficient form of update.

In degraded mode, a user read requesting data on the failed disk is serviced by reading all the units in the parity stripe, including the parity unit, and XORing them together to reconstruct the requested data unit(s). User reads that do not request data on the failed disk are serviced normally. User write requests updating data on the failed drive are serviced via reconstruct-writes, independently of the number of units being updated, with the write to the failed disk suppressed. Since the data cannot be written, this method of update causes the new data to be reflected in the parity, so that the next read will return the correct data. User write requests not updating data on the failed drive are serviced normally, except in the reconstruct-write case, where the parity needs to be read. When a user write request updates data for which the parity has failed, the data is simply written in-place, since no parity-maintenance operations are possible.

2.2.2.4. Comparing the performance of the RAID levels

Table 2.1, adapted from Patterson, Gibson, and Katz [Patterson88], compares the fault-free performance and capacity overhead of the RAID levels. The values are all first-order approximations, since there are a wide variety of effects related to seek distance, head synchronization, access patterns, etc., that influence performance, but the table provides a baseline comparison. It’s clear that RAID Level 1 offers better performance on concurrent, small-access workloads, but does so at a high cost in capacity overhead.

2.2.2.5. On-line reconstruction

The preceding has shown how a disk array operates, and how it may continue to operate in the presence of a single disk failure. The next step to take is that the array should have the ability *recover* from the failure, that is, restore itself to the fault-free state. Further, a disk array should be able to effect this recovery without taking the system off-line. This is implemented by maintaining one or more on-line spare disks in the array. When a

RAID Level	Large Accesses			Small Accesses			Capacity Overhead (%)	Max Concurrency
	Read	Write	RMW	Read	Write	RMW		
0	100	100	100	100	100	100	0	*N*
1	100+	50	66	100+	50	66	100	*N*
3	100	100	100	n/a	n/a	n/a	100/*N*	1
5	100	100	100	100	25	33	100/*N*	*N*

Table 2.1: First-order comparison between the RAID levels for an *N*-disk array.

The table reports performance numbers as percentages of RAID Level 0 performance. The "RMW" column gives the performance of the array when the application reads each data unit before writing it, which eliminates the need for the data preread. The capacity overheads are expressed as a percentage of the user data capacity of the array. The concurrency figures indicate the maximum number of user I/Os that can be simultaneously executed. The table reports the maximum concurrency numbers for Levels 1 *and* 5 as N *because such arrays can support* N *concurrent reads, but writes involve multiple I/O operations, and this reduces the maximum supportable concurrency.*

disk fails, the array switches to degraded mode as described above, but also invokes a *background reconstruction process* to recover from the failure. This process successively reconstructs the data and parity units that were lost when the disk failed, and stores them on the spare disk. The mechanism by which this is accomplished is called the *reconstruction algorithm*. Once all the units have been recovered, the array returns to normal performance and is once again single-failure tolerant, and so the recovery is complete. The primary topic of this thesis is the design of disk arrays and reconstruction algorithms that minimize both the duration and the user-performance impact of the reconstruction process.

2.2.2.6. Related work: variations on these organizations

This section summarizes industrial and academic research on disk arrays. It defines nine categories of investigation, and presents brief summaries of some papers in each. These studies do not relate specifically to the topic of availability, but rather serve as background in the area of redundant disk arrays. Subsequent chapters describe in detail prior studies that are specifically related to one of the topics covered in this dissertation.

2.2.2.6.1. Multiple failure toleration

Each of the RAID levels defined above is only single-failure tolerant; in each organization there exist pairs of disks such that the simultaneous failure of both disks results in irretrievable data loss. This is adequate in most environments, because the reliability of the component disks is high enough that the probability of incurring a second failure before a first is repaired is low. There are, however, three reasons why single-failure tolerance may not be adequate for all systems. First, recalling that the reliability of the array falls as the number of disks increases, the reliability of very large single-failure tolerating arrays may be unacceptable [Burkhard93]. Second, applications in which data loss has catastrophic consequences may mandate a higher degree of reliability than can be delivered using the RAID architectures described above. Finally, disk drives sometimes exhibit *latent sector failures*, in which the contents of a sector or group of sectors is irretrievably lost, but the failure is not detected because the data is never accessed. The rate at which this occurs is very low, but if a latent sector failure is detected on a surviving disk during the process of reconstructing the contents of a failed disk, the corresponding data becomes unrecoverable. Multiple-failure toleration allows recovery even in the presence of latent sector failures.

The drawback of multiple failure toleration is that it degrades write performance: in an *n*-failure tolerating array, every write operation must update at least *n*+1 disks, so that some record of the write will remain should *n* of those *n*+1 disks fail [Gibson89]. Thus the write performance of the array decreases in proportion to any increase in *n*.

Gibson et. al. [Gibson89] treated multiple failure tolerance as an error-control coding problem [Peterson72]. They restricted consideration to the class of codes that (1) do not encode user data, but instead simply store additional “check” information in each parity stripe, (2) use only parity operations (modulo-2 arithmetic) in the computation of the check information, and (3) incur exactly *n*+1 disk writes per user write. They defined three primary figures of merit on the codes used to protect against data loss: the *mean time to data loss*, which is the expected time until unrecoverable failure in an array using the indicated code, the *check disk overhead*, which is the ratio of disks containing ECC to disks containing user data, and the *group size*, which is the number of units in a parity stripe, including check units, supportable by the code. They demonstrated codes for double- and triple-error toleration based on three primary techniques, which they call *N-dimensional parity*, *full-n codes*, and the *additive-3* code. Each of these is a technique for defining the equations that relate each check bit to a set of information bits. In comparing the techniques according to the figures of merit, they show multiple-order-of-magnitude reliability enhancements in moving from single- to multiple-failure toleration, and achieve this using relatively low check disk overheads ranging from 2% to 30%.

Burkhard and Menon [Burkhard93] described two multiple-failure tolerating schemes as examples of *maximum distance separable* (MDS) codes [MacWilliams78]. The first uses a *file dispersal matrix* to distribute a block of data (a *file* in their terminology) into n fragments such that any $m \leq n$ of them suffice to reconstruct the entire file. An array constructed using such a code can tolerate $(n\text{-}m)$ concurrent failures without losing data. The second, described fully by Blaum et. al. [Blaum94], clusters together sets of N-1 parity stripes, where N is the number of disks in the array, and stores two parity units per parity stripe. The first parity unit holds the same information as in RAID Level 5, and the second holds parity computed using one data unit from each of the parity stripes in the cluster. Blaum et. al. showed that this scheme tolerates two simultaneous failures, is optimal with respect to check disk overhead and update penalty, and uses only XOR operations in the computation of the parity units.

2.2.2.6.2. Addressing the small-write problem

Recall from Section 2.2.2.3 that small write operations in RAID Level 5 incur up to four disk operations: data preread, data write, parity preread, and parity write. This degrades the performance of small write operations by a factor of four when compared to RAID Level 0. Several organizations have been proposed to address this problem.

Menon and Kasson [Menon89, Menon92a] proposed a technique based on *floating* the data and/or parity units to different disk locations upon each update. Normally, the controller services a small write operation by pre-reading the old data, waiting for the disk to spin through one revolution, writing the new data back to the original location, and then repeating this process for the parity unit. In the floating data/parity scheme, the controller reserves (leaves unoccupied) some number of data units on each track of each disk. After each preread operation, the array controller writes the new data to a rotationally convenient free location, rather than writing it in-place. This saves up to one full rotation (10-17 milliseconds of disk time) per preread/write pair. An analytical model in the paper shows that a free unit can typically be found within about two units of the location of the old data.

This makes each preread/write pair take only slightly longer than a single access, and thus can potentially nearly double the small-write performance of the array. Menon and Kasson concluded that the best capacity/performance tradeoff is achieved by applying this floating only to the parity unit, rather than to both data and parity. A potential problem with this approach is that the array controller must be intimately familiar with the geometry and performance characteristics of the component disks, as well as the latencies involved in communicating with them. This requires a high degree of predictability from the disks, and makes the design difficult to verify, tune, and maintain.

Another technique proposed to address the small-write problem is to eliminate them from the workload. The *Log-Structured File System (LFS)* [Rosenblum91, Seltzer93] has the potential to achieve this by organizing the file system as an append-only log. The motivation behind this file system is that a disk drive is able to service sequential accesses at about twenty times the bandwidth of random accesses. All user writes are held in memory until enough have accumulated to allow them to be written to disk using a single large update. Over time, this causes the disk to fill with dead data, and so a *cleaner* process periodically sweeps through the disk, compacts live files into sequential extents, and reclaims dead space. This technique improves write performance by causing all writes to be sequential, and can potentially improve read performance by causing files written contiguously to end up contiguous on the disk. When the underlying storage mechanism is a disk array, the only writes that are encountered are large enough to span entire parity stripes, and thus the large-write optimization always applies.

Stodolsky et. al. [Stodolsky93] adapted the ideas behind LFS to the problem of parity maintenance, and proposed an approach based on logging the parity changes generated by each write operation, rather than immediately updating the parity upon each user write. In this scheme, the controller reads the old data (or acquires it from the buffer cache), and writes the new data as before. It then XORs together the old and new data to produce a *parity update record*, which it appends it to a write-only buffer, rather XORing it with the old parity. The controller spills the entire buffer to disk when it becomes full. No parity operations are performed for each user write, but some of the array's capacity (about one disks' worth) must be reserved to hold the parity update logs. Eventually the log space in the array becomes full, at which time the controller empties it by reading the log records and the corresponding parity units, XORing them together, and writing the result back out to the parity locations. Note that the controller buffers only parity information, and so is not vulnerable to data loss due to power failure. The advantage of this approach is that in RAID Level 5, parity is updated using a large number of small, random accesses, whereas in parity logging, it is updated using a smaller number of large, sequential accesses. The paper showed simulation results indicating that this technique can allow the performance of RAID Level 5 arrays to approach, or under certain conditions even exceed, that of mirroring.

Menon and Cortney [Menon93] described the architecture of a controller that improves small-write performance by deferring the actual update operations for some period of time after the application performs the write. In this approach, the controller stores the data associated with a write in a nonvolatile, fault-tolerant cache memory in the array controller. Immediately upon storing the data in the cache, the host computer is told that the write is complete, even though the data has not yet been sent to disk. The controller maintains the data block in the cache until another block replaces it, at which time it is written (“destaged”) to disk using the four-operation RAID Level 5 update. This improves write performance in two ways. First, if the host performs another write to the same unit prior to destage, the new data can simply replace the old in the cache, and the first write need not occur at all. Second, if the host writes several units in the same track, they are all destaged at the same time, which greatly improves disk efficiency. This is an expensive solution, suitable only for large-scale systems, because of the necessity of incorporating the large, nonvolatile, fault-tolerant cache.

2.2.2.6.3. Spare space organizations

RAID Level 5 arrays typically maintain one or more on-line spare disks, so that reconstruction can be immediately initiated should one of the primary disks fail. This spare disk can be viewed as a system resource that is grossly underutilized; the throughput of the array could be increased if this disk could be used to service user requests.

Menon and Kasson [Menon92b] described and evaluated three alternatives for organizing the spare space in a RAID Level 5 disk array. The first, *dedicated sparing*, is the default approach of dedicating a single disk as the spare. In the second, called *distributed sparing*, the spare space is distributed amongst the disks of the array, much in the same manner as parity is distributed in RAID Level 5. In the third technique, *parity sparing*, the array is divided into at least two independent groups, and when a failure occurs, the affected group is merged with another, with the parity space in the surviving group serving as the spare space for the group containing the failure. In the latter two organizations, the completion of reconstruction returns the array to fault-free mode, but in a different configuration than before the failure. For this reason, they require a separate *copyback* phase in the reconstruction process, to restore the array to the original configuration when the failed disk has been physically replaced. The paper concluded that distributed sparing was preferable to parity sparing due to improved reconstruction-mode performance.

2.2.2.6.4. Distributing the functionality of the array controller

The existence of a centralized array controller in both of the architectures shown in Figure 2.5 has two disadvantages: it constitutes either a single point of failure or an expensive system resource that must be duplicated, and its performance and connectivity limit the scalability of the array to larger numbers of disks. Cao et. al. [Cao93] described a disk array architecture they call *TickerTAIP* that distributes the controller functionality amongst several loosely-coupled controller nodes. Each node controls a relatively small set of disks (one SCSI string, for example), and communicates with the other nodes via a small, dedicated interconnect network. Under the direction of the distributed controllers, data and parity units, as well as control information, pass through the interconnect to effect the RAID read and write algorithms. The paper demonstrated the elimination of several performance bottlenecks through the use of the distributed-control architecture.

2.2.2.6.5. Striping studies

A variety of studies have looked at how to select the striping unit in a redundant disk array. The choice is always made based on the characteristics of the expected workload.

Gray, Horst, and Walker [Gray90] objected to the notion of striping the data across the disks comprising an array, arguing that fine-grain striping is inappropriate for transaction processing systems because it causes more than one arm to be used per disk request, and that coarse-grain striping has several drawbacks when compared to non-striped arrays. These drawbacks stem primarily from the inability to address individual disks directly from software. They include the inability to archive and restore a single disk, the software problems inherent in re-coding existing device drivers to enable them to handle the abstraction of one very large, highly concurrent disk, the problem of designing single channels fast enough to absorb all bandwidth produced by the array, etc. They proposed instead an organization in which the parity is striped across the array in large contiguous extents at the end each disk. The data is not striped at all; the controller allocates sequential user sequentially on each disk, and fills each disk with data before using the next. This is essentially equivalent to RAID Level 5 with a very large striping unit, but allows each disk to be addressed individually. The paper conceded that none of these problems are insurmountable in RAID arrays, but asserted that designers cannot ignore the problem of retrofitting existing systems to use disk arrays.

Chen and Patterson [Chen90b] developed simple rules of thumb for selecting the striping unit in a nonredundant disk array. They expect that these rules will hold, perhaps with some modification, for redundant arrays as well. The study used simulation to evaluate the performance of a block-striped RAID Level 0 on many different synthetically-generated workloads, and then investigated choices of the striping unit that maximize the minimum observed throughput across all these workloads. They found that a good rule of thumb is to select the striping unit according to the formula

$$Size = S \cdot \textit{avg positioning time} \cdot \textit{disk xfer rate} \cdot (concurrency - 1) + 1\ \textit{sector}$$

where *S* is a constant typically around 1/4. Note that the stripe unit size takes on its minimum value (one sector) at concurrency one, in order to assure that the single requesting process is able to utilize all the disks. The size of the striping unit increases as the concurrency rises in order to gradually reduce the probability that any particular access will use more than one disk arm.

Lee and Katz [Lee91] described several different strategies for placing the parity units amongst the striped data units. They found that the most significant performance effect of varying parity placement was the number of disks used for large reads and writes; some placement strategies caused fewer than the maximum number of possible disks to be used on large accesses, and these suffered in performance. The left-symmetric parity placement illustrated in the RAID Level 5 case of Figure 2.6 was among the best of the options.

Merchant and Yu [Merchant92b] noted that it is common for a database workload to consist of two components: transactions, and ad hoc, read-only queries into the database. Transactions generate small, randomly distributed accesses into the array, whereas the ad hoc queries often scan significant portions of the database. To efficiently handle this workload combination, they proposed a dual striping strategy for mirrored arrays, where the size of the stripe unit is small in one copy (4 KB) and large in the other (32 KB). The authors note that using a large stripe unit is efficient for relatively large accesses because it reduces the number of actuators used, but under a small-access model, it can cause workload imbalance amongst the disks. They assert that the converse is true as well: a small stripe unit achieves good workload balance, but causes too many actuators to be used per large access. Thus they service the transactions using the small-stripe-unit copy of the data, and the ad hoc queries with the large-stripe-unit copy. Merchant and Yu evaluated this organization using both analytical modeling and simulation, with a synthetically generated workload that adhered to the assumptions made in designing the striping strategy. They found substantial benefits to this approach.

2.2.2.6.6. Disk array performance evaluation

Chen et. al. [Chen90a] tackled the thorny problem of comparing RAID Level 5 to RAID Level 1. The comparison is difficult to make because equating the number of actuators causes the array capacities to differ, and vice versa. The authors addressed this problem by choosing to equate user data capacity and reporting two metrics: throughput at a fixed 90th percentile response time, and throughput per disk at a fixed 90th percentile response time. Their motivation for this was the assumption that systems will dictate a minimum acceptable capacity and level of responsiveness, and will desire the maximum possible throughput subject to these constraints. The authors evaluated the architectures by implementing them in real hardware, and applying synthetically-generated workloads that varied in the parameters of interest. The results largely validated the simple model of Patterson et. al. [Patterson88], which is approximated in Table 2.1. They further showed that due to the shortest-seek optimization, the RAID Level 1 outperformed the RAID Level 5 on small-access dominated-workloads, whereas the reverse was true on large-access workloads due to more efficient write operations in RAID Level 5.

2.2.2.6.7. Reliability modeling

Patterson et. al. [Patterson88] derived a simple expression for the mean time to data loss (MTTDL) in a redundant disk array:

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

where $MTTF_{disk}$ is the mean time to failure of a component disk, N_{groups} is the number of independent groups in the array, each of which contains $N_{diskspergroup}$ disks, including the (possibly distributed) parity disk, and $MTTR_{disk}$ is the mean time to repair (reconstruct) a disk failure. This model assumes that disk failure rates are identical, independent, exponentially distributed random variables. In arrays that maintain one or more on-line spare disks, the repair time can be very short, a few minutes to half an hour, and so the mean time to data loss can be very long.

Schulze et. al. [Schulze89] noted that the time until data loss due to multiple simultaneous disk failures, which is the only failure mode modeled by the above equation, is not an adequate measure of true reliability because the failure of other system components (array controllers, string controllers, cabling, air conditioning, etc.) can equally well cause data to be lost or become temporarily inaccessible. This paper estimated the reliability of each such component, and derived simple techniques for building redundancy into the controllers, cabling, cooling, etc., so as to maximize the overall system reliability.

Modeling the reliability of disk arrays was the one of the primary topics of Gibson’s Ph.D. dissertation [Gibson92, Gibson93]. He analyzed all of the assumptions behind the simple equation given above, identified the conditions under which they do and do not hold, and derived new reliability models for conditions not previously covered. Specifically, he investigated whether disk failure rates are truly exponentially distributed, derived reliability models for disk arrays with dependent failure modes, extended these models to take into account the possibility of spare-pool exhaustion, and investigated the reliability implications of both the number and the connectivity of the spare drives. He verified the models using Monte Carlo simulation of disk lifetimes, and found good agreement between the two. This work theoretically and empirically validated the use of the models and disk array structures described above.

2.2.2.6.8. Improving the write-performance of RAID Level 1

As shown in Table 2.1, mirrored systems achieve only 50% of the write performance of nonredundant arrays, because each write must be sent to two disks. This section describes several studies intended to improve this performance. Most of the ideas here relate to caching and deferring updates, and so apply to parity-encoded arrays as well.

Solworth and Orji proposed several variations on an organization to improve mirrored-array write performance. They first proposed implementing a large, nonvolatile, possibly fault-tolerant *write-only disk cache* dedicated exclusively to write operations [Solworth90]. In this scheme, the controller defers user write operations by holding the corresponding data in the cache until a user read operation moves the disk heads to the vicinity of the data to be written, at which time it destages the data to disk. In this sense, this scheme is similar to the deferred-updated techniques described by Menon and Corney [Menon93], with the primary difference being that reads are not cached in Solworth and Orji's proposal, and the cache replacement policies are adapted to account for this. The authors do not address the question of whether some of the memory used for write-caching would be better used for read-caching.

In two follow-on studies, Solworth and Orji proposed *distorted mirrors* [Solworth91] and *doubly distorted mirrors* [Orji93]. In the former, the controller updates data in-place on the primary disk in a mirror pair, but writes the data to any convenient location on the secondary drive. The controller maintains a data structure in memory describing the location of each block on the secondary drive. This approach reduces the total disk-arm time consumed in servicing a write request. The controller services small reads from either copy, but services large reads from the primary copy only, since consecutive blocks on the secondary are not in general sequential on the disk. In the latter (doubly distorted mirrors), the authors combined the ideas of a write-only cache and write-anywhere semantics on the secondary drive to eliminate the necessity that the cache be nonvolatile and fault-tolerant.

Polyzois, Bhide, and Dias [Polyzois93] proposed a modification to the deferred-write technique in which the two disk arms in a mirror pair alternate between reading and writing. Deferred writes accumulate in the cache for some period of time, and then the controller batches them together and writes them out to one drive. During this period, the other drive services all read operations. The two drives then switch roles: the first services reads, and the second destages deferred writes. This scheme yields very low latency access to data at moderate workloads, because there is always one disk arm available to service user read requests, and write operations incur only the latency required to install the data in the cache.

2.2.2.6.9. Network file systems based on RAID

Several studies have looked at extending the ideas of striping and parity protection to network file systems. This allows the file system to operate in the presence of server and/or network failures, and provides for disaster recovery should all data stored at one site be permanently destroyed. It achieves this at lower disk cost that the standard approach of file duplication on multiple servers.

Stonebraker and Schloss [Stonebraker90] proposed an organization that is essentially identical to RAID Level 5, with each disk replaced by a server in a network file system. They evaluated the performance, overhead, and reliability of several variations on this idea, and concluded that distributed RAID has many reliability advantages, but performs poorly in the presence of failures. Other studies [Cabrera91, Hartman93] have extended this idea to network file systems that stripe data for performance.

2.3. Evaluation methodology

This section describes the techniques and workloads used to evaluate the availability techniques developed in this dissertation.

2.3.1. Simulation methodology

We performed nearly all the performance analyses in this paper using an event-driven disk array simulator called *raidSim* [Chen90b, Lee91], originally developed for the RAID project at U.C. Berkeley [Katz89]. We chose simulation because hardware was not available to allow real implementation, and because accurate analytical models of disk array performance are difficult to formulate due to the fact that a single user operation can invoke multiple disk operations [Lee93].

As disks get smaller and less expensive, and as systems demand increased I/O rates,

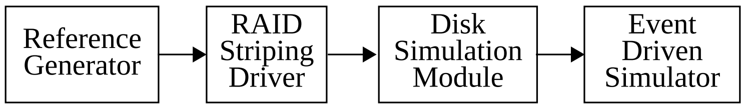


Figure 2.10: The structure of raidSim.

the number of disks in a typical array will increase. For this reason, we focus our simulations on array sizes that are larger than are common today. Specifically, the simulations reported in subsequent sections use a default array size of 40 disks. In order to verify that our conclusions are not specific to a particular array size, we also ran 20-disk simulations in most cases. The performance of the 20 disk array was identical to that of the 40-disk array for a given user workload measured in accesses per second per disk, and so in general we report only the 40-disk results here.

All reported simulation results represent averages over five independently seeded simulation runs. In nearly 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 show the actual intervals. Appendix C gives the 95% confidence interval computed for each run. For simulations of fault-free and degraded-mode arrays, the simulation was not terminated until the 95% confidence interval on the user response time had fallen to less than 3% 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.

2.3.2. The *raidSim* disk array simulator

RaidSim consists of four primary components, illustrated in Figure 2.10. At the top level of abstraction is a *reference generator.* This module simulates a set of independent processes performing read and write operations in the data space of the array. In most cases we configured the reference generator to produce synthetic workloads according to prespecified distributions (described in the next subsection). This allows us to identify and isolate specific performance effects in the array. This module also has the ability to apply references traces taken from real applications. We used this ability to validate conclusions by evaluating performance on more realistic workloads.

RaidSim sends the requests produced by this workload generator to a *RAID striping*

Array size:	40 disks
Stripe unit size:	24KB
Reconstruction unit:	24KB
Head scheduling:	FIFO
User data layout:	Sequential in address space of array
Disk spindles:	Synchronized

Table 2.2: Default array parameters for simulation.

driver, whose function is to translate each user request into the corresponding set of disk accesses. This code in this module was copied directly from the RAID device driver in the Sprite operating system [Ousterhout88] used for the RAID-I prototype at U.C. Berkeley [Lee90]. The original version of this code correctly controlled real hardware, and the Berkeley RAID group extracted it from Sprite and installed in raidSim with little or no modification. We extended this driver to support the disk array architectures and recovery techniques described in the remainder of this dissertation. This assured that the set of disk requests generated by each user request is identical to that which would be observed in a real system, which validated the accuracy of the simulation. We made every extension with the intention of preserving the property that this code could be re-installed in a device driver and would continue to function. We did not actually perform this re-installation due to lack of appropriate hardware. Table 2.2 shows the configuration of our extended version of this striping driver.

RaidSim sends low-level disk operations generated by the striping driver to a *disk simulation module*, which accurately models all significant aspects of each disk access (seek time, rotation time, cylinder layout, etc.). Table 2.3 shows the characteristics of the 314 MB, 3-1/2 inch diameter IBM 0661 Model 370 (Lightning) [IBM0661] disks on which we based the simulations. This disk was relatively current when we initiated this study, but has since been superceded [IBM0664].

At the lowest level of abstraction in raidSim is an *event-driven simulator*, which raidSim modules invoke to cause simulated time to pass. This is fundamentally a co-routine (lightweight process) package and a set of scheduling and queueing routines.

2.3.3. Default workload

This dissertation reports on many performance evaluations. In order to assure that the

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

Table 2.3: Parameters of the IBM 0661 Model 370 (Lightning) drive.

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:	5 x (number of disks)
Think time distribution:	Exponential, mean varied to adjust offered load

Table 2.4: Default workload parameters for simulations.

results of different sets of simulations are directly comparable, it's necessary to applied a consistent workload to the array. We therefore developed a default workload and apply it in most cases. Since the techniques developed in this thesis are most applicable in environments where the non-availability of data can lead to significant disruption in the service provided by the application, we chose to model an on-line transaction processing (OLTP) workload.

OLTP workloads are characterized by a large number of independent processes, concurrently reading and writing data in relatively small units [TPCA89, Menon92c, Ramakrishnan92]. They typically mandate a minimum level of responsiveness; for example, the TPC-A transaction processing benchmark [TPCA89] requires that 90% of all transactions complete in under two seconds. In many such environments, the system must service ad hoc database queries simultaneously with the transaction workload, which leads to a small percentage of larger accesses in the workload. Because of the random nature of the workload, caching is not highly effective on the actual account records comprising the database, and so the workloads observed by the storage subsystem are typically read-dominated [Ramakrishnan92]. This is the workload captured in the simulations.

Table 2.4 shows the default workload generated for the simulations, which we based loosely on access statistics measured on an airline-reservation OLTP system [Ramakrishnan92]. A majority of the studies in this dissertation use this workload, or one derived from it by modifying a single parameter.

Chapter 3: Disk Array Architectures and Data Layouts

This chapter describes disk array architectures and data layout mechanisms that can be used to design storage subsystems that exhibit good failure-recovery performance. The primary focus is on a technique we call *parity declustering* that reduces the amount of work required of each surviving disk to reconstruct each unit of data on a failed disk. This improves reconstruction time by reducing the total number of input/output operations needed to recover the contents of a failed drive, and also improves user response time by reducing the number of operations necessary to effect the on-the-fly reconstruction of a data unit requested by a user. Further, it reduces the failure-induced per-disk load increase from about 60%, as experienced by a RAID Level 5 array ([Ng92a], see also Section 3.3.3.1) to an arbitrarily-small fraction, thereby allowing the fault-free disk utilization to be higher than in RAID Level 5 without risking saturation should a failure occur. Further still, this technique balances all failure-induced workload over the disks comprising the array, and thereby both minimizes reconstruction time and maximizes user performance during recovery.

The remainder of this chapter is organized as follows. Section 3.1 describes previous and related work on the topic. Section 3.2 describes parity declustering, the main approach to the problem of on-line failure recovery taken in this thesis, and its implementation. Section 3.3 presents a comprehensive series of performance evaluations of parity declustering, demonstrating its advantages over previous solutions. Section 3.4 discusses the problem of configuring a system, which is the process of selecting the values of the main parameters under parity declustering. Having established and evaluated the basic technique, Section 3.5 discusses a number of optimizations and improvements that can be applied to improve specific performance aspects of the approach. Section 3.6 concludes and summarizes the chapter.

3.1. Related work

Previous and related work on architectures and data layout techniques for availability in storage subsystems stems primarily from mirrored-disk research in the database community. This section provides an overview of that work, and then describes the previous efforts aimed at parity-based arrays.

3.1.1. Availability techniques in mirrored arrays

Many researchers have proposed techniques to improve the performance of mirrored-disk storage systems [Bitton88, Copeland88, Orji93, Merchant92b], but only two such techniques, *interleaved declustering* and *chained declustering*, relate specifically to performance during failure recovery. Both of these techniques improve failure-recovery performance by distributing the failure-induced workload over more than the minimum number of surviving disks.

Traditionally, mirrored disk systems allocate one disk as a primary and another as a secondary or backup disk. To improve performance during failure recovery, Teradata Corporation implemented a scheme called *interleaved declustering* in their DBC/1012 database computer [Teradata85]. In this approach, the disks comprising the storage subsystem are divided into groups called *clusters*. Half of each disk is allocated to primary copies of user data (in the Teradata case, database relations), and the other half contains a portion of the secondary-copy data from each of the primaries on all other disks in the cluster. This insures that the controller can recover from a failure 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 cluster. Copeland and Keller [Copeland89] compared this approach to the more standard *mirrored declustering*, where each disk has a backup disk that stores identical data, and concluded that interleaved declustering provided significant improvements in recovery time, mean time to data loss, throughput during normal operation, and response time during recovery.

A problem with the interleaved declustering approach is that in order to distribute the reconstruction workload across a relatively large number of disks, the array must be configured with a large cluster size (the number of disks over which the backup copies are distributed). This reduces the system mean-time-to-data-loss (MTTDL) by increasing the probability that a second disk will fail before the data on the first can be recovered. Hsiao and DeWitt [Hsiao90, Hsiao91] describe a scheme called *chained declustering* which is able to distribute the reconstruction workload across a larger number of disks without affecting the MTTDL. In this organization, the controller maintains a single backup copy of each block of data on disk *i* on disk *(i*+1*) mod N*, where *N* is the total number of disks. When a disk fails, its backup disk absorbs all its workload, and the controller shifts a large percentage of the backup's regular workload to the next disk in the line (the backup's backup). The workload-shifting process repeats, with each disk in the line absorbing a decreasing amount of the regular workload of the disk for which it serves as the backup. The controller pre-computes the workload-shift percentage distribution to balance the resulting workload across all disks. This scheme can tolerate multiple failures in a cluster, as long as no two adjacent disks fail, and so it is attractive in systems where the cost of mirroring is warranted by the need for data integrity.

Both of these techniques rely on the existence of two copies of all data, and so they do not apply directly to parity-based arrays. However, as will be seen, techniques similar to these can be (and have been) applied to the design of non-mirrored redundant disk arrays.

3.1.2. Availability techniques for parity-based arrays

This section describes the technique currently used in the industry to improve failure-recovery performance, and demonstrates that it does not solve the failure recovery problem as defined in Section 2.1.4. The section then describes two previous array proposals aimed at addressing this problem. One of these proposals, which we call *parity declustering*, forms the foundation of the new architecture and data-layout techniques presented in the remainder of this chapter. Section 3.1.2.2 describes it in detail.

3.1.2.1. Multiple independent groups

The technique currently used in the disk-array industry to improve the failure-recovery performance of redundant disk arrays is to divide the set of disks comprising an array into a number of independent groups, and to compute parity over the disks in each group rather than over all the disks in the array [Rudeseal92]. Figure 3.1 illustrates one such approach for a fifteen-disk array consisting of three independent groups of five disks each[1].

1. To simplify the presentation, the user data units in the figure are laid out according to the left-asymmetric mapping scheme [Lee91]. There exist other arrangements of user data that provide better overall performance [Lee90, Lee91].

	Group 0					Group 1					Group 2				
	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
0	D0.0	D0.1	D0.2	D0.3	P0	D1.0	D1.1	D1.2	D1.3	P1	D2.0	D2.1	D2.2	D2.3	P2
1	D3.0	D3.1	D3.2	P3	D3.3	D4.0	D4.1	D4.2	P4	D4.3	D5.0	D5.1	D5.2	P5	D5.3
2	D6.0	D6.1	P6	D6.2	D6.3	D7.0	D7.1	P7	D7.2	D7.3	D8.0	D8.1	P8	D8.2	D8.3
3	D9.0	P9	D9.1	D9.2	D9.3	D10.0	P10	D10.1	D10.2	D10.3	D11.0	P11	D11.1	D11.2	D11.3
4	P12	D12.0	D12.1	D12.2	D12.3	P13	D13.0	D13.1	D13.2	D13.3	P14	D14.0	D14.1	D14.2	D14.3

Figure 3.1: Striping data across multiple independent groups in RAID Level 5.

The figure shows the mapping of consecutive parity stripes to physical disks in the array. Shaded blocks represent parity computed over the corresponding data blocks within each group. The notation Di.j *represents stripe unit number* j *of parity stripe number* i, *and* Pj *denotes the parity for parity stripe* j.

Since multiple-group arrays stripe consecutive user data across the groups, a workload that accesses the data space of the array uniformly sends an equal number of requests to each group. When a disk fails, only the user accesses that address the failed group experience degraded performance. As long as the disks in the affected group do not saturate, the failure affects only one out of every N_{groups} accesses. Increasing the number of groups in the array increases the capacity overhead for redundancy, since more blocks are consumed by parity, but in the absence of disk saturation it reduces degraded-mode performance degradation by a factor of N_{groups} over that of a single-group configuration.

This approach, although widely used, is not an adequate solution to the failure recovery problem for a number of reasons. It does not allow the fault-free utilization on any disk to be above about 60% (as derived in Section 2.1.4) because, should a failure occur, the resultant disk saturation in the affected group would cause unacceptably long user response time for all accesses that address that group. Second, a group that contains a failure becomes a severe system performance bottleneck. Assuming that user processes access the data space of the array uniformly, every process periodically accesses the failed group. Since the disk utilizations are high in this group, the queueing time for these accesses is long. Therefore, the processes are not able to access the array at their normal rate because they regularly get stuck for long periods trying to perform I/O in the affected group. Finally, this technique causes reconstruction time to be very long because the only disks that participate in the reconstruction of a failed drive are those in the failed group, and these are precisely the drives on which the load has increased.

3.1.2.2. Distributing the failure-induced workload

It's clear from the above discussion that in order for a disk array to maintain adequate performance in the presence of disk failure, the load increase due to the failure must be both minimized and distributed over the array rather than concentrated in a small number of disks. Recognizing this, Muntz and Lui [Muntz90] applied ideas similar to that behind interleaved declustering [Copeland89] to parity-based arrays to derive an organization we call *parity declustering*[2]. Whereas interleaved declustering distributes the backup copies of data over more than the minimal set of disks, parity declustering distributes fixed-width parity stripes across an array whose size (number of disks) is larger than the number of units in a parity stripe. Muntz and Lui's proposal forms the foundation of our work on architectures to improve failure-recovery performance, and so this section describes it in detail.

Referring again to Figure 2.6, 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 3.2, 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 physical 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. To see this in Figure 3.2, assume that physical disk number 3 has failed; in order to reconstruct the parity stripe labeled 'S', it is only necessary to read units from disks zero, four, and six. Disks one, two, and five are not involved at all.

In parity declustering, the reconstruction of each unit on the failed drive requires G-1 reads. Since there are C-1 surviving drives, each surviving disk sees a failure-induced read load increase of $(G\text{-}1)/(C\text{-}1)$ instead of $(C\text{-}1)/(C\text{-}1)$ = 100% as in RAID Level 5 (refer to Section 2.1.4)[3]. Similarly, each disk sees an approximate failure-induced write load

2. Muntz and Lui use the term *Clustered RAID* to describe this approach. Their use may be derived from "clustering" independent RAIDs into a single array with the same parity overhead. Our use follows the earlier work on mirrored arrays, where redundancy information is "declustered" over more than the minimal collection of disks.

3. This assumes the failure-induced workload is evenly balanced over all disks.

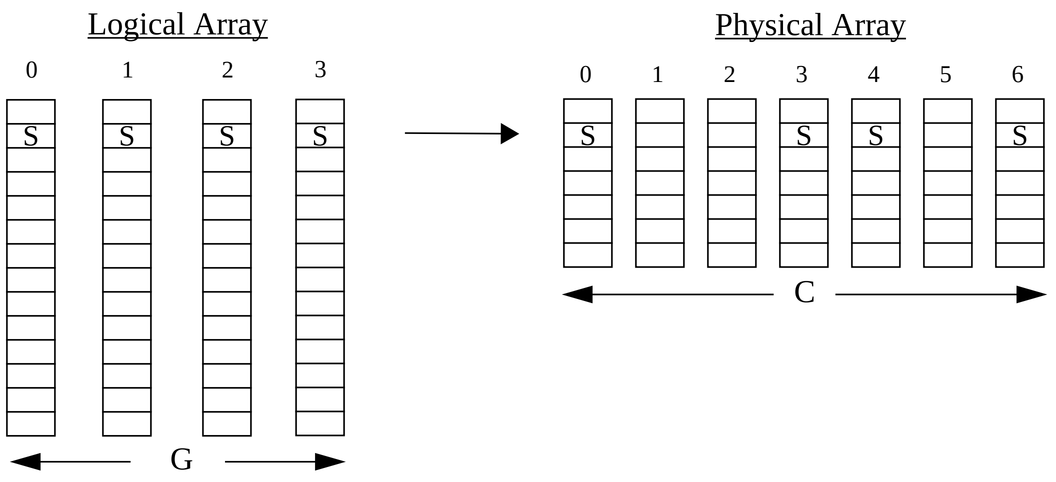


Figure 3.2: Declustering a parity stripe of size four over an array of seven disks.

increase of $\frac{1}{4}(G\text{-}1)/(C\text{-}1)$ instead of $\frac{1}{4}(C\text{-}1)/(C\text{-}1)$ = 25% as in RAID Level 5. The fraction *(G*-1*)/(C*-1*)* is referred to as the *declustering ratio* and is denoted by α. Parity declustering therefore reduces the degraded-mode workload increase due to user reads from a factor of 2.0 to a factor of (1+α), and reduces the load increase due to user writes from a factor of 1.25 of a factor of 1+0.25α. Thus the performance degradation due to failure recovery diminishes toward zero as α is reduced.

The declustering ratio can be made smaller, and thus failure-recovery performance improved, either by increasing *C* for a fixed *G* as shown in Figure 3.2, or by decreasing *G* for a fixed *C*. As α is made smaller, performance during failure recovery improves but more of the array's capacity is consumed by parity. When *G*=2 (the minimum possible value), declustered parity reduces to mirroring since the controller computes the parity unit for each parity stripe as the XOR over only one data unit. Thus declustered parity with *G*=2 is essentially equivalent to interleaved declustering [Copeland89, Hsiao91] in that the controller maintains two copies of each data unit, with the mirror data being distributed over the other disks in the array. At the other extreme, when *G*=*C* (α = 1.0), parity declustering is equivalent to RAID Level 5. Thus parity declustering can be seen as defining a continuum of design points between RAID Level 5 and mirroring, exhibiting improved performance in the presence of failure but increased capacity overhead as *G* is reduced.

Reddy and Bannerjee [Reddy91] proposed an organization essentially equivalent to a limited version parity declustering. Their technique assigns each unit on each disk in an array to one of a set of *parity groups*, in such a way as to distribute the groups evenly across the array. They derive the assignment of units to parity groups from the incidence matrix of a *balanced incomplete block design* (refer to Section 3.2.2) for the case when the array is divided into exactly two parity groups, that is, when the number of units in a parity stripe is equal to half the number of disks in the array. This is equivalent to a parity-declustered organization with $\alpha \approx 0.5$. They discuss the possibility of deriving layouts for other values of α, but do not make the mechanism clear in the general case.

3.1.3. Summary

In order to achieve the degraded- and reconstruction-mode performance advantages of declustering, it is necessary to find a mapping of logical parity stripes onto sets of physical disks such that the failure-induced workload is balanced over the C disks comprising the array. This is referred to as *data layout*. Muntz and Lui left it as an open problem, and Reddy and Bannerjee address it only for special case where α is approximately 0.5. One of the contributions of this dissertation is to solve this problem. Ng and Mattson [Ng92a] independently proposed a data layout solution very similar to the one described in the next section. This study, concurrent with our own, formulates the implementation of a parity declustered layout using essentially the same technique as is described in Section 3.2.2. This dissertation provides a more thorough treatment of the implementation issues, and improves on the basic technique in several ways. Merchant and Yu [Merchant92a] have also addressed this problem; Section 3.2.3 discusses their solution and contrasts it to our own.

This dissertation further extends prior work by providing comprehensive analyses of failure recovery techniques in the larger context of overall system performance, and using the results to derive improved disk array architectures and data layouts. Issues addressed in this chapter include deriving a declustered data layout that allows reconstruction accesses to be of arbitrary size, and combining declustering with the multiple independent groups approach to yield arrays that simultaneously provide high levels of degraded- and reconstruction-mode performance, reliability, and availability, while consuming only a relatively small fraction of the capacity of the array for redundancy.

3.2. 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. We refer to this mapping of an application's logical unit of stored data to physical disk locations and associated parity locations as the disk array's *layout*. In this section we discuss goals for a disk array layout, present a layout for declustered parity based on balanced incomplete block designs, and contrast it to a layout proposed by Merchant and Yu [Menon92a] which supports more combinations of array size (C) and number of units per parity stripe (G) in large arrays, at the cost of higher complexity.

3.2.1. Layout goodness criteria

Extending from prior studies [Lee90, Dibble90, Reddy91, Merchant92a], we have identified six criteria for a good disk array layout.

1. *Single failure correcting.* No two units (whether data or parity) contained in the same parity stripe may reside on the same physical disk. This is the basic characteristic of any failure-tolerating organization. In arrays in which groups of disks have a common failure mode, such as the case where a break in a communication link causes multiple disks to become unavailable, this criteria should be extended to prohibit the allocation of data or parity 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, the user workload that it had serviced prior to the failure should be evenly distributed across all disks that share any parity stripe with the affected disk. In a parity-based array, each user access to data on a failed drive translates into multiple physical disk accesses in order to effect the on-the-fly reconstruction of the data. Criterion 2 states that, over time, each surviving disk in the array should service an equivalent number of these

accesses. When a failed disk is replaced or repaired and the background reconstruction process is started, the reconstruction workload should also be evenly distributed.

3. *Distributed parity.* In order that the parity update load should be evenly balanced, an equivalent number of parity units should be assigned to each disk in the array.

4. *Efficient mapping*. The functions mapping a file system's logical block address to the physical disk addresses for the corresponding data unit and parity stripe, and the appropriate inverse mappings, must be efficiently implementable. These functions are executed each time a user data unit is accessed, and so it is necessary that they consume neither excessive computation nor memory resources.

5. *Large write optimization*. The mapping functions should have the property that user data units that are contiguous in the address space of the array map to contiguous data units within contiguous parity stripes on the physical drives. This insures that whenever a user performs a write operation that is the size of the data portion of a parity stripe and starts on a parity stripe boundary, it is possible to update the corresponding parity unit without pre-reading the prior content of any data or parity units. When this criterion holds, the controller can service such a write operation by simply computing the cumulative XOR over the newly-written data units, and writing the result to the corresponding parity unit. When this criterion does not hold, that is, when a contiguous region of user data maps to a set of data units in differing parity stripes, it is necessary to perform read-modify-write operations on the data and parity units in each affected parity stripe. Thus the total number of I/O operations necessary to service large user write operations is reduced when this criterion holds.

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 the controller can achieve maximum parallelism.

There are a number of points to be made about these criteria. Firstly, criteria two and three make recommendations about balancing workload over the array. In general, this means that considering the array as a whole (all data and parity units on all disks), the workload should be balanced. Note, however, that if the applied workload is localized to a particular region of the array, rather than being evenly distributed across the entire data space, then the failure-induced and parity-update workload can be imbalanced despite the fact that the array meets both criteria two and three. For this reason, these criteria should be extended to apply to the smallest possible subsection of the address space, so that localized workloads see the same benefits.

Secondly, criterion six should not be interpreted as placing constraints on the size of the data unit in the array; it makes recommendations only about the assignment of consecutive data units to disks. Using more than one disk to service a read operation increases the total positioning overhead (seek time and rotational delay) incurred by the read, but reduces the data transfer time. If the amount of data transferred from each drive is relatively small, and other requests are waiting to access the array, then the parallel transfer of the access will lead to significantly lower throughput because of this extra positioning overhead. In this case, the controller could achieve higher throughput by servicing multiple accesses concurrently, with each accesses using fewer drives. However if a very large read is serviced by a small number of disks, the response time of the read will be very long due to the lack of parallel data transfer. Therefore, the stripe unit size should be selected according to the characteristics of the expected workload [Chen90b], and the layout policy should not influence this selection.

The best way to understand the value of criterion six is to consider the ramifications of disregarding it. After the characteristics of the expected workload have been used to determine the appropriate data unit size, it may still be the case that there occur some user accesses large enough to span all the disks in the array. If criterion six is ignored, the resultant layout could allocate the data units of a very large contiguous read over a possibly small subset of the disks. (This is consistent with criterion five if G is much smaller than C.) This could render the file system or application program unable to achieve high transfer bandwidth even for very large contiguous reads, and so the response time of these reads would be many times longer than necessary. Criterion six provides a very simple model for file systems and applications to ensure fast transfer for large objects.

Finally, note that the first four criteria deal exclusively with relationships between stripe units and parity stripe membership, and thus are properties only of the functions that map units in the address space of the array to physical disk locations. The last two make recommendations for the relationship between the allocation of contiguous user data into the address space of the array and parity stripe organization. Unfortunately, the disk array controller (whether implemented as hardware or a device driver) has no way of identifying a region of user data as contiguous, because a file system is not required to allocate contiguous user data contiguously in the array's address space. For example, in order to manage storage space efficiently, file systems generally divide each user data file into fixed-size blocks, and store each in a separate and often unrelated location. Although the data in the file is logically contiguous, and is perhaps always read and written contiguously, there is no way for the array controller to identify this, and thus no way for it to guarantee that criteria five and six are always met. The best that can be done is to attempt to meet these last two criteria for data units that *are* contiguous in the address space of the array. This issue is discussed more fully in Section 3.5.3.

3.2.2. Layouts based on balanced incomplete block designs

Our primary goal in designing a layout strategy for parity declustering was to meet criterion two: 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 required from each surviving disk during the reconstruction of the entire contents 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. In other words, the failure-induced workload will be balanced 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.

3.2.2.1. Block designs

A block design is an arrangement of v distinct objects into b tuples[4], each containing

4. These tuples are called *block*s in the block design literature. We avoid this name as it conflicts with the common 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 either of the common usages of λ as the rate of arrival of user accesses at the array or the failure rate of some system component.

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 3.3. An example parity-declustered layout.

k objects, such that each object appears in exactly *r* tuples, and each pair of objects appears in exactly λ_p tuples. For example, Table 3.1 gives a block design with $b = 5$, $v = 5$, $k = 4$, $r = 4$, and $\lambda_p = 3$, using non-negative integers as objects.

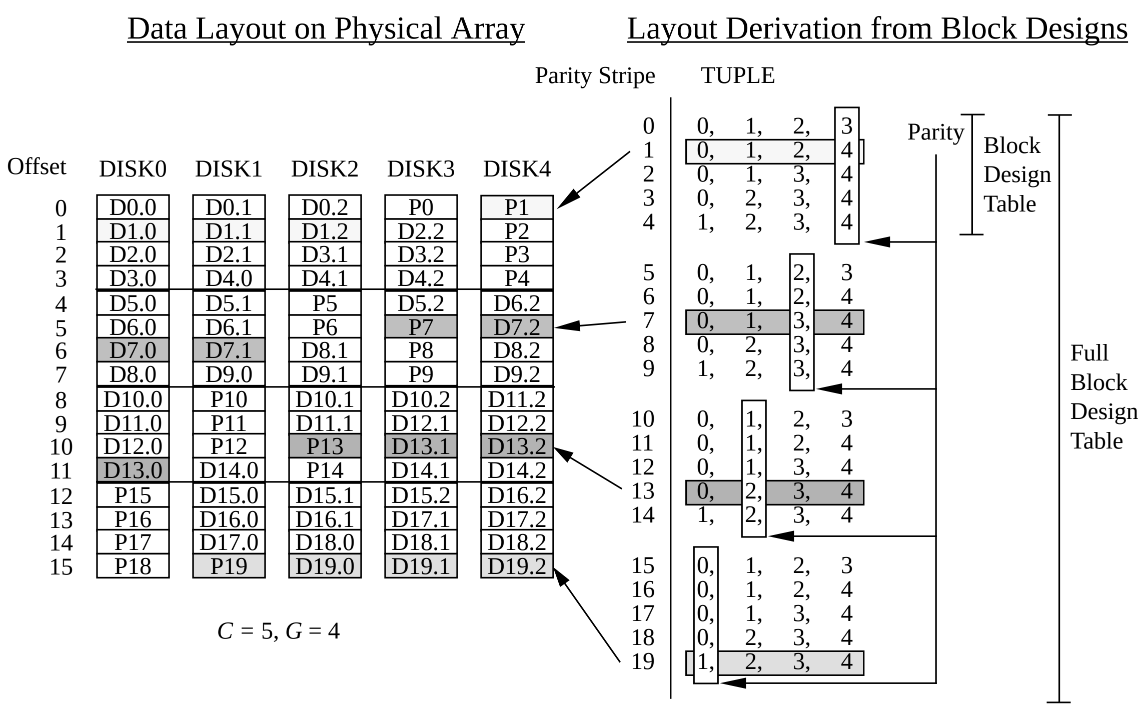
Tuple Number	Tuple
0	0, 1, 2, 3
1	0, 1, 2, 4
2	0, 1, 3, 4
3	0, 2, 3, 4
4	1, 2, 3, 4

Table 3.1: 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 objects 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.

3.2.2.2. Deriving a layout from a block design

The layout associates disks with objects and parity stripes with tuples. For clarity, the following discussion is illustrated by the construction of the layout in Figure 3.3 from the block design in Table 3.1. To build a parity layout, we find a block design with $v = C$, $k = G$, and the minimum possible value for *b*. The mapping identifies a tuple in the block design with a parity stripe: each object in a tuple identifies the disk on which the corresponding data or parity unit of the parity stripe resides. In Figure 3.3, the first tuple in the design of Table 3.1 is used to lay out parity stripe 0: the three data blocks in parity stripe 0



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
4	D5.0	D5.1	P5	D5.2	D6.2
5	D6.0	D6.1	P6	P7	D7.2
6	D7.0	D7.1	D8.1	P8	D8.2
7	D8.0	D9.0	D9.1	P9	D9.2
8	D10.0	P10	D10.1	D10.2	D11.2
9	D11.0	P11	D11.1	D12.1	D12.2
10	D12.0	P12	P13	D13.1	D13.2
11	D13.0	D14.0	P14	D14.1	D14.2
12	P15	D15.0	D15.1	D15.2	D16.2
13	P16	D16.0	D16.1	D17.1	D17.2
14	P17	D17.0	D18.0	D18.1	D18.2
15	P18	P19	D19.0	D19.1	D19.2

Parity Stripe	TUPLE
0	0, 1, 2, 3
1	0, 1, 2, 4
2	0, 1, 3, 4
3	0, 2, 3, 4
4	1, 2, 3, 4
5	0, 1, 2, 3
6	0, 1, 2, 4
7	0, 1, 3, 4
8	0, 2, 3, 4
9	1, 2, 3, 4
10	0, 1, 2, 3
11	0, 1, 2, 4
12	0, 1, 3, 4
13	0, 2, 3, 4
14	1, 2, 3, 4
15	0, 1, 2, 3
16	0, 1, 2, 4
17	0, 1, 3, 4
18	0, 2, 3, 4
19	1, 2, 3, 4

Figure 3.4: Full block design table for a parity declustering organization.

The right half of the figure illustrates how G iterations of the block design table are used to construct the full block design table, with the parity rotating through the columns of the design. The left half shows the corresponding full block design table layout on five drives, with a few parity stripes shaded for clarity.

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, the layout assigns stripe unit *j* of parity stripe *i* (labeled *Di.j* in Figure 3.3) to the lowest available offset on the disk identified by the j^{th} object of tuple *i mod b* in the block design, and the last object of each tuple is used to map the parity unit.

It is apparent from Figure 3.3 that this approach produces a layout that violates the distributed parity criterion (3). To resolve this violation, we duplicate the above layout *G* times (four times in this example), assigning parity to a different object of each tuple in each duplication, as shown in Figure 3.4. This layout, the entire contents of Figure 3.4, 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 3.4) as a *block design table*, and one complete cycle (all blocks in Figure 3.4) as a *full block design table*, or simply a *full table*.

Note the following from Figure 3.4. First, the units comprising a parity stripe do not all reside at the same disk offset in the array, as they typically do in a RAID Level 5 array (refer back to any of the groups in Figure 3.1). Second, a single instance of a block design maps an identical number of units on each physical disk. To see this, note that from the definition, each object appears in exactly r blocks in the design, and hence one instance of the block design maps exactly r units onto each physical disk. Since the layout maps the entire array via copies of the block design table, it assigns an equivalent number of units to each disk. Third, 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, and so it is necessary to find an appropriately small design for each combination of C and G. Finally, defining the *full table depth* as the number of units assigned to each disk over the course of one full table ($k \cdot r$), Figure 3.4 seems to imply that the number of units on each disk must be an exact multiple of the full table depth. This is not actually the case, since there need not be an integral number of full tables, nor even of tables in the array. As long as the full table depth is sufficiently small, the slight parity and workload imbalances caused by the existence of one partial full table at the end of the array are negligible. Section 3.5.2.3 discusses this issue in more detail.

3.2.2.3. Evaluating the layout

It is easy to verify that the layout of Figure 3.4 meets the first four 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 object 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. While it is not guaranteed that a block design will exist for every possible combination of C and G, nor that the number of tuples 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 about 40 disks, and for many of the possible combinations beyond.[5] Section 3.4 discusses the problem of designing larger arrays.

5. Appendix B describes a large database of block designs publicly available via the internet.

(3) Parity is balanced because over the course of one full table, the layout assigns parity to each object of each tuple in the block design exactly once (refer to the boxes labelled "parity" in Figure 3.4), and since each object appears exactly r times in the block design table, the layout assigns a parity unit to each disk exactly r times over the course of the full table. Again, since parity is balanced in every full table, it is balanced over the entire array. Note that if the number of units on a disk is not a multiple of $k{\cdot}r$ (the number of units assigned to each disk over the course of the full table), then there will a small imbalance in the parity distribution due to the existence of a partial full block design table at the end of the array. As long as the number of units assigned to each disk in one full table is a relatively small fraction of the number of units per disk in the array, this imbalance will be negligible. However, in arrays consisting of small disks mapped by large block design tables, the imbalance may become significant. Section 3.5.2 discusses a technique to reduce the size of the full block design table in order to alleviate this problem.

(4) The data mapping algorithms presented in Appendix A operate in constant time, and consist of a few dozen multiply, divide, and modulo operations. Thus they do not consume excessive computational resources. For efficiency, the algorithms use three tables in memory; two of them are integer arrays of size equal to the number of objects in the block design ($b{\cdot}k$), and one is an integer array of size equal to the number of data and parity units in one table[6]. As long as the block design for a particular layout is sufficiently small, the mapping tables are also small (nearly always less than about 20 KB), and so the algorithms are also memory efficient.

As previously mentioned, criteria five and six depend on the assignment of user data units to units in the address space of the array, and so no data layout mechanism can guarantee that they will always 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 in the linear address space of the array, 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 3.4

6. At this point in the discussion, these two table sizes appear to be the same since $bk = vr$. Section 3.5.1 shows an extension of the declustered parity architecture in which they can differ.

shows that reading *C* (5, in this case) successive user data units starting at the unit marked *D0.0* causes two units to be read from each of disks 0 and 1, and does not use disks 3 and 4 not at all. Hence the layout violates criterion six.

Criterion six is important when a significant fraction of the workload consists of read operations of size approximately equal to the number of disks in the array multiplied by the size of one data unit. Note that reads of less than *G* data units always use the maximum number of disks, and reads of a very large number of units typically span enough parity stripes to use most or all the disks in the array. Typical OLTP transactions access data in small units [TPCA89, Ramakrishnan92], and large accesses account for a small fraction of the total workload, typically deriving from ad hoc queries (decision-support functions) or array-maintenance functions rather than actual transactions [Merchant92b]. 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]. This means that the benefits of achieving criterion six in our layout would be marginal at best in OLTP workloads, and so we do not consider the failure to meet criterion six a significant drawback of the layout policy. However, under a user workload where large read operations are more common, the failure to meet criterion six, combined with the fact that a declustered parity array must skip over more parity units when servicing a read large enough to access multiple data units from multiple disks, causes the response time of these large reads to be longer in parity declustering than in RAID Level 5. To explore this issue further, Section 3.3.4 analyzes the performance of the layout strategy for non-OLTP workloads, and Section 3.5.3 investigates the possibilities of simultaneously meeting both criteria five and six.

3.2.2.4. Finding block designs for layout

This section describes the process of finding a block design once the values of *C* and *G* have been determined. The problem of selecting the values of these two parameters is discussed in Section 3.4.

Complete block designs such as the one in Table 3.1 are easily generated, but the number of tuples in a complete design, $\binom{C}{G}$, is in general so large that the layout fails to have an efficient mapping. For example, a 40 disk array with 10% parity overhead (*G*=10) mapped by a complete block design will have close to 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 criterion two (distributed reconstruction) nor criterion three (distributed parity) 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. This section describes the process of finding such a design for a given parameter set. Section 3.4 discusses the problem of generating layouts for arrays in which C is large enough that no block design is known for a given combination of C and G.

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, researchers generate designs 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.

When a block design on a particular parameter set is needed, the first step in finding it is to check these tables. The Hanani table presents designs that can be used to generate layouts for all values of G as long as C is at most 43. Appendix B summarizes the remaining tables for larger values of C. When a required design does not exist in any table, it is often possible to vary C and/or G slightly from the desired values in order to find a design. For example, we know of no design that can be used to generate a layout for C=33 and G=11, but designs on C=33 exist for both G=10 and G=12, as do designs on G=11 with C=32 and C=34. Using any of these four alternative designs results in a layout that is nearly identical to the C=33, G=11 case for every possible figure of merit. In general, as long as the variations in C and G are small (recall that this is nearly never necessary for arrays of less than 44 disks), adjusting either parameter has negligible impact on all aspects of system performance. When none of the above techniques yield a design with an acceptably small number of tuples, a layout can be generated using the alternative layout strategy described in the next section.

3.2.3. A related study: layout via random permutations

Section 3.4 will show that it is possible, using only designs that are known to exist, to construct parity declustered disk arrays with an arbitrary number of disks and a very small degree of performance degradation during failure recovery. However, it will also show that in systems with very stringent requirements (performance during recovery, capacity overhead, and data reliability), it may be necessary to use a system configuration for which no block design can be found. In this case, there is an alternative declustered parity layout strategy that can be used. Developed by Merchant and Yu [Merchant92a], this approach is based on randomizing the assignment of parity stripes to disks, and so does not use block designs at all. This layout is therefore capable of mapping a declustered array using any combination of C and G, but it does so at the expense of much larger computational overhead. Section 3.4 describes the process of selecting between layout strategies.

Merchant and Yu's 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 n^{th} 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:

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

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 identifies 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 the layout meets criterion one. If C is not a multiple of G, then using each permutation R = 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. The layout meets criterion four 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 certainly requiring substantial computation, this algorithm's asymptotic computation needs grow slowly (logarithmically) with respect to C and G. The random-permutation layout meets or fails criteria five and six 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 mapping algorithms in Appendix A compute physical disk addresses via a lookup in a small table and a few dozen arithmetic operations.

3.2.4. Summary

This section demonstrated layout techniques that allow the per-disk load increase experienced during failure recovery to be arbitrarily small (a factor of α). These techniques improve failure recovery performance by increasing the amount of redundant information stored in the array, that is, by trading off array capacity for performance during failure recovery. The next section verifies via simulation that these improvements are actually achievable.

3.3. Primary evaluations

This section presents the results of four simulation studies, using the simulation system and workloads described in Chapter 2. Each focuses on five primary figures of merit: performance when all disks in the array are functional (fault-free mode), performance in the presence of a single disk failure (degraded mode), performance during the reconstruction of a failed drive (reconstruction mode), total time required to reconstruct a failed drive (reconstruction time), and the probability of data loss within a fixed period of time (data reliability). The declustered parity simulations use the block-design based layout described in Section 3.2.2.2 in all cases.

The first study compares a declustered-parity disk array to an equivalent-capacity multiple-group RAID Level 5 array. Since both of these techniques operate by trading off array capacity for improved failure recovery performance, equating both the number of disks and the total data capacity of the arrays allows a direct comparison. This study demonstrates that the declustering solution is uniformly superior for nearly every figure of merit.

Once the advantages of declustering relative to RAID Level 5 are established, the second study looks into the benefits that can be obtained by reducing G while keeping C fixed. Reducing G results in less available data capacity in the array, but improves performance in the presence of failure by reducing the number of I/O operations required to reconstruct any particular data unit.

The third study verifies the claim that declustering allows a fault-free array to be driven to higher degree of utilization than a RAID Level 5 array, without risking unacceptable response time during recovery. The section derives a simple analytic model for the amount by which the fault-free workload in a declustered array can be increased over a RAID Level 5 array, as a function of α. It then presents the results of a set of simulations comparing the degraded-mode response time between arrays with varying declustering ratios, where the user workload applied to the array is increased commensurate with each decrease in α. The simulation results validate both the model and the ability of declustered arrays to support higher fault-free workloads.

Since the first three studies evaluate declustering using a small-access workload typical of OLTP applications, the fourth looks at performance on other workloads. The first part of this study uses synthetic workloads to characterize the performance of declustering

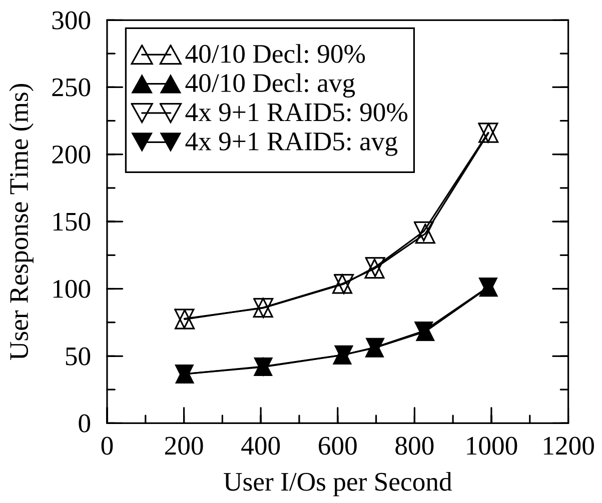


Figure 3.5: Comparing organizations: fault-free performance.

with respect to user access size, read fraction, and locality of reference, and the second part evaluates performance using workloads derived from real applications.

3.3.1. Comparing declustering to RAID Level 5

The comparison examines arrays with a total of 40 disks, and keeps constant the fraction of the array's capacity consumed by parity. It uses the workload described in Section 2.3.3. Specifically, we fix the size of a parity stripe at 10 units (10% parity overhead), which means the simulations compare a 4-group 9+1 RAID Level 5 (α=1.0) to a C=40, G=10 declustered array (α=0.23).

3.3.1.1. No effect on fault-free performance

Figure 3.5 plots the average and ninetieth-percentile user response time versus the achieved user I/O operations per second when the declustered parity and RAID Level 5 arrays are fault-free. This figure shows that for OLTP workloads, parity declustering causes no fault-free performance degradation with respect to RAID Level 5. This is a necessary condition, since improving failure-recovery performance at the expense of fault-free performance is unacceptable. Section 3.3.4 characterizes the performance of fault-free declustered arrays over a wider range of workloads.

3.3.1.2. Declustering greatly benefits degraded-mode performance

Figure 3.6 plots the respective disk arrays' user response-time against achieved user I/Os per second when each array contains one failed disk, but reconstruction has not yet

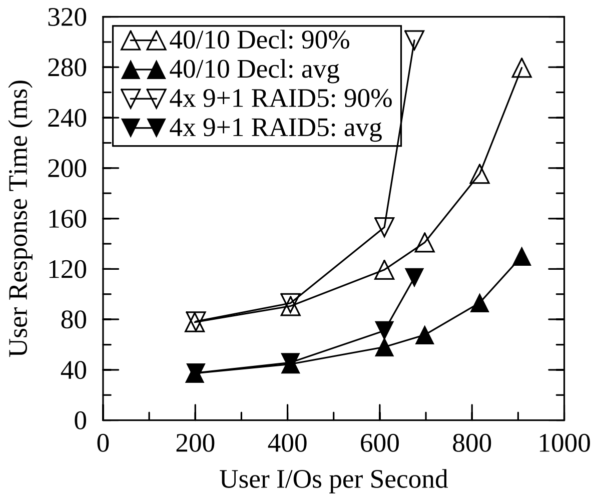


Figure 3.6: Comparing organizations: degraded-mode performance.

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. As the workload climbs, the failure-recovery problem in RAID Level 5 arrays becomes evident: 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. Once the affected group has saturated, all requesting processes are affected because user data is striped over all the groups of the array to achieve load balance. Thus every user process periodically requests data from the affected group, and is forced to wait in the queue for that group for a long period of time. This yields underutilization of the remaining disks, and so the system is not able to deliver all of its potential bandwidth to the users. Because the declustered-parity array distributes failure-induced work across all disks in the array, it is 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.

3.3.1.3. Declustering benefits persist during reconstruction

Figure 3.7 shows average and 90th percentile user response times while reconstruction is ongoing. In contrast to the degraded-mode performance shown in Figure 3.6, Figure 3.7 shows that at low user workloads, parity declustered arrays deliver very slightly (a few percent) 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

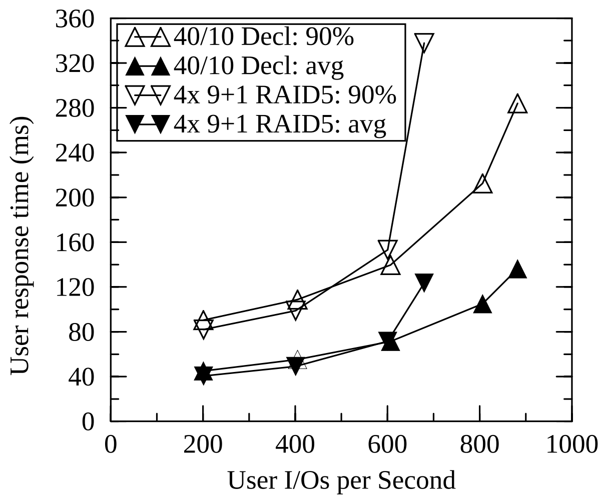


Figure 3.7: Comparing organizations: response time during reconstruction.

have plenty of available bandwidth. But, as described in the previous section, this group quickly becomes saturated as the on-line user load increases. This dramatically increases average and 90th percentile response times, and limits total throughput.

Turning to the issue of time until reconstruction completes, Figure 3.8 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 contents of this failed disk, reconstruction time is poor at all levels of 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 (on which the load has nearly doubled), 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. The 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 3.8, consider the point corresponding to 15 user I/Os per second per disk. This load (600 total I/Os per second into 40 disks) drives the array's disks to about 50% utilization in fault-free

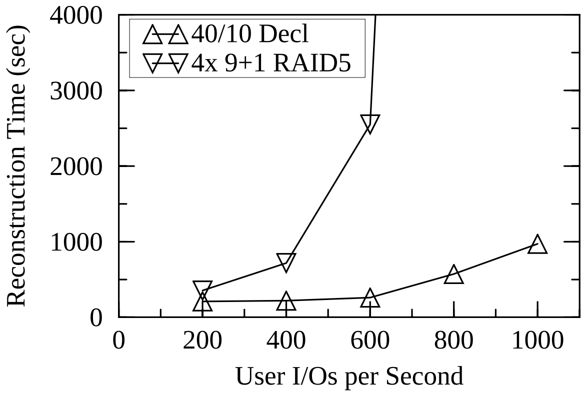


Figure 3.8: Comparing organizations: reconstruction time.

mode, and so constitutes a reasonable operating point. Reconstruction time in the declustered parity organization at this load level 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 3.7 and Figure 3.8 show response time and reconstruction time in the same on-line reconstruction event – they show that parity declustering provides huge savings in reconstruction time, and simultaneous savings in response time for moderately and heavily loaded disk systems.

3.3.1.4. Declustering also benefits data reliability

Our final figure of merit is the probability of losing data because of 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, or, equivalently, 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[7]. From this, the probability of data loss

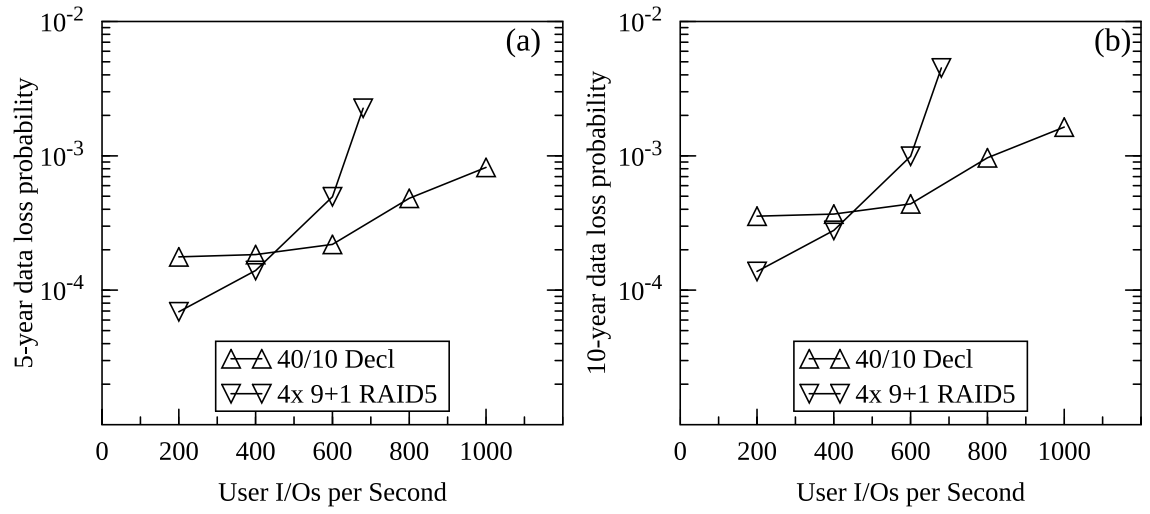


Figure 3.9: Comparing organizations: reliability.

The figures show the probability of data loss within (a) 5 years and (b) 10 years. Note that the Y-axis is log-scaled.

in a time period T can be modeled as

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

Figure 3.9 shows the probability of losing data within 5 and 10 years (optimistic estimates 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, rise 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).

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

3.3.1.5. Summary

This section 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. 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. This makes a compelling case for the use of parity declustering in on-line systems that cannot tolerate substantial degradation during failure recovery.

3.3.2. Varying the declustering ratio

Having established the benefits of declustered parity over multiple-group RAID Level 5, this section investigates how much failure recovery performance can be improved by relaxing the constant-data-capacity assumption. Specifically, it examines the effect on failure recovery performance of varying the declustering ratio 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. This determines the parity overhead, $1/G$, and correspondingly, the fraction of storage available for 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. This section shows that declustering ratios larger than 0.25, which provide relatively low parity overhead, yield much of the performance benefits of the example in the last section. It also shows that in systems very sensitive to performance during failure recovery, the special case of parity declustering with G=2 (the minimal possible value) yields failure-recovery performance advantages unavailable in most other declustered organizations, at the cost of high parity overhead.

This section considers the same array size as the previous section, 40 disks, and reports on the performance of the arrays on the workload described in Table 2.4, using a fixed user access rate of 14 user I/Os per second per disk. We selected this rate because it is approximately the maximum that a RAID Level 5 array (α=1.0) can support during reconstruction. It causes the disks to be utilized at slightly less than 50% in the fault-free case.

The simulations reported here evaluate the arrays at α=1.0, α=0.75, α=0.5, α=0.25, and the 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. This is simply the application of the *reconstruct-write* technique described in Section 2.2.2.3.3.

The case *G*=2 is important because it is equivalent to disk mirroring, except that the layout distributes the backup copy of each disk over the other disks in the array instead of locating it 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 controller replaces the four accesses normally associated with a small-write operation 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]. The raidSim 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.

These optimizations can be significant in terms of performance, but they apply only in the *G*=2 and *G*=3 cases, which are expensive in terms of capacity overhead.

3.3.2.1. Fault-free performance

Figure 3.10 shows that the response time of a fault-free array is independent of α in all cases except *G*=2 and *G*=3, where the above-described optimizations can be applied.

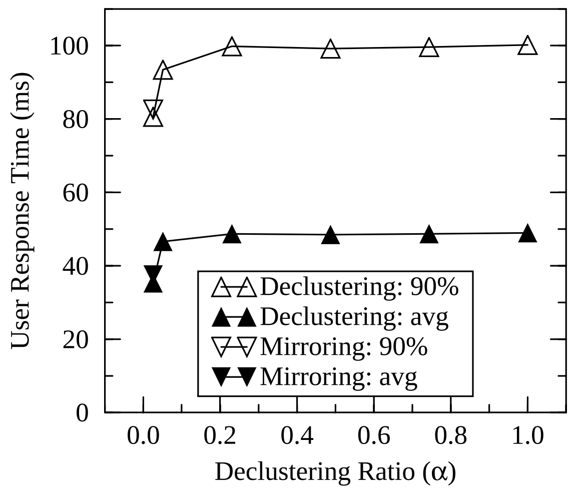


Figure 3.10: Varying declustering ratio: user response time in fault-free mode.

This figure confirms the result of Section 3.3.1.1 that declustering parity does not negatively effect fault-free performance. Similarly, declustered parity with *G*=2 performs essentially identically to mirroring. Figure 3.10 does not show a significant benefit for the three-access update when *G*=3 because the OLTP-like workload used in the simulations 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 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.

3.3.2.2. Degraded- and reconstruction-mode performance

Figure 3.11 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. Comparing Figure 3.11 to Figure 3.10 shows that the response times that occur at low 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, and thus the disk utilization remains at approximately 50%, the same level as in the fault-free case. However, when α=1.0 the degraded-mode utilization is close to 100% because extra I/O required to maintain operation in the pres-

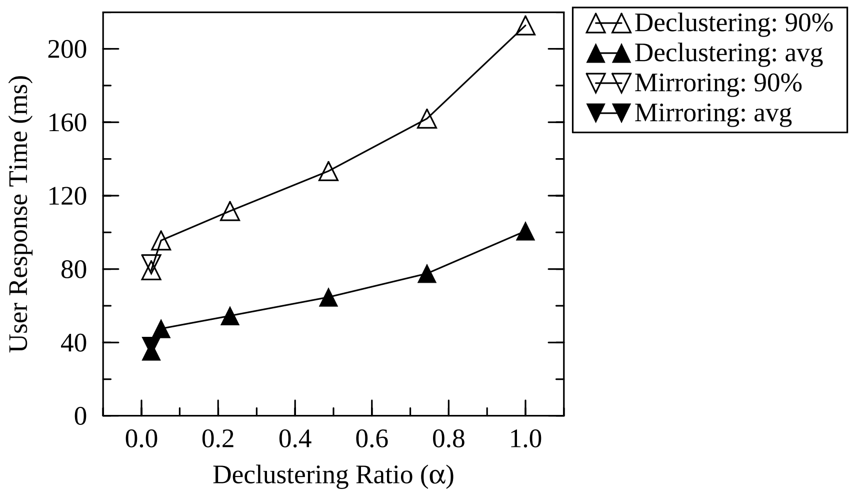


Figure 3.11: Varying declustering ratio: user response time in degraded mode.

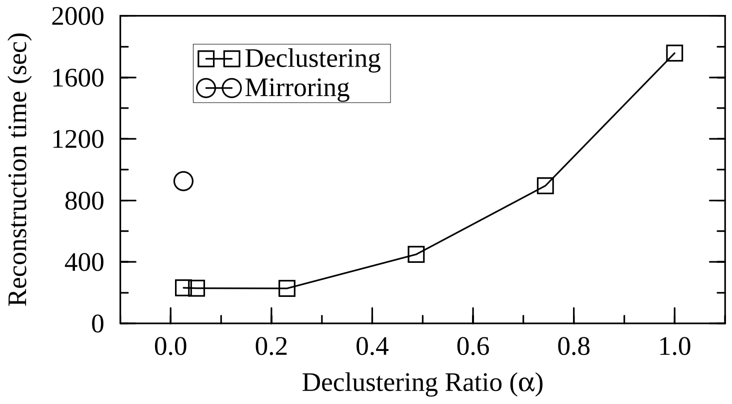


Figure 3.12: Varying declustering ratio: reconstruction time.

ence of failure. 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 3.12 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[8]. Since a high declustering ratio causes surviving disks to be saturated with work, recon-

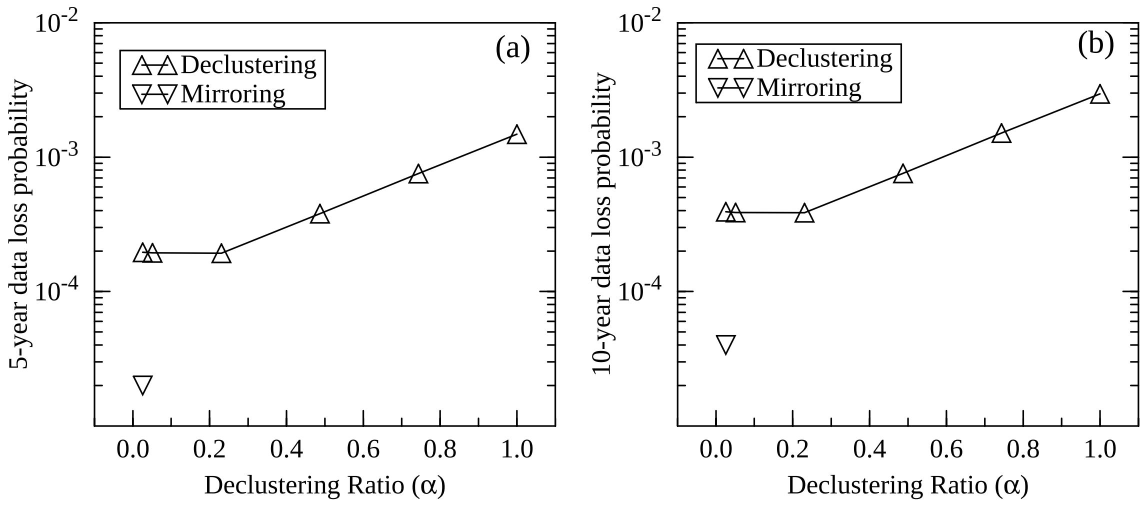


Figure 3.13: Varying declustering ratio: reliability.

The figures show the probability of data loss within (a) 5 years and (b) 10 years. Note that the Y-axis is log-scaled.

struction time falls off steeply with decreasing α, flattening out at the point where the 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.

3.3.2.3. High data reliability

Figure 3.13 shows the probability of losing data within 5 and 10 years due to a disk failure occurring while the reconstruction of another disk is ongoing (refer to Section 3.3.1.4). 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 3.12.

3.3.2.4. Summary

In contrast to parity declustered arrays with fixed declustering ratios determined by a

8. Chapter 5 investigates a technique to remove this bottleneck by distributing the capacity of spare disks throughout the array.

fair cost comparison to multi-group RAID Level 5 arrays in Section 3.3.1, this section examined the choices available if an array's declustering ratio is varied. It showed that the special case of parity declustering with *G*=2 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 reconstruction-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.

3.3.3. Improving fault-free performance by increasing disk utilization

In a continuous-operation system, it is necessary to guarantee that the failure of a disk causes no significant degradation in the throughput of the system, and causes only the minimum possible response time degradation. Since both throughput and response time are primarily functions of disk utilization, these guarantees are made by maintaining the fault-free disk utilization at a sufficiently low level such that the load increase experienced during a failure will not cause the utilizations to rise above some pre-defined boundary. The per-disk load increase experienced during a failure is less in a parity declustered array than in a RAID Level 5, and so the declustered array can maintain a higher level of fault-free disk utilization than can the RAID Level 5. This section derives and verifies a simple model for the factor by which the fault-free load in a declustered array can be increased over a RAID Level 5 array such that the degraded-mode disk utilization remains constant.

At first glance, it may seem that the fault-free load can increase in direct proportion to the decrease in α from 1.0. The model and simulations will show that while this is true for read operations, the presence of writes in the user workload forces the actual load increase factor to be slightly less. The section shows that at low declustering ratios, the fault-free disk utilization can be increased by a factor between 1.2 and 2.0, depending on the characteristics of the applied user workload, and that for read-dominated workloads, a typical load increase factor for low values of α is about 1.6.

3.3.3.1. A simple model for the load-increase factor

The derivation is similar to that of Ng and Mattson [Ng92b], and starts with a simple model for the degraded-mode disk utilization in a declustered array assuming a small-access model, that is, assuming that each user read or write addresses only one data unit.

When a disk array is fault-free, a user read operation invokes a single disk read on one drive, while a user write operation invokes a read-modify-write on the addressed data unit and then a read-modify-write on the corresponding parity unit. In degraded mode, the translation of user accesses into disk access is more complicated. The following description summarizes the specifications in Section 2.2.2.3.3. A read or write that does not access the failed disk at all operates as if the array were fault-free. A read of failed data causes all of the non-failed data and parity in the parity stripe to be read and XORed together to produce the failed unit. The controller services a write to failed data by reading all surviving units in the parity stripe, XORing them together with the new data to produce the new parity unit, and then writing the new parity unit to disk. Finally, a write to a data unit for which the parity unit has failed results in a single write to the data unit, since there is no parity to update.

Table 3.2a and Table 3.2b summarize the disk accesses that occur in degraded mode for user read and write operations, respectively. The tables should be read as follows: the first column gives the failure conditions, that is, whether the addressed data or parity unit resides on a failed disk or a surviving disk. The second column gives the percentage of all accesses that invoke the indicated failure condition, or, equivalently, the probability that the failure condition will occur on any given access. The third column gives the number and type of I/O operations that occur in that failure condition, and the fourth column gives the number of disks over which the operations specified in column three are distributed.

Defining λ as the total user access rate in accesses per second, w as the fraction of user accesses that are write operations, $T_{r/\mathrm{w}}$ as the average time (seconds) required to perform a disk read or write operation, T_{rmw} as the average time required to perform a disk read-modify-write, and recalling that $\alpha = (G\text{-}1)/(C\text{-}1)$, the following disk utilization model for declustered parity arrays in degraded mode is easily derived from Table 3.2:

$$U = w\left[\frac{\lambda\alpha}{C}T_{\mathrm{r/w}} + \frac{\lambda}{C\,(C-1)}T_{\mathrm{r/w}} + \left(\frac{C-2}{C}\right)\left(\frac{2\lambda}{C-1}\right)T_{rmw}\right] + (1-w)\left[\frac{\lambda}{C}T_{\mathrm{r/w}} + \frac{\lambda\alpha}{C}T_{\mathrm{r/w}}\right] \qquad (1)$$

Failure Conditions On Data Unit	Probability of Occurrence	Disk I/Os Invoked	No. of Disks Distributed Over
Surviving	(C-1)/C	1 Read	C-1
Failed	1/C	G-1 Reads	C-1

(a) User Read Operations

Failure Conditions		Probability of Occurrence	Disk I/Os Invoked	No. of Disks Distributed Over
Data Unit	Parity Unit			
Failed	Surviving	1/C	G-2 Reads 1 Write	C-1
Surviving	Failed	1/C	1 Write	C-1
Surviving	Surviving	(C-2)/C	2 RMWs	C-1

(b) User Write Operations

Table 3.2: I/O operations in degraded mode.

Part (a) shows the number and type of disk operations that occur upon each user read operation, and part (b) gives the same information for user write operations.

The first term is the disk utilization caused by user write operations, and the second term by user read operations. Within each term, each sub-term models the corresponding row in Table 3.2. Since the controller services a read-modify-write via a regular read, a rotation back around to the start of the data, and then a regular write, it's clear that $T_{rmw} = T_{r/w} + T_{rot}$, where T_{rot} is the time for one complete revolution. To simplify the above expression, define $T = T_{r/w}/T_{rot}$, and use the approximations $(C-2)/(C-1) \approx 1$ and $\alpha + 1/(C-1) \approx \alpha$. The latter approximation is justified by noting that when α is close to 1.0, 1/(C-1) is small by comparison, and when α is small, the term in which this expression appears has little impact on the overall expression for disk utilization. Applying these simplifications and expressing U as a function of α and λ yields:

$$U(\alpha, \lambda) = \frac{\lambda T_{rot}}{C}\left[wT\alpha + 2w(T+1) + T(1-w)(1+\alpha)\right] \qquad (2)$$

Recall that the goal of this analysis is to derive the factor by which the fault-free workload in a declustered parity array with some fixed declustering ratio α can be increased over that of a RAID Level 5 array, while still maintaining the same degraded-

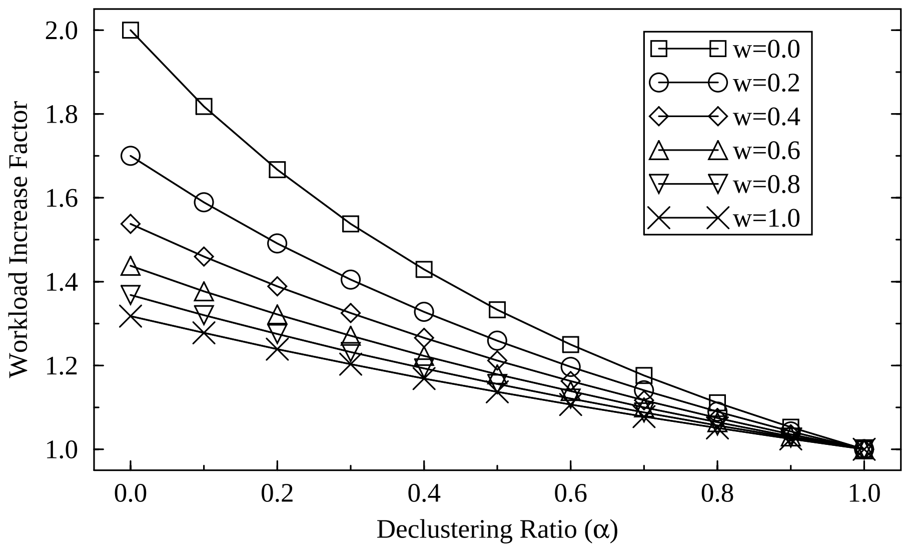


Figure 3.14: Fault-free workload increase factor versus the declustering ratio.

The workload increase factor for a declustered array is the amount by which the fault-free user workload can be increased over that of a RAID Level 5 array such that degraded-mode disk utilization will be constant. "w" is the fraction of all user accesses that are write operations. The plot assumes T = 7/4. Note that the Y-axis starts at 1.0 rather than 0.0.

mode disk utilization. Accordingly, we now set $U(1.0, \lambda_{RAID5}) = U(\alpha, \lambda_{decl})$, and solve the resultant equation for the ratio between λ_{decl} and λ_{RAID5}. After simplification this yields a workload increase factor of

$$\frac{\lambda_{decl}}{\lambda_{RAID5}} = \frac{T(w+2)+2w}{T(1+\alpha+w)+2w} \tag{3}$$

Figure 3.14 plots this function across the range of possible values of w and α, using $T = 7/4$, which is a reasonable value for a modern disk running a small-access dominated workload. When there are no writes in the workload ($w = 0$), the workload increase factor reduces to $2/(1 + \alpha)$, implying that $(1 + \alpha)\lambda_{decl} = 2\lambda_{RAID5}$. This is intuitive since for a 100% read workload, the RAID Level 5 disk utilization doubles in the presence of a failure, whereas the utilization in a declustered array increases by only a factor of α. This implies that in the absence of write operations, the fault-free user workload can be increased in direct proportion to the decrease in α from 1.0. However Figure 3.14 shows that in the presence of write operations, the fault-free user workload increase factor is

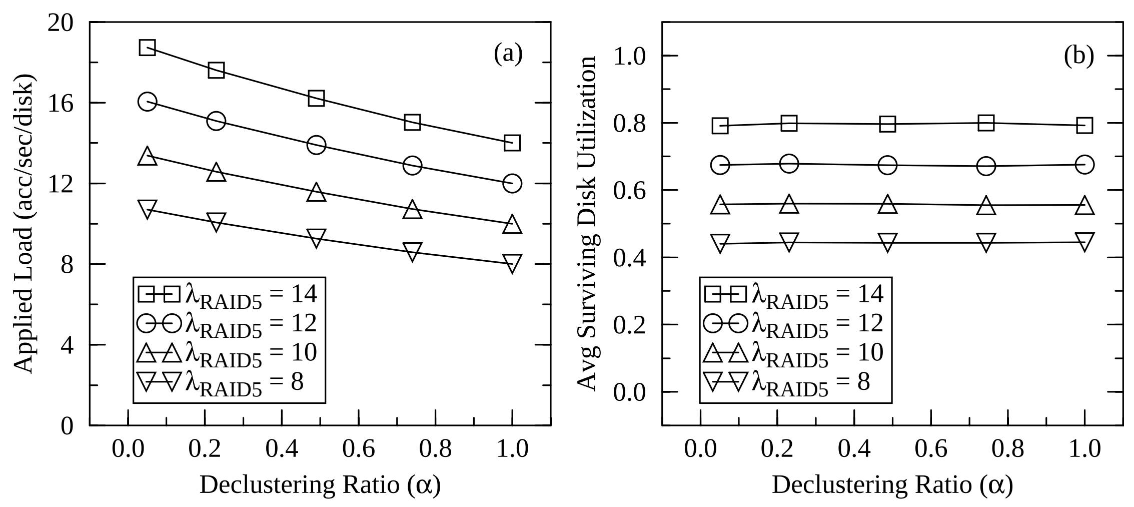


Figure 3.15: Scaling the applied workload with the declustering ratio.

The simulations scale the load applied to the array with the declustering ratio by multiplying a baseline access rate (λ_{RAID5}) by the workload increase factor using r=0.82 and T=7/4. This yields an average disk utilization in degraded mode that is constant with respect to the declustering ratio. Part (a) shows the applied load, and part (b) shows the resultant average surviving disk utilization (simulation results) in degraded mode versus the declustering ratio, for a 40-disk array.

smaller than this intuitive value of $2/(1 + \alpha)$. Referring back to Table 3.2, as the fault-free user workload is increased, the majority of the new accesses invoke two read-modify-write operations, and a small minority invoke either one or *G*-1 simple operations (reads and/or writes). Decreasing the value of α reduces the penalty of the latter type of write, but since the double read-modify-write operation is much more expensive than a simple read or write, the reduction in α fails to compensate for the increase in disk utilization caused by the increased number of read-modify-write operations. This limits the overall workload increase factor to a smaller value than would be expected.

3.3.3.2. Verification via simulation

This section demonstrates the accuracy of the above derivation via simulation. Figure 3.15 plots the applied workload and the degraded-mode disk utilization for the 40-disk example array described in Section 2.3.1, using a user access rate that scales with α according to the workload increase factor in Equation (3) above. The user workload applied was the one described in Section 2.3.3. It's clear that the disk utilization is constant across all values of α, except for the leftmost data point in the figure, where the utili-

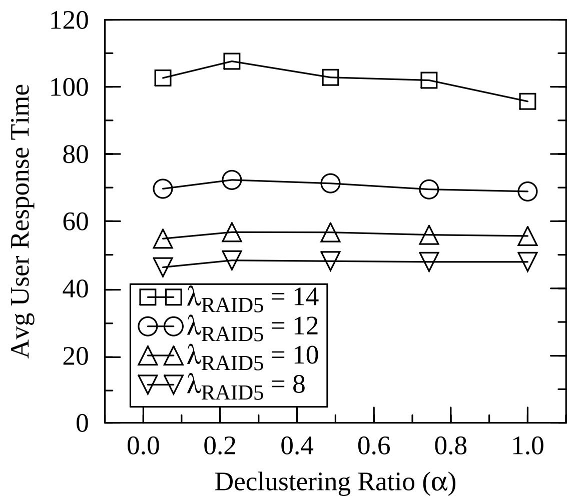


Figure 3.16: User response time using scaled user access rates.

The figure shows the average user response time in degraded mode versus the declustering ratio for a 40-disk array, using the scaled user access rates of Figure 3.15a.

zation is very slightly lower than predicted. As in the previous sections, this data point corresponds to the G=3 case, where the model derived from Table 3.2 is not strictly accurate due to differences in the handling of writes (refer to the description of the RAID Level 5 update techniques in Section 2.2.2.3.3).

Since the user response time in a disk array is primarily (but of course, not exclusively) a function of the disk utilization, the flat utilization plot in Figure 3.15b should imply that the average user response time in degraded mode should also be constant with respect to the declustering ratio. Figure 3.16 verifies that this is the case by plotting average user response time in degraded mode versus the declustering ratio, using the scaled user access rate shown in Figure 3.15a. At low and moderate disk utilizations, the response time is flat with respect to the declustering ratio, but at λ_{RAID5} = 14, the disks are near saturation, and so the response time becomes very sensitive to slight variations in the access rate. This explains the increasing nonuniformity of the upper lines in the figure. The slight dip in the response time at the G=3 point (α = 0.05) is more apparent in this figure than in Figure 3.15.

3.3.3.3. Summary

This section has shown that parity declustering can improve the fault-free array performance over RAID Level 5 by allowing the disks to be operated at significantly higher utilization while maintaining the same levels of throughput and responsiveness should a failure occur. This means that parity declustering allows a system with a given performance level to be implemented using fewer disks, and hence can reduce overall system cost. Of course, increasing the fault-free disk utilization in a disk array causes the fault-free response time to increase, and so a balance must be struck between fault-free responsiveness, responsiveness in degraded mode (that is, worst-case responsiveness), and array capacity overhead for parity.

Since the system designer has the freedom to choose the declustering ratio, declustering provides an additional degree of freedom in the design process. For example, if the system is to be highly responsive at all times, a designer might choose to configure the array with a low declustering ratio and operate the array at a low fault-free disk utilization. This assures that, should a failure occur, the performance degradation will be essentially unnoticeable to the users. If, however, the system specifications emphasize cost over responsiveness, then declustering allows the operation of the array at a high utilization without risking any throughput degradation due to saturation should a failure occur. Section 3.4 summarizes the process of system configuration.

3.3.4. Performance on non-OLTP workloads

The previous evaluations focused on a workload dominated by small, random accesses, since these are typical of the OLTP environment where parity declustering is most attractive. However, OLTP is not the only application area requiring high availability from the data storage subsystem, and so this section evaluates the performance of the organization under differing workload conditions. The first subsection characterizes performance with respect to user access size and read-to-write ratio, the second subsection investigates locality of reference, and the third evaluates performance using workloads traced from specific applications.

3.3.4.1. Performance versus user access size and read-write ratio

Section 3.3.1.1 showed that the fault-free performance of declustered parity redundant disk arrays is insensitive to the declustering ratio for workloads dominated by accesses that touch only a single data unit. This is not true for workloads containing a significant fraction of larger accesses, for three primary reasons. First, note that for any given large access, the maximum number of disks that can be used to service the access is equal to the access size divided by the size of the stripe unit, plus one disk if the access is not aligned to a stripe unit boundary. Since the layout does not meet criterion six (Section 3.2.2.3), large read accesses are not able to consistently utilize this maximum number of disks, and so the data transfer rate for these accesses is less than it would be in an array that meets criterion six. Second, the number of parity units dispersed in the array increases as the declustering ratio (α) decreases. Read accesses that are large enough to span multiple data units on any particular disk are forced to skip over these parity units while transferring the requested data. Since there is more parity at lower α, each disk spends more time skipping over these units, which reduces the data transfer bandwidth available to the users. The third aspect of declustered parity arrays that influences large-access performance is that reducing the value of G increases the fraction of write accesses to which the large-write optimization applies. The first two of these effects cause performance to degrade as the declustering ratio is reduced, while the third causes performance to be improved.

The interaction of these three effects is the primary topic of this section, and consequently the figure of merit that is used here is the data transfer bandwidth of the array. For the evaluations that follow, we set the concurrency of the workload to one, that is, the simulations consisted of a single process synchronously requesting large read and/or write accesses. This represents the worst case for the declustered arrays, since any higher concurrency would allow more disks to be utilized at any one time, and thus increase the cumulative large-access bandwidth available to the users.

Figure 3.17 plots the maximum data rate (total megabytes of user data moved per second) achieved in a 40 disk array as a function of the user access size, for several values of α. The left hand plot is for a 100% read workload, and the right-hand plot is for 100% writes. Note that the x-axis is log-scaled. The size of the stripe unit was fixed at 24 KB, and so the rightmost data point in each plot, corresponding to a user access size of 4 MB, accesses between 4 complete stripes (for G=40) and 85 complete stripes (for G=3).

For access sizes less than about half a megabyte, there is no significant performance difference between any of the configurations, but the curves begin to separate for access sizes above this value. The access size at which the curves begin to deviate from one

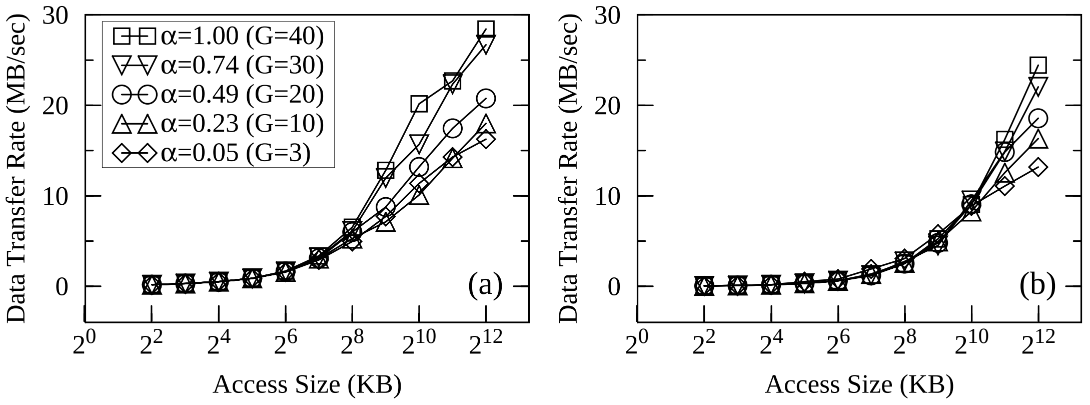


Figure 3.17: Fault-free data transfer rate versus access size.

Part (a) shows the response for a 100% read workload, and part (b) for a 100% write workload. The stripe unit size was 24 KB. Note that the x-axis is log-scaled. The small spike in the data transfer rate for α = 1.0 in part (a) at an access size of 2^{10} KB is due to fluctuations in the disk utilization due to the simulated access sizes not being exact multiples of the stripe unit size times the number of disks in the array.

another can, of course, be increased by increasing the size of the striping unit. Above the deviation point, the failure to meet criterion six and the necessity of skipping over more parity per access allows the higher-α configurations to transfer more data each second than those with a lower value of α. In the 100%-write workload, the deviation between the curves is less pronounced and occurs at a larger user access size (about 2 MB). This is because the improved write performance of smaller-α configurations compensates for the degraded transfer rates. Note that as the access size increases, the write-performance benefits of a smaller value of α diminish, but the associated transfer-rate penalties do not. Simulations for mixed read-write workloads exhibit the expected behavior: the deviation between the curves has magnitude and starting point intermediate to the values shown in Figure 3.17.

It's clear, then, that for low-concurrency workloads characterized by very large access sizes, declustered parity arrays do not perform as well as RAID Level 5 arrays, and that the degradation is worse for lower values of α. The next step toward understanding and addressing this problem is to investigate whether it is the failure to meet criterion six or the necessity of skipping over parity units that contributes most to this degradation. The two effects can be separated by looking at the disk utilizations; at low workload concurrency, the failure to meet criterion six causes disks to be underutilized, whereas the time

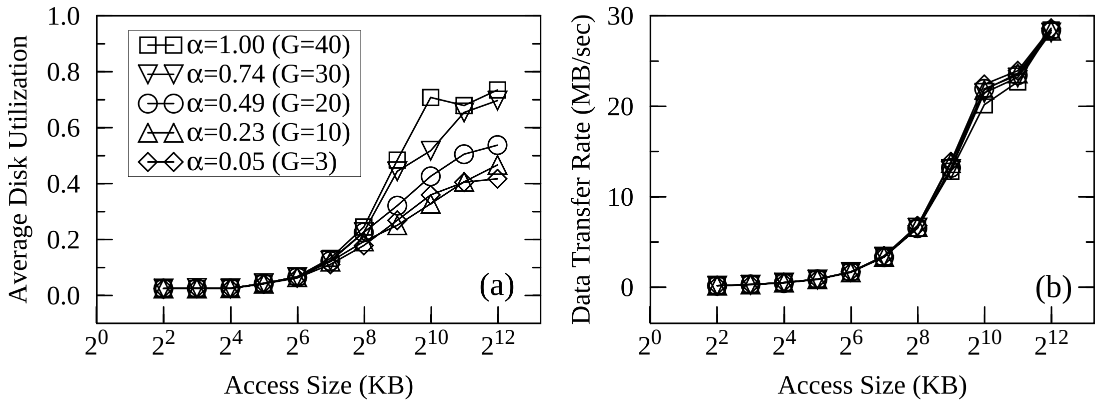


Figure 3.18: Disk utilization and normalized transfer rate in a fault-free array.

The applied workload was 100% reads. The normalized transfer rate in part (b) is equal to the transfer rate in Figure 3.17a multiplied by the ratio between the disk utilization for the given declustering ratio and the disk utilization for the RAID Level 5 case. The point of this plot is that all the lines are essentially co-incident, indicating that the failure to meet criterion six is the dominant factor by which large-access performance is lost in declustered parity arrays.

required to skip over parity units manifests as extra head positioning time, and hence affects data rate but not utilization.

Figure 3.18a shows the average disk utilization for a 100% read workload as a function of the access size[9]. From the significant deviations in utilization, it's clear that the failure to meet criterion six is the dominant component. To quantify this for the example array, Figure 3.18b plots the data transfer rates of Figure 3.18a scaled so as to normalize the disk utilization to the RAID Level 5 case. More specifically, if $D_\alpha(s)$ is the data transfer rate at access size *s* and declustering ratio α (Figure 3.17a), and $U_\alpha(s)$ is the corresponding disk utilization (Figure 3.18a), then the normalized data transfer rate (Figure 3.18b) is equal to $D_\alpha(s) \cdot U_{1.0}(s)/U_\alpha(s)$. The normalized transfer rate reveals, to a first-order approximation, what the data rate would be if the declustered layouts could meet criterion six and thus match the disk utilization obtained by the RAID Level 5 case. The point of the figure is that the normalized data rates are nearly identical, and so essentially all of the large-access performance degradation is due to the failure to meet criterion six. To some degree, this result is an artifact of the choice of striping unit to be exactly one

9. Since the left-symmetric layout is used for the $\alpha = 1.0$ (RAID Level 5) case, criterion six is met and so it appears at first glance that the disk utilization at $\alpha = 1.0$ should be 1.0 for all access sizes above $40 \cdot 24$K = 960 KB. The actual RAID Level 5 disk utilization is lower than this because the workload concurrency is 1 and none of the simulated access sizes are exact multiples of 960 KB.

disk track, which allows a parity unit skipped in time equal to two track skews. However, it's clear from the utilization plot that adherence to criterion six remains the dominant problem. Based on this result, Section 3.5.3 discusses a technique whereby the adherence to criterion six can be improved in declustered parity arrays.

3.3.4.2. Performance versus locality of reference

This section investigates whether the performance of declustered parity architectures is sensitive to locality of reference in the access stream. The conclusion drawn is that the performance of declustered arrays is not significantly sensitive to locality. To avoid bludgeoning the reader with still more performance plots, this section presents summary results only.

We simulated fault-free-, degraded-, and reconstruction-mode performance for the 40-disk array using an 80/20 read/write ratio and an access size of 4 KB, for two locality patterns. In the first pattern, accesses were uniform within the address space of the array. In the second, deliberately selected as an extreme case, 90% of all accesses were uniformly distributed within the first 10% of the array and the remainder were uniformly distributed over the entire array. The remaining workload parameters matched those in Table 2.4. Simulation results showed the following:

- User response time in all modes of operation improved by about 20% due to shorter average seek operations, with the degree of performance improvement independent of the declustering ratio, and
- Reconstruction time was essentially unaffected, since user locality does not significantly influence the average seek distance for a reconstruction access.

In the specific case of α=1.0 (RAID Level 5), the simulation results showed a few exceptions to the above:

- User response time in the degraded and reconstruction modes improved by more than 20%, that is, by more than the improvement when α was less than 1.0, because the uniform workload caused the array to be close to saturation in the presence of a failure, whereas the 90/10 workload did not, and
- Reconstruction time was significantly lower in the 90/10 workload, again due to the fact that the uniform workload causes the array to be on the brink of saturation in the presence of failure.

These results validate the conclusion that the performance of parity declustered disk arrays is not strongly dependent upon the degree of locality in the user workload. Put another way, any locality-related performance benefits that appear in non-declustered arrays will also appear in declustered arrays.

3.3.4.3. Performance on specific workloads

The previous sections analyzed the performance of parity declustering using synthetic workloads. This allowed specific performance issues to be isolated and clearly defined, but by the same token it glossed over many effects that may influence the performance of the array. The purpose of this section is to validate that the conclusions reached in the previous sections apply to more realistic workloads. The technique employed is to drive the simulator with reference traces taken from real applications. We made several attempts to acquire traces taken from real applications in commercial settings, but none were successful. We therefore collected traces locally.

Using the *dfstrace* [Kistler92] tracing package, we captured traces of the I/O behavior of several I/O-intensive applications running UNIX applications on a DECStation 5000 under the Mach operating system. This machine has only one disk, and so all the accesses are serialized in the trace. Further, the machine does not have sufficient resources (memory, memory bandwidth, I/O bus bandwidth, etc.) to run a highly concurrent I/O workload, and so the approach adopted to achieve a concurrent workload characteristic of a UNIX workstation environment was to trace a set of nine applications running in isolation, and then simulate all the traces running concurrently.

Although each process was traced independently, there can be a significant degree of concurrency in the actual workload applied to (and subsequently serialized by) the disk queue, which stems from three sources. First, if the application consists of a set of independent processes, or makes remote procedure calls to an asynchronous program such as a network file daemon, the separate processes can access the I/O system in parallel. Second, the operating system by default performs *single block readahead*, in which after each demand read operation it attempts to fetch the next sequential block into the buffer cache. This improves the overlap between processing and I/O in the common case where the application is reading a file sequentially. It can cause concurrent I/O because if a user request arrives while a readahead request is pending, the I/O concurrency will be at least two. Finally, and most significantly, every thirty seconds the operating system writes all dirty blocks in the buffer cache to disk, in order to guard against data loss due to crashes or power failures. All such blocks are queued for write asynchronously, that is, without waiting for each to complete before queuing the next, and this can cause the I/O concurrency to rise, momentarily, into the hundreds.

Since the goal of this study is to validate prior conclusions using these I/O workloads, it’s not adequate to run the serialized traces directly through the simulator, because the concurrency in the real workloads would cause them to perform substantially differently on a disk array than they perform on a single disk. In order to recapture the single-application workload concurrency, we pre-processed each of the nine independent traces prior to simulation as follows. We configured *dfstrace* to record the time that each I/O operation arrives at the buffer cache, as well as the time that each access starts and completes on the disk. The preprocessor assumes the disk queue to be empty at the start of the trace, and processes each record in the trace file sequentially. If, at the time an I/O passed through the buffer cache, there was another I/O pending (that is, another I/O has passed through the buffer cache layer but not yet completed disk service), the pre-processor assumed that the newly arrived I/O is asynchronous to the application, and marked the I/O as such in the trace file. As raidSim read each trace entry, it forked an independent process to handle each asynchronous I/O, but serialized the synchronous operations (those on the main access stream) on a per-process basis. The forked processes simply delayed for the inter-access time specified in the trace record, performed the indicated access, and then terminated. This preprocessing allowed the trace to reflect both intra- and inter-application concurrency in the I/O workload.

The applications, which we deliberately selected to stress the I/O system, were as follows:

- *Andrew*, the Andrew File System benchmark [Howard88],
- *Bigfile read*, a sequential read of a 20 MB data file into memory,
- *Compress*, the UNIX data compression program applied to a 19 MB input file and producing a 4.6 MB compressed file,
- *Uncompress*, run on the file generated in the *compress* trace,
- *Grep,* a text-searching program, applied to 1532 small files (e-mail messages) contained in 27 directories and totalling 3.7 MB,

- *Make*, the UNIX program build utility, building the *raidSim* simulator, which consists of 125 source files totalling about 25,000 lines of code,
- *Paging*, a 30-second trace of paging behavior invoked by forcing an application (*raidSim*) to allocate and use more memory than is available on the workstation,
- *Tar create*, the UNIX archiving program, archiving to the local disk a directory containing 257 files in 15 subdirectories, and totalling about 7 MB, and
- *Tar extract*, extracting (de-archiving) the same directory.

Table 3.3 lists some summary statistics about these applications. The table omits the

Trace	Number of Reads (%)	Number of Writes (%)	Avg Delay (ms)	Total MB Touched	Sequential Accs (%)	Mean/ Median/ 90% Seek Dist (Cyls)	% Async Accs
Andrew	120 (33%)	243 (66%)	60	1.04	45 (12%)	55/3/179	34
Bigfile read	2690 (98%)	42 (1%)	8	20.61	2423 (88%)	18/0/0	48
Compress	2458 (74%)	845 (25%)	29	23.10	1644 (49%)	161/0/642	22
Grep	1700 (91%)	148 (8%)	10	4.28	77 (4%)	27/0/58	9
Make	204 (18%)	913 (81%)	91	2.12	146 (13%)	86/1/455	27
Paging	1518 (56%)	1370 (43%)	590	12.29	1172 (40%)	32/0/171	62
Tar create	1248 (56%)	972 (43%)	22	14.13	123 (5%)	103/87/137	59
Tar extract	984 (30%)	2227 (69%)	20	13.75	10 (0%)	167/166/414	50
Uncompress	724 (22%)	2468 (77%)	73	23.10	1703 (53%)	96/0/406	74
Averages	**1294 (53%)**	**1025 (46%)**	**100**	**12.7**	**815 (29%)**	**83/28/273**	**43**

Table 3.3: Summary statistics on traces collected from a UNIX workstation.

For each of the traces, the table gives the number and percentage of read accesses, the number and percentage of write accesses, the average delay between "synchronous" accesses, the total number of distinct megabytes read or written by the trace, the number and percentage of accesses that were sequential, the mean, median, and ninetieth percentile seek distances for the synchronous accesses, and the percentage of all accesses that were determined to be asynchronous by the preprocessor. Note that the UNIX file system interleaves consecutive blocks on the disk [Leffler89], rather than laying them out completely sequentially, and so the "sequential accesses" column reports those accesses targeting a disk location one interleave factor (16 sectors in this case) away from the previous access, rather than immediately adjacent to the previous access.

average access sizes because, under Mach, all I/O accesses made by a user process are passed through the buffer cache software layer, which uses a block size of 8 KB, and so the accesses that arrive at the disk are at most 8 KB in size. Except for the *grep* trace, where the typical file size was small and there were many accesses to metadata, most accesses were 8 KB, and the rest were fairly evenly distributed between 1 and 7 KB. The buffer cache size on the workstation was set to 12 MB, and was purged of all information prior to starting each application, so that the traces would reflect “cold-cache” behavior. Note the severely skewed seek distributions: in the *bigfile read* trace, for example, the average seek distance was 16 cylinders, but over 90% of the accesses did not seek at all. This is due to the file system [Leffler89] allocation policy, which is effective at locating consecutive blocks of a file close together, and unrelated data far apart. Note also the high percentages of asynchronous accesses, indicating that there is significant I/O concurrency even in a single application.

In order to generate a workload sufficient to keep a large array busy, we combined together three copies of each of the above traces, adjusting the sector offsets within each trace so that each copy of each trace accessed a different region of the array. The resulted in a workload sufficient to keep a 20-disk array utilized at about 50% in the fault-free case, and so this is the array size used in the simulations that follow. Note that this differs from the previous sections, which used a 40-disk array. We assigned each trace to a different process within *raidSim*, resulting in 27 processes generating both synchronous and asynchronous accesses. The traces differ in length (total number of accesses), and so some processes finish their trace more rapidly than others. When a process completes a trace, *raidSim* restarts it at the beginning, and continues to restart it until all processes have completed their trace at least once. The reconstruction unit size was one track (24 KB), in accordance with the results of Section 3.5.1. In accordance with Chen’s results [Chen90b], the simulations set the stripe unit size to 24 KB to make the probability that an access will span a stripe unit boundary small, without clustering too much sequential data on a single disk.

Figure 3.19 shows the results of this study. Part (a) shows the reconstruction time, and part (b) shows the average and 90th percentile user response time in the fault-free, degraded, and reconstruction modes. These results agree well with the previous simulations that used synthetic models. The remainder of this section compares these results to the previous ones.

First, declustering has little effect on fault-free performance, except that there is a

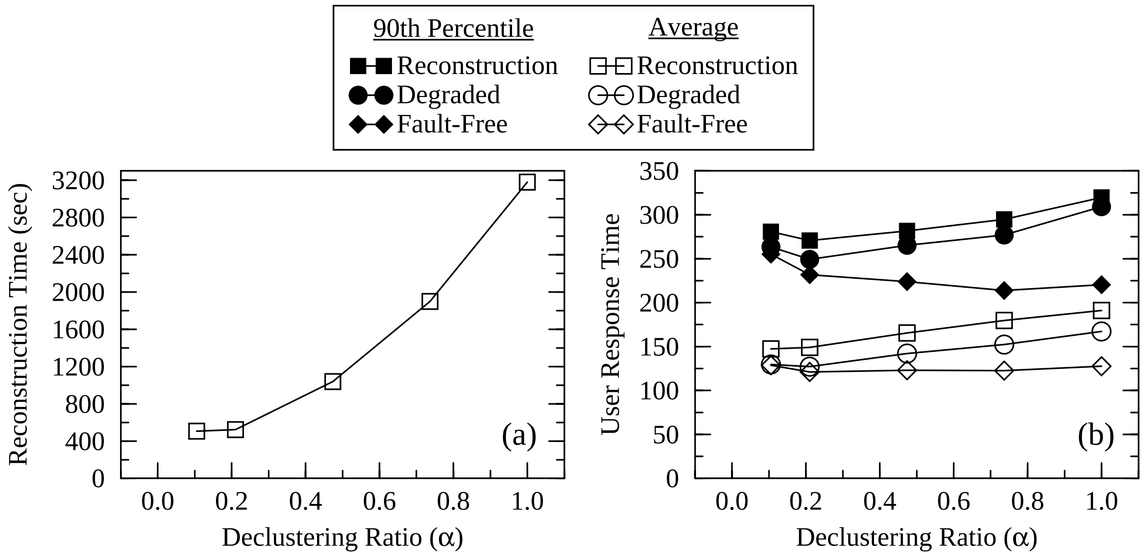


Figure 3.19: Parity-declustered array performance under a UNIX workload.

Part (a) shows reconstruction time, and part (b) shows average and 90th percentile user response time in the three different modes of operation (b), for a 20-disk array running the UNIX file system traces.

slight increase in 90th percentile response time as α is reduced to 0.2 ($G = 5$), and then a marked increase at $\alpha = 0.1$ ($G = 3$). As α is decreased to 0.2, 90th percentile user response time increases from 220 ms to 231 ms, which is approximately 5%. This is not significant in that it is close to being within the error bars of the plot[10]. The significant increase in 90th percentile response time at $G = 3$ is because of the different mechanism for handling user write operations in this case (refer to the opening remarks in Section 3.3.2); when $G = 3$ there are only two data units in a parity stripe, and so the controller can service a write by directly over-writing the old data with the new, reading the data unit that is *not* being written, XORing this unit with the newly-written unit, and writing the result to the parity unit. In this case, the new-data write and the old-data read occur concurrently, whereas with $G > 3$, the two read-modify-write operations used to service the request are serialized. This causes the increase in 90th percentile response time by increasing the average queueing time for an access. This optimization also causes the trace to be retired faster; in other words, the throughput was higher at G=3 than at other values. Except for these effects, the plot of fault-free response time versus declustering ratio is flat, which

10. Recall from Section 2.3.1 that in fault-free mode, the simulations were run until the 95% confidence interval on user response time had fallen to less than 3% of the mean.

agrees with the results of Section 3.3.1.1.

Second, declustering reduces the degraded-mode response time significantly; average user response time was 167 ms at $\alpha = 1.0$, but only 129 ms at $\alpha = 0.1$, which corresponds to a 23% decrease. This improvement is smaller than was observed in Section 3.3.2.2 due to the higher percentage of write operations in the traced workloads. Recall from Section 2.1.4 that the failure-induced workload increase experienced by a disk array decreases in severity as the write fraction increases.

Finally, the reconstruction-mode plots agree well with those in Section 3.3.2.2: reducing the declustering ratio to 0.2 improves reconstruction time by about a factor of 6 (3200 seconds down to 510 seconds), and causes the reconstruction-mode response time to be very nearly the same as the response time when the array is fault-free.

3.3.4.4. Summary of non-OLTP performance evaluations

The primary conclusion to be drawn from this section is that the benefits of declustering apply to other workloads as well as OLTP, but several performance effects need to be considered. First, declustering negatively impacted the fault-free read-performance of the array on large accesses with low concurrency. This effect only occurred for very large accesses. Additionally, Section 3.5.3 will show that it is often possible to recapture a significant fraction of this lost performance. Second, the degradation in the fault-free performance of large write operations was less severe than the read degradation, due to the fact that the large-write optimization applies at smaller access sizes in a declustered array. Third, locality of reference was shown to have very little impact on any of the conclusions about declustering. Finally, evaluating declustering using real workloads largely validated the previous conclusions.

3.3.5. Overall performance evaluation summary

The comprehensive set of performance evaluations presented in this section demonstrated four primary facts about declustering. First, at moderate to high user accesses rates, the declustering approach is superior to the multiple-group RAID Level 5 approach for every figure of merit. Second, declustering offers order-of-magnitude improvements in reconstruction time, simultaneous with large improvements in user response time during recovery, at the cost of increased capacity overhead for parity. Third, a declustered array can be operated at substantially higher fault-free utilization than can a multiple-group RAID Level 5 array without risking the unbounded queueing delays caused by saturation should a failure occur. Finally, the benefits of declustering apply to non-OLTP workloads as well.

3.4. System configuration

Section 3.3.1 showed that for arrays of up to about 40 disks, a single declustered group organization yields better performance in the presence of failure than an organization that separates disks into a set of independent RAID Level 5 groups. This section revisits 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. Of particular interest is the question of 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}. The section shows 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 [Schulze89, Gibson93]. For example, if the bus-connected disk array architecture shown in Figure 2.5 provides only one path to each disk but shares this path over multiple disks, the failure of a path renders multiple disks unavailable, although not damaged, for long periods of time. This 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 criterion one in Section 3.2.1, these disks may not reside in the same redundancy group. A cost effective way to do this is to organize each rank[11] of drives as an independent parity group. It follows 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

11. A disk array typically contains several disk busses, each of which is shared by several disks. A “rank” of drives is defined as a set of disks, one from each bus, that have a common bus ID. Organizing each rank as a distinct parity group assures that no two disks in a group will share a bus.

case, the results of Section 3.3.1 are directly applicable: the group size (C) should be maximized to minimize performance degradation during failure recovery. Further, since the number of disks in a rank is limited to a relatively small number, an adequately small block design always exists, and so there is no need to use any of the alternative layout strategies.

In disk arrays with sufficient redundancy in non-disk components, such as the fully duplicated versions of Figure 2.5, 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 tradeoffs are between cost, data reliability, capacity (parity overhead), fault-free performance, and on-line failure recovery performance. Remaining with the OLTP-like model of such an array's workload, the following discussion assumes that the goal of any configuration is to achieve the lowest cost array which meets specific I/O throughput and response time requirements.

We derive a set of constraint equations and simple analytical models from which it is possible to compute values for C, G, and N_{groups}. We assume that a designer has specified four primary constraints on the design of the array:

1. a minimum acceptable I/O rate, which must be maintained in both fault-free and degraded mode,
2. a maximum acceptable average[12] user response time in fault-free mode,
3. a maximum acceptable average user response time in degraded mode, and
4. a minimum acceptable mean time to data loss (MTTDL).

We further assume that a system designer has selected a particular disk from which to construct the array; if this is not the case, the analysis below can be repeated for each disk under consideration.

In some applications, there may be additional constraints on the design of the array, such as a minimum capacity requirement, or a maximum acceptable parity group size. For this reason, the configuration process presented below should be viewed as a baseline approach which can be augmented or modified to meet the specific goals of an implemen-

12. Constraints 2 and 3 can be easily changed to use 90th percentile response time instead of an average value. This involves changing only a few terms in the response-time equations.

tation.

Symbol	Type	Description
N_{groups}	Result	Number of independent groups in the array.
N_{tot}	Result	Total number of disks = $N_{groups} \cdot C$
C	Result	Number of disks per group.
G	Result	Number of units per parity stripe.
λ	Constraint	Min acceptable user access rate (I/Os/second).
T_{ff}	Constraint	Max acceptable average response time, fault-free (sec).
T_{deg}	Constraint	Max acceptable average response time, degraded (sec).
MTTDL	Constraint	Min acceptable mean time to data loss (sec).
B_d	Specification	Single-disk capacity (MB).
r	Specification	Read fraction in user workload.
$MTTF_d$	Specification	MTTF of component disk (sec).
T_S, T_R, T_X	Specification	Expected disk seek, rotation, and transfer time (sec).
R_X	Specification	Max disk transfer rate (MB/sec).
α	Term	Declustering ratio ((G-1)/(C-1)).
$f(r,\alpha)$	Term	Failure-induced user workload increase function.
μ_d	Term	Disk service rate at 100% utilization ($1/(T_S+T_R+T_X)$).
T_{recon}	Term	Reconstruction time (sec).
$U_{fault\text{-}free}$	Term	Average fault-free disk utilization.
U_{deg}	Term	Average degraded-mode disk utilization.
U_{repl}	Term	Average recon-mode utilization on replacement disk.
U_{surv}	Term	Average recon-mode utilization on surviving disk. A synonym for U_{deg}, used in a different context.
w	Term	Write fraction in user workload (1-*r*).

Table 3.4: Symbols used in the configuration equations.

"Results" are values computed by the configuration model. "Constraints" are system requirements. "Specifications" are system parameters supplied by the designer. "Terms" are intermediate values computed in the equations.

Table 3.4 presents the parameters used in deriving the configuration constraint equations. First, we assume that each user write operation translates into four disk operations, so the disk access rate corresponding to user access rate λ is $\lambda(r+4(1-r)) = \lambda(4-3r)$. In the presence of failure, this access rate is increased by a factor of $f(r,\alpha)$, which will be derived below. This workload must be serviced by the surviving set of $N_{groups} \cdot C-1$ disks, each of which can service a maximum of μ_d I/Os per second, and operates at utilization U_{deg}:

$$\lambda \cdot (4-3r) \cdot f(r,\alpha) \le U_{deg} \cdot (N_{groups} \cdot C - 1) \cdot \mu_d \qquad (1)$$

To derive an equation for the second constraint, we model each disk as an M/M/1 queue [Jain91]. Such a queue has response time $1/(\mu \cdot (1-U))$, where μ is the service rate and U the utilization. In order to map this disk response time back to user response time, we make the very pessimistic assumption that each of the four I/O operations associated with a user write operation occurs sequentially, and the write is not considered complete until all four have completed. This makes the user read response time equal to the disk response time, and the user write response time equal to four times the disk response time. The average user response time is therefore $(r+4w) = (3-4r)$ times as long as the disk response time. The second constraint is therefore:

$$\frac{3-4r}{\mu_d(1-U_{fault\text{-}free})} \leq T_{ff} \tag{2}$$

Similarly the third constraint is modeled by:

$$\frac{3-4r}{\mu_d(1-U_{deg})} \leq T_{deg} \tag{3}$$

We assume that user accesses are given priority over reconstruction accesses, and that reconstruction accesses are perfectly preemptible. This lets us use T_{deg} as the average user response time in both degraded mode and in reconstruction mode.

The fourth constraint is modeled by the reliability equation presented in Section 3.3.1.4, using the reconstruction time as the repair time:

$$\frac{MTTF_d^2}{N_{groups} \cdot C \cdot (C-1) \cdot T_{recon}} \geq MTTDL \tag{4}$$

Equation (4) requires an estimate of the reconstruction time for a particular configuration. For generality, we use a simple analytical model to derive this estimate; more accurate results can be obtained via detailed simulation, as is done in the rest of this dissertation[13]. The reconstruction time model works by computing the expected disk utilization caused by user requests at a given access rate, for one surviving disk and the for the replacement disk, and then assuming that all the disk time not consumed by user accesses is used to effect reconstruction. First, considering only a surviving disk serving a user

13. Our modified version of the *raidSim* simulator described in Section 2.3.2 is available via anonymous ftp from the machine *ftp.cs.cmu.edu*, internet address 128.2.206.173, in the directory *project/nectar-io*.

access rate of λ, disk requests arrive at a rate of $(\lambda/(N_{groups}\cdot C))\cdot(4\text{-}3r)\cdot f(r,\alpha)$, and are serviced at rate $1/(T_S+T_R+T_X)$. Thus the surviving disk utilization due to user-invoked accesses is:

$$U_{surv} = \frac{\lambda}{N_{groups}C} \cdot (4-3r) \cdot f(r,\alpha) \cdot (T_S+T_R+T_X) \tag{5}$$

If the disk queue is empty of user requests when a user request arrives, the new request incurs an average seek (T_S), an average rotational latency (T_R), an average transfer time (T_X), an average seek back to the reconstruction point (T_S), and an average rotational latency to initiate the next reconstruction access (T_R). If, however, the user request arrives while another user request is in-service or queued, the new request does not incur the latter two components, because no seek back to the reconstruction point is invoked between two user accesses. At a given disk utilization U, the probability that an M/M/1 queuing system is empty is 1-U [Jain91], and so the total utilization consumed by all activity other than the transfer of reconstruction data is:

$$U_{tot} = U_{surv} + (1-U_{surv})(T_S+T_R) \tag{6}$$

Subtracting the above value from unity yields the utilization available for reconstruction transfer, which we assume occurs at the drive bandwidth of R_X. The fraction of each surviving disk that must be read to effect reconstruction is α, and so the time required to read all the necessary data from a surviving drive is

$$T_{read} = \frac{B_d \cdot \alpha}{R_X(1-U_{tot})} \tag{7}$$

If the surviving disks represent the reconstruction bottleneck, then Equation (7) represents a good estimate of the reconstruction time. If, however, the replacement disk represents the reconstruction bottleneck, then the above model will be highly optimistic. To resolve this, we perform a similar computation for the total time taken to write the entire contents of the replacement disk. Since user writes to previously-reconstructed data must be serviced by the replacement disk, we assume that the only user-induced workload that the replacement disk observes is from half of all user write operations invoking two operations (a pre-read and a write) on the replacement disk[14]. Therefore

14. Section 4.4.1 (page 135) discusses fully the workload that the replacement disk might see under various conditions. We assume the simplest model here.

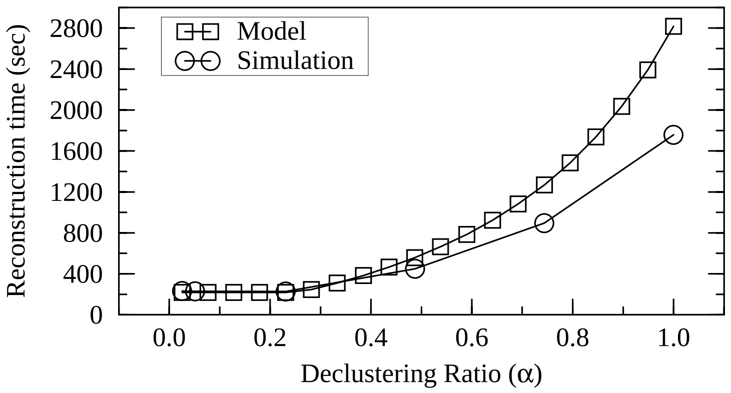


Figure 3.20: Comparing the reconstruction model to simulation data.

$U_{repl} = (\lambda \cdot w)/(N_{groups} \cdot C \cdot \mu_d)$. Using exactly the same strategy as was used to derive Equation (6), the time required to write the replacement drive is:

$$T_{write} = \frac{B_d}{R_X\left(1 - \frac{\lambda w}{N_{groups} C}\left(T_S + T_R + T_X + (1 - U_{repl})(T_S + T_R)\right)\right)} \qquad (8)$$

Reconstruction time is therefore $T_{recon} = \mathrm{MAX}(T_{read}, T_{write})$. Figure 3.20 plots this function versus the declustering ratio using the same configuration and workload as was used to derive Figure 3.12. The data from Figure 3.12 is included for comparison. The figure shows acceptable although imperfect agreement between this very simple model and the simulations.

We derive the function $f(r,\alpha)$ representing the degraded-mode workload increase on the surviving disk from the model in Section 3.3.3.1, with the small modification that we assume for simplicity that fault-free user writes require four I/Os instead of two atomic read-modify-write operations. We derive the expected surviving-disk utilizations in degraded- and fault-free mode, and then take the ratio to find the load increase factor. Using $T_{acc} = T_S + T_R + T_X$, the utilizations are:

$$U_{fault\text{-}free} = \frac{\lambda(4 - 3r)}{C} T_{acc}$$
$$U_{deg} = w\left[\frac{\lambda\alpha}{C} + \frac{\lambda}{C(C-1)} + \frac{4\lambda(C-2)}{C(C-1)}\right]T_{acc} + r\left[\frac{\lambda}{C} + \frac{\lambda\alpha}{C}\right]T_{acc} \qquad (9)$$

The latter of these two simplifies to $w(\alpha+4) + r(1+\alpha)$, where we use the approximation $(4C\text{-}7)/(C\text{-}1) \approx 4$. Note that Equation (9) makes no mention of N_{groups}; it implicitly assumes that there is only one group in the array. This represents a worst-case assumption that greatly simplifies the model. If there is more than one group in the array, performance will be better than the values computed above because any access that does not touch the affected group will not incur any performance penalty at all. Dividing the array into multiple groups will not cause a bottleneck to form in the array because the declustering ratio (α) will be computed to assure adherence to the specified throughput and response time in the worst case.

From Equation (9), derive $f(r,\alpha)$ as:

$$f(r,\alpha) = \frac{U_{deg}}{U_{fault\text{-}free}} = \frac{\alpha - 3r + 4}{4 - 3r} \quad (10)$$

Equations (1) through (10) define all the terms necessary to derive the values of N_{groups}, C, and G. Starting with Equation (2), compute the average fault-free disk utilization as $U_{fault\text{-}free} = 1 - 1/(\mu_d \cdot T_{ff})$. Similarly use Equation (3) to find the average degraded-mode disk utilization as $U_{deg} = 1 - 1/(\mu_d \cdot T_{deg})$. Using these values and the formula for $f(r,\alpha)$, derive the declustering ratio as:

$$\alpha = (4 - 3r)\left(\frac{U_{deg}}{U_{fault\text{-}free}}\right) + 3r - 4 \quad (11)$$

Using U_{deg} and α, derive the total number of disks in the system ($N_{tot} = N_{groups} \cdot C$) from Equation (1):

$$N_{tot} = \frac{\lambda (4 - 3r) f(r,\alpha)}{U_{deg} \cdot \mu_d} + 1 \quad (12)$$

At this point, T_{recon} is computable from known values. Using T_{recon} as the mean time to repair, compute C from Equation (4) as:

$$C = \frac{MTTF_d^2}{N_{tot} \cdot MTTDL \cdot T_{recon}} + 1 \quad (13)$$

The value of C derived in Equation (13) may be very large. It may even be larger than N_{tot}, which is of course a contradiction. This is because the target array reliability (MTTDL) may be lenient enough to place essentially no restrictions on C. Therefore, after computing the value of C in Equation (13), C should be reduced to at most the value

($N_{groups} \cdot C$), and can be reduced further if required to find a block design. Since G remains to be calculated, this will not affect any of the previous constraint computations.

N_{groups} is easily computed from the values of C and ($N_{groups} \cdot C$). Similarly G can now be computed as $\alpha \cdot (C\text{-}1) + 1$. This yields all the configuration parameters.

At this point it is necessary to evaluate any other constraints that apply to a particular design. For example, the total user data capacity of the array is easily computed as $(N_{groups} \cdot C \cdot B_d) \cdot (G\text{-}1)/G$. If this capacity is not adequate, the analysis can be re-done using a different (larger) component disk, more disks, a larger value for C (which will allow a larger value of G), or a less stringent requirement on the degraded-mode workload increase.

The configuration methodology presented here can also be used to revisit the question presented in Section 3.2.3. That section discussed selecting between a declustered parity layout based on balanced incomplete block designs and any of the alternative layouts. Pessimistically, if a declustered parity group size exceeds about 40 disks, it cannot be guaranteed that a sufficiently small block design exists for all possible values of G; for such a guarantee, Schwabe and Sutherland's approximately balanced designs [Schwabe94], or Merchant and Yu's random permutations layout [Merchant92a] can be used. The primary conditions under which these mechanisms become necessary are when the design mandates both very high failure-recovery performance (low declustering ratio) and very low capacity overhead (high G). It's important to note once again that acceptably-small block designs exist for many combinations of C and G beyond 40 disks; the only caveat is that such designs are not known for *every* possible combination.

3.5. Optimizations and improvements

This section describes and evaluates a set of variations that can be applied to the layout strategy to improve specific aspects of its performance.

3.5.1. Optimizing the reconstruction unit size

The rate at which a disk drive is able to read or write data increases with the access

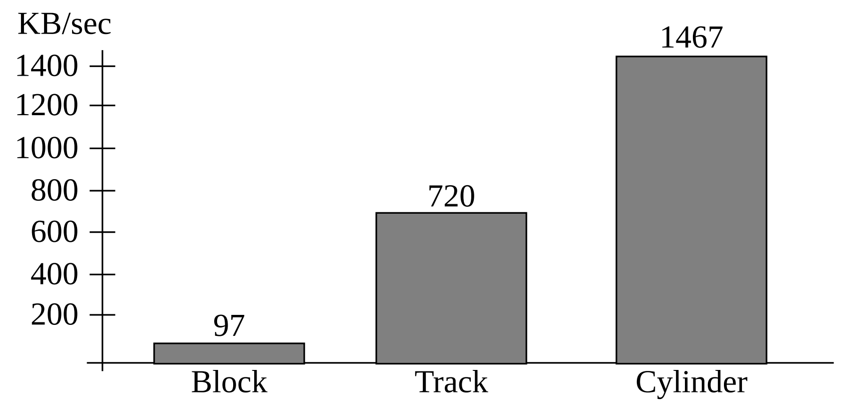


Figure 3.21: Single-disk I/O bandwidth versus access size.

The figures shows the peak I/O bandwidth (total kilobytes read or written per second) for an IBM Model 0661 (Lightning) drive, when the access size is one block (2KB), one track (24 KB), and one cylinder (336 KB).

size. This is illustrated in Figure 3.21, which shows the total kilobytes per second that can be read from or written to an IBM Lightning drive (refer to Table 2.3) using random accesses of size equal to one block (2 KB), one track (24 KB), and one cylinder (336 KB). The figure shows that track accesses move data about ten times faster than block accesses, and cylinder accesses about twice as fast as track accesses.

This implies that it might be possible to speed up reconstruction by increasing the size of the accesses used by the reconstruction process. In order to maximize fault-free performance, the size of a data unit should be selected according to the characteristics of the expected workload [Chen90b] rather than the characteristics of the reconstruction process, and so it’s desirable that the reconstruction unit size be selectable independently of the data unit size. The block-design based layout described above requires a simple modification to support this decoupling. This section describes this modification, investigates the sensitivity of reconstruction-mode performance to the size of the reconstruction unit, and then describes a technique for determining the optimal reconstruction unit.

3.5.1.1. Layout modification

Referring back to Figure 3.3, assume that the reconstruction unit is four times as large as the data unit, and that disk number 1 has failed. If a 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 1. In general, since the units

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

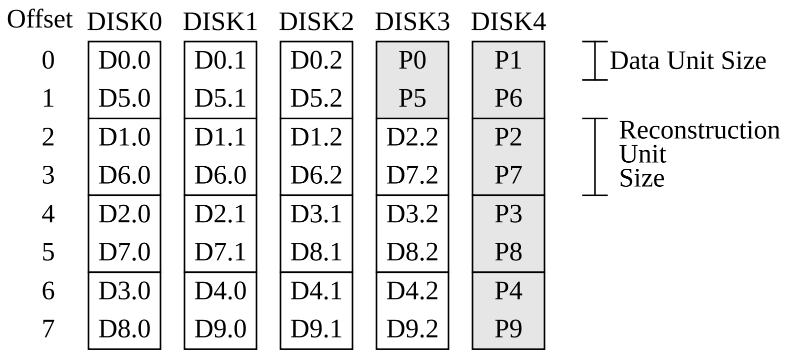
Figure 3.22: Doubling the size of the reconstruction unit.

The figures shows the first block design table for a layout in which the size of the reconstruction unit has been doubled by packing two data or parity units into each reconstruction unit.

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. Note that this problem does not occur in RAID Level 5 arrays because every unit on every surviving disk must be read at some point in time in order to reconstruct a failed drive.

This problem can be eliminated by repeating the tuple assignment pattern so as to pack multiple data stripe units into a single reconstruction unit. This modified layout is illustrated in Figure 3.22, which shows a layout where the reconstruction unit size is twice the data unit size, using the same block design (Table 3.1) as used in Figure 3.3. While the layout of Figure 3.3 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. Using Figure 3.22 as an example, parity stripes zero and one are laid out using the first tuple in Table 3.1, parity stripes two and three using the second tuple, and so on.

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. However, this layout has a drawback in that it causes relatively large regions of sequential user data to be clustered on a small number of drives.



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 3.23: Striping sequential units across parity stripes.

The figure shows the first block design table for a layout where the reconstruction unit size is twice the data unit size, and where the problem of excessive clustering of sequential user data on small groups of drives has been fixed by striping sequential data units across tuples.

Note, for example, that in Figure 3.22 the first twelve user data units, which map to the units labelled *D0.0* through *D3.2*, all reside on disks 0, 1, and 2. This problem of course gets worse as the size of the reconstruction unit increases with respect to the size of the data unit, and as the number of units in one parity stripe (G) decreases with respect to the number of disks in the array (C).

The layout in Figure 3.23 addresses this problem by striping data units across reconstruction units, instead of filling each reconstruction unit with data units before switching to the next. This means that stripe zero occupies the first slot in tuple 0, stripe one occupies the first slot in tuple 1, and so on up to stripe four, which occupies the first slot in tuple 4. Stripe five then occupies the second slot in tuple 0, stripe 6 the second slot in tuple 1, and so on to the end of the table. This avoids excessive clustering of consecutive user data units onto small sets of disks because it restores the condition that successive parity stripes are laid out using successive tuples.

3.5.1.2. Evaluating the benefits of larger reconstruction units

Figure 3.24 plots the reconstruction time and user response time during reconstruction versus the declustering ratio for a 40-disk array, using reconstruction units of size equal to one track (24 KB), half a cylinder (seven tracks), and one full cylinder (14 tracks). The figure does not consider reconstruction units smaller than a track because of the strongly diminishing efficiency of the drives for small accesses, and because it is desirable to take advantage of the zero-latency read optimization (refer to page 15 in Section 2.2.1) if the disks support it, although the IBM 0661 (Lightning) drives simulated in this study do not. The user workload was again the one described in Table 2.4, and the user access rate was 14 accesses per second per disk, approximately the maximum supportable when $\alpha = 1.0$.

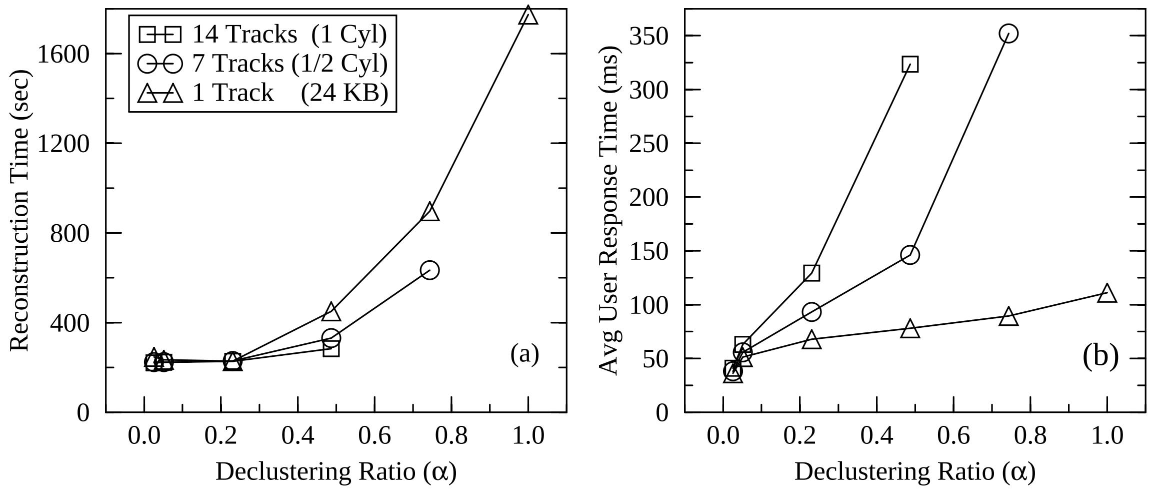


Figure 3.24: Reconstruction performance versus reconstruction unit size.

The figure shows reconstruction time (a) and average user response time during reconstruction (b) for various sizes of the reconstruction unit.

The figure shows that using a larger reconstruction unit improves reconstruction time by up to about 25% by utilizing the disks more efficiently, but also degrades user response time, in many cases quite severely, by monopolizing the disk arms for long periods of time. Since large reconstruction accesses are non-preemptible and take a long time to complete, user accesses tend to queue behind them, thereby increasing the observed response time. At higher declustering ratio, using the larger reconstruction units consumes so much bandwidth that the array fails to support the user access rate of 14 accesses per second per disk, as indicated by the missing points in the figure. This occurs despite the fact that user accesses are given strict priority over reconstruction accesses, because once a reconstruction access is initiated, many user accesses end up queueing behind it for long periods of time.

The obvious question that arises from Figure 3.24 is how much user response time degradation is acceptable to improve reconstruction time. The next section proposes a metric to answer this question, and finds that the answer is actually independent of the declustering ratio.

3.5.1.3. Determining the optimal reconstruction unit

To quantify the trade-off between improved reconstruction time and degraded user response time, Figure 3.25 plots the cumulative response time degradation during reconstruction versus the declustering ratio for the array and workload described above. The cumulative degradation is the product of the reconstruction time (Figure 3.24a) and the increase in average user response time (Figure 3.24b) 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 at any declustering ratio. This demonstrates that in systems that mandate a minimum level of responsiveness, the size of the reconstruction unit can not be selected strictly for maximum disk bandwidth.

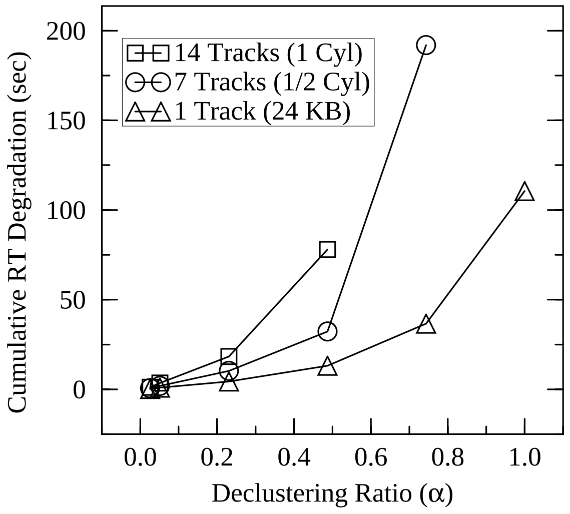


Figure 3.25: Cumulative response time degradation during reconstruction.

$$CumDeg = (RespTime_{recon} - RespTime_{fault\text{-}free}) * ReconTime$$

A reconstruction unit that is *n* tracks in size takes *n* full disk revolutions plus the second-order component of *n* track and/or cylinder skews to transfer. Therefore for a fixed disk revolution rate and reconstruction unit size, the time that user requests block waiting for reconstruction requests to complete will be largely independent of the declustering ratio, array size, and other array parameters. The conclusion from this is that for most arrays, track-sized reconstruction units are the most appropriate.

3.5.2. Compacting the full block design table

Recall from Section 3.2.2 that the “full table depth” (the number of units assigned to each disk in one full table) should be minimized for three reasons. First, if the depth exceeds the number of units on a single disk, then the block-design-based layout strategy will fail to balance parity over the array, and so will fail to balance the fault-free update workload. Second, recall that in order to utilize the entire capacity of each disk in the array, the last full table in the array may be incomplete, that is, may contain fewer than k block design tables. If the full table depth is a significant fraction of the number of units on one disk, then the existence of a partial full table at the end of the array may also imbalance the parity distribution. Third, if the user workload exhibits a high degree of locality, accessing only the data in a portion of one full table over some period of time, then the parity update workload will not be balanced across the array during that time, since parity is only balanced over complete full tables. Thus reducing the depth of the full table both improves the balance of parity update workload across the disks of the array, and allows the array to be constructed from smaller-capacity disks, if that is desired. This section describes a mechanism by which the full table can be compacted to its minimum possible size.

This section makes extensive reference to the five block design parameters, so we summarize them here for convenience. A block design consists of b tuples, each containing k objects selected from a set of v possible objects, such that each object appears in exactly r tuples, and each pair of objects appears in exactly λ_p tuples. By counting objects and pairs in the design, it is easy to derive that $bk = vr$ and $r(k\text{-}1) = \lambda_p(v\text{-}1)$. The disk array layout mechanism associates objects with disks and tuples with parity stripes, so $v = C$ and $k = G$.

The remainder of this section presents a technique for compacting the full table to it’s minimum size. The first subsection outlines the approach and presents an example, the second describes the process of reordering the tuples and elements of the block design to achieve parity balance, and the third evaluates the effectiveness of the approach at reducing the size of the full table.

3.5.2.1. Balancing parity in the minimum number of block design tables

Recall that the block-design based layout described in Section 3.2.2 constructs a full table from *k* block design tables solely in order to balance the distribution of parity across the array. There is no other reason to rotate the parity assignment through the columns of the block design. The question that arises from this observation is whether it is possible to balance the parity distribution in fewer than *k* repetitions of the block design table. This would shrink the size of the full table, and thus improve the layout with respect to all three issues described above. Since each block design table contains *b* tuples and each tuple contains one parity unit, there are *b* parity units in one block design table. In order to balance these parity units across the *v* disks comprising the array, it's clear that at least LCM(*b*,*v*) tuples are required, where LCM is the least-common-multiple function. Since LCM(*b*,*v*) is of course a multiple of *b*, it's clear that this number of tuples comprises an integral number of block design tables, and that this number of tables (LCM(*b*,*v*)/*b*) is the minimum over which parity can be balanced. This follows immediately from the fact that concatenating together any integral number of block designs yields a block design.

The full block design table can be compacted to its minimum size as follows. From a block design *B* on parameters *v, k,* and *b,* construct an expanded design *B'* consisting of N_c = LCM(*b*,*v*)/*b* copies of *B*. Reorder the tuples of *B*' and the objects within each tuple according to the reordering algorithm (presented below) to achieve the condition that the last object in tuple *i* of *B*' is object *i mod v,* for all tuples *i* in *B'*. This reordered version of *B*' has the property that its last column identifies each disk exactly LCM(*b*,*v*)/*v* times. Thus if the last column of the block design is always selected to be the parity object, that is, if the rotation of the parity through the columns of the block design is suppressed, then parity will be evenly balanced in the array. Figure 3.26 illustrates the compaction process for an example block design.

The data mapping algorithms given in Appendix A are capable of optionally suppressing the rotation of the parity unit, and so they can be used directly to implement the compacted-full-table layout by simply using *B*' instead of *B* as the block design. The only cost of this implementation over the straightforward block-design implementation described in Section 3.2.2 is the small amount of extra memory (a few tens of kilobytes, at

Tuple Number	(a) Original Design	(b) Expanded Design	(c) Re-ordered Design
0	0 1 2 3	0 1 2 3	3 1 2 **0**
1	0 1 2 4	0 1 2 4	0 4 2 **1**
2	0 1 2 5	0 1 2 5	0 1 5 **2**
3	0 1 3 4	0 1 3 4	0 1 4 **3**
4	0 1 3 5	0 1 3 5	0 1 5 **4**
5	0 1 4 5	0 1 4 5	0 1 3 **5**
6	0 2 3 4	0 2 3 4	4 2 3 **0**
7	0 2 3 5	0 2 3 5	4 2 3 **1**
8	0 2 4 5	0 2 4 5	0 5 4 **2**
9	0 3 4 5	0 3 4 5	0 5 4 **3**
10	1 2 3 4	1 2 3 4	1 2 5 **4**
11	1 2 3 5	1 2 3 5	1 2 3 **5**
12	1 2 4 5	1 2 4 5	5 2 3 **0**
13	1 3 4 5	1 3 4 5	5 3 4 **1**
14	2 3 4 5	2 3 4 5	5 3 4 **2**
15		0 1 2 3	0 1 2 **3**
16		0 1 2 4	0 1 2 **4**
17		0 1 2 5	0 1 2 **5**
18		0 1 3 4	4 1 3 **0**
19		0 1 3 5	0 5 3 **1**
20		0 1 4 5	0 4 3 **2**
21		0 2 3 4	0 2 5 **3**
22		0 2 3 5	0 1 5 **4**
23		0 2 4 5	0 2 4 **5**
24		0 3 4 5	5 3 4 **0**
25		1 2 3 4	4 2 3 **1**
26		1 2 3 5	1 5 3 **2**
27		1 2 4 5	1 5 4 **3**
28		1 3 4 5	1 2 5 **4**
29		2 3 4 5	2 3 4 **5**

Figure 3.26: Compacting the full block design table with *v*=6, *k*=4, and *b*=15.

LCM(b,v)/b = 2 copies of the original design (a) are concatenated together to form the expanded design (b), which is then re-ordered (c) such that object (i mod v) appears as the last object of tuple i, for all tuples i. This reduces the full table size by 50% in this example.

most, for any array of up to 43 disks) needed to store the expanded design.

3.5.2.2. Reordering algorithm

The algorithm to re-order the expanded block design *B'* to achieve the "ordering property", that is, the condition that object *i mod v* appears as the last object of tuple *i*, for all tuples *i*, is straightforward. First note that each object appears $N_c \cdot r = k \cdot \mathrm{LCM}(b,v)/v$ times in the expanded design, but is required to serve as a parity unit only

$N_c \cdot b/v$ = LCM(b,v)/v times. Thus there are k times as many copies of each object in the expanded design as are needed to serve as parity units. The re-ordering algorithm simply selects successive tuples that have the desired property. When it encounters the condition that it needs to select a tuple containing a particular object i, but no copies of i are left unselected, it finds a swapping of selected tuples with unselected tuples that moves a copy of i into the unselected set, while still maintaining the ordering property on the selected set. The algorithm occasionally needs to perform a more complex swap on three tuples rather than just two, but never more than three. Figure 3.27 defines the algorithm more precisely.

Schwabe and Sutherland [Schwabe94] have also investigated whether or not it is possible to balance parity without duplicating the block design table G times and rotating the parity slot. They proved that parity can be balanced if and only if b is a multiple of v. Note that the expanded design (B') is simply a new block design with $b' = N_c \cdot b$ and $v' = v$. Therefore, Schwabe and Sutherland's result implies (1) $N_c = LCM(b,v)/v$ is the minimum number of copies of the block design over which parity can be balanced, and (2) any block design, duplicated N_c times, can be reordered to achieve the condition that tuple i contains object *(i mod v)*. Conclusion (1) arises because N_c is the minimum number of copies of the block design required to assure that b' is a multiple of v, which is necessary to balance parity. Conclusion (2) arises because B' is a block design in which the number of tuples (b') is a multiple of the number of elements (v), and this is sufficient to guarantee the balance.

It's clear from the above that it is always possible to reorder an expanded design such that is has the ordering property. In practice, we have never observed this simple algorithm to fail to find such an ordering. An interesting topic for future work would be to prove that the above algorithm always works. More specifically, the result to be proven is that when selecting the i^{th} tuple, it is always possible to find a three-way tuple swap that moves a tuple containing a copy of element *(i mod v)* into the necessary position.

In practice, the algorithm is very efficient: in deriving the reordering for 433 block designs with $v \leq 50$ and $\alpha < 1.0$, it performed an average of ten tuple swaps per block design, two designs required one three-way swap each, and no designs required more than one three-way swap. This efficiency stems from the fact that there are many more copies of each element in the expanded design than are actually needed to serve as parity.

```
Algorithm Reorder
Precondition:   the expanded block design $B'$ with parameters $N_c$, $v$, $k$, and $b$.
Define:         an ordered set $S$ holding currently selected tuples
                an unordered set $US$ holding currently unselected tuples
Postcondition:  $S$ contains an expanded design having the ordering property
Notation:       L($t$) represents the last object of tuple $t$
$US = B'$
for $i = 0$ to $(N_c \cdot b - 1)$ {
    if ($\exists$ a tuple $t \in US$ such that object $(i \bmod v) \in t$) then {
        remove $t$ from $US$ and install it as tuple $i$ of $S$
        move object $(i \bmod v)$ so that it is the last object of tuple $t$
    } else if ($\exists t_1 \in S$ such that $(i \bmod v) \in t_1$ and $\exists t_2 \in US$ such that $L(t_1) \in t_2$) {
        /* simple swap of two tuples */
        move $t_2$ to the position in $S$ occupied by $t_1$
        install $t_1$ as tuple $i$ of $S$
        adjust object ordering in $t_1$ and $t_2$
    } else {
        /* three-way tuple swap */
        find $t_1, t_2 \in S$ and $t_3 \in US$ such that $L(t_2) \in t_3$, $L(t_1) \in t_2$, and $(i \bmod v) \in t_1$
        move $t_3$ to the position in $S$ occupied by $t_2$
        move $t_2$ to the position in $S$ occupied by $t_1$
        install $t_1$ as tuple $i$ of $S$
        adjust object ordering in $t_1$, $t_2$, and $t_3$
    }
}
```

Figure 3.27: The block design reordering algorithm.

3.5.2.3. Compaction results and conclusions

Figure 3.28 shows a histogram of the percentage reduction in full table size when we applied the compaction algorithm to the 433 designs described above. In about 15% of the designs (65 out of 433), there was no benefit to compacting the full block design table because, in these cases, LCM(b,v)/$b \geq k$. However, the percentage compaction for the other 85% of the designs ranged from a minimum of 50% to over 99% for some of the larger designs. The average compaction was about 70%, including the designs which did

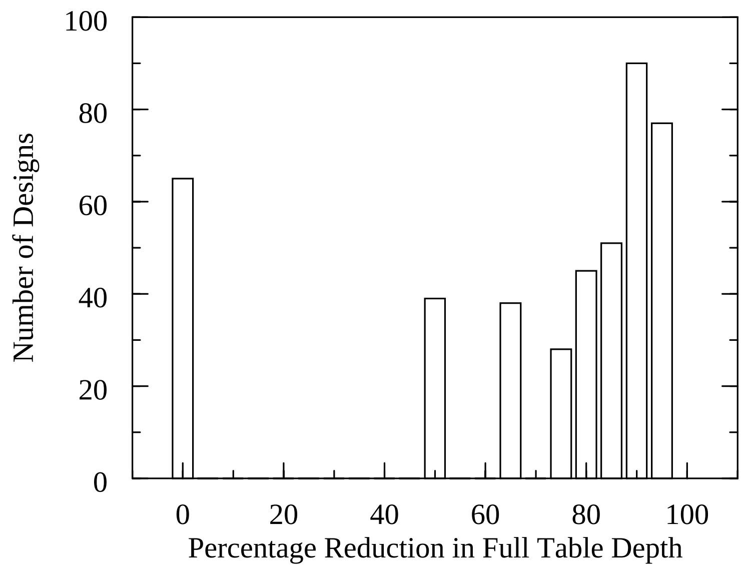


Figure 3.28: Histogram of full block design table compaction percentage.

not compact at all.

It's expected that designs with larger α will compact more than other designs, since α is proportional to *k* and an uncompacted block design full table consists of *k* repetitions of a block design table. Since the advantages of declustering are much more pronounced at low values of α, it's worthwhile to investigate the compaction percentage as a function of the declustering ratio. Figure 3.29 shows, for the same set of 433 designs, the average compaction percentage as a function of α, with α quantized to multiples of 0.1. It's clear that the trend toward greater compaction at higher declustering ratios is slight, indicating that the benefits of compaction apply at all values of α.

The final question to be answered with respect to the size of the full block design table is how large the component disks of the array must be in order to assure an even balance of parity. As discussed previously, if the depth of the full block design is too large with respect to the number of units on a disk, then the existence of a partial full table at the end of the array may imbalance the parity distribution. Figure 3.30 plots, for each of the 433 block designs ($v \leq 50$ and $\alpha < 1.0$) the minimum number of units required on each disk in the array to assure that the number of units on each disk is at least ten times the compacted full table depth. When the component disks are at least as large as the values given in the

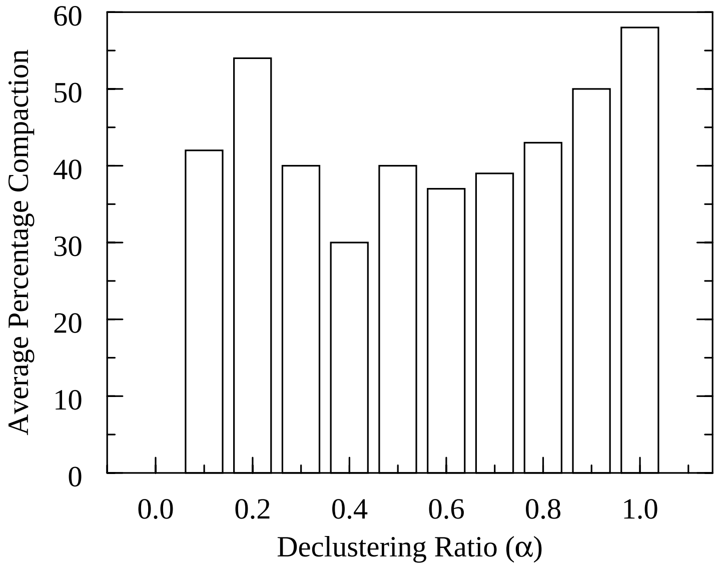


Figure 3.29: Histogram of compaction percentage versus declustering ratio.

Each bar represents an average computed over designs whose declustering ratios are within 0.05 of the center point of the bar on the x-axis. There were 433 designs altogether, evenly distributed around the point $\alpha = 0.5$.

figure, the array consists of at least ten full tables, and so the parity imbalance effects of the partial full table, if any, are guaranteed to be small. Assuming the size of the unit of layout to be 32 KB (about one track for typical disks), then the peak value of the graph of about 18,000 units corresponds to a minimum disk size of 580 MB. Note, however, that about 70% of the designs allow for component disks of any size larger than 5,000 units, or about 160 MB.

It is important to note that having fewer than ten full block design tables in the array does not necessarily imply that the parity distribution will be unbalanced. On the contrary, so long as there is at least one full block design table in the array, it is always possible to balance parity exactly by truncating the disks at the boundary of the last full table. Further, the actual degree of parity imbalance caused by the existence of a partial full table varies with the block design used and the size of the component disks, and can be small even when the full table depth is large. The plot in Figure 3.30 should be interpreted as the minimum disk size to *guarantee* that any parity imbalance will be small.

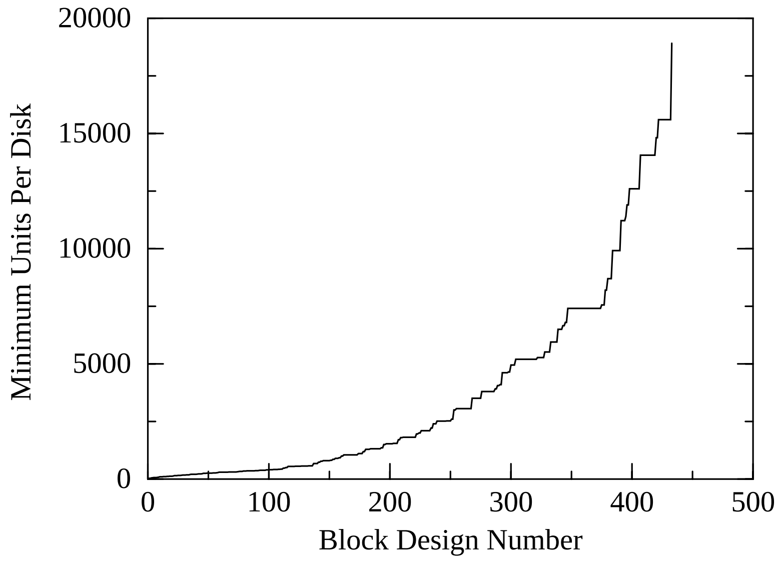


Figure 3.30: Minimum disk size to guarantee good parity balance.

The figure shows the minimum number of units per disk to ensure that the array consists of at least ten full block design tables, which guarantees that any parity imbalance due to the existence of a partial full table is small. The value 10 is arbitrary, but reasonable. Note that parity is balanced within each full table, and so parity can be balanced over the array by truncating the disks to ensure that there are an integral number of full block design tables in the array. This reduces the values in the above figure by a factor of 10.

3.5.3. Improving adherence to criterion six

This section discusses a technique for optimizing a block design to improve adherence to criterion six. The first subsection motivates the need for optimization by demonstrating that it is not possible to simultaneously meet criterion five and six using the block-design-based layout. The second describes the optimization approach, the third evaluates the results, and the fourth concludes.

3.5.3.1. The need for optimization

As illustrated in Figure 3.31, it is possible to meet criterion six by employing a data mapping similar to Lee’s left-symmetric layout for non-declustered arrays [Lee91], but this causes the layout to violate criterion five. This mapping works by pre-assigning the parity units to physical disk locations in the same manner as Figure 3.4, and then assigning each successive data unit in the address space of the array to the first available data unit on each successive disk, wrapping around to disk zero after using disk four. This generates a layout that meets criterion six since all disks are used before any is re-used. It causes crite-

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0	D1	D2	P0	P1
1	D5	D6	D7	D3	P2
2	D10	D11	D12	D8	P3
3	D15	D16	D17	D13	P4
4	D20	D21	P5	D18	D4
5	D25	D26	P6	P7	D9
6	D30	D31	D22	P8	D14
7	D35	D36	D27	P9	D19
8	D40	P10	D32	D23	D24
9	D45	P11	D37	D28	D29
10	D50	P12	P13	D33	D34
11	D55	D41	P14	D38	D39
12	P15	D46	D42	D43	D44
13	P16	D51	D47	D48	D49
14	P17	D56	D52	D53	D54
15	P18	P19	D57	D58	D59

Figure 3.31: Meeting criterion six via left-symmetric parity-declustered layout.

The figure shows the data units that are allocated by the full block design table in Figure 3.4, where data units are mapped in the style of Lee's left-symmetric layout. Note that this figure shows the mapping of data units in the address space of the array to physical disks, rather than the mapping of parity stripes to disks as in Figure 3.4.

rion five to be violated because successive user data blocks are assigned to differing parity stripes. For example, a write of three units starting at unit D3 is one parity stripe in length and parity-stripe aligned, but, referring back to Figure 3.4, unit D3 is in parity stripe 2, D4 is in parity stripe 6, and D5 is in parity stripe 1. Therefore, the controller must pre-read the corresponding data and parity units to effect the write operation, which violates criterion five. At first glance it seems possible to solve this problem by re-defining the data units that each parity unit protects. For example, P1 could be defined to be the parity unit for data units D3 through D5, no matter where they reside. Unfortunately, this leads to a violation of criterion one: Figure 3.31 shows that under this definition, D4 and P1 reside on the same disk.

It's easy to show by counterexample that, in general, it is not possible in this layout to simultaneously meet both criterion five and criterion six. Consider the block design on $v = 10$ and $k = 4$ given in Figure 3.32a. In order to meet criterion five, the layout must assign sequential data units in the address space of the array to sequential data units of parity stripes in the physical array. Tuple number zero shares at least one object with every other tuple in the design, and so no matter which tuple is selected as tuple number one in

(a) Block design on $v = 10$, $k = 4$, $b = 15$

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

(b) The violation of criterion six

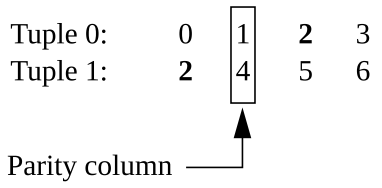


Figure 3.32: The mutual incompatibility of criteria five and six.

The figure shows a block design in which it is impossible to simultaneously meet both criterion five and six, using the layout defined in Section 3.2.2. The bold typeface in part (a) identifies an object that each tuple has in common with tuple 0. As shown in part (b), at some point within the full block design table the parity column will be selected as a column containing neither copy of the common element between tuples 0 and 1, and thus there will be an access of size six user data units that re-uses a disk. Since all tuples have an element in common with tuple 0, this holds no matter which of them appears as tuple number 1.

the block design, it will be the case that two successive parity stripes have at least one disk in common. Each parity stripe contains G-1 = 3 user data units, and so the only way that the six user data units contained in the two parity stripes defined by tuples zero and one can all reside on different disks is if the common element is always selected to be the parity unit in at least one of the two tuples. However, as illustrated in Figure 3.32b, the layout has the property that every column in the design is selected as the parity column at some point within the full block design table, and so this cannot always be the case. Thus, in this example, it is always possible to find a user access of size 2(G-1) = 6 user data units that uses fewer than six disks, which violates criterion six.

3.5.3.2. Optimizing the designs

The above discussion leads to the idea of optimizing the tuple and object ordering within a block design in order to maximize the adherence to criterion six in a given design. Note that large-access performance is the only figure of merit of a parity-declustered redundant disk array that is affected by the tuple and object ordering of the block design, and so these orderings can be freely permuted without affecting any of the other performance metrics.

Since there are $b!$ possible orderings of the tuples in a design, and $(k!)^b$ possible object orderings for each such tuple ordering, it's clearly infeasible to exhaustively search for the ordering that yields optimal adherence to criterion six. Instead, the approach taken here is to use a generic "global optimization" technique, *simulated annealing* [Kirkpatrick83]. This approach, modeled after the physical process by which a hot metal cools to a minimum energy state, can be summarized as follows. Define a cost function on the problem being considered, such that the function assigns to a given problem state a number indicating its "badness" with respect to the optimization goal. Then define a set of legal permutations on the problem state, with the intent that all possible problem states can be reached by some set of permutation applications. Finally, define the current "temperature" as an abstract number that is initially set to a high value and is gradually reduced as the annealing process proceeds. The annealer selects permutations at random and applies them to the problem state, monitoring the cost function as it does so. Permutations that reduce the cost function ("downward moves") are always "accepted", that is, the algorithm updates current problem state to reflect the permutation. Permutations that increase the cost ("upward moves") are accepted or rejected according to a probability distribution defined on the current temperature. The probability of accepting an upward move is high at high temperature, and diminishes, typically exponentially, as the temperature is reduced. This process continues until the point where the temperature is too low to accept any more upward moves, and no more downward moves can be found within some number of attempts. The goal of accepting upward moves at high temperature is to allow the optimization process to move out of local minima in the error surface. Of course, the annealing algorithm is not guaranteed to find the global optimum, and so the process is generally repeated several times with different random seed values, and the best solution thus identified is selected.

To optimize a block design for large-access performance, define the cost function as the reciprocal of the average number of disks used by an access of size C data units, where the average is taken over the accesses starting at each data unit within a full block design table, including those accesses that cross the full table boundary. Since the array is laid out using copies of the full table, this cost function suffices to optimize large-access performance throughout the array. It is of course possible to define the cost function so as to optimize the block design with respect to any access size less than or equal to C data units, if the expected workload is known. Define two permutations, one that swaps the ordering of two randomly selected tuples, and one that swaps the ordering of two randomly selected objects within a randomly selected tuple. The annealing then proceeds as described above.

3.5.3.3. Results

Figure 3.33 shows the results of this study for a few block designs on $v = 40$. The left-had plot in each part of the figure shows the cumulative distribution function (CDF) for the number of disks used by an access of size 40 stripe units, both before and after optimization. In each such plot, each y-axis value represents the fraction of all possible accesses to the array that use fewer disks than the corresponding x-axis value. The figure also gives the average number of disks used by an access in each case (the numbers in parentheses in the legend). The right-hand plot in each part shows the corresponding simulation results for large-access performance, measured in total megabytes moved per second on a 100% read workload with concurrency one. A workload of 100% writes yielded similar results, and so we omit the plots.

There are a number of points to be made about these plots. First, recall from Section 3.2.1 that the goal of meeting criterion six is to be able to guarantee a high data transfer bandwidth for accesses large enough to span many stripe units in the array. The CDF plots show that after optimization, the probability of using fewer than about 25 of the 40 disks in the array on any one large access is zero for all values of G, and so it is possible to provide such a guarantee. Second, note that in all cases the probability of using all 40 disks to service an access is zero, indicating that, even after optimization, it is impossible in the given declustered-parity layout examples to use all the disks for any access of size 40. This empirical result is even stronger than the one proven in the previous section. However, it's clear that optimizing the designs does in general improve the adherence to criterion six, with the degree of improvement depending with the specific case being considered. For the examples considered, the improvement varied from essentially zero to about a 10% increase in the average number of disks used by an access of size 40. In all cases, this improvement in the adherence to criterion six translated directly into improved large-access performance due to higher disk utilization. These observations are reflected in the plots on the right-hand side of Figure 3.33, which show that the large-access performance of an array using an optimized block design exceeds that of an array using an unop-

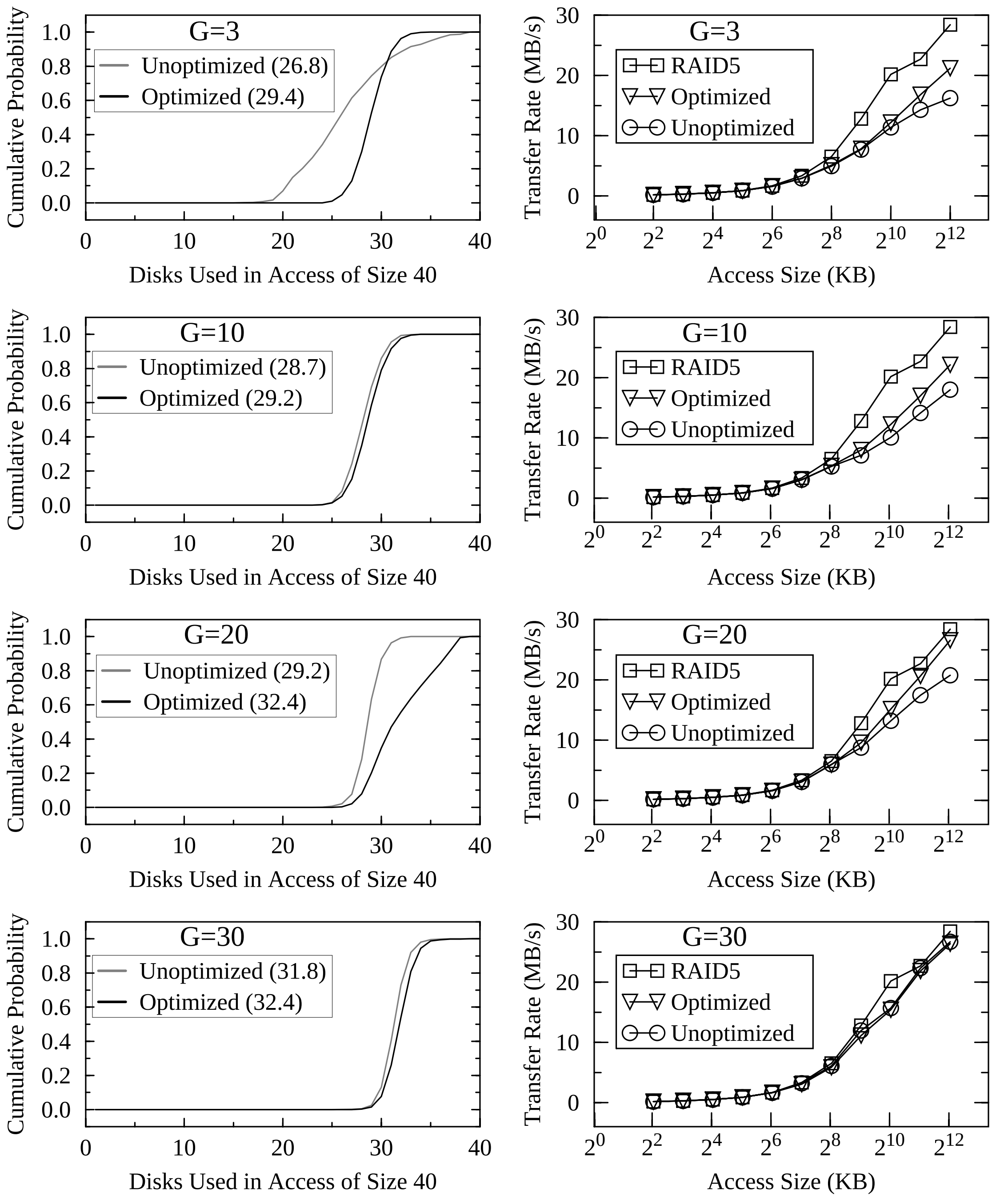


Figure 3.33: The results of the annealing study.

The left-hand plots show cumulative distribution functions on the number of disks used by an access of size 40 stripe units, and the right-hand plots show the large-access read performance of optimized block designs as compared to unoptimized designs, for a 40-disk array. Numbers in parentheses are average values. For each value of G, the left-hand plot shows the probability of using fewer than n *disks to service an access of size 40 stripe units, as a function of* n. *The right hand plot shows the comparative performance gains obtained by optimizing the block design, with the RAID Level 5 case included in each plot for comparison. Note the log scale on the x-axis.*

timized design, but still falls short of that attained in a RAID Level 5 array that meets criterion six.

Finally, and most interestingly, consider the $G = 10$ case in Figure 3.33. For this example, optimizing the block design yielded only about 2% improvement in adherence to criterion six, but the read performance of the optimized design at an access size of 4 MB (2^{12} bytes) was about 20% better than the corresponding unoptimized case (about 22 MB/second for the optimized case, versus about 18 unoptimized). The disk utilization plot (not shown) indicates correspondingly higher utilization using the optimized design. This effect is caused by variations in the degree of *head synchronization* allowed by the two designs. Since the workload concurrency is one, the single requesting process issues one access and then waits for it to complete. If some disks finish their component of the user access before others because of variations in seek distance and/or amount of data transferred from each drive, those disks will be idle until the slowest disk completes its component of the access. If, on the other hand, all the disk heads remained perfectly synchronized and each disk involved in a user access transferred exactly the same amount of data, then all disks would complete their component of the user access at approximately the same time, and so the disk utilization would remain high on all drives. This higher utilization would allow for higher data transfer bandwidth.

This shows that the large-access performance of a declustered-parity array at low workload concurrency is sensitive to the *variability* in the layout of data and parity. As illustrated in Figure 3.34, the two primary factors that influence this performance are (1) the variability in the physical disk offset of each component of a large access, and (2) the variability in the amount of data transferred from any one disk for a large access. The former causes variation in the completion time of the access components by inducing variation in the seek times, and the latter by inducing variation in the data transfer times.

To analyze this effect for the examples given above, Table 3.5 gives the coefficient of variation ("COV", the standard deviation divided by the mean) in the starting physical disk offset on each disk, and in the amount of data transferred from each disk, for an access of size 40 stripe units. There are many possible accesses of size 40 in each array, characterized by their starting position within the full block design table. Each such access has a distinct COV for the two metrics, and so the table reports the "average COV" across

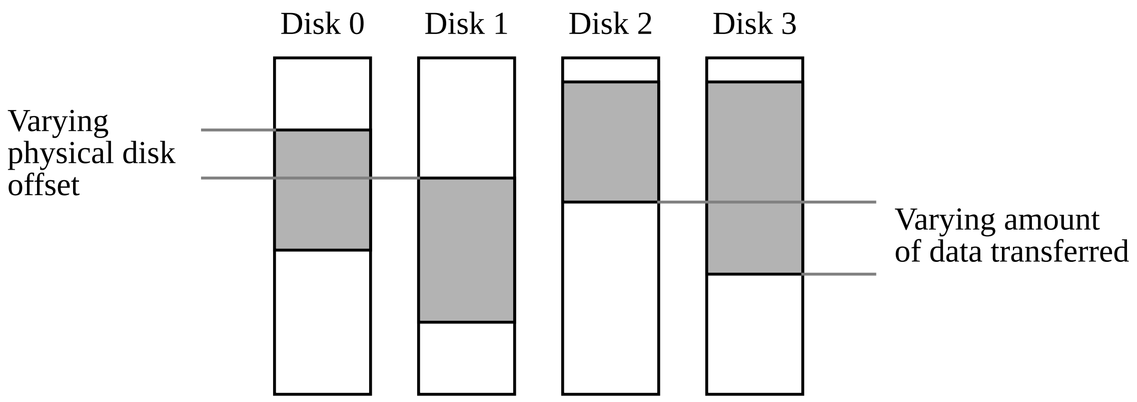

Disk 0
Disk 1
Disk 2
Disk 3
Varying physical disk offset
Varying amount of data transferred

Figure 3.34: Two sources of variation in the individual disk access times.

The figure shows examples of physical-disk-offset and amount-of-data-transferred variations between the components of a large access in a declustered-parity redundant disk array. The shaded regions represent the units read by a hypothetical large access.

all the possible size-40 accesses within a full block design table. In a strict sense, this average COV is not statistically meaningful since there is a COV associated with the averaging process itself, but since the only goal is to minimize the variability, it serves as a convenient metric.

	COV of Physical Disk Offset		COV of Amount of Data Transferred Per Disk	
Block Design	Unoptimized	Optimized	Unoptimized	Optimized
G=3	0.04	0.07	0.44	0.46
G=10	0.03	0.03	0.56	0.42
G=20	0.01	0.01	0.37	0.33
G=30	0.005	0.004	0.34	0.34
G=40 (RAID5)	0.003	N/A	0.00	N/A

Table 3.5: Tabularizing the degree of variation.

The table gives the coefficients of variation on physical disk offsets and amount of data transferred for the example designs on C = 40.

The table shows that the COV on the physical disk offset is small in both the optimized and unoptimized case, indicating that there is little variability in the component access times due to differing seek times. However, the variations in the amount of data transferred per disk are larger, ranging up to almost 60% of the mean value. The entry for $G = 10$ shows that the optimization process reduced the standard deviation in amount of data transferred from 56% of the mean to 42%. The resultant increase in disk utilization caused the improved performance observed in Figure 3.33. Note that reducing this variance was not the goal of the optimization; the cost function used by the annealer considered only the average number of disks used by an access of a particular size. Note also that optimizing the block design according to this cost function can cause the variability to increase, as happened in the $G = 3$ case. An interesting topic for future work would be to revise the cost function to include both number-of-disks and variability terms, and re-evaluate the performance gains using the blocks designs so generated.

3.5.3.4. Conclusions

In summary, the block design optimization process was able to improve the large-access performance of parity declustered redundant disk arrays by up to about 30% in the examples considered. Expressed a different way, Figure 3.33 shows that optimizing the block design allowed the array to recapture a significant fraction of the performance lost due to the failure to meet criterion six; for example, optimizing a block design for an array with $C = 40$ and $G = 20$ improved the read data transfer rate from 20.7 to almost 26.6 MB/second at an access size of 4 MB. The corresponding RAID Level 5 array achieved about 28.4 MB/second on the same workload, indicating that optimizing the block design recaptured about 75% of lost performance. The actual degree of performance improvement was case-specific, and there were cases where little benefit was observed by optimizing the design. However in these cases, the unoptimized design performed comparatively well (consider the $G = 30$ case in Figure 3.33), and so there was little benefit to be achieved in any case.

3.6. Conclusions

This very long chapter first showed that the performance of RAID Level 5 disk arrays in the presence of a failed disk is not acceptable, and then described and thoroughly evaluated parity declustering as a solution. The fundamental idea behind this approach is that each parity unit should protect fewer than C-1 data units, where C is the number of disks in a group. Having made this observation, the problem reduced to finding a layout for the data and parity units that balances the failure-induced workload over the disks comprising the array. We derived such a layout using *balanced incomplete block designs*.

Having defined the declustered parity organization, the question arose of how to select the values for C and G, its two primary parameters. The chapter showed that, using certain very reasonable assumptions about the requirements of the system, this problem reduced to a tradeoff between capacity overhead and reliability.

Evaluating parity declustering turned up several shortcomings, such as the necessity to decouple the size of the reconstruction unit from that of the striping unit and the fault-free performance degradation on very large read accesses, and so the subsequent sections proposed modifications to address them. These sections showed that all of the shortcomings were addressable via some technique; at the end of the chapter there remained no significant drawbacks to using parity declustering.

Chapter 4: Reconstruction Algorithms

This chapter addresses the design and implementation of the reconstruction algorithm, which is the technique used to recover the data that is lost when a disk fails. The primary problems to be addressed are (1) selecting the order in which to reconstruct lost units, (2) managing concurrency in the reconstruction process (determining the read and write operations to perform in parallel), (3) managing the buffer memory used to hold partially-reconstructed units, and (4) controlling the interaction of reconstruction accesses with normal user accesses in order to guarantee correct results and good performance. These functions can be implemented in either the host computer or in the array controller, or can be distributed amongst the local disk controllers in the array [Cao93]; the techniques discussed in this chapter apply to all implementations.

The chapter is organized as follows. Section 4.1 discusses prior and related work on the topic. There have not been many studies specifically addressing the design of the reconstruction algorithm, so this short section essentially summarizes assumptions that other researchers have made about the recovery process. Section 4.2 describes what has been the default reconstruction algorithm, *stripe-oriented reconstruction*, and discusses its limitations. This leads to the development of *disk-oriented reconstruction*, the algorithm that forms the basis of the work in this chapter. Section 4.3 provides a comprehensive set of performance evaluations, demonstrating the benefits of disk-oriented reconstruction. Section 4.4 describes and evaluates a set of optimizations and improvements that can be applied to the reconstruction algorithm to improve specific aspects of its performance. Section 4.5 concludes and summarizes the chapter.

4.1. Prior work

There is little or no existing literature on the design of the reconstruction algorithm, neither for mirrored nor parity-based arrays. Most failure-recovery studies make a number of assumptions about the characteristics of the reconstruction algorithm, but none specify how they would be implemented. Both the mirroring studies [Copeland89, Bitton88] and the parity-based studies [Muntz90, Merchant92a] assume that the recovery algorithm is able to maintain an even balance of reconstruction requests across the disks comprising the array at all times, that the reconstruction process is able to absorb all of the array's excess bandwidth, and that buffer memory management does not pose a problem. The implementation of a reconstruction algorithm that adheres to these assumptions is one of the topics of this chapter. Hou et. al. [Hou93] describe a simulation-based study that presumably models the operation of the reconstruction algorithm, but the study focuses on other aspects of the recovery process and hence does not describe or analyze the algorithm in detail.

4.2. Stripe-oriented and disk-oriented reconstruction

This section describes the reconstruction algorithm assumed by most studies, and uses it to motivate the development of a better algorithm.

4.2.1. Stripe-oriented reconstruction and its parallelized version

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

```
for each unit on the failed disk (sequentially)
    1. Identify the parity stripe to which the unit belongs.
    2. Lock all units in the stripe against accesses by the users.
    3. Issue low-priority read requests for all other units in the stripe, including
       the parity unit.
    4. Wait until all reads have completed.
    5. Compute the XOR over all units read.
    6. Issue a low-priority write request to the replacement disk.
    7. Wait for the write to complete.
    8. Unlock the stripe.
end
```

In this context, a "low-priority request" is one that has lower queueing priority; that is, a low-priority request will not be initiated on a disk until there are no normal priority requests in the queue for that disk, but will be initiated immediately upon the occurrence of this condition. Once initiated, a low-priority access cannot be interrupted by a higher-priority access. The algorithm uses the low-priority requests in order to minimize the impact of reconstruction on user response time, since commodity disk drives do not support preemptive access. The algorithm uses a low-priority request even for the write to the replacement disk, because as will be discussed in Section 4.4.1.1, this disk services writes in the user request stream as well as reconstruction writes. The algorithm locks the stripe currently under reconstruction against user accesses because (1) a user write operation can cause a data unit to be temporarily out of date with respect to its parity unit, and it must be guaranteed that the reconstruction process reads neither data nor parity in this period, and (2) there is a potential race condition on the replacement disk between the write of the reconstructed data and a write of new user data. Note that as it is currently specified, there is a potential livelock problem in the algorithm: a user write request targeting a parity stripe that is currently under reconstruction is forced to wait for the low-priority reconstruction accesses to complete. If the array is operating at a high utilization, the reconstruction can be starved and hence the user write may be forced to wait indefinitely. We address this issue in Section 4.2.3.2.

The main problem with this algorithm is that it is unable to consistently utilize all the disk bandwidth that is not absorbed by users. This inability stems from three sources. First, it does not overlap reads of the 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 will not be identical for all disks and since the disk heads will be in essentially random positions with respect to each other in an OLTP workload. 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 organization, not every disk is involved in the reconstruction of every parity stripe, and so uninvolved disks remain idle with respect to reconstruction since the reconstruction algorithm works on only one parity stripe at a time.

These deficiencies can be partially overcome by parallelizing this algorithm, that is, by simultaneously reconstructing a set of parity stripes instead of just one [Holland92]. In this approach, the host or array controller creates a set of *P* identical, independent reconstruction processes. Each process executes the stripe-oriented algorithm, except that the next parity stripe to reconstruct is selected by accessing a shared list of as-yet unreconstructed parity stripes, so as to avoid duplication. Since different parity stripes use different sets of disks, the reconstruction process is able to absorb more of the array's unused bandwidth than in the single-process case, by allowing concurrent accesses on more than *G*-1 disks.

Although this approach yields substantial improvement in reconstruction time [Holland92], it does so in a haphazard fashion. Disks may still idle with respect to reconstruction because the set of data and parity units that comprise a set of *P* parity stripes is not guaranteed to use all the disks in the array evenly. 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. Supporting a large number of reconstruction processes requires a large amount of memory and computation power in the host or array controller. The next subsection describes a better algorithm.

4.2.2. Disk-oriented reconstruction

The deficiencies of both single-stripe and parallel-stripe reconstruction can be addressed by restructuring the reconstruction algorithm so that it is *disk-oriented* instead of *stripe-oriented*. A few previous studies [Merchant92a, Hou93] have suggested the main ideas behind this approach; we fully define and evaluate it in this section. Instead of creating a set of reconstruction processes associated with stripes, 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 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 XOR, or
       Block the process if buffer manager has no memory to accept the unit.
until (all necessary units have been read)
```

The process associated with the replacement disk executes:

```
repeat
        1.  Request the next sequential full buffer from the buffer manager.
            Block the process if none are 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)
```

The buffer manager provides a central repository for data and parity from parity stripes that are currently “under reconstruction.” When a new buffer arrives from a surviving-disk process, the 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 the active list, and returns it to the replacement-disk process.

The advantage of this approach is that it is able to maintain one low-priority request in the queue for each disk at all times, which means that it will absorb all of the array’s bandwidth that is not absorbed by users. Section 4.3.1 demonstrates that this approach yields substantially faster reconstruction than the parallel-stripe oriented approach.

4.2.3. Implementation of disk-oriented reconstruction

There are two implementation issues that need to be addressed in order for the above algorithm to perform as expected. The first relates to the amount of memory needed, and the second to the interaction of reconstruction accesses with updates in the normal workload.

4.2.3.1. Buffer memory management

In the stripe-oriented algorithm, the host or array controller requires exactly G reconstruction units worth of buffer memory per reconstruction process[1], where a reconstruction unit is defined as the amount of data read or written per reconstruction access (refer to Section 3.5.1). There is no need to manage this memory in any way, since each buffer is used for only one purpose. However, in the disk-oriented algorithm, transient fluctuations in the arrival rate of user requests at various disks can cause some reconstruction pro-

1. A well-designed controller might actually require only G-1 buffers per reconstruction process, since a buffer used to read data could be re-used to compute parity.

cesses to read data more rapidly than others. This buffer manager must store this information until the corresponding data or parity arrives from slower reconstruction processes, and thus the buffering requirements of each individual reconstruction process vary over time. It's possible to construct pathological conditions in which a substantial fraction of the data space of the array needs to be buffered in memory, and so it's necessary to define a buffer memory management policy for the disk-oriented algorithm.

The amount of memory needed for disk-oriented reconstruction can be bounded by enforcing a limit on the number of buffers employed. If no buffers are available, a requesting process blocks until a buffer is freed by some other process. In the implementation described here, the buffer pool is divided into two parts: each surviving-disk reconstruction process has one buffer assigned for its exclusive use, and all remaining buffers are assigned to a "free buffer pool." A surviving-disk process always reads units into its exclusive buffer, but then upon submission to the buffer manager, the buffer manager transfers the data to a buffer from the free pool, and then installs this buffer in its data structures. This division of buffers simplifies the code by assuring that there is always a free buffer into which to read data or parity when a reconstruction access arrives at the head of a disk queue. A buffer stall condition occurs only when there are no free buffers available into which to transfer the incoming unit, at which point the corresponding reconstruction process has no outstanding I/O requests. Only the first process submitting data for a particular parity stripe must acquire a free buffer, because subsequent submissions for that parity stripe can be XORed into this buffer. Thus this approach is able to maintain as many parity stripes under reconstruction as there are buffers in the free buffer pool.

Forcing reconstruction processes to stall when there are no available free buffers causes the corresponding disks to idle with respect to reconstruction. In practice, a relatively small number of free buffers suffices to achieve good reconstruction performance. There should be at least as many free buffers as there are surviving disks, so that in the worst case each reconstruction process can have one access in progress and one buffer submitted to the buffer manager. Section 4.3.2 demonstrates that using this minimum number of buffers is in general adequate to achieve most of the benefits of the disk-oriented algorithm, and using about twice as many free buffers as disks reduces the buffer stall penalty to nearly its minimum.

The alternative to this approach is to permanently assign buffers to disks as is suggested by Merchant and Yu [Menon92a] and by Hou et. al. [Hou93]. In other words, each reconstruction buffer available to the host or array controller is permanently assigned to one of the disks in the array, and no other disk may use it. The drawback of this simpler scheme is that it causes a reconstruction processes to stall when it exhausts its supply of buffers, even when there are many unused buffers assigned to other disks. Simulations showed that this approach yielded substantially worse reconstruction time than the “free buffer pool” approach.

4.2.3.2. Interaction with writes in the normal workload

The reconstruction accesses for a particular parity stripe must be interlocked with user writes to that parity stripe, since a user write can potentially invalidate data that has been previously read by a reconstruction process. This problem applies only to user writes to parity stripes for which some (but not all) data units have already been fetched; if the parity stripe is not currently “under reconstruction,” then the user write can proceed independently.

There are a number of approaches to this interlocking problem. The controller can flush buffered partially-reconstructed units when a conflicting user write is detected. Alternatively, the controller can treat buffered units as a cache, and user writes can update any previously-read information before a replacement-disk write is issued. A third option is to delay the initiation of a conflicting user write until the desired stripe’s reconstruction is complete.

We rejected the first option as wasteful of disk bandwidth. We rejected the second because it requires that the host or array controller buffer each individual data and parity unit until all have arrived for one parity stripe, rather than just buffering the accumulating XOR for each parity stripe. This would have multiplied the memory requirements in the host or controller by a factor of at least G-1. The third option is memory-efficient and does not waste disk bandwidth, but if it is implemented as stated, a user write may experience a very long latency when it is forced to wait for a number of low-priority accesses to complete. This drawback can be overcome if it is possible to expedite the reconstruction of a parity stripe containing the data unit that is about to be by the user. This is what we implemented. When the controller detects a user write to a data unit in a parity stripe that is currently under reconstruction, it elevates all pending accesses for that reconstruction to the priority of user accesses. If there are any reconstruction accesses for the indicated parity stripe that have not yet been issued, the controller issues them immediately, at regular priority rather than low priority. The user write triggering the re-prioritization stalls until the expedited reconstruction is complete, and the controller allows it to proceed normally.

Note that a user write to a lost and as-yet unreconstructed data unit implies that an on-the-fly reconstruction operation must occur, because the written data must be incorporated into the parity, and there is no way to do this without the previous value of the affected disk unit. Thus, this approach to interlocking reconstruction with user writes does not incur any avoidable disk accesses. Also, in practice, forcing the user write to wait for an expedited reconstruction does not significantly elevate average user response time, because the number of parity stripes that are under reconstruction at any given moment (typically less than about $3C$, as will be seen in Section 4.3.2) is small with respect to the total number of parity stripes in the array (many thousand).

A potential problem arises if a free reconstruction buffer has not yet been acquired for the parity stripe whose reconstruction is to be expedited, and none are available. The current implementation simply allocates a new buffer and frees it when the reconstruction is complete. This may not be acceptable in some implementations because the amount of buffer memory available may be strictly limited and completely in use. There are a number of potential solutions to this problem, ranging from reserving a few buffers for this purpose to stealing an in-use buffer and forcing the reconstruction of the corresponding parity stripe to be restarted. We did not pursue these avenues as the problem is minor and highly transient.

4.2.4. Summary

This section demonstrated a reconstruction algorithm designed to absorb for reconstruction all of the disk array bandwidth not absorbed by the users. The described implementation keeps every surviving disk busy with reconstruction reads at all times, unless blocked by the inability to acquire a buffer to hold the reconstruction unit. Splitting the buffer pool into “exclusive” and “free” parts and forcing processes to block only at buffer submission time assures maximally efficient buffer usage, since a reconstruction process cannot block unless there are zero free buffers in the system. The approach of expediting the reconstruction of parity stripes for which a user write is pending is simple to implement, and also preserves software boundaries in that the code controlling the user write operations is maintained separately from the code controlling the reconstruction process. The only modification required to the user-write code is that it must make a single call into the reconstruction module prior to initiating a write operation so that a pending reconstruction operation, if any, can be forced to completion before the write occurs.

4.3. Performance evaluations

The section presents the results of simulations evaluating the above algorithms. It reports on both reconstruction performance and the sensitivity of this performance to the amount of buffer memory in the controller.

4.3.1. Comparing reconstruction algorithms

Figure 4.1 shows the reconstruction time and user response time during recovery versus the declustering ratio (α) for single-thread, 8-way parallel, and 16-way parallel stripe-oriented reconstruction, and for disk-oriented reconstruction, using the array and workload parameters from the tables in Section 2.3. The figure shows that the disk-oriented algorithm makes more efficient use of the system resources: reconstruction is uniformly faster under disk-oriented reconstruction than under any of the stripe oriented algorithms, with the difference ranging up to a maximum of about 40% at $\alpha = 0.5$ (refer to Appendix C). The graphs converge at low α because, as discussed in Section 3.3.2.2, saturation on the replacement drive becomes the dominant factor influencing reconstruction performance, and so the differences in the efficiencies of the reconstruction algorithms become less significant.

Figure 4.1b shows that the improvement in reconstruction time comes at a moderate cost in user response time. The stripe-oriented algorithms yield slightly better user response time because they cause disks to idle more 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. Neglecting the RAID Level 5 ($\alpha = 1.0$) case, where disk saturation causes

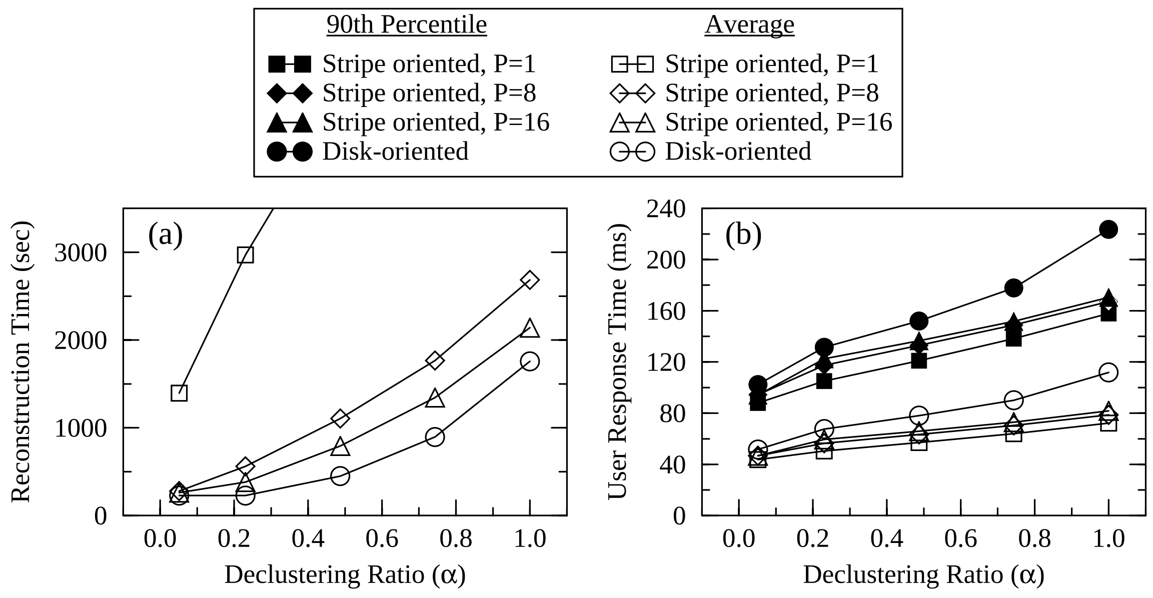


Figure 4.1: Comparing reconstruction algorithms.

The figure shows reconstruction time (a) and user response time during reconstruction (b) for the different reconstruction algorithms. The figure deliberately cuts off the upper portion of the reconstruction time plot in order to expand the scale. Reconstruction time with one process (P=1) *and* α=1.0 *was about 9000 seconds (refer to Appendix C).*

the response times to be unstable, the maximum difference in average user response time between the disk-oriented algorithm and the 16-way parallel stripe-oriented algorithm is about 20% at α = 0.75. However, the performance degradation caused by reconstruction lasts longer using these stripe-oriented algorithms, and this offsets the response-time benefit that they provide.

Note that the response time degradation experienced when using the disk-oriented algorithm is fundamentally a priority inversion problem, because low-priority reconstruction requests force high-priority user requests to stall until the low-priority requests complete. An interesting topic for future work would be to investigate the possibilities of recapturing the lost responsiveness by implementing pre-emption in the local controllers of the disks comprising the array. If the disks had the ability to suspend and subsequently restart low-priority accesses with relatively low overhead, the controller could maintain responsiveness during reconstruction at essentially the same level as in fault-free mode.

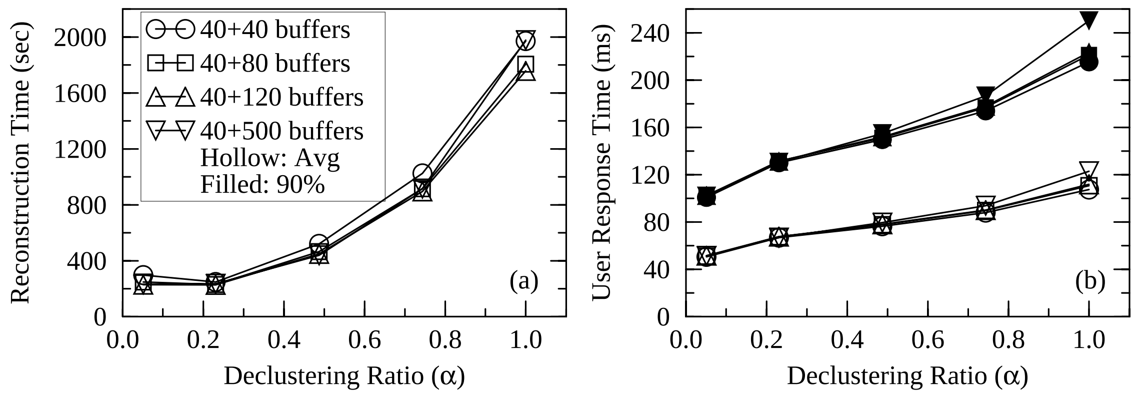


Figure 4.2: Sensitivity to available buffer memory.

The figure shows the sensitivity of reconstruction time (a) and user response time during reconstruction (b) to the number of reconstruction buffers employed. In the legend, the caption "40+X" means that the simulation was run using 40 exclusive and X free reconstruction buffers.

4.3.2. Sensitivity of disk-oriented algorithm to available buffer memory

As stated in Section 4.2.3.1, about 3*C* total reconstruction buffers are sufficient to obtain most of the benefit of the disk-oriented reconstruction algorithm. Figure 4.2 demonstrates this by showing the reconstruction time and average user response time during reconstruction for a varying number of free reconstruction buffers. Figure 4.2a shows that the reconstruction time can be slightly improved by using 120 rather than 80 buffers for a 40-disk array, but further increases do not yield any significant benefit. Using a very large number of reconstruction buffers causes reconstruction time to be slightly worse than the other cases. This is because with 540 reconstruction buffers in use there can be up to 540 parity stripes actively under reconstruction, which causes an excessive number of user accesses to invoke expedited reconstruction. This lengthens reconstruction time by forcing parity stripes to be reconstructed in non-sequential order, thereby increasing the average head positioning time incurred by a reconstruction access. Figure 4.2b shows that the number of reconstruction buffers has virtually no effect on user response time, except in the case where 540 buffers were used, where an excessively large fraction of user accesses are forced to wait for expedited reconstruction.

The non-sensitivity of reconstruction performance to available buffer memory is explained by noting that at declustering ratios close to 1.0, the reconstruction rate is limited by the rate at which data can be read from the surviving disks, while at low values of α, the replacement disk is the bottleneck. When the surviving disks are the bottleneck, total buffer stall time will be small because the controller cannot read data from the surviving disks as fast as it can write data to the replacement. This means that the buffer manager will recycle buffers faster than the reconstruction processes will require them to accept new data or parity units, and so a blocked process will not have to wait long to acquire a buffer. Under this condition, it's clear that increasing the size of the buffer pool will not improve performance. The simulations bear this out, showing that when α is close to 1.0, each surviving-disk process spends only a small fraction of its total time waiting to acquire reconstruction buffers. At low values for α, the surviving-disk processes generate reconstructed units faster than the replacement-disk process can write them out to disk, and so all free reconstruction buffers fill up very early in the reconstruction run, no matter how many of them there are. In this case, the rate at which the buffer manager frees buffers is throttled by the rate at which the replacement-disk process can write them to disk. Since the total number of free buffers has no effect on this rate, there is no benefit in increasing the total number of these buffers.

The conclusion from this is that using about three times as many reconstruction buffers as there are disks is in general sufficient to achieve the full benefits of a disk-oriented reconstruction algorithm. For an array of 40 disks using 24 KB stripe units and a total of 120 reconstruction buffers (40 exclusive + 80 free), the total buffer requirement in the host or controller is about 2.9 MB. Assuming that buffer memory costs 25 times as much as disk, and using the example 314 MB disks, the total cost of the buffer memory needed for reconstruction is less than 1% of the cost of the disks. Figure 4.2 shows that even this represents a relatively generous usage of buffer memory; the controller can use fewer buffers without experiencing any significant decrease in reconstruction performance.

4.3.3. Comparing memory requirements between algorithms

A P-way parallel stripe-oriented algorithm requires PG controller memory buffers, where P should be at least 8 or 16, while Figure 4.2 shows that a disk-oriented algorithm requires about $3C$. Thus except at low declustering ratios, the disk-oriented algorithm requires fewer buffers than the stripe-oriented algorithm with significant parallelism, and yet delivers faster reconstruction. In the example 40-disk array with α=0.5, the disk-oriented algorithm requires about 120 buffers, while the 8-way parallel stripe-oriented algorithm requires 160. Figure 4.1 shows that the disk-oriented algorithm is able to reconstruct about twice as fast under these conditions. However, at very low α the opposite condition holds; at $\alpha = 0.05$ (G=3), an 8-way parallel disk-oriented algorithm requires only 24 reconstruction buffers, but the disk-oriented algorithm still requires about 120. Figure 4.1 shows that saturation on the replacement drive causes reconstruction time to be essentially the same between the two algorithms at this declustering ratio. However, Chapter 5 will demonstrate a mechanism by which the replacement-disk bottleneck at low α can be eliminated, which in turn eliminates this apparent advantage for a stripe-oriented algorithm.

It is important to note that the total buffer memory requirements of the disk-oriented algorithm are relatively small, and so the controller can typically borrow the required memory from the controller or host buffer cache. If a reconstruction buffer is the size of one track (as indicated by the results of Section 3.5.1) and a disk contains 10,000 tracks, then the 120 buffers required for the example 40-disk array total about 1% of the size of one disk. If buffer memory costs 25 times as much per megabyte as disk, a buffer cache of 10% of the size of one disk costs about 6% of the total disk cost in the example array, and so is affordable in either the host or controller. Therefore, borrowing the 1% needed to effect reconstruction rapidly will not, in most cases, significantly alter the performance of the cache. This makes the disk-oriented algorithm preferable at all values of the low declustering ratio.

4.4. Optimizations and improvements

Paralleling the structure of Chapter 3, this section describes and evaluates a set of optimizations and improvements that can be applied to the reconstruction algorithm. The first subsection evaluates the effects of applying a set of well-known variations to the disk-oriented algorithm. The second discusses a technique called *head following*, where the reconstruction algorithm attempts to minimize head positioning time by reconstructing data and parity in the region of the array currently being accessed by the users.

4.4.1. Work reducing variations to reconstruction algorithms

Muntz and Lui [Muntz90] identified two simple modifications to a reconstruction

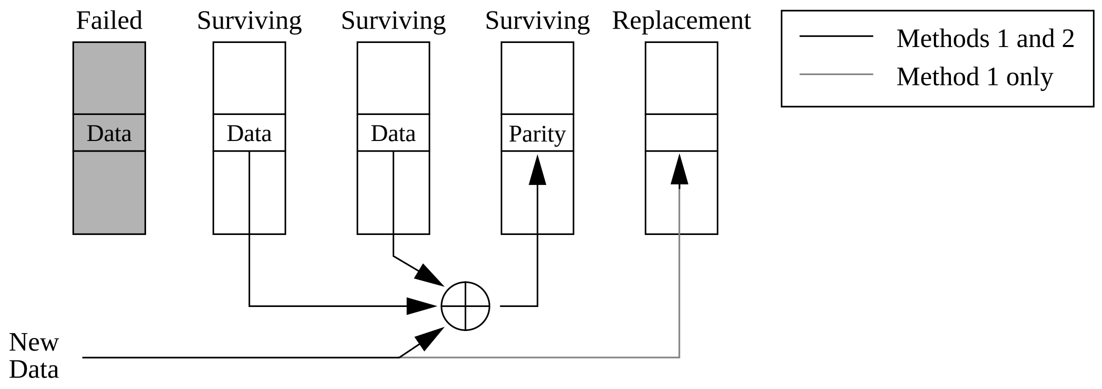


Figure 4.3: Two methods for servicing a user write to unreconstructed data.

Method 1 writes the new data to the replacement and updates the parity. Method 2 updates only the parity, and allows the background reconstruction process to later install the new data on the replacement drive.

algorithm, each intended to improve reconstruction-mode performance or reduce reconstruction time by reducing the total work required of surviving disks. This section evaluates their effects under the disk-oriented reconstruction algorithm.

4.4.1.1. Defining the variations

In the first variation, called *redirection of reads*, the controller services user read requests for failed data units that have already been reconstructed by reading from 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 1. Although this seems to be an obvious thing to do, Section 4.4.1.2 shows that it can actually lengthen reconstruction time under certain conditions. In the second variation, *piggybacking of writes,* when a user read request causes a data unit to be reconstructed on-the-fly, the controller writes that data unit to the replacement drive as well as delivering it 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 the following 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 controller writes new data directly to the replacement drive, and updates the parity to reflect this change. In the second, the controller updates only the parity, and does not write the data to disk at all. Figure 4.3 illustrates the two approaches more specifically: in the first method the controller writes the new data to the replacement disk, and updates the parity 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 controller updates the parity in the same manner as the first option, but does not write the new data to the replacement drive. In the latter case, the data unit being updated remains invalid until recovered by the background reconstruction process. We view sending user writes to the replacement disk (the former approach) as a third modification that the controller can apply, and refer to it as the *user writes* option. Note that writing to the replacement disk eliminates the need to reconstruct the corresponding data later on, but exacerbates the replacement-disk reconstruction bottleneck at low declustering ratios.

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, and the surviving drives service the remainder of the workload. 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. This section analyzes these options using the disk-oriented reconstruction algorithm and the array and the workload configuration described in the tables in Section 2.3.

The following evaluations investigate five of the eight possible combinations of these three reconstruction algorithm options: all options off, each option on with the other two off, and all options on. They show that only one option, the redirection of reads option, achieves its intended benefit for the workload considered here.

4.4.1.2. Evaluating the options on OLTP-like workloads

Figure 4.4 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 user-writes options have little effect on user response time. To understand this, note that updating a particular unit on the replacement drive can improve response time only if a user process re-accesses that unit prior to the completion of reconstruction. For a random workload, the probability of re-accessing the same data unit

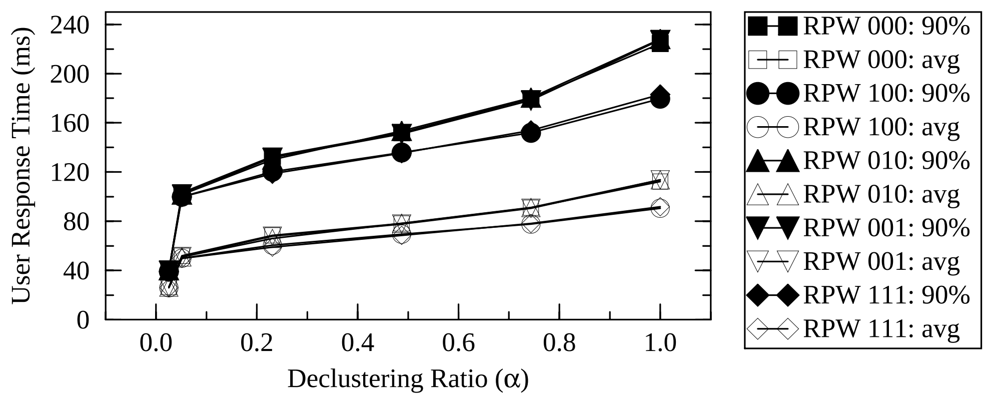


Figure 4.4: User response time for five combinations of the variations.

In the 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.*

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 the 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. Reducing α, however, causes both these effects diminish: at low α 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 4.5 shows the reconstruction time for the five combinations of options. The piggybacking of writes and user-writes options again make little difference. In this case, it is because nearly all (96%) of the accesses in the workload are smaller than one reconstruction unit. This means that typical user- or piggybacked-write operation updates and marks as reconstructed only a fraction of a reconstruction unit. When a reconstruction process examines a partially-reconstructed unit, it has the option of reconstructing only the

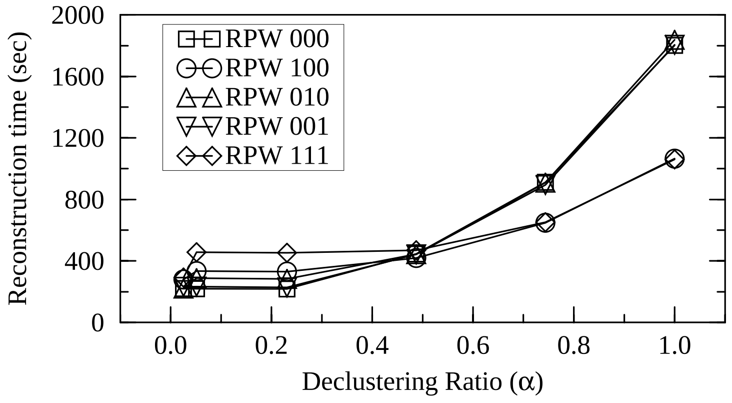


Figure 4.5: Reconstruction time for five combinations of the variations.

Refer to Figure 4.4 for a description of the legend.

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, the implementation evaluated here always chooses the latter option. Hence, most of the potential benefits to reconstruction time from user- and piggy-backed-write options cannot be realized. 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 4.5 shows that enabling 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. Enabling any of the reconstruction options incurs a significant reduction in the efficiency of the replacement disk by forcing it to service far more randomly located accesses. This accounts for the significant increase in reconstruction time at low α when the reconstruction options are enabled.

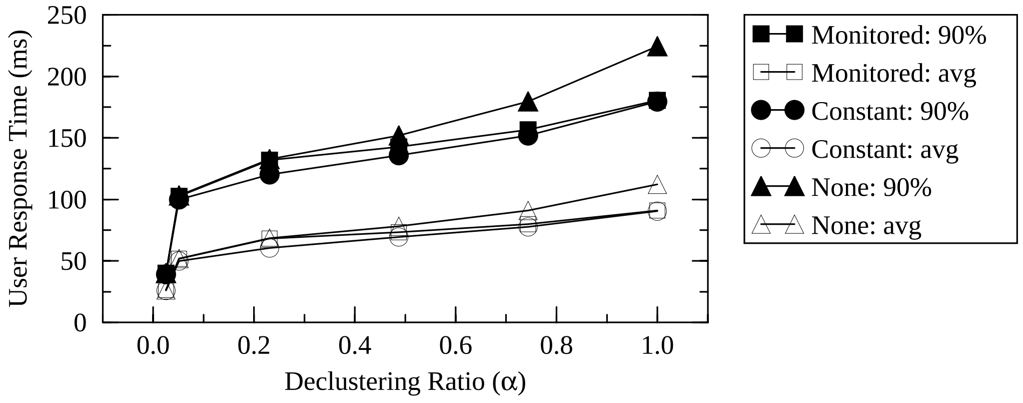


Figure 4.6: Evaluating monitored redirection of reads: response time.

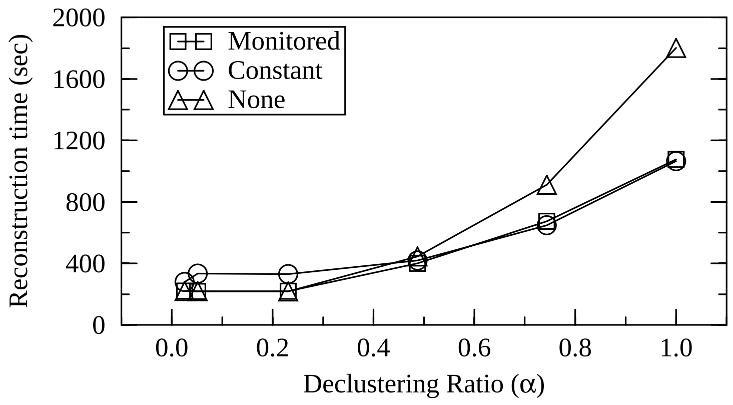


Figure 4.7: Evaluating monitored redirection of reads: reconstruction time.

4.4.1.3. Dynamic use of reconstruction options

As Figure 4.5 shows, the value of each reconstruction algorithm option depends on whether the replacement disk or the surviving disks limit the rate of reconstruction. This effect is dependent upon both the array's declustering ratio and the amount of the failed disk's data reconstructed so far. 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 4.6 and Figure 4.7 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. The simulations dynamically apply only the redirection of reads option because it is the only option that significantly affects recovery mode performance for the OLTP workload. This dynamic reconstruction algorithm is called the *monitored redirection* option. The monitoring scheme employed is as follows: the controller records the duration of each disk busy and idle period, and every 300 accesses it generates a new estimate for the utilization of each disk. If the replacement disk utilization is higher than the average surviving disk utilization, the controller declares the replacement disk to be the bottleneck, and disables redirection of reads until the next time the estimates are updated. If the opposite is true, the controller declares the surviving disks to be the bottleneck, and enables redirection of reads until the next utilization estimate update.

As Figure 4.6 shows, the response-time performance of monitored redirection is actually worse at moderate and low declustering ratios than the constant-redirection case. This is because redirection of reads is uniformly beneficial to response time, but monitored redirection disables it at low α. Figure 4.7, however, shows that monitored redirection minimizes reconstruction time because it is always the case that either the replacement disk or the surviving disks limit the reconstruction rate.

4.4.1.4. Summary

For the OLTP-like workload considered here, the only effective work-reducing variation to the disk-oriented reconstruction algorithm is redirection of reads. This option improves user response time by as much as 10% - 20% at high declustering ratios, while reducing reconstruction time by as much as 40%. However at a low declustering ratio, redirection of reads benefits response time by only a very small amount, and lengthens reconstruction time by over-utilizing the replacement disk. A dynamic application of this option based on monitoring disk utilizations achieves much of its benefits without its costs, independent of the declustering ratio.

4.4.2. Head following

User accesses and reconstruction accesses interfere with each other by causing the disk heads to be continually moved between the current reconstruction point and the region of the array being accessed by the users. More specifically, when a user access arrives at the array while reconstruction is ongoing, it incurs a potentially long seek from the current reconstruction point to the requested data, performs exactly one access there, and then, if no other user access arrives, incurs another long seek back to the reconstruction point. Thus reconstruction can lengthen average user response time by breaking up any locality of reference in the user access stream, and user accesses can elongate reconstruction time by causing more of the disk's time to be spent positioning rather than transferring data. This observation leads to the idea of modifying the reconstruction algorithm so that it tracks the user-induced movement of the disk heads, and attempts to select parity stripes to reconstruct so as to minimize the seek time induced by the reconstruction accesses.

This section develops two such algorithms and evaluates their efficacy. These techniques both yield negative results, that is, they degrade reconstruction performance rather than improve it, and analyzing of the reasons behind this leads to the conclusion that head-following is not viable in random-workload environments such as OLTP. The first subsection describes the basic approach to the problem, and demonstrates a severe buffer-memory management problem inherent in it. The second and third subsections propose and evaluate potential solutions to this problem, but show that neither approach achieves the intended benefits. The fourth subsection categorizes and summarizes the reasons why head following is not viable in random-workload environments. Given these reasons, the fifth subsection investigates whether head following might be effective under other workload conditions. The final subsection summarizes the results.

4.4.2.1. Basic head-following algorithm, and its shortcomings

Each disk in a block-stripe redundant disk array typically services a different user access, and so there is, in general, no correlation between the position of any two heads. It is therefore necessary to apply the head-following technique on a per-disk basis, rather than a per-parity-stripe basis. This means that the reconstruction process associated with each surviving disk should track the head movement on its assigned disk, and pick a surviving data or parity unit to fetch so as to minimize the seek time.

Modern disks typically hide their geometry, and thus their absolute head position, from the outside world in order to facilitate modular design; the typical interface exported by a disk consists of a linear address space of sectors, each of which has an address and can be read or written [ANSI86]. Furthermore, the head settling time may vary from accesses to access, and the rotational speed of the media may vary by a few percent over time [Maxtor89]. These facts make it difficult or impossible for the host or array controller to know the head position at any given time with absolute certainty. Fortunately, however, sectors that are sequential in the address space exported by the disk are typically sequential on the physical media in order to maximize performance. This means that the controller can use the address of the last sector accessed on a disk as an indicator of that disk's head position,[2] and thus can monitor the head position without incurring any overhead on the drive itself. The address of the last sector accessed is, of course, only an approximation to the current head position, since the media rotates continually under the heads. This means that the head following algorithm described below may not actually select the unit that is actually closest to the current head position, in the sense that some other needed unit may be accessible in a slightly shorter period of time. However, since the goal of head following is to reduce the seek time penalties incurred by the concurrent servicing of user requests and reconstruction requests, the accuracy of the approximation is sufficient.

The basic head-following algorithm is as follows. Each reconstruction process submits a low-priority request to its disk queue, as in the regular disk-oriented algorithm. Each such request is tagged with the address of a "callback" routine, which is invoked with a specific set of parameters immediately before the reconstruction access is initiated. When invoked, the callback function obtains the address of the last sector accessed, either from its parameter list or from a locally-maintained data structure, and assumes the disk heads are positioned over this sector. From the disk sector address and the disk identifier, it computes the logical sector address of the reconstruction units[3] immediately adjacent to this sector. It then consults the reconstruction map to see if any of these reconstruction units are both needed to reconstruct some unit on the failed drive, and not previously fetched. The callback continues to search the reconstruction units adjacent to the current head location, widening the search at each step, until it finds such a reconstruction unit or

2. Most modern disks also incorporate a *read-ahead* function, whereby after an access ends, they continue to read sequential sectors from the media into an internal buffer, so that if the user asks for these sectors, they can be supplied with very low latency. Since the read-ahead buffer is typically less than one cylinder in size, this does not invalidate the use of the last sector address accessed as an indicator of the disk head position.

3. Recall that the "reconstruction unit" is the unit of data or parity read or written per reconstruction access, and its size may be different in size than the data and parity units in the array.

determines that no sectors on the indicated drive are still needed for reconstruction. The callback initiates a low-priority reconstruction read on the first such unit that it finds, or signals completion if it finds none.

The problem with this approach is that, as described, it leads to almost immediate deadlock of the reconstruction process. Since the workload causes the disk heads to be uncorrelated with respect to each other and there are generally many thousand parity stripes in the array, head following causes each reconstruction process to fetch a reconstruction unit from a different parity stripe. When the read operations complete, these units are submitted to the buffer manager, and consume *C* reconstruction buffers. Each process then picks another (uncorrelated) reconstruction unit to fetch, and *C* more buffers are used. Thus the reconstruction process rapidly exhaust the pool of available buffers, usually within the first two or three fetches after the initiation of reconstruction. When no reconstruction buffers are available to accept a reconstruction unit that has been fetched, the submitting process blocks until a buffer is freed by another process. Buffers are only freed when all the surviving reconstruction units in the corresponding parity stripe have been fetched and XORed together, and the results written to the replacement disk. Since all the reconstruction processes are fetching units for different parity stripes, no buffers are ever filled and freed, and so reconstruction deadlocks.

The central problem is that in order to make progress, the reconstruction processes must work on a correlated set of parity stripes. In other words, when a reconstruction process selects a reconstruction unit to fetch, there must be a high probability that the other reconstruction processes will fetch units for the same parity stripe within a relatively short period of time, so that the corresponding reconstruction buffer can be filled, written, and recycled for use by another parity stripe. The next two subsections describe mechanisms that a reconstruction process can use to select the next reconstruction unit to fetch, so as to avoid the deadlock condition described above.

4.4.2.2. First approach: fetch closest active parity stripe

Recall from Section 4.2.3.1 that only the first reconstruction process submitting a unit for a particular parity stripe needs to acquire a reconstruction buffer. Subsequent submissions can simply XOR the fetched data into this buffer, and hence need not allocate a new one. This leads to the idea of causing the reconstruction processes to initiate reconstruction on a new parity stripe (one for which no other units have been fetched, and so a reconstruction buffer will be required at submission time) only if there is a buffer available to accept the fetched unit when it is submitted. If no buffers are available, the reconstruction process must fetch a reconstruction unit from a parity stripe that is already under reconstruction, in order to guarantee that it will not have to acquire a buffer at submission time. Since there are many parity stripes under reconstruction at any one time, the reconstruction process generally has a choice of which one to select. The obvious candidate is the one that is closest to the current position of the disk heads.

More specifically, this approach to head following is as follows. When the callback occurs, indicating that a disk is free to service a reconstruction access, the head-following code checks to see if any reconstruction buffers are available for use. If so, the callback routine selects the next reconstruction unit to be fetched to be the as-yet-unfetched unit that is closest to the current position of the disk heads. It then acquires a reconstruction buffer, zeroes the data portion of it, marks the buffer as empty, and submits it to the buffer manager. This places the corresponding parity stripe “under reconstruction”, meaning that other processes may now submit units for that parity stripe without blocking, since a buffer has already been acquired. The callback then initiates the fetch for the indicated reconstruction unit, and proceeds normally. If, however, no buffers are available at the time of the callback, the routine scans the list of parity stripes that are currently under reconstruction, and identifies the parity stripes that still require a unit from the indicated disk. From the parity stripes so identified, the callback routine picks the one that is closest to the current disk head position, and initiates a fetch on the corresponding reconstruction unit.

In the unusual case that there are no available reconstruction buffers and no parity stripes currently under reconstruction that still require a unit from the indicated disk, the reconstruction process has two choices; it can release the disk, delay for some fixed period of time and then re-execute the callback routine, or it can just select a unit to be fetched and accept the buffer stall that may occur upon submission of the unit to the buffer manager. The implementation in raidSim uses the latter option, with the following modification. When this situation is encountered, the callback routine selects the closest needed reconstruction unit, and places the corresponding parity stripe identifier on the *buffer wait*

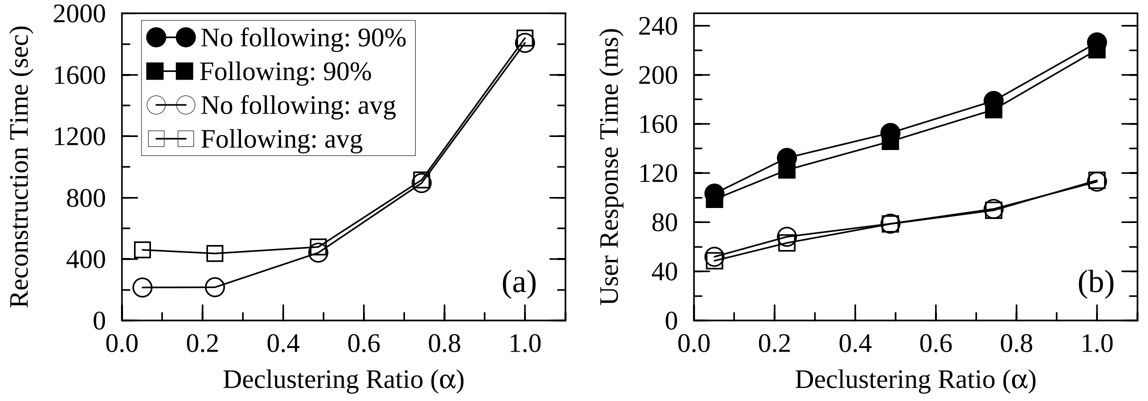


Figure 4.8: Evaluating the “fetch closest active” type of head following.

Part (a) shows reconstruction time, and part (b) shows average and 90th percentile user response time during reconstruction.

list, which is a list of those parity stripes for which reconstruction has been initiated but no reconstruction buffers have been acquired. Parity stripes identified on this list are given first priority for reconstruction buffers as they are freed, so as to minimize any buffer stall time that occurs when processes submit units for these parity stripes. Because of this prioritization, reconstruction processes that find no reconstruction buffers available, and no parity stripes under reconstruction still requiring a unit from their disk, check the buffer wait list before initiating reconstruction on a new parity stripe. If any parity stripe in the buffer wait list requires a unit from the disk assigned to the reconstruction process, the closest such unit is selected for fetch rather than initiating reconstruction on a new parity stripe.

The goal behind this approach to head following is to distribute the parity stripes currently under reconstruction throughout the array, rather than having them concentrated at a single point. This assures that when no buffers are available, there will be a high probability of finding something close by to be fetched, and the average seek distance incurred by a reconstruction access will be reduced.

Unfortunately, Figure 4.8a shows that this approach actually lengthens reconstruction time over the no-head-following case, by a significant amount at low declustering ratios. The corresponding response-time plot (Figure 4.8b) indicates that there is essentially no difference between the head-following and no-head-following cases. This behavior is explained by noting that when head following is disabled, reconstruction is largely a sequential process. Although user accesses cause potentially long seeks to and from the reconstruction point, it is often possible when head following is disabled to fetch several sequential reconstruction units before a user access arrives and moves the disk heads again. These sequential reconstruction accesses occur at a very high bandwidth, since the seek and rotate penalties incurred between each access are essentially zero. When head-following is enabled, there is no single reconstruction point, and so reconstruction processes can fetch units sequentially only if there is a constant supply of available buffers. This is not the case at any value of α (that is, no matter whether the surviving disks or the replacement disk is the reconstruction bottleneck), because as soon as a buffer is freed, some reconstruction process grabs it. Thus the average number of free buffers available at any moment in time is close to zero at all values of the declustering ratio, and so the reconstruction processes are forced to fetch the “closest active unit” most of the time. This means that nearly every reconstruction access incurs a seek and rotate penalty. These penalties are smaller than the average seek and rotate penalties listed in Table 2.3 because the closest active reconstruction unit is always selected, but it is still the case that the reconstruction process incur some positioning penalty on nearly every reconstruction access. By way of contrast, although the per-access seek and rotate penalties paid in the no-head-following case are larger, they occur much less frequently since many reconstruction units can often be fetched sequentially. This accounts for the poor performance of the head following algorithm with respect to the default disk-oriented algorithm.

The reconstruction performance of the head-following algorithm is worst at low α because the user access rate is fixed at 14 user I/Os per second per disk in these simulations. This implies that total per-disk access rate is lower at low values of α., because the per-disk load increase is less at a low declustering raio than at a high one. At lower access rates, the non-head-following algorithm is able to perform more sequential reconstruction accesses per user access, and so the difference between the two algorithms increases as the declustering ratio is decreased.

Figure 4.9 verifies this explanation by showing a histogram of the time taken to complete a reconstruction access in one specific example (C = 40, G = 10), for one surviving disk and for the replacement disk. The figure shows that with head-following disabled,

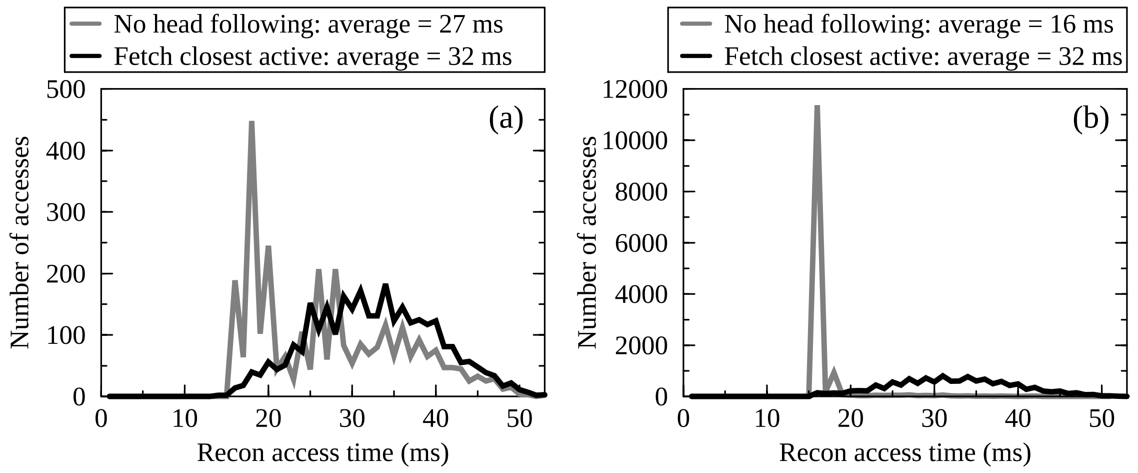


Figure 4.9: Example reconstruction access time histograms.

Part (a) shows the histogram for a surviving disk, and part (b) for the replacement disk. The example shown here is for C = 40 and G = 10 (α = 0.23). Note that the two plots have different scales on the y-axes.

many reconstruction accesses occur sequentially and thus get done very quickly, whereas with head-following turned on, most accesses incur a seek and rotate penalty, and this substantially lengthens the average reconstruction access time.

It's clear, then, that the strategy of head following and fetching the "closest active reconstruction unit" when there are no buffers available to fetch a new unit is not viable under OLTP-like workloads, because it ruins the sequentiality of the reconstruction process. The next section describes a head-following approach intended to avoid this problem.

4.4.2.3. Second approach: multiple reconstruction points

In order to maintain sequentiality, it's necessary to structure the reconstruction algorithm such that each process fetches sequentially forward from a well-defined point within the array. In order to get the advantages of head-following, the processes must be able to select the next reconstruction unit to fetch from a set of possibilities that are distributed over the address space of the array. These two facts lead to the idea of maintaining multiple independent "current reconstruction points", that is, points within the address space of the array from which a reconstruction process can fetch sequentially forward, instead of just one such point as in the default disk-oriented algorithm. When a user access moves the disk heads away from the current reconstruction point and then completes its I/O operation, the corresponding reconstruction process can reduce the seek time back to the reconstruction point by selecting the closest reconstruction point, rather than returning to the original point.

If each process selects its current reconstruction point solely on the basis of shortest seek distance, then the deadlock condition discussed above can occur. To see this, consider a system with two reconstruction processes P1 and P2, and a parity stripe S requiring a reconstruction unit from the disks assigned to both P1 and P2. The following sequence causes deadlock. P1 fetches its unit from S and installs it in the buffer manager, and then a user access intervenes and moves the heads on disk 1 to the vicinity of another reconstruction point. At this point in time, the reconstruction buffers become exhausted. P1 begins fetching at the new reconstruction point, encounters a parity stripe for which no other units have been fetched, and blocks upon submission. Meanwhile, P2 encounters a "new" reconstruction unit at any reconstruction point other than the one at which it will encounter the required unit in S, and hence P2 blocks upon submission as well. In order to free a buffer to release either process, one of the two processes must visit another reconstruction point and fetch the required unit. Since both processes are blocked, deadlock ensues.

This problem of buffer memory management extends beyond the issue of deadlock; it also manifests itself as poor reconstruction performance. Inherent in the reconstruction task is the requirement that multiple reconstruction processes work in conjunction on the same parity stripe, or set of parity stripes, in order to make progress. When there are multiple reconstruction points in the array, partially-filled buffers (buffers for which some but not all required units have been fetched and XORed) tend to languish in the buffer manager queues, because there is no impetus for the reconstruction processes required to fill and free them to move back to the reconstruction point where they will encounter the corresponding parity stripes. By way of contrast, with a single reconstruction point, all reconstruction processes are forced to work on the same relatively small set of parity stripes, and so the processes fill, write, and recycle buffers at a much more rapid pace.

In order to avoid these problems, a reconstruction process cannot select the next reconstruction point to visit solely on the basis of shortest seek distance; it must also consider buffer memory management issues. The remainder of this section describes two approaches to this problem, and provides the reasons why they did not perform as hoped.

The simplest approach to the buffer management problem is for each reconstruction process to select the closest reconstruction point only when there are available reconstruction buffers, or when the next few parity stripes encountered at that point are already under reconstruction. When neither of these conditions hold, a process selects the next reconstruction point to visit by scanning the list of parity stripes currently under reconstruction to determine how many buffers can be freed by moving to each reconstruction point, and moving to the one that will free the maximum number of buffers. When no buffers can be directly freed by moving to any reconstruction point, that is, there are no parity stripes under reconstruction that require *only* a unit from the disk under consideration to be full, the scanning process selects the reconstruction point at which it will encounter the most active parity stripes still requiring a unit from the indicated disk.

Simulating this approach yielded results very similar to those shown in Figure 4.8; response time was not improved, reconstruction time was not improved at high α, and reconstruction time was degraded at low α. The reason is simple: reconstruction buffers are grabbed by a reconstruction process very soon after being freed, and so it is rare that there are ever any buffers on the free list. It takes only one fetch by one reconstruction process to consume a buffer, but it takes several coordinated fetches by several reconstruction processes to free a buffer. Since there are rarely any available reconstruction buffers, reconstruction process are rarely able to move to the closest reconstruction point, and so they nearly always end up moving to the reconstruction point where they can free the most buffers. Thus, this algorithm does not actually head-follow at all; it “buffer-follows”. In a typical example ($C = 40$, $G = 10$, two reconstruction points), there were 517,179 total invocations of the head-following callback function, of which only 3,352 (less than 1%) found a buffer available. Even at $\alpha = 1.0$, reconstruction processes found free buffers only about 50% of the time, which was not sufficient to offset the languishing-buffers problem described above. Thus there is no advantage to having multiple reconstruction points using this approach, since having a single reconstruction point maximizes the algorithm’s ability to recycle buffers.

The above approach is based on avoiding the condition that causes an excessive number of reconstruction processes to block on buffer submissions. An alternative approach would be allow this condition to occur, but to detect it when it happens and invoke some mechanism to break the potential deadlock. The premise here is that if the number of reconstruction points is small, a random user workload should keep the reconstruction processes moving between the reconstruction points, and thus the actual occurrence of a condition where an excessive number of reconstruction processes have blocked on buffer submission should be relatively rare. If this were the case, then even an expensive mechanism for detecting this condition and releasing the affected process would be acceptable, since it would not occur very often.

The easiest way to release a process that is stalled trying to submit a buffer is to acquire a buffer, tag it with the identifier of the parity stripe for which the process is trying to submit, and submit it to the buffer manager. This causes the manager to wake up all processes that are waiting on that parity stripe, after which each returns to its normal fetching of reconstruction units. Accordingly, in this approach, the buffer manager reserves a small number of reconstruction buffers for the purpose of breaking deadlock conditions as they arise. When the number of processes waiting on a particular parity stripe crosses some pre-defined threshold, the buffer manager commits a buffer from this reserved set to the corresponding parity stripe, thus releasing all the waiting processes[4]. Similarly, when the total number of reconstruction processes waiting on buffers crosses a threshold, the buffer manager commits a reserved buffer to the parity stripe upon which the most processes are waiting. When the number of available reserved buffers drops below some pre-defined number, it indicates that excessive buffer stall conditions are arising faster than reserved buffers are being freed by the completion of reconstruction, and so the buffer manager switches from this *deadlock-breaking mode* to *deadlock-avoidance mode*. In avoidance mode, the reconstruction processes operate as described previously: a process moves to the closest reconstruction point only if it finds a regular (that is, non-reserved) buffer available, and otherwise moves to the reconstruction point at which it can free up the most buffers. When the number of reserved buffers rises back above the threshold, the buffer manager re-enters deadlock-breaking mode.

The goal behind this is to allow the reconstruction processes to move to the closest

4. In order to avoid reserving these buffers, the buffer manager could instead choose to steal an in-use buffer to break the deadlock, and force the reconstruction on the corresponding parity stripe to be restarted.

reconstruction point in all cases except when this policy genuinely causes an excessive number of processes to stall attempting to submit. However, simulating this approach revealed that its primary premise is flawed: it is not the case that excessive buffer stalls are relatively rare events so long as the number of reconstruction points is small. Rather, the buffer-stall thresholds are crossed constantly, and thus the supply of reserved reconstruction buffers is rapidly depleted, and the algorithm reverts to buffer-stall avoidance mode. In avoidance mode, the reserved buffers are recycled but never re-committed to break a deadlock, and so the supply builds back up, and eventually the algorithm re-enters deadlock-breaking mode. However, as soon as this occurs, the supply of reserved buffers is depleted at the same rate as before, and so the algorithm is unable to achieve its goals.

4.4.2.4. Summarizing: head following is not viable under a random workload

After describing the failure of both the fetch-closest-active and multiple-reconstruction-points approaches, it makes sense to step back and summarize the problems encountered in trying to improve reconstruction performance via head-following in a random-workload environment. There are two fundamental issues:

1. Head following interferes with the sequentiality of the reconstruction process. It makes it difficult for a reconstruction process to issue multiple consecutive accesses in the absence of user activity on the disk, and this drastically reduces the bandwidth available for reconstruction (see Figure 3.21). This problem is especially severe on the replacement disk, which observes an access pattern that is almost completely sequential using a single-point reconstruction algorithm, but almost completely random using a head-following algorithm.

2. Head following obstructs the efficient recycling and re-use of reconstruction buffer memory. Reconstruction progress requires that multiple reconstruction processes coordinate on a single parity stripe. Using a single-reconstruction-point algorithm forces all reconstruction processes to work on the same small set of parity stripes, whereas any form of head following reduces or eliminates this tendency to focus the reconstruction resources (processes and memory) on a single point in the array.

For these reasons, head following is not viable under workloads (such as OLTP) that are characterized by a large number of independent processes concurrently accessing small, randomly distributed units of data.

4.4.2.5. Evaluating head following on other workloads

Having found that head following yielded negative results for the primary workload under consideration, it’s worthwhile to briefly investigate whether there exist workload conditions under which head following is effective. This section evaluates the performance of the two head following algorithms under three workloads: a synthetic, concurrent, random workload that exhibits a high degree of spatial locality, a synthetic, single-process workload that exhibits a high degree of sequentiality, and the traced UNIX workstation workload described in Section 3.3.4.3. All unspecified parameters were set to the values given in the tables in Section 2.3. To keep the discussion brief, the results are presented in summary form only.

The first workload was the same as the one described in Table 2.4, with the modification that in 80% of the accesses, the starting points within the array were uniformly distributed within a contiguous region comprising 10% of the array’s data space, while in the other 20% the starting points were uniformly distributed throughout the array’s data space. This causes the disk heads to remain relatively close together, which might make it easier for the head-following code to coordinate the processes on the same set of parity stripes. However, the simulation results showed behavior very similar to that exhibited in Figure 4.8; there was no benefit to reconstruction time at high α, reconstruction time was worse at low α, and response time was not strongly affected. The reason for this is that when the disk heads are largely confined to small region of the array, head following benefits reconstruction-mode performance only for that region of the array. The problems described in the preceding section still apply when reconstructing the remaining 90% of the lost data, and the small benefit achieved within the local region is not sufficient to offset the loss of sequentiality and the additional positioning overhead due to buffer memory management. Furthermore, since the amount of buffer memory available in the controller is typically less than 1% of the size of the array, the region of the array to which the heads are confined must be extremely small for head-following to be effective even within the local region.

The results on the random-but-local workload suggest head following might be effective if a workload had a form of “roving locality”, that is, if the application accessed data within a very small region of the array for a short period of time, and then moved to another small region. This leads to the idea that a low-concurrency application that sequentially reads a set of large contiguous files might induce sufficient synchronization between the disk heads to render head following effective. To model this, we simulated a single process performing 1 MB read accesses, each of which has a 90% probability of being sequential with respect to the previous access.[5] This should be a very favorable workload for head following, because each access uses most of the disk arms, and thus the heads should be very well synchronized. Simulating this workload indeed shows a limited benefit to the "fetch-closest-active" type of head following (Section 4.4.2.2). Reconstruction time was reduced by about 20% when α was near 1.0, but was again significantly degraded at low α. User response time was improved by between 5% and 10% at all values of α. Although this performance could perhaps be improved somewhat by specifically tuning the head-following algorithm to the high degree of sequentiality in the user workload, the extreme specificity of the workload, and the less-than-stellar performance improvements achieved, make the additional complexity and computation overhead of head-following unattractive.

Finally, for completeness we simulated the performance of the head-following algorithms on the UNIX workstation workloads described in Section 3.3.4.3. The results were very similar to those reported in Figure 4.8, with the primary difference being that head following had less of a negative effect at low α. This was due to the large fraction of writes in the traced workloads. Recall that a user write operation that targets a previously reconstructed unit must always update the data on the replacement disk, in order to assure that the corresponding parity unit does not become out of date. This interferes with the sequentiality of the reconstruction writes to the replacement disk, and thus slows reconstruction. Since the traced workload had a higher fraction of writes than the synthetic workloads, this interference was more severe, and thus the interference due to head-following was less noticeable.

4.4.2.6. Summary

This section described techniques for causing the reconstruction process to track the

5. This represents a geometric distribution, and thus the expected number of sequential megabytes read between each non-sequential access is 1.0/0.1 = 10 MB.

user-induced motion of the disk heads, and attempt to reconstruct in the areas of the array currently being accessed by the users. The intended benefit is that the head positioning time incurred by both user- and reconstruction-accesses should be reduced. Two problems became evident: head following ruins the sequentiality of the reconstruction process, and interferes with the efficient recycling and re-use of the reconstruction buffers. The section described several attempts to overcome these problems, none of which were successful. The conclusion to be drawn is that head following is not viable in random-workload environments. Analysis of other workloads showed only limited benefits in very specific cases. Although this section did not explicitly analyze the issues of computational overhead and implementation complexity, the head following techniques were all significantly slower and more complicated that the simple disk-oriented algorithm previously described. Thus, we found no compelling reason to adopt this approach.

4.5. Conclusions

This chapter demonstrated that there are three primary considerations in the design of the reconstruction algorithm. First, it is essential that the algorithm absorb as much as possible of the array's bandwidth that is not absorbed by the user accesses. Second, the algorithm must preserve the inherent sequentiality of the reconstruction process, since a disk drive is able to service sequential accesses at many times the bandwidth of random accesses. Finally, the algorithm must concentrate its resources (work on a relatively small set of parity stripes at any one time) in order to avoid severe buffer memory management problems. These considerations lead to the development of the disk-oriented algorithm, and to the rejection of head following.

The second major concern in the design of a reconstruction algorithm is determining whether the reconstruction bottleneck will be the surviving disk or the replacement disk. The *redirection of reads, piggybacking of writes,* and *user-writes* reconstruction variations allow some degree of control over the division of user-induced workload between the replacement and surviving disks, and thus can, under certain conditions, improve the reconstruction-mode performance of the array by diverting work from the bottleneck resource to the one with available bandwidth. We found that in OLTP-like workloads (random, read-dominated, small-access), the latter two variations had little effect on the overall performance. Taking all this into account, the strategy of *monitored redirection* yielded optimal reconstruction time, and good, but not quite optimal, user response time.

The tertiary issues discussed in this chapter included the amount of memory required in the disk controller, and the interaction of user requests with reconstruction requests. We showed that the algorithms presented here were largely insensitive to the amount of buffer memory available, which allows them to provide good performance at low cost. We argued that, since there is no motivation to use a large amount of memory for reconstruction in the controller, the number of parity stripes under reconstruction at any moment will be relatively small, and thus the frequency with which user accesses will interact with reconstruction accesses will be very low. Thus we proposed a very simple approach to handling this interaction, based on forcing a user access to wait for an expedited reconstruction to complete before performing its disk operations.

Chapter 5: Distributed Sparing

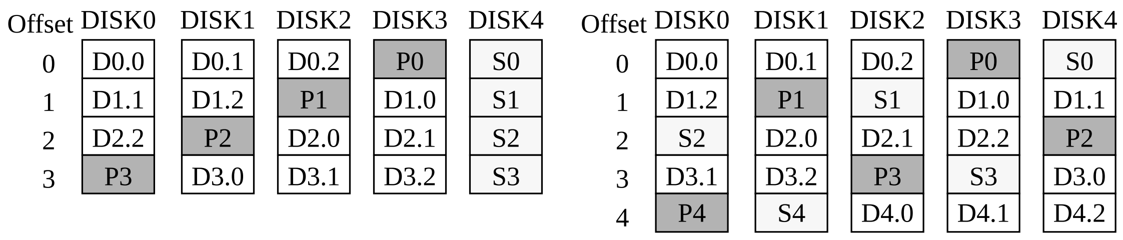
As discussed in Chapter 2, redundant disk arrays typically maintain one or more on-line spare disks, so that reconstruction can be immediately initiated when a failure occurs. This chapter will show that there is strong motivation to distribute the capacity of the on-line spare disk(s) amongst all the disks in the array, instead of dedicating one or more disks as spares. The chapter shows that combining this *distributed sparing* technique with parity declustering (as developed in Chapter 3) eliminates the spare disk as the reconstruction bottleneck at low values of the declustering ratio (α), and thereby allows for extremely rapid failure recovery. A few prior studies have investigated the benefits of distributed sparing in RAID Level 5 arrays; Section 5.1 describes them and summarizes their conclusions. In order to introduce the details of the technique, Section 5.2 describes its implementation in RAID Level 5 arrays. Unfortunately, the simple layout solution described in Section 5.2 does not apply directly to parity-declustered arrays, and so Section 5.3 develops a new layout mechanism, again based on balanced incomplete block designs, that achieves the required distribution. Section 5.4 addresses the re-design of the disk-oriented reconstruction algorithm to support distributed sparing. The primary issue in this section is that distributing the spare disks' capacity (hereafter referred to as the *spare space*) eliminates the distinction between surviving-disk processes and the replacement-disk process. Together, Sections 5.3 and 5.4 put into place all the elements needed to implement distributed sparing, and so Section 5.5 evaluates via simulation the reconstruction-mode benefits of this approach. The analysis shows that the technique does indeed eliminate the spare disk as the reconstruction bottleneck at low declustering ratios. The results show reconstruction times that decrease monotonically with α, dipping as low as about 30 seconds in large arrays. Section 5.6 describes a related study that uses complete block designs to implement distributed sparing, showing that the approach taken in this study has several advantages, but is only feasible for small arrays. Section 5.7 concludes and summarizes the chapter.

5.1. The benefits of distributed sparing

There are two primary motivations for distributing spare space. First, as Menon and Mattson [Menon92b] noted, the spare disks in a RAID Level 5 disk array remain idle except during failure recovery, and so represent a grossly underutilized system resource. Distributing the spare space allows the idle drives to service user requests, thereby improving the performance of the array without increasing the number of disks. The benefit of using these extra actuators clearly decreases with the number of disks in the array; adding one extra actuator to an N-disk array yields a performance improvement factor of $1/N$. Menon and Mattson also evaluate a third alternative, *parity sparing*. In this approach, upon detecting a failure, the controller merges two parity groups, and uses the parity units in one group as the spare units for the other. The study concluded that distributed sparing was a better alternative as it allowed for faster reconstruction, and so the remainder of this chapter does not consider parity sparing.

The second advantage of distributed sparing, more pertinent to the topic of this dissertation, is that it allows for significantly improved reconstruction performance in parity declustered arrays at low values of the declustering ratio (α). Recall from Section 3.3.2 that for low values of α, the reconstruction time is limited by the rate at which recovered data and parity units can be written to the replacement disk. This enforces a lower bound on the reconstruction period equal to the minimum time required to write the entire contents of one disk (about 200 seconds for the IBM 0661 drives described in Section 2.3.2). Distributing the spare space amongst the disks of the array increases the maximum reconstruction-write bandwidth from that of a single disk toward the aggregate write bandwidth of the array, eliminating the spare disk as the reconstruction-time bottleneck. This promises extremely fast reconstruction. Note that distributed sparing does not significantly improve reconstruction time in declustered arrays with a high declustering ratio (recall that parity declustering with $\alpha = 1.0$ is equivalent to RAID Level 5), because in this case it is the surviving disks, rather than the replacement, that constitute the reconstruction bottleneck.

The ultimate goal of any high-availability system is that failure recovery should be entirely transparent to the system's users, in terms of both functionality and performance. A data storage subsystem that combines parity declustering using a low declustering ratio



Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	S0
1	D1.1	D1.2	P1	D1.0	S1
2	D2.2	P2	D2.0	D2.1	S2
3	P3	D3.0	D3.1	D3.2	S3

(a) Dedicated Sparing

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	S0
1	D1.2	P1	S1	D1.0	D1.1
2	S2	D2.0	D2.1	D2.2	P2
3	D3.1	D3.2	P3	S3	D3.0
4	P4	S4	D4.0	D4.1	D4.2

(b) Distributed Sparing

Figure 5.1: Contrasting sparing alternatives in RAID Level 5 arrays.

The areas labeled P *and* S *represent parity and spare units for each parity stripe, respectively. These layouts are derived by first placing the data units of each parity stripe on disks in such a way as to adhere to criterion six of Section 3.2.1 (maximal read parallelism), and then placing the parity and/or spare units in the remaining locations.*

with distributed sparing is uniquely suited to this goal. A low declustering ratio implies that degraded- and reconstruction-mode performance are not significantly different from fault-free performance. Distributed sparing eliminates the replacement-disk bottleneck, thereby minimizing the duration of this performance degradation as well.

5.2. Distributed sparing and its implications on failure recovery

Figure 5.1 contrasts a dedicated-spare RAID Level 5 array to an equivalent distributed-spare array. The spare space in the distributed-sparing case is laid out by simply allocating one extra unit per parity stripe to contain the spare unit for that stripe, and rotating this unit along with the parity unit. This guarantees an even distribution of both parity and spare space over the disks comprising the array.

In fault-free mode, the spare units are unused. When a failure occurs, the host or array controller reconstructs the data that was lost from each parity stripe, and stores it in the spare unit for that parity stripe. Note that the spare units lost due to a disk failure do not have to be reconstructed, and so the host or array controller needs to reconstruct only C-1 out of every C units, where C is the total number of disks in the array. When all lost data and parity units have been recovered, the array is said to “reconfigured”, since it is once again single-fault tolerant, but now contains one fewer disk and has no spare space. The array operates at normal performance in reconfigured mode, except that parity is typically no longer balanced over the array, and adherence to layout criterion six (Section 3.2.1) is generally lost. At some point in time after the failed disk is physically replaced by a new drive, the reconstructed data and parity units must be copied from the spare locations to their original locations on the new drive. Since the reconfigured array operates at normal performance and is single-fault tolerant, this “copyback” operation can be deferred until a time when the user access rate to the array is low, such as midnight, a weekend, or a period of scheduled downtime, and so is not significant with respect to reconstruction performance. Either of the layouts of Figure 5.1 can be extended to support multiple spare disks per group, and thereby tolerate additional disk failures between the time at which the first drive fails and the point at which the copyback is initiated, so long as no drive fails while another is under reconstruction.

The advantages of distributed sparing in RAID Level 5 arrays are limited to the use of the extra actuator or actuators in fault-free mode, and the necessity to reconstruct only C-1 out of every C units on a failed drive. These effects are small for all but the smallest-scale arrays. In particular, distributed sparing does not yield significantly faster reconstruction in RAID Level 5 arrays because, as was observed in Section 3.3.2, the reconstruction rate is limited by the rate at which data and parity can be read from the surviving disks, rather than the rate at which reconstructed units can be written to the distributed spare space. The next section derives a layout for distributed sparing in parity declustered arrays that allows the array to reap the additional benefit of extremely fast reconstruction.

5.3. Implementation in declustered parity arrays

To simplify the exposition of our layout strategy, we consider the process of deriving a distributed-sparing layout for declustered arrays to have two distinct steps, as illustrated in Figure 5.2.

The first step consists of statically allocating spare units to disks, where the term “static” refers to the fact that the set of units selected to be reserved as spares does not depend on which disk has failed. Allocation is the process of determining which units on which disks in the array will be permanently reserved for use as spare units. Although it is not strictly necessary that the allocation be static, static allocation simplifies the algo-

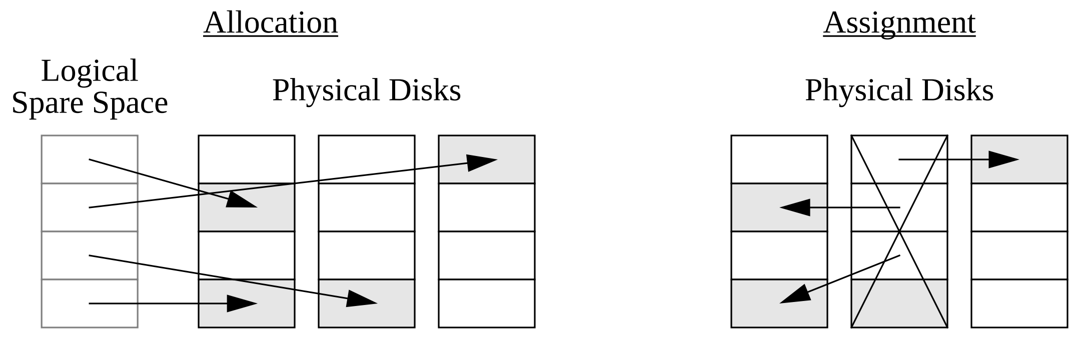


Figure 5.2: Spare-space allocation versus spare-unit assignment.

Spare units are permanently allocated to physical locations (shaded) in the array. Upon detecting a failure, the controller assigns each (non-spare) failed unit to one of the pre-allocated spare units on another disk. As each failed unit is reconstructed, it is written to its assigned spare unit. When the failed disk is replaced and the reconstructed data copied from the spare locations to the new disk, the assignment is discarded, but the allocation remains.

rithms that map logical addresses to physical addresses (refer to layout criterion 4 on page 52). Logically, the process of allocation occurs when the layout is designed; a static allocation is permanent, and inherent in the structure of any given layout.

When a failure occurs, the controller assigns to each failed unit a spare unit that will replace it in reconfigured mode. Thus the second step in determining the layout is to compute, based on the identifier of the failed disk, an assignment of failed units to spare units. Logically, this computation occurs only after a disk has failed, but in practice we pre-compute the assignment of failed units to spare units for every possible spare disk. This assures that reconstruction will not have to be deferred while the controller performs the spare-assignment computation.

There are five primary considerations in deriving the allocation and the assignment, which we list as criteria seven through eleven to distinguish them from the six layout criteria in Section 3.2.1:

7. *Distributed spare space.* Each physical disk should contain the same number of pre-allocated spare units. This assures that after reconfiguration, each disk will absorb an equivalent fraction of the failed disk's workload.

8. *Fault tolerance after reconfiguration.* When a disk fails, it must be possible to

assign each reconstructed unit to a spare unit on a disk that contains no data or parity units from the parity stripe containing the failed unit. This assures that the reconfigured array will be single-fault tolerant.

9. *Balanced parity after reconfiguration.* The assignment of reconstructed units to spare units must ensure that the same number of reconstructed parity units are assigned to each surviving disk. This assures that parity will be balanced in the reconfigured array.

10. *No wasted space.* A layout should not allocate more spare space than it will ever assign. More specifically, if an array is designed to recover from f non-overlapping disk failures and subsequent reconstructions without an intervening copyback, then after exactly f such failures and reconstructions, there should be no unassigned spare space in the array. Consider, for example, an array with $f = 1$ in which distributed sparing is achieved by simply using one unit in each parity stripe as a spare unit for that parity stripe. If the number of units per parity stripe (G) is 5 and the number of disks in the group (C) is 40, then this approach would allocate 1/5 of all units in the array as spare units (which are unavailable to store user data), whereas only 1/40 of the total number of units is actually required to recover from the failure of any single disk. Criterion 10 eliminates such inefficient layouts from consideration.

11. *Spare units physically close to failed units.* The spare unit for each reconstructed unit should reside at a disk offset that is close to the offset of the failed unit. This avoids the situation where a large user access that includes both an unfailed and a spared unit on the same disk incurs a long seek in traversing from one to the other. It also assures that the reconstruction algorithm can maintain the disk heads relatively well synchronized during reconstruction, thereby avoiding long seeks as it switches between reading surviving data and writing reconstructed data. The motivation behind this criterion is also discussed in Section 4.2.

In the remainder of this dissertation, we restrict attention to the case where $f = 1$, deferring a more general layout solution to future work. As discussed above, the simple solution of allocating one unit in each parity stripe to serve as the spare for that stripe is not acceptable, because it violates criterion ten. Our goal, therefore, is to find a mechanism

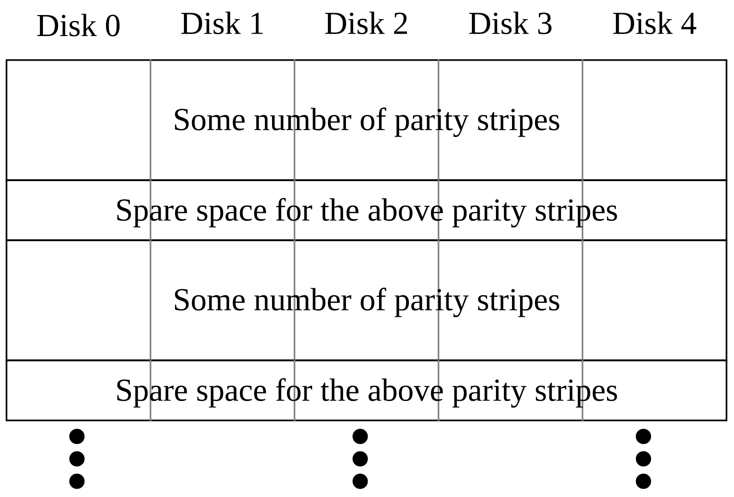

Disk 0
Disk 1
Disk 2
Disk 3
Disk 4
Some number of parity stripes
Spare space for the above parity stripes
Some number of parity stripes
Spare space for the above parity stripes

Figure 5.3: Allocating spare space in contiguous bands to simplify the mapping.

whereby the layout can distribute exactly one physical disk’s worth of spare space over the array.

Figure 5.3 shows an allocation of spare units to disks in which the spare space is allocated in bands of contiguous units at a constant disk offset. This approach has two benefits: it guarantees that criterion seven will be met, and it simplifies the algorithms used to map logical array addresses to physical disk addresses (refer to Appendix B). In what follows, we derive the number of parity stripes to be spared by each band.

In order to assure that each band of spare space occurs at a constant disk offset, the number of parity stripes spared by each band must be an integral multiple of the number of parity stripes in one block design table (refer to page 57). This greatly simplifies the mapping algorithms. In order to meet criterion nine, it is also necessary that after reconfiguration (reconstruction to distributed spare space), parity should be balanced within each band of spare space. As will be seen, these two requirements lead directly to a determination of the number of parity stripes in between each band of spare space.

When a disk fails, exactly *r* parity units are lost from each full block design table, where *r* is the number of tuples in which each object occurs in the block design. (The block design parameters are defined on page 55.) To meet criterion nine, these *r* units need to be evenly distributed over the surviving *v*-1 disks. In general, *r* need not be a multiple of

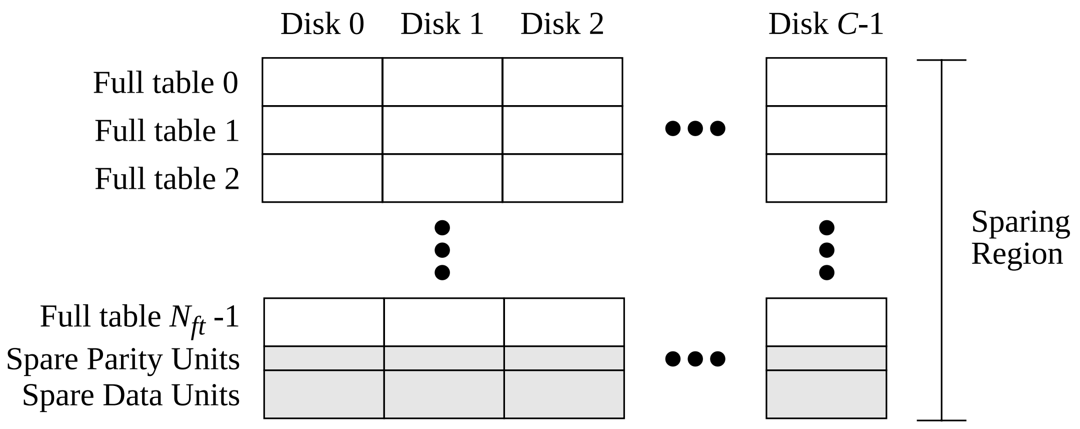


Figure 5.4: The allocation of spare space.

Each set of N_{ft} = LCM(v-1,r)/r contiguous full block design tables are grouped together to form a sparing region, and enough spare space is reserved at the bottom to accommodate the data and parity units that are lost from the region when a disk fails. Shaded units represent spare space.

v-1. The approach we take is therefore to place N_{ft} copies of the full block design table above each band of spare space, where N_{ft} is selected so that $N_{ft} \cdot r$ will be a multiple of v-1.

The minimum-sized layout region over which the layout policy can allocate spare space in order to balance parity after a failure is N_{ft} = LCM(v-1,r)/r copies of the full block design table. Allocating spare space over N_{ft} full tables assures that the number of parity units lost in this region is the smallest possible multiple of the number of surviving disks, v-1. The array loses exactly k-1 data units for each parity unit lost in a full table, where k is the number of objects per tuple, and so the number of data units lost in the region is also a multiple of v-1. This implies that the layout policy can also even distribute the spare units for the lost data units over the surviving disks.

One way to allocate the spare units is illustrated in Figure 5.4. The figure defines a “sparing region” to be N_{ft} full tables and their associated spare space, and shows how the allocation policy subdivides the spare space into distinct data and parity areas. This segregation of data and parity allows the assignment policy to balance each separately over the disks in the array. The allocation policy used in this section locates the spare space at the end of the region in order to simplify the mapping algorithms, but could equally well place it elsewhere within the sparing region to improve adherence to layout criterion eleven.

.

Full Tables Per Sparing Region (N_{ft})	Number of Designs	Fraction of Designs	Cumulative Fraction
1	264	0.61	0.61
2	71	0.16	0.77
3	39	0.09	0.86
4	18	0.04	0.90
5	15	0.03	0.94
6	4	0.01	0.95
7	7	0.02	0.96
8	3	0.01	0.97
9	3	0.01	0.98
10	2	0.00	0.98
11	1	0.00	0.98
13	3	0.01	0.99
15	1	0.00	0.99
17	1	0.00	1.00
19	1	0.00	1.00

Table 5.1: Number of full block design tables per sparing region for 433 designs.

For each value of N_{ft}, the table gives the number of designs having that value, the fraction of all designs having that value, and the fraction of all designs having N_{ft} less than or equal to that value. The 433 block designs had $v \leq 50$ and $\alpha < 1.0$.

Referring back the discussion of the full table depth in Section 3.5.2, this approach is viable as long as the depth (number of units per disk) of the sparing region is smaller than the number of units per disk in the array, and will achieve good parity balance as long as the array consists of a reasonably large number of sparing regions (about ten). Fortunately, for typical block designs, it's rare that N_{ft} is large. Table 5.1 gives the value of N_{ft} for 433 designs on $C = v \leq 50$ with $\alpha < 1.0$. It shows that it is unusual for there to be more than one full block design table per sparing region, and quite rare for there to be more than five. The average value of N_{ft} over the 433 designs was 2.1.

Having specified the allocation of spare units to disks, it's necessary to find an assignment of actual failed units to spare units that meets the criteria above. In parity-declustered arrays, the set of parity stripes that are affected by a failure varies with the particular disk that has failed, and so the layout policy must specify this assignment for each possible failed disk. Figure 5.5 illustrates the construction of this assignment for an example block design (Figure 5.5a) and its associated sparing region (Figure 5.5b). The design is first replicated $k \cdot N_{ft}$ times to produce a list of the tuples used in the layout of one sparing region (Figure 5.5c). The factor of k arises from the duplication of the block design to balance parity within a full block design table. The tuples that do not contain the identifier of the failed disk are discarded from the list (Figure 5.5d). Each remaining tuple is then complemented, that is, a new tuple is constructed for each that contains all the objects *not* contained in the original tuple.

A complement tuple can be viewed as a list of disk identifiers on which the controller can place the spare unit for the corresponding parity stripe in order to achieve single fault tolerance after reconfiguration (criterion eight). Since the complement tuple is guaranteed not to have any objects in common with the original tuple, the spare unit for the parity stripe laid out by the original tuple can be selected as any one of the objects in the complement tuple. It is then necessary to select one object from each complement tuple to serve as the spare disk for the corresponding parity stripe, in such a way as to evenly balance both the parity and data units. This is achieved using a slightly modified version of the *Reorder* algorithm from Section 3.5.2.

First, the complement tuples corresponding to lost parity units are identified and separated out from the other complement tuples (Figure 5.5e). The *Reorder* algorithm is applied to these tuples, with the modification that tuples and elements are ordered such that the last column contains the objects in sequence, skipping the object identifying the failed disk (Figure 5.5f). The last column of the reordered tuples then contains the identifier of each surviving disk exactly the same number of times, and so the disks thus identified can be used to generate an assignment of spare units to failed units that balances the distribution of spared parity units across the surviving disks. The reordering process is repeated for the tuples corresponding to failed data units, to assure that each surviving disk is assigned the same number of spare data units.

Although we have not formally proven that it is always possible to reorder each of the two sets of segregated tuples to achieve the balance condition, we have successfully computed the spare map for every possible failed disk for all of the designs in our block design database.

(a) Original block design
(v=6, k=3, b=10)

Tuple Number	Tuple
0	0 1 2
1	1 3 4
2	1 3 5
3	0 3 5
4	1 2 5
5	2 3 4
6	2 4 5
7	0 1 4
8	0 2 3
9	0 4 5

(b) Sparing Region Layout (N_{ft} = 1)

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
D0.0	D0.1	P0	D1.1	P1	P2
D3.0	D1.0	D4.1	D2.1	P5	P3
D7.0	D2.0	D5.0	D3.1	D6.1	P4
D8.0	D4.0	D6.0	D5.1	P7	P6
D9.0	D7.1	D8.1	P8	D9.1	P9
D10.0	P10	D10.1	P11	D11.1	D12.1
D13.0	D11.0	P14	P12	D15.1	D13.1
D17.0	D12.0	D15.0	P13	P16	D14.1
D18.0	D14.0	D16.0	P15	D17.1	D16.1
D19.0	P17	P18	D18.1	P19	D19.1
P20	D20.0	D20.1	D21.0	D21.1	D22.1
P23	P21	D24.0	D22.0	D25.1	D23.1
P27	P22	P25	D23.0	D26.0	D24.1
P28	P24	P26	D25.0	D27.1	D26.1
P29	D27.0	D28.0	D28.1	D29.0	D29.1
Spare Parity Space					
Spare Data Space					
Spare Data Space					

(c) Replicated design

Tuple No.	Tuple	Tuple No.	Tuple	Tuple No.	Tuple
0	0 1 2	10	0 1 2	20	0 1 2
1	1 3 4	11	1 3 4	21	1 3 4
2	1 3 5	12	1 3 5	22	1 3 5
3	0 3 5	13	0 3 5	23	0 3 5
4	1 2 5	14	1 2 5	24	1 2 5
5	2 3 4	15	2 3 4	25	2 3 4
6	2 4 5	16	2 4 5	26	2 4 5
7	0 1 4	17	0 1 4	27	0 1 4
8	0 2 3	18	0 2 3	28	0 2 3
9	0 4 5	19	0 4 5	29	0 4 5

(d) Tuples in replicated design
affected by failure of disk 2.

Tuple No.	Tuple	Complement Tuple
0	0 1 2	3 4 5
4	1 2 5	0 3 4
5	2 3 4	0 1 5
6	2 4 5	0 1 3
8	0 2 3	1 4 5
10	0 1 2	3 4 5
14	1 2 5	0 3 4
15	2 3 4	0 1 5
16	2 4 5	0 1 3
18	0 2 3	1 4 5
20	0 1 2	3 4 5
24	1 2 5	0 3 4
25	2 3 4	0 1 5
26	2 4 5	0 1 3
28	0 2 3	1 4 5

Figure 5.5 (part 1): Generating the assignment of reconstructed units to spare units.

The example assumes disk number 2 has failed. The block design (a) is replicated $k \cdot N_{ft}$ times to produce a complete list of tuples in the sparing region (b and c). From this, the tuples unaffected by the failure are eliminated, and the remaining tuples complemented (d) to produce a list of potential spare units for each failed unit. The figure continues on the next page.

(e) Segregating the complement tuples

Tuples that lost parity units	
Tuple No.	Complement Tuple
0	3 4 5
14	0 3 4
18	1 4 5
25	0 1 5
26	0 1 3

Tuples that lost data units	
Tuple No.	Complement Tuple
4	0 3 4
5	0 1 5
6	0 1 3
8	1 4 5
10	3 4 5
15	0 1 5
16	0 1 3
20	3 4 5
24	0 3 4
28	1 4 5

(f) Reordering the complement tuples

Tuples that lost parity units	
Tuple No.	Complement Tuple
25	1 5 **0**
18	4 5 **1**
26	0 1 **3**
14	0 3 **4**
0	3 4 **5**

Tuples that lost data units	
Tuple No.	Complement Tuple
4	3 4 **0**
5	0 5 **1**
6	0 1 **3**
8	1 5 **4**
10	3 4 **5**
15	1 5 **0**
16	0 3 **1**
20	4 5 **3**
24	0 3 **4**
28	1 4 **5**

(g) Assignment of failed units to spare units within sparing region

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Spare Parity Space	P25	P18		P26	P14	P0
Spare Data Space	D4.1	D5.0		D6.0	D8.1	D10.0
	D15.0	D16.0		D20.1	D24.0	D28.0

Figure 5.5 (part 2): Generating the assignment of failed units to spare units.

The complement tuples are segregated such that those that lost a parity unit are separate from those that lost a data unit (e). Each set of tuples is then reordered using the Reorder *algorithm (f), with the modification described in the text. In this example, the tuple ordering for the data units case did not have to be modified. The last column of each reordered set of tuples is used to assign failed units to spare units (g).*

In practice, we pre-compute the assignment of spare units to reconstructed units for each possible failed disk so that the controller can immediately instantiate it upon disk failure. But even this pre-computation of the reordering for the spare assignment is rapid and efficient: in deriving the reordering for every possible failed disk for the 433 designs described above, there were an average of 160 tuple swaps and 0.06 three-way tuple swaps per computation of a spare assignment, and no computations required more than eight three-way swaps. Computing the spare assignment for a particular design and failed disk rarely takes more than a few seconds on an engineering workstation.

5.4. Disk-oriented reconstruction algorithm to support distributed sparing

When a controller uses the disk-oriented algorithm (Section 4.2.2) with dedicated sparing, the reconstruction processes that read surviving units never have to write reconstructed units, and vice versa. Under distributed sparing, however, each reconstruction process must both read surviving data and parity units and write reconstructed units to the spare space on the disk to which it is assigned. Further, distributed sparing eliminates the distinction between surviving-disk processes and the replacement-disk process; all processes now execute the same algorithm. Finally, under distributed sparing, a reconstruction process frequently has the choice of whether to write a reconstructed unit or read a surviving unit. This section describes the modifications to the disk-oriented algorithm that are required to address these issues.

Recall from Section 4.2.3.1 that when a reconstruction process attempts to submit to the buffer manager the first reconstruction unit from a parity stripe, and there are no free buffers available to accept the unit, the process blocks in a FIFO queue until a buffer becomes available. This causes the corresponding disk to idle with respect to reconstruction during the period that the process is blocked, which can lengthen reconstruction time. It is therefore necessary to write full buffers (buffers for which all units have arrived and been XORed in) to the corresponding spare units as soon as possible, so that the buffer can be freed for use by another parity stripe. This is the key issue to be addressed in designing a disk-oriented algorithm for use with distributed sparing.

Define a buffer to be "applicable" to a reconstruction process if the spare unit to which the buffer is to be eventually written resides on the disk assigned to the indicated process. In order to minimize the period of time that full buffers remain in memory, each reconstruction process must write all applicable full buffers to its assigned disk before reading any new surviving units. Furthermore, a reconstruction process must not remain blocked trying to submit a new unit to the buffer manager if an applicable buffer becomes full while the process is blocked.

To achieve these two properties, each reconstruction process must query the buffer manager for applicable full buffers prior to each read of surviving data. If any such buffers exist, all such buffers must be written to the process' assigned disk prior to reading any new data. Further, when a reconstruction process blocks attempting to submit a buffer, the buffer manager must wake up that process if an applicable buffer becomes full prior to the successful completion of the buffer submission.

The following disk-oriented reconstruction algorithm achieves these goals. It is executed by all reconstruction processes.

```
while (∃ required surviving units or unwritten spare units on the assigned disk)
    while (∃ applicable full buffers)
        acquire next sequential full buffer from buffer manager
        issue low-priority write of buffer contents to disk
        wait for the write operation to complete
    end
    if (there remain any required and unfetched units on the assigned disk)
        if (there are no unresolved buffer submissions by this process)
            identify the next sequential required unit
            issue a low-priority read request for that unit
            wait for the read to complete
            inform buffer manager of desire to submit a buffer
        end
        block on a buffer-manager-maintained synchronization variable
        upon wakeup, check status set by buffer manager
        if (status == ok to submit) submit buffer to buffer manager
    end
end
```

In addition to it’s XORing responsibility, the buffer manager has several synchronization functions. First, it maintains a FIFO queue of all processes that have indicated a desire to submit buffers, and delivers available buffers to these processes in order. Second, it monitors all buffer submissions and signals the appropriate synchronization variable when either a free buffer has been assigned to the parity stripe upon which a process is waiting, or when a buffer applicable to a particular process has become full. Since the buffer manager may wake up a process for one of two reasons, it must communicate the actual reason to the process. The status flag referred to in the above algorithm serves this purpose. The manger sets this flag to “ok to submit” only after a free buffer has been assigned to the parity stripe upon which the indicated reconstruction process is waiting. This assures that the process will not need to block upon the actual submission. If the “ok to submit” flag is off at the time the reconstruction process returns from the block, it indicates that an applicable buffer has become full, and so the algorithm returns to the top of the outer loop.

Note that the manager may assign a buffer to the requesting process while that process is in the process of writing a full buffer to its disk. In this case, the buffer remains assigned to the indicated parity stripe and available for submissions by other processes, but unused by the requesting process until it (the process) has completed all possible write operations.

This discussion points out another benefit of distributed sparing: by eliminating the replacement disk as the reconstruction bottleneck, it reduces the total buffer stall time experienced during reconstruction. In dedicated sparing a reconstruction process will stall, and hence a disk will idle with respect to reconstruction, whenever there are no free reconstruction buffers at the time a new parity stripe is submitted to the buffer manager. In distributed sparing, the process will only stall if this condition is true *and* there are no full buffers to write to the indicated disk.

5.5. Performance analysis

This section analyzes the efficacy of distributed sparing at alleviating the replacement-disk bottleneck in declustered parity arrays, evaluates the large-access performance degradation experienced in reconfigured mode caused by the failure to meet layout criterion six (Section 3.2.1), and then re-evaluates the reconstruction options (Section 4.4.1) under distributed sparing.

5.5.1. Reconstruction performance

Figure 5.6 plots the reconstruction time and user response time during recovery for distributed sparing, and contrasts it to the dedicated-sparing case, using the array configuration and workload parameters described in Chapter 2. It shows that distributed sparing does provide the intended benefit of extremely fast reconstruction at low declustering ratios: the minimum reconstruction time was 34 seconds for the Lightning drive in the distributed sparing case, but 215 seconds under dedicated sparing. The response time plot shows that this comes at a slight cost: since distributed sparing eliminates the replacement

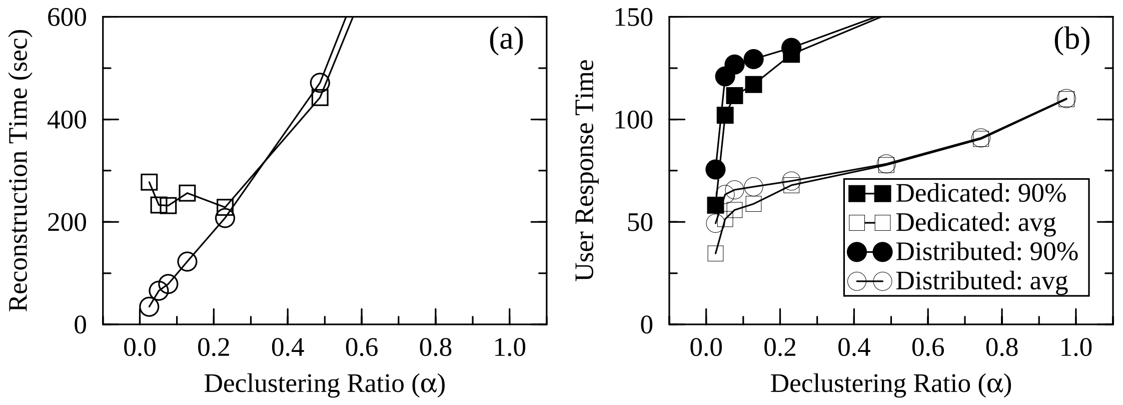


Figure 5.6: Comparing sparing alternatives.

Part (a) shows reconstruction time, and part (b) user response time during recovery, for both distributed and dedicated sparing. The graphs deliberately cut off the upper end of the curves in order to expand the scale. The curves are essentially co-incident at all points not shown in the figure. The array configuration was as described in Section 2.3.3.

disk as the reconstruction bottleneck, it keeps all disks busier with reconstruction workload. This means that even though the controller gives user requests higher priority than reconstruction requests, distributing the spare space causes the controller to initiate reconstruction requests more often, which causes user requests to be more often forced to wait for them to complete. The figure shows that distributed and dedicated sparing perform essentially identically at higher values of the declustering ratio (α). This is expected since at higher α the reconstruction rate is limited by the reads from the surviving disks rather than the writes to the spare locations.

5.5.2. Ultra-fast reconstruction in large arrays

Reconstructing the contents of a failed disk in a parity-declustered disk array requires at most $B \cdot G$ disk operations, where B is the number units on a single disk and G is the number of units in a parity stripe[1]. The distributed sparing layout causes these I/Os to be evenly distributed across all disks in the group containing the failure, which means that the full aggregate bandwidth of the group is available to reconstruct missing data. Thus, as the number of disks in a group (C) grows, the total disk bandwidth available to effect reconstruction grows, but the total amount of work to be done remains constant. This indicates

1. The reconstruction options described in Section 4.4.1 can cause the total number of reconstruction I/Os to be less than $B \cdot G$, but there need never be more than this number.

that under distributed sparing with a fixed parity stripe size (*G*), reconstruction time should be a monotonically decreasing function of the group size (*C*). Thus reconstruction time can be made arbitrarily small in large arrays by fixing the value of *G* and increasing *C*.

This conclusion that reconstruction time can be arbitrarily small is based on the assumption that the reconstruction rate is limited only by the rate at which data can be read from or written to the disks in the array. In reality, the reconstruction rate may also be limited by the ability of the array controller to flow data into its buffers, through its XOR engine, and back out to the drives. Further, the array controller must often process several interrupts per disk I/O, and so reconstruction time may also be limited by the instruction overhead associated with each interrupt. An array controller for a large disk array should be capable of handling the full aggregate bandwidth of the array however, since if the controller limits the throughput of an array, there is little point in connecting that many disks to it. Note that the array controller's implementation can be distributed rather than centralized [Cao93], which reduces the problem to that of designing an interconnect network with sufficient bandwidth. Thus, in what follows, we assume that the array controller can always keep up with the reconstruction rate, implying that the disks represent the dominant reconstruction bottleneck.

There is also a potential problem in finding block designs to implement a parity-declustered disk array with very large group size (*C*). Our database is only dense when $C \leq 43$. As outlined in Section 3.2.2.4, there are a number of solutions to this problem. Schwabe and Sutherland [Schwabe94] demonstrate approximately-balanced designs for *C* ranging up into the thousands, and so a designer can adopt their solution in cases where she cannot find an applicable block design. In this section, we limited the values of $C > 43$ evaluated to powers of prime numbers, which assures the existence of a design for all possible *G* [Hanani75].

Figure 5.7 shows the reconstruction-mode performance of a parity-declustered disk array with 4 units per parity stripe ($G = 4$, implying 25% of the array devoted to parity), as a function of the group size (*C*). It shows that reconstruction time is indeed a monotonically decreasing function of *C*, and falls as low as 24 seconds for the example disk (refer to Table 2.3) when $C = 151$. Increasing *G* to reduce the parity overhead shifts the recon-

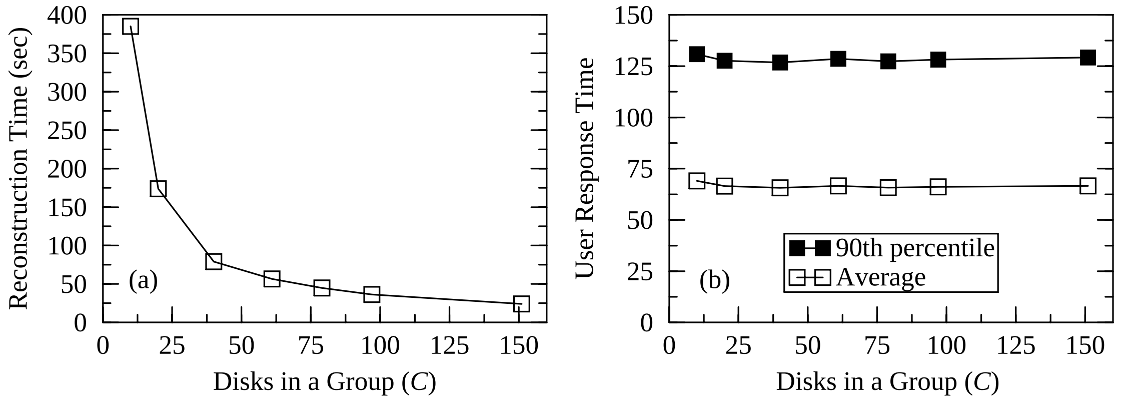


Figure 5.7: Distributed-sparing reconstruction performance versus *C* with *G*=4.

Part (a) shows the reconstruction time, and part (b) shows user response time during failure recovery.

struction time curve upward, but does not change its shape. Response time is essentially flat with respect to *C*, since, for nearly all points in the plot, the declustering ratio is so low that reducing it further by increasing *C* has little effect.

The reliability implications of making the group size (*C*) very large to achieve rapid recovery remain to be investigated. As in Section 3.4, we assume here that the total number of disks in the array is dictated by the performance and/or capacity requirements of the system, and thus it is only necessary to determine the division of these disks into independent groups. Letting N_{disks} be the total number of disks in the system, the reliability equation presented in Section 3.3.1.4 reduces to

$$MTTDL = \frac{(MTTF_{disk})^2}{N_{disks}(C-1)\,MTTR}$$

Figure 5.7 shows that under distributed sparing, average recovery time (*MTTR*) is inversely proportional to the number of disks in the array[2], and thus the term $(C\text{-}1)\cdot MTTR$ is also constant. Therefore, when N_{disks} and G are fixed, the reliability of the array

2. Regression-fitting a line to the inverse of the data in Figure 5.7a yields a correlation coefficient of 0.9988. The resultant curve is $ReconTime = 1/(\,2.7\text{e-}4\cdot C + 7\text{e-}4\,)$.

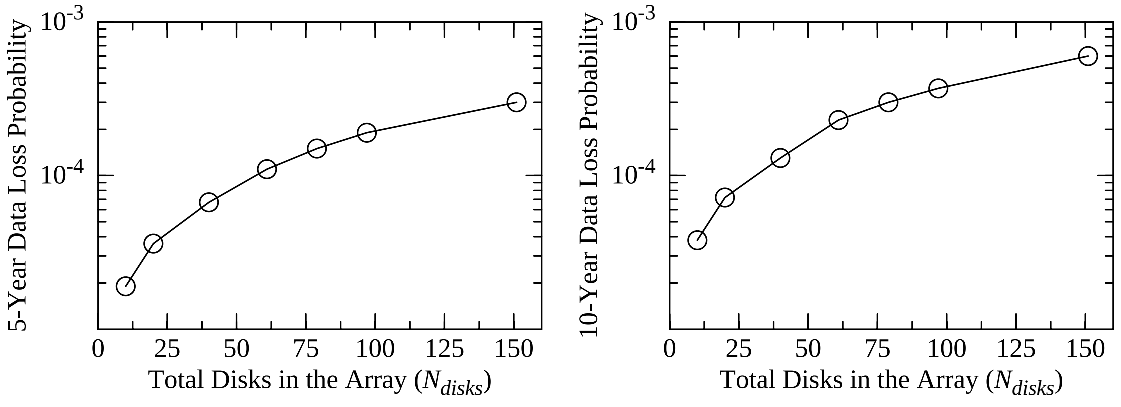


Figure 5.8: 5- and 10-year data loss probabilities versus C with G=4.

Note that the y-axis is plotted on a log scale, and does not end at 1.0.

becomes insensitive to C under distributed sparing:

$$\left(MTTDL_{array} \propto \frac{1}{N_{disks}\,(C-1)\,MTTR}\right) \quad \text{and} \quad \left(MTTR \propto \frac{1}{C}\right) \Rightarrow$$

$$\left(MTTDL_{array} \propto \frac{1}{N_{disks}}\right)$$

Thus, for a fixed total number of disks in the array, increasing the group size to achieve rapid reconstruction has no effect on the array's reliability. Figure 5.8 plots the probability of data loss over 5- and 10-year periods (note that these are not the MTTDL values) using the array sizes and recovery times in Figure 5.7. It's clear from this figure that, even for the largest array, the probability of data loss due to multiple concurrent disk failures is so low that some other factor, for example, a software bug, is almost certain to dominate the overall reliability equation for the array.

5.5.3. Large-access performance in reconfigured mode

Reconstructing the data of a failed disk in a distributed sparing array changes the assignment of data units to disks for all the units that were on the failed drive. For this reason, the array's post-reconfiguration large-access performance may be degraded, since reconfiguration may reduce the degree to which the layout adheres to criterion six. This is especially true if the layout has been optimized (see Section 3.5.3) to maximize adherence

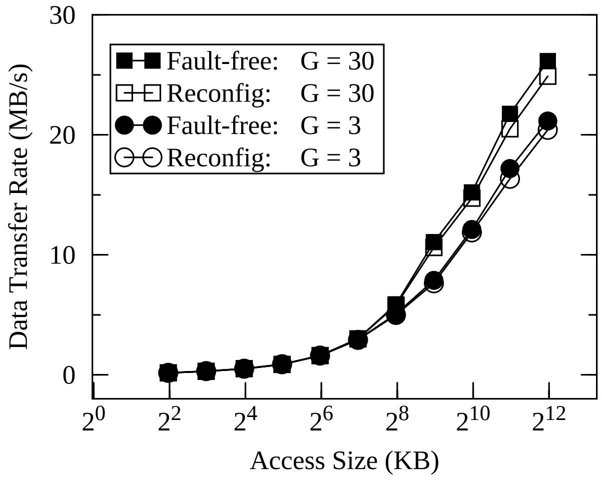


Figure 5.9: Large access performance in reconfigured mode.

*The plot shows the maximum data transfer rate achieved by a single user process versus the user access size in a 40 disk array, in both fault-free and reconfigured mode. The workload was 100% reads and the plot shows two representative values for parity stripe size (*G*). Simulations of workloads containing writes and of other values for* G *yielded similar results.*

to criterion six in the fault-free state.

To investigate this effect, Figure 5.9 plots the maximum throughput achieved at a user concurrency of one versus the access size, for a 40-disk array running 100% read workload, and two selected values of the declustering ratio, $\alpha \approx 0.05$ ($G = 3$) and $\alpha \approx 0.75$ ($G = 30$). In each of these simulations, all accesses were of the same size. Simulations of workloads containing writes and different values of the declustering ratio yielded similar results; these two examples are representative of all the simulations performed. The block designs used for layout were optimized via the technique described in Section 3.5.3. It's clear from the figure that the large-access throughput of the array in reconfigured mode is not significantly lower than in fault-free mode.

To explain this effect, consider the set of disks used by one possible large access, as illustrated in Figure 5.10. Reconfiguring the array to spare space is equivalent to uniformly distributing all the accessed (shaded) units that reside on the failed disk into the bottom portion of the sparing region. Since a large access typically uses a large fraction of disks in the array (refer to Figure 3.33), it is likely that each access encounters at most a few remapped units. Thus, in order to service the access, most surviving disks simply read

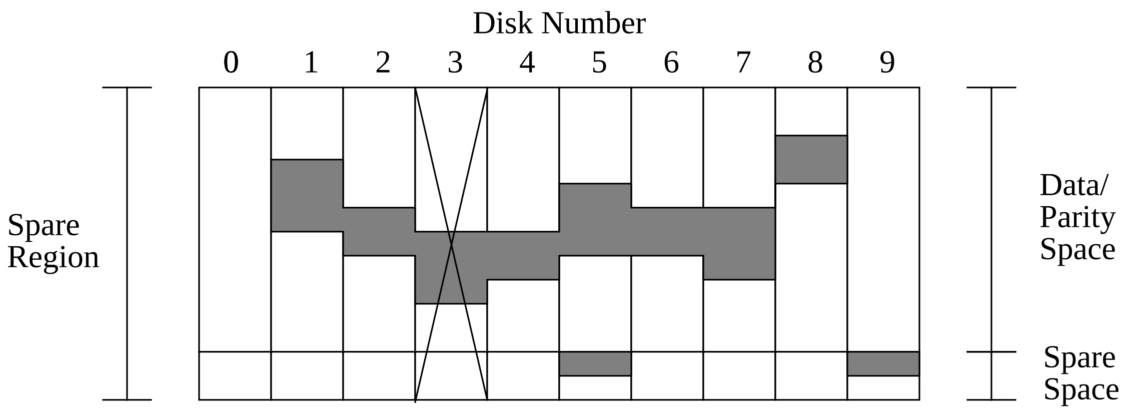


Figure 5.10: A hypothetical large access in reconfigured mode.

The figure shows one sparing region in a 10-disk array. The shaded areas represent the units read or written by the large access. Disk 3 has failed, and so its units have been remapped into the spare space at the bottom of the sparing region. The two lined regions represent the spare locations for the accessed units on disk 3. The figure shows that in servicing the access, most disks are unaffected, but a few (disks 5 and 9) must skip down to the end of the sparing region after accessing the unfailed portion of the access assigned to that disk (if any). Since only a few disks are typically affected, and since the sparing region is not typically very deep, the large-access performance in reconfigured mode is only slightly worse than the performance in fault-free mode.

or write the portion of the data assigned to them, but a few must access their portion, and then skip down to the bottom of the sparing region to access the remapped units. Since the number of disks that must perform this extra work is typically small, and since the depth of the sparing region is seldom large (refer to Table 5.1), this extra work causes only a small performance penalty.

5.5.4. Re-evaluating the reconstruction options under distributed sparing

Section 4.4.1.1 described three options that can be applied to the reconstruction algorithm: *redirection of reads*, *piggybacking of writes,* and *user writes to the replacement*. The conclusion of that section was that for workloads dominated by small accesses, only redirection of reads had significant impact on reconstruction performance. That section went on to demonstrate that redirection is beneficial to user response time at all values of the declustering ratio (α), but that it lengthens reconstruction time at low α by sending additional work to the over-utilized replacement disk. Since distributed sparing eliminates the replacement disk as the reconstruction bottleneck, it is worthwhile to re-evaluate this reconstruction option in this context.

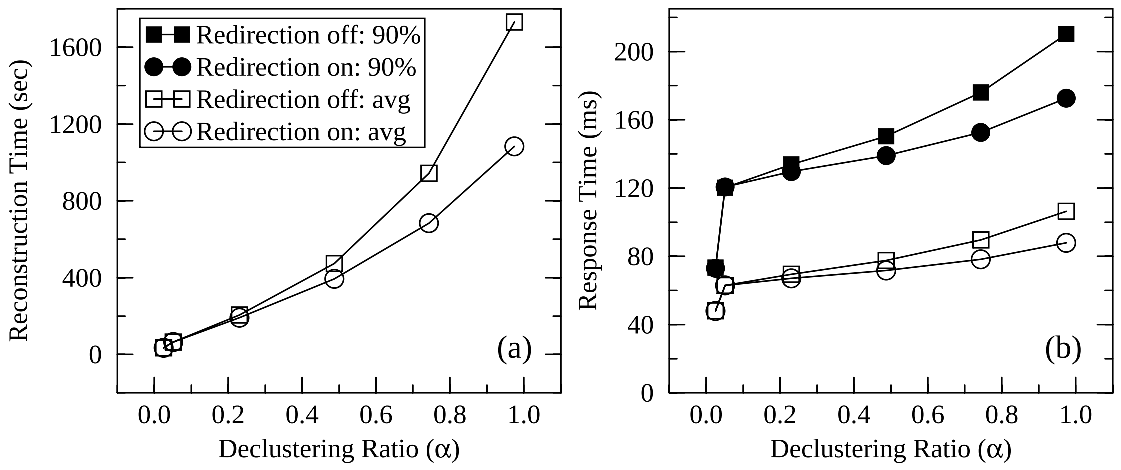


Figure 5.11: Evaluating redirection of reads under distributed sparing.

Part (a) shows reconstruction time, and (b) user response time during reconstruction.

As in the original evaluation of the options, simulations showed that the piggybacking and user-writes options have little effect on reconstruction performance under distributed sparing. This is expected behavior, because the reasons for it (described in Section 4.4.1.2) have nothing to do with the replacement disk being the reconstruction bottleneck. Therefore, this section considers only redirection of reads. These evaluations again use the array configuration and workload parameters described in Section 2.3.

Figure 5.11 shows the reconstruction time and user response time during reconstruction versus the declustering ratio, for a 40-disk array containing one disks' worth of distributed spare space. With the elimination of the spare disk as the reconstruction bottleneck, redirection has become uniformly beneficial to both reconstruction time and user response time during recovery, with the primary benefit occurring at high values of the declustering ratio where the load on the surviving disks is greatest. This eliminates the need for the "monitored" versions of the reconstruction options developed for the dedicated-sparing case (Section 4.4.1.3). Note that redirection in the distributed-sparing context means that the controller redirects a user read operation of a previously-reconstructed data unit to whichever disk holds the spare unit for that data unit. This means that the redirected workload is also distributed over the disks of the array, rather than being concentrated on one replacement disk, and so redirection does not interfere with the very low reconstruction time possible at low α.

5.6. A related study

Ng and Mattson [Ng92a, Ng92b] also investigated the benefits of combining distributed sparing with parity declustering, which they refer to as “Uniform Parity Group Distribution”, and also use a layout mechanism based on combinatorial block designs. This section very briefly outlines their solution and contrasts it with the one described above.

In this approach, the array is divided into “domains”, each consisting of some number of parity stripes. Ng and Mattson leave the actual number unspecified, but it must be at least k (refer to page 55) since they assume that the layout balances parity within each domain. They use the tuples of a block design to lay out the domains in the same manner as parity stripes are laid out in Section 3.2.2.2.

Ng and Mattson add one more criterion to the list given in Section 5.3: they require that when a disk failure occurs while the array is in reconfigured mode, the layout must also evenly balance the resulting failure-induced workload across the disks, without relocating any of the surviving units at the time of failure[3]. In order to achieve this, they lay out the fault-free array using a complete block design on $v = C$ and $k = G$, rather than an incomplete design.

Prior to any failure, the array is mapped using some number of copies of the complete design. When a disk fails, Ng and Mattson eliminate the identifier of the failed disk from all tuples in the duplicated complete design. They then select the assignment of spare units to failed units in such a way that, after reconfiguration, the layout maps the array using some larger number of copies of the complete design on one fewer disk ($v = C$-1 and $k = G$). The number of tuples in a complete design is $\binom{v}{k}$, and it’s straightforward to derive that $\binom{v}{k} = \frac{v}{v-k}\binom{v-1}{k}$, and so in order for this approach to work, it must be the case that $v/(v\text{-}k)$ is an integer. Since this is not true for all v and k, it’s necessary in the general case to lay out the fault-free array using some number of copies of the complete design, such

3. Ng and Mattson’s approach generalizes to the case where multiple disks worth of spare space are distributed in the array, and multiple failures occur prior to replacement and copyback. That is, by distributing multiple disks worth of spare space, they are able to maintain balance of the failure-induced workload through multiple failures and reconfigurations without replacement. In the interest of simplicity, the description here discusses only a single spare disk.

that $N_{copies} \cdot \binom{v}{k}$ is a multiple of $\binom{v-1}{k}$. The logical choice is of course

$$N_{copies} = \frac{LCM\left[\binom{v}{k}, \binom{v-1}{k}\right]}{\binom{v}{k}}$$

The above example considered only the case of balancing the failure-induced workload through two failures. In general, the required number of copies of the original block design increases with the number of failures to be survived and reconfigured (*f*), because each increase in *f* adds one more term to the LCM function above.

Since (1) the number of tuples in a complete block design grows exponentially with v, (2) it is necessary in the general case to use multiple copies of the complete design to lay out the domains of a fault-free array, and (3) each domain must consist of at least k parity stripes, this approach is viable only for configurations with a relatively small number of disks. For example, with $v = C = 15$ and $k = G = 7$, the minimum possible value of N_{copies} is 8, and so the block design used to lay out the fault-free array has 51,480 tuples, which translates into at least 360,360 parity stripes. Laying out these parity stripes over 15 disks uses 168,168 units per disk. Assuming, as in Section 3.5.1, that the data and parity unit size is 32K (about one track), the minimum sized disk that can be used in this layout is about 5.4 GB, which is too large for many configurations. The approach developed in this dissertation allows implementation using much smaller disks: the block design on $v = 15$ and $k = 7$ has $r = 14$ and $N_{ft} = 2$, and so each sparing region uses $2 \cdot 14 \cdot 7 = 196$ units per disk for data and parity, and $196/14 = 14$ units per disk for spare space, for a total of 210 units per disk per sparing region. With units of size 32 KB, the approach is viable for any disk size larger than about 6.7 MB.

The drawback of accommodating larger arrays using the approach developed here is that it neglects entirely the issue of degraded- and reconstruction-mode balance during second-failure recovery in reconfigured arrays. However, any particular group in the array should spend only a relatively small percentage of its total time in reconfigured mode, and hence the probability of a failure occurring while in this mode should be small. This is not intended to imply that fault tolerance in reconfigured mode is not important, since a small probability of relatively poor performance is acceptable, whereas a small probability of data loss is not.

Table 5.1 shows that since N_{ft} is generally small, our approach can be implemented for arrays with much larger values of the group size (*C*). The conclusion to be drawn from this is, of course, dependent upon the requirements of the system. If *C* is small and system environment is such that failed disks will not be replaced for long periods of time, then Ng and Mattson's approach is superior for its ability to balance failure-induced workload in the event of non-overlapping multiple failures. The solution derived in this chapter allows broader application of this technique.

5.7. Conclusions

The techniques described in this chapter used distributed sparing to eliminate the replacement disk as the reconstruction bottleneck at low declustering ratios, and thereby allowed for extremely fast reconstruction. Using a reasonably complicated layout mechanism, but a very simple reconstruction algorithm, it was possible to reduce reconstruction time for the example 40-disk array from a minimum of about 200 seconds to about 35 seconds. The 200 second limit in the dedicated-sparing case is an absolute lower bound; this is the amount of time it takes to write the entire contents of one disk at the maximum bandwidth supported by the drive. This factor of six reduction in reconstruction time was achieved at a slight cost (about 10% in the worst case) in user response time during reconstruction, but without penalty to response time in reconfigured mode. Distributing the spare space and deferring the copyback of the reconstructed data to the replacement drive until a period of low utilization or scheduled downtime can thus allow disk failure events to go essentially unnoticed by the users of the system.

Beyond the material presented in this chapter, there are several modifications to the layout and the reconstruction algorithm that might further improve reconstruction performance. First, the current layout policy locates the spare space at the end of each sparing region. Recalling that each spare unit serves as spare space for a number of possible data or parity units, depending on which disk has failed, it might be possible to derive a layout that locates each spare unit closer to the set of all units that it protects. This would improve large-access performance by reducing the positioning penalty described in the caption on

Figure 5.10, and could potentially improve small-access performance in highly local workloads as well. Second, the reconstruction algorithm developed here always gives writes priority over reads. This policy might be modified to take other factors into account, such as the current head position and the availability of reconstruction buffers. Finally, the writes of full buffers to spare locations currently occur one at a time, even if there are several full buffers available to write to a particular disk. Reconstruction time might be improved, at the cost of degraded response time, by batching together sets of full buffers for specific disks and writing them in bulk. This would potentially reduce the positioning overhead incurred by the write operations.

Chapter 6: Conclusions

This dissertation demonstrated techniques by which it is possible to design parity-based data storage subsystems that exhibit arbitrarily-small performance degradation during failure recovery, and further that allow this recovery to be completed very rapidly. Toward this end, it made three primary contributions. First, it demonstrated an implementation of *parity declustering*, which is a disk array organization that allows the performance degradation experienced during failure recovery to be reduced, essentially continuously, to nearly zero by evenly distributing the failure-induced workload over a larger-than-minimal set of disks. Second, it developed a *disk-oriented* reconstruction algorithm that minimizes recovery time by allowing the failure recovery process to absorb essentially all the array's bandwidth that is not absorbed by the users. Finally, it combined the above two techniques with *distributed sparing*, which resulted in a disk array exhibiting extremely rapid failure recovery, and achieved this without sacrificing performance to any significant degree. Table 6.1 on page 190 summarizes the contributions of the thesis.

The introductory chapters of this thesis made the case that the ability to gracefully tolerate component failures is essential to many applications, notably transaction processing systems and file servers, because in these areas, any interruption in the accessibility of data causes significant disruption in the service provided or supported by the computing system. They further showed that technology trends are leading inevitably to the condition where the rapidly increasing demands for I/O throughput will be met by systems with larger numbers of small disks, rather than the converse. The conclusion drawn from these two observations was that existing redundant disk array technology provided insufficient availability, and this served as the primary motivation for the work reported here.

The parity declustered disk array organization described in Chapter 3 provided the most effective tool for improving recovery-mode performance. In a standard disk array organization (RAID Level 5), each parity unit protects exactly C-1 data units, where C is the number of disks in each of the independent groups comprising the array. In parity declustering, each parity unit instead protects G-1 data units, where G is a parameter of the array that can be selected arbitrarily between 2 and C. Reducing G improves degraded- and reconstruction-mode performance by reducing the number of I/O operations required to reconstruct any particular unit, but increases the capacity overhead for redundancy. At the extremes, selecting G=2 yields a layout equivalent to disk mirroring (two copies of all data), while selecting G=C yields a RAID Level 5 array. To achieve an even balance of failure-induced workload across the disks comprising a group, Chapter 3 used *balanced incomplete block designs* to derive the mapping of data and parity units to disks. This chapter showed that a parity declustered array provides superior performance in the presence of failure than a RAID Level 5 disk array with an equivalent number of disks and total user data capacity. It also showed that order-of-magnitude improvements in both recovery time and response time during recovery are possible by reducing G to its minimum value, while keeping the number of disks in the array fixed. The chapter further showed that using a low ratio of G to C allowed reconstruction time to be reduced to very nearly its miminum possible value, which is the time required to write the entire contents of one disk at the maximum possible transfer rate of the drive.

Defining a disk array architecture to support high performance operation during failure recovery addresses only half of the overall availability problem. Chapter 4 addressed the other half; the design of the algorithm used to recover lost data. It showed that the standard approach of reconstructing a single parity stripe, or set of stripes, at one time yields inefficient utilization of the disks, especially when applied to parity declustered disk arrays. The *disk-oriented* reconstruction algorithm addressed this problem by structuring the reconstruction process around disks, rather than parity stripes. Chapter 4 showed that this algorithm is able to absorb nearly all of the disk bandwidth not absorbed by user processes. It resulted in improvements in reconstruction time of up to 40% over a 16-way parallel stripe-oriented algorithm, using only a small and bounded amount of buffer memory, and without significantly degrading user response time when compared to the stripe-oriented case.

The goal of any availability study is to minimize the impact of the failure recovery process on the systems' users. This leads to the idea of attempting to tailor the reconstruction algorithm to the observed user workload, rather than simply viewing this workload as an arbitrary set of accesses that must be completed while reconstruction is ongoing. Thus the latter portion of Chapter 4 investigated a set of *head following* techniques, whereby the reconstruction algorithm attempts to track the user-induced motion of the disk heads, and reconstruct the parity stripes that are close to the position of the disk heads after each user access. The intended benefit was that, at the time a disk switches between performing a user access and a reconstruction access, the seek time to and from the reconstruction point should be reduced, and thus reduce the positioning overhead incurred by both types of accesses. However, evaluating this approach revealed that the efficiency of the reconstruction algorithm is tied most strongly to its sequentiality, and thus that head following actually degraded recovery-mode performance by interfering with this sequentiality. After investigating several techniques for ameliorating this problem, Chapter 4 concluded that head following is not viable in random-workload environments.

It is possible to view parity declustering and disk-oriented reconstruction as general techniques for improving the degraded- and reconstruction-mode performance of any ECC-based redundant disk array, rather than as a specific disk array organization and a specific approach to reconstruction, respectively. Under this view, parity declustering simply amounts to the ability to decouple the number of units in a parity stripe (G), from the number of disks in an independent group in the array (C), with the other details of the disk array organization left unspecified. Similarly it is possible to view disk-oriented reconstruction as simply using one recovery process per disk, with the actual code that each executes left unspecified. This observation leads to the idea that it is possible to couple these two techniques with essentially any other technique proposed for improving some aspect of the performance of redundant disk arrays. In this light, the techniques presented in Chapters 3 and 4 constitute an application of parity declustering and disk-oriented reconstruction to RAID Level 5 arrays.

Chapter 5 investigated combining these two techniques with another type of disk array, RAID Level 5 with distributed sparing. This approach to organizing spare space allows the array to use the actuator on the spare drive to service user requests, and also eliminates the replacement disk as the reconstruction bottleneck. Combining distributed sparing with parity declustering and disk-oriented reconstruction achieved all the advantages of each. Specifically, this combination led to the development of an array that exhibits extremely fast reconstruction at low declustering ratios. Note that a low declustering ratio does not necessarily imply a high capacity overhead, since it is possible to reduce α by increasing C as well as by decreasing G.

Of course, this observation about parity declustering and distributed sparing leads directly to a number of ideas for future work. Foremost among them is the ability to combine these two techniques with parity logging [Stodolsky93]. RAID Level 5 arrays have always been viewed as having two primary drawbacks: they exhibit poor performance on small write operations, and they exhibit poor performance during failure recovery. Thus, combining parity declustering with parity logging would yield an array organization that overcomes both these limitations. The layout and control mechanisms for such an organization would be complex, but the resultant design would have advantages that are not achievable by other organizations: low redundancy overhead, high performance on all types of operations, low performance degradation during reconstruction, and rapid reconstruction.

Another promising area for future work would be to investigate the possibilities and performance implications of relaxing the requirement that parity and failure-induced workload be perfectly balanced across the disks comprising the array. This idea is motivated by the observation that parity-update workload and failure-induced workload will be perfectly balanced across the disks of the array only if the workload applied by the application is perfectly balanced, which is unlikely. One approach would be to use an approximately-balanced block design, that is, a design in which the number of tuples containing each pair of objects can vary slightly, to lay out the data and parity. An alternative to this would be to use a layout which perfectly balances the failure-induced workload, but which relaxes the requirement that parity be perfectly balanced. Schwabe and Sutherland [Schwabe94] have promising results that these techniques might greatly expand the range of combinations of C (number of disks in the array) and G (number of units per parity stripe) that are supportable in the block-design-based layout, but the performance implications of this remain to be investigated.

A third area worth investigating is the implication of parity declustering on the architecture of arrays containing very large numbers of disks. In order to achieve both low capacity overhead and good failure-recovery performance, it may be necessary to select C

to be a large value. For example, to keep the capacity overhead below 10% in a parity declustered array with $\alpha = 0.1$, C must be approximately 100. This would be difficult or impossible in a bus-based disk array controller such as the one in Figure 2.5a, because of the requirement that no two disks sharing a path to the controller should reside in the same group (criterion 1 in Section 3.2.1). Thus the desire to achieve a low declustering ratio might have strong implications on the overall architecture chosen for the storage subsystem. The main challenge in designing such a system would be to assure that there are at least two independent paths to all disks, without paying the cost of full duplication of the interconnect.

Fourthly, several of the techniques used in this dissertation to reduce reconstruction time caused the user responsiveness to be slightly degraded (see, for example, Sections 4.3.1 and 5.5). In all cases, this was caused by the fact that disk accesses are non-preemptible. In order to reduce reconstruction time, it's necessary to initiate reconstruction accesses more often, which causes user accesses to more often block in the disk queues waiting for a reconstruction access to complete. Much of this loss in responsiveness could be recaptured by supporting preemptible disk accesses. If the component disks had the ability to suspend an ongoing reconstruction access in order to service a user access, the user accesses would not spend as much time blocked in the disk queues. It is possible that the overhead associated with preemption would eliminate any benefit, but this remains to be investigated.

There are a number of less significant items that might be pursued. As mentioned in Sections 3.5.2.2 and 5.3, we have never observed the block design reordering algorithm used to compact the full block design table and to balance parity in reconfigured mode to fail, but we have not formally proven that it always succeeds. Constructing these proofs would validate the methods used.

Finally, Chapter 5 essentially defined away the problem of copying reconstructed data from distributed spare space to a new disk when a failed disk is physically replaced. The assumption that this copyback can be deferred until a period of low utilization, or even scheduled maintenance, is valid for most environments. However, in some environments there may be no such periods, or the system requirements may make it desirable to initiate the copyback as soon as the failed disk is replaced, so as to minimize the (already small) possibility of encountering the poor performance that would occur should a second failure occur while the array operates in reconfigured mode. Defining goals for the copyback operation and designing algorithms to meet them would improve the overall availability of the system.

Technique	Objectives	Summary	Results
Parity declustering	Faster reconstruction. Improved responsiveness.	Distribute parity stripes over larger-than-minimal collection of disks. Use block designs to balance workload.	Up to 10x faster reconstruction. Up to 6x shorter response time. Better MTTDL at moderate-to-high load. No fault-free degradation. Supports higher fault-free workload. Small penalty for large accesses.
Disk-oriented reconstruction	Absorb all unused disk bandwidth for reconstruction.	Use one reconstruction process per disk, instead of one per stripe.	Up to 40% faster reconstruction. Small response time penalty.
Distributed sparing	Remove spare disk as reconstruction bottleneck. Ultra-fast reconstruction in large arrays.	Modify layout to distribute spare space over array.	Reconstruction time monotone-decreasing in *C*. ~25 second reconstruction in large array.
Adjusting the reconstruction unit size	Optimize tradeoff between reconstruction time and response time.	Modify layout to pack multiple stripe units into a reconstruction unit.	Track-sized units yield best tradeoff.
Compacting the full block design table	Better balance of parity-update workload. Support smaller disks.	Re-order tuples and objects to balance parity.	Minimum possible table size achieved for all *C*,*G* combinations.
Improving adherence to criterion 6	Improved fault-free large-access performance.	Use simulated annealing to optimize tuple and object ordering.	Mixed results (case-specific). Typically about 50% of lost large-access performance reclaimed.

Table 6.1: Summary of thesis contributions.

Technique		Objectives	Summary	Results
Reconstruction options	Redirection	Faster reconstruction. Improved responsiveness.	Service reads of previously reconstructed data from spare disk.	Halves reconstruction time at high α, but doubles it at low α. Uniform benefit to response time.
	Piggybacking		Write data reconstructed to service a user read to spare disk.	Little effect on OLTP-like workloads.
	User-writes		Send user writes of unreconstructed data to spare disk.	Little effect on OLTP-like workloads.
Monitored redirection		Optimal reconstruction time at all α.	Redirect only if spare disk utilization < average surviving disk utilization.	Optimal reconstruction time at all α. Small response time penalty at low α.
Head following		Minimize positioning overhead incurred by reconstruction.	Reconstruct in region of array being accessed by users.	Negative results on nearly every workload due to loss of sequentiality and inability to focus memory resources.

Table 6.1: Summary of thesis contributions.

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.

[Bell89] C.G. Bell, "The Future of High Performance Computers in Science and Engineering," *Communications of the ACM*, vol. 32 no. 9, 1989, pp. 1091-1101.

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

[Bitton89] D. Bitton, "Arm Scheduling in Shadowed Disks," *Proceedings of the Computer Society International Conference (COMPCON 89)*, 1989, pp. 132-136.

[Blaum94] M. Blaum, J. Brady, J. Bruck, and J. Menon, Evenodd: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures, *Proceedings of the International Symposium on Computer Architecture*, 1994, pp. 245-254.

[Burkhard93] W. Burkhard and J. Menon, "Disk Array Storage System Reliability," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993, pp. 432-441.

[Buzen87] J.P. Buzen and A.W. Shum, "A Unified Operational Treatment of RPS Reconnect Delays," *Proceedings of the 1987 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Performance Evaluation Review*, vol. 15 no. 1, 1987.

[Cabrera91] L.-F. Cabrera and D. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, vol. 4 no. 4, 1991, pp. 405-439.

[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*, vol. 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, pp. 322-331.

[Copeland88] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba," *Proceedings of the ACM Conference on Management of Data*, 1988, pp. 99-108.

[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.

[DEC86] Digital Equipment Corporation, *Digital Large System Mass Storage Handbook*, 1986.

[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.

[Fujitsu2360] Fujitsu Corporation, Model M2360A product information.

[Gelsinger89] P.P. Gelsinger, P.A. Gargini, G.H. Parker, A.Y.C. Yu, "Microprocessors circa 2000," *IEEE Spectrum*, October 1989, pp. 43-74.

[Gibson89] G. Gibson, L. Hellerstein, R. Karp, R. Katz, D. Patterson, "Coding Techniques for Handling Failures in Large Disk Arrays," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 123-132.

[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*, vol. 17, 1993, pp. 4-27.

[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*, vol. 11, 1975.

[Harker81] J.M Harker, D.W. Brede, R.E. Pattison, G.R. Santana, L.G. Taft, "A Quarter Century of Disk File Innovation," *IBM Journal of Research and Development*, vol. 25 no. 5, 1981, pp. 677-689.

[Hartman93] J. Hartman and J. Ousterhout, "The Zebra Striped Network File System," *Proceedings of the Symposium on Operating System Principles*, 1993.

[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-Toler-*

ant Computing, 1993, pp. 422-431.

[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.

[Howard88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems*, vol. 6 no. 1, 1988, pp. 51-81.

[HPC3013] HP Corporation, Disk Drive Model HP C3013 (Kittyhawk) product information.

[Hsiao90] H. Hsiao and D. DeWitt, “Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines,” *Proceedings of the International Data Engineering Conference*, 1990.

[Hsiao91] H. Hsiao and D. 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.

[IBM0664] IBM Corporation, IBM 0664 Disk Drive Product Information.

[IBM3380] IBM Corporation, *IBM 3380 Direct Access Storage Introduction*, Manual GC26-4491-0, 1987.

[IBM3390] IBM Corporation, *IBM 3390 Direct Access Storage Introduction*, Manual GC26-4573-0, 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.

[Jain91] R. Jain, *The Art of Computer Systems Performance Evaluation,* John Wiley & Sons, 1991.

[Jones91] J. Jones, Jr., and T. Liu, “RAID: A Technology Poised for Explosive Growth,” Montgomery Securities Industry Report, Montgomery Securities, San Francisco, 1991.

[Katz89] R. Katz, et. al., “A Project on High Performance I/O Subsystems,” *ACM Computer Architecture News*, vol. 17 no. 5, 1989, pp. 24-31.

[Katz93] R. Katz, P. Chen, A. Drapeau, E. Lee, K. Lutz, E. Miller, S. Seshan, and D. Patterson, “RAID-II: Design and Implementation of a Large Scale Disk Array Controller,” *Symposium on Integrated Systems,* 1993.

[Katzman77] J. Katzman, “System Architecture for Nonstop Computing,” *Proceedings of the Computer Society International Conference (COMPCON 77)*, 1977.

[Kim86] M. Kim, “Synchronized Disk Interleaving,” *IEEE Transactions on Computers*, vol. 35 no. 11, 1986, pp. 978-988.

[Kirkpatrick83] S. Kirkpatrick, D. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, 1983.

[Kistler92] J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions on Computer Systems*, vol. 10 no. 1, 1992, pp. 3-25.

[Kung86] H.T. Kung, "Memory Requirements for Balanced Computer Architectures," *Proceedings of the International Symposium on Computer Architecture*, 1986, pp. 49-54.

[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.

[Lee93] E. Lee and R. Katz, "An Analytic Performance Model of Disk Arrays," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1993.

[Leffler89] S. Leffler, M. McKusick, M. Karels, J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[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.

[MacWilliams78] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes***,** North Holland, 1978.

[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*, vol. 30, 1990.

[Maxtor89] Maxtor Corporation, *XT-8000S Product Specification and OEM Technical Manual*, Document 1015586, 1989.

[McKeown83] D. McKeown, *MAPS: The Organization of a Spatial Database System Using Imagery, Terrain, and Map Data*, Department of Computer Science Technical Report CMU-CS-83-136, Carnegie Mellon University, 1983.

[Menon89] J. Menon and J. Kasson, *Methods for Improved Update Performance of Disk Arrays*, IBM Research Division Computer Science Report RJ 6928 (66034), 1989.

[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, pp. 318-329.

[Menon92c] J. Menon and D. Mattson, "Performance of Disk Arrays in Transaction Processing Environments," *Conference on Distributed Computing Systems*, 1992, pp. 302-309.

[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.

[Merchant92a] A. Merchant and P. Yu, "Design and Modeling of Clustered RAID," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1992, pp. 140-149.

[Merchant92b] A. Merchant and P. Yu, "Performance Analysis of A Dual Striping Strategy for Replicated Disk Arrays," *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, 1992.

[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.

[Myers86] G.J. Myers, A.Y.C. Yu, D.L. House, "Microprocessor Technology Trends," *Proceedings of the IEEE*, vol. 74 no. 12, 1986.

[Ng92a] S. Ng, and R. Mattson, "Maintaining Good Performance in Disk Arrays During Failure Via Uniform Parity Group Distribution," *Proceedings of the First International Symposium on High-Performance Distributed Computing*, 1992, pp. 260-269.

[Ng92b] S. Ng and R. Mattson, *Uniform Parity Group Distribution in Disk Arrays*, IBM Research Division Computer Science Research Report RJ 8835 (79217), 1992.

[Orji93] C. Orji and J. Solworth, "Doubly Distorted Mirrors," *Proceedings of the ACM Conference on Management of Data*, 1993, pp. 307-316.

[Ousterhout88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, vol. 21 no. 2, 1988, pp. 23-36.

[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 ACM Conference on Management of Data*, 1988, pp. 109-116.

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

[Polyzois93] C. Polyzois, A. Bhide, and D. Dias, "Disk Mirroring with Alternating Deferred Updates," *Proceedings of the Conference on Very Large Data Bases*, 1993, pp. 604-617.

[RAID93] *The RAIDBook, A Source Book for RAID Technology*, published by the RAID Advisory Board, Lino Lakes, Minnesota, 1993.

[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.

[Rangan93] P.V. Rangan and H.M. Vin, "Efficient Storage Techniques for Digital Continuous Multimedia," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5 no. 4, 1993.

[Reddy91] A.L.N. Reddy and P. Bannerjee, "Gracefully Degradable Disk Arrays," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1991, pp. 401-408.

[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.

[Rudeseal92] A. Rudeseal, Storage Technology Corporation, Presentation at Carnegie Mellon University, March 5, 1992.

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

[Schwabe94] E. Schwabe and I. Sutherland, personal communications, and "Improved Parity-Declustered Layouts for Disk Arrays," draft submission to the *Symposium on Parallel Algorithms and Architectures*, 1994.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the Winter USENIX Conference*, 1993, pp. 201-220.

[Siewiorek92] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems Design and Evaluation*, Digital Press, 1992.

[Solworth90] J. Solworth and C. Orji, "Write-Only Disk Caches," *Proceedings of the ACM Conference on Management of Data*, 1990, pp. 123-132.

[Solworth91] J. Solworth and C. Orji, "Distorted Mirrors," *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1991, pp. 10-17.

[ST9096] Seagate Corporation, Disk Drive Model ST9096 product information.

[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.

[Stonebraker90] M. Stonebraker and G. Schloss, "Distributed RAID -- A New Multiple Copy Algorithm," *Proceedings of the IEEE Conference on Data Engineering*, 1990, pp. 430-437.

[Stonebraker92] M. Stonebraker, "An Overview of the Sequoia 2000 Project," *Proceedings of the 37th IEEE Computer Society International Conference (COMPCON)*, 1992, pp. 383-388.

[Teradata85] Teradata Corporation, "DBC/1012 Data Base Computer System Manual, Release 1.3," *C10-0001-01, Teradata Corporation*, 1985.

[TMC87] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, Thinking Machines Technical Report HA87-4, 1987.

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

[Wood93] C. Wood and P. Hodges, “DASD Trends: Cost, Performance, and Form Factor,” *Proceedings of the IEEE*, vol. 81 no. 4, 1993, pp. 573-585.

Appendix A: Data Mapping Algorithms

This appendix gives the algorithms used to perform the mapping from logical array addresses to physical disk addresses, and vice versa. Section A.1 presents some preliminary information about the mapping functions, Section A.2 gives the algorithms for the declustered parity organization described in Chapter 3, and Section A.3 gives the algorithms for the distributed-sparing approach of Chapter 5. The four subsections in each present (1) the variables and data structures used, and the values to which they are initialized, (2) *MapSector*, which maps a logical array address to a physical disk address, (3) *MapParity*, which maps a logical array address to the physical disk address of the corresponding parity unit, and (4) *MapPhysicalToStripeID*, which maps a physical disk address to the identifier of the stripe containing it. These mapping functions are sufficient to implement all of the techniques described in the previous chapters.

A.1. Preliminaries

The mapping code assumes that the storage subsystem is a two-dimensional array of disks, so that each disk is identified by its row and column number. It further assumes that each row of the array is an independent sub-array (refer to layout criterion 1 in Section 3.2.1).

Recall that under declustered parity, data is laid out in *full block design tables*. The mapping code stripes the full tables across the rows comprising the array, to avoid clustering consecutive data on the disks in one row. For an array with *R* rows, full table number 0 is in row 0, number 1 is in row 1, …, number *R*-1 is in row *R*-1, number *R* is in row 0, etc. This allows user data to be striped across all disks in the array, even when the array contains multiple independent sub-arrays. An alternative would be to interleave (stripe) the parity stripes across the full block design tables, instead of filling each full block design table with consecutive user data before switching to the next. The required modifications to the mapping algorithms would be relatively minor, and we did not pursue them.

The number of units on a disk may not be an exact multiple of the number of units on one disk in a full block design table. In order to reduce the wasted space at the end of each disk, the mapping algorithms allow the last full block design table in each row to be incomplete, that is, to contain fewer than *k* copies of the block design table[1]. This leads to a special case in the mapping functions, which is handled by the call to *adjust_parameters* in *MapSector* and *MapParity*, and is handled in-line in *MapPhysicalToStripeID*. The problem is that the identifier of the full block design table is computed by dividing the logical offset by the number of data stripe units in one full table. Since the last full table may be incomplete, it may contain a different number of stripe units. To simplify the code, this case is handled by adjusting the values of the layout parameters used to perform the mapping, so that the same mapping code works for all regions of the array. Section A.4 gives the code for *adjust_parameters*.

A.2. Mapping code for declustered parity

A.2.1. Data structures

Table A.1 shows the variables and data structures used by the three parity declustering mapping routines, and how they are initialized from the parameters of the array and the block design. Many of these are redundant, that is, they are easily derived from other parameters, but including them makes the layout code more efficient and readable. For reference, recall that a block design consists of *b* tuples, each containing *k* objects selected from a set of *v* objects, such that the number of tuples containing each element is *r*, and the number of tuples containing each pair of elements is λ_p.

Recall from Section 3.5.1 that the layout decouples the size of the reconstruction unit from that of the stripe unit. Thus the mapping code uses two different measures of the size of a unit on a disk: one SU is the number of sectors in a stripe unit, and one RU is the number of sectors in a reconstruction unit.

In addition to the above terms, the layout code uses three tables, each proportional in size to the number of elements in the block design. The *LayoutTable* is a *b* by *k* array holding the block design, and is used by the mapping to code identify the disk on which a par-

1. Note that if desired, the amount of wasted space at the end of each disk could be reduced further by allowing the last block design table in the last full block design table to be incomplete as well. We did not implement this in order to avoid further complicating the mapping algorithms.

Variable	Initialization	Description
CompleteFTsPerRow	*PUsPerDisk/(r·k)*	Number of full tables in each row, not including any partial full table
ExtraTablesPerRow	*PUsPerDisk/r* mod *k*	Tables in the partial full table
FullTableDepthInPU	*r·k*	PUs on one disk in one full table
FullTableLimitInSU	*CompleteFTsPerRow· k · SUsPerFullTable*	Number of stripe units per row that occur in non-partial full tables
LastFTOffsetInSU	*CompleteFTsPerRow · FullTableDepthInPU · SUsPerPU*	Disk offset of start of last full table
NoRotate	Config parameter	Suppresses parity rotation
NumCol	Config parameter	Columns in the array
NumParityReps	*k*	Tables in one full table
NumRow	Config parameter	Rows in the array
PUsPerDisk	Config parameter	Size of a disk in PUs
PUsPerTuple	*k*-1	Total PUs of user data in one tuple
StripeUnitsPerDisk	Config parameter	Total SUs on one disk, adjusted to ensure an integral number of tables
SUsPerFullTable	*k·SUsPerTable*	Total SUs of user data in one full table
SUsPerPU	Config parameter	Number of SUs in one PU
SUsPerTable	*b(k*-1*)(SUsPerPU)*	Total SUs of user data in one table
TableDepthInPU	*r*	PUs on one disk in one table
TuplesPerTable	*b*	Parity stripes in one table

Table A.1: Variables used in declustered parity mapping functions.

ticular unit is located. The *OffsetTable* is also *b* by *k*, and holds, for each unit in a table, the disk offset of that unit from the start of the table. Recalling that the layout is constructed by assigning each successive unit to the lowest available offset on the disk indicated by an element of the block design, the offset table is initialized as follows:

```
for (i=0; i<v; i++) first_avail_unit[i] = 0;
  for (i=0; i<b; i++) {
      for (j=0; j<k; j++) {
          OffsetTable[i][j] = first_avail_slot[ LayoutTable[i][j] ];
          first_avail_unit[ LayoutTable[i][j] ]++;
      }
  }
```

The third table, called the *TupleTable*, is used only by *MapPhysicalToStripeID* to implement the inverse mapping. It has one entry for each data or parity unit in a block design table (that is, it is *r·SUsPerPU* by *v*), and identifies the tuple within the block design that was used to lay out that unit. The following code initializes it:

```
  TupleID=0;
  for (l=0; l<SUsPerPU; l++) {
      for (i=0; i<b; i++) {
          for (j=0; j<k; j++) {
              TupleTable[ OffsetTable[i][j]*SUsPerPU +l ][ LayoutTable[i][j] ] = TupleID;
          }
          TupleID++;
      }
  }
```

A.2.2. *MapSector*

The code to map a logical array address to a physical array address is as follows. Local variables can be distinguished from the parameters given in Table A.1 by the fact that they all start with a lower-case letter.

```
void MapSector(logicalSU, col, row, offset)
    unsigned long logicalSU;                    /* input: logical stripe unit address */
    long *col, *row;                                        /* output: disk identifier */
    unsigned long *offset;                     /* output: physical stripe unit offset */
{
    unsigned long fullTableID, fullTableOffset, tableID, tableOffset;
    unsigned long tupleID, tupleOffset, repIndex;
    unsigned long sus_per_fulltable = SUsPerFullTable;
    unsigned long fulltable_depth = FullTableDepthInPU * SUsPerPU;
    unsigned long base_suid = 0, outSU;

    adjust_params(&logicalSU, &sus_per_fulltable, &fulltable_depth, &base_suid);

    /* find fulltable ID within array (across rows) */
    fullTableID = logicalSU / sus_per_fulltable;
    *row = fullTableID % numRow;

    /* convert to fulltable ID on this disk */
    fullTableID /= numRow;

    /* find offset within full block design table */
     fullTableOffset = logicalSU % sus_per_fulltable;

    /* compute offsets into block design table and tuple within table */
    tableID = fullTableOffset / SUsPerTable;
    tableOffset = fullTableOffset % SUsPerTable;
    tupleID = (tableOffset / PUsPerTuple) % TuplesPerTable;
    tupleOffset = tableOffset % PUsPerTuple;

    /* compute parity slot in table. optionally supress rotation */
    repIndex = PUsPerTuple - tableID;
    if (!NoRotate) tupleOffset += ((tupleOffset >= repIndex) ? 1 : 0);

    /* compute the disk identifier */
    *col = LayoutTable[tupleID][tupleOffset];

     /* sum components to find the stripe unit offset within disk */
    outSU   = base_suid;
    outSU += fullTableID * fulltable_depth;
    outSU += tableID * TableDepthInPU * SUsPerPU;
    outSU += OffsetTable[tupleID][tupleOffset] * SUsPerPU;
    outSU += tableOffset / (TuplesPerTable * PUsPerTuple);

    *offset = outSU;
}
```

A.2.3. *MapParity*

MapParity is identical to *MapSector* except that the column is selected according to the location of the parity rather than the location of the data.

```
void MapParity(logicalSU, col, row, offset)
    unsigned long logicalSU;                    /* input: logical stripe unit address */
    long *col, *row;                                     /* output: disk identifier */
    unsigned long *offset;                          /* physical stripe unit offset */
{
    unsigned long fullTableID, fullTableOffset, tableID, tableOffset;
    unsigned long tupleID, tupleOffset, repIndex;
    unsigned long sus_per_fulltable = SUsPerFullTable;
    unsigned long fulltable_depth = FullTableDepthInPU * SUsPerPU;
    unsigned long base_suid = 0, outSU;

    adjust_params(&logicalSU, &sus_per_fulltable, &fulltable_depth, &base_suid);

    /* this section is identical to MapSector */
    fullTableID     = logicalSU / sus_per_fulltable;
    *row            = fullTableID % numRow;
    fullTableID     /= numRow;
    fullTableOffset= logicalSU % sus_per_fulltable;
    tableID         = fullTableOffset / SUsPerTable;
    tableOffset     = fullTableOffset % SUsPerTable;
    tupleID         = (tableOffset / PUsPerTuple) % TuplesPerTable;
    tupleOffset     = tableOffset % PUsPerTuple;

    /* the parity block is in the position indicated by repIndex */
    repIndex        = (NoRotate) ? PUsPerTuple : PUsPerTuple - tableID;
    *col            = LayoutTable[tupleID][repIndex];

    /* compute as before, except use repIndex instead of tupleOffset */
    outSU   = base_suid;
    outSU += fullTableID * fulltable_depth;
    outSU += tableID * TableDepthInPU * SUsPerPU;
    outSU += OffsetTable[tupleID][repIndex] * SUsPerPU;
    outSU += tableOffset / (TuplesPerTable * PUsPerTuple);

    *offset = outSU;
}
```

A.2.4. ***MapPhysicalToStripeID***

Unlike *MapSector* and *MapParity, MapPhysicalToStripeID* does not call *adjust_-params* to handle the case where the locations to be mapped reside in the partial full block design table at the end of a row. The functionality required here is slightly different, and so the code to handle this case is included in-line.

```
void MapPhysicalToStripeID(col, row, offset, outStripeID)
    long col, row, offset;                              /* input: physical disk address */
    unsigned long outStripeID;                              /* output: stripe identifier */
{
    unsigned long fullTableID, tableID, tupleID;
    unsigned long ft_limit = NumCompleteFullTablesPerDisk;
    unsigned long ftDepthInSU = FullTableDepthInPU * SUsPerPU;
    unsigned long tDepthInSU = TableDepthInPU * SUsPerPU;

    /* compute fulltable, table, and tuple this disk unit resides in */
    fullTableID = (offset / ftDepthInSU) * NumRow + row;
    tableID = (offset % ftDepthInSU) / tDepthInSU;
    tupleID = TupleTable[offset % tDepthInSU][col];

    if (fullTableID >= ft_limit) {

        *outStripeIDPtr = ft_limit * NumParityReps * TuplesPerTable * SUsPerPU +
                            row * (ExtraTablesPerDisk * TuplesPerTable) * SUsPerPU;
        *outStripeIDPtr += tableID * TuplesPerTable * SUsPerPU + tupleID;

    } else {

        *outStripeIDPtr = fullTableID * NumParityReps * TuplesPerTable * SUsPerPU +
                            tableID * TuplesPerTable * SUsPerPU + tupleID;
    }
}
```

A.3. Mapping code for declustered parity with distributed sparing

The distributed-sparing layout code uses the following terms in addition to those listed in Table A.1.

Variable	Initialization	Description
FTPerSpareRegion	LCM(*v*-1,*r*)/*r*	Number of full block design tables in one spare region
TablesPerSpareRgn	*k*·LCM(*v*-1,*r*)/*r*	Block design tables per spare region
SpDepthPerRgnInSU	(*k*·LCM(*v*-1,*r*)/(*v*-1)) · *SUsPerPU*	SUs of spare space on one disk in one spare region
SpRegionDepthInSU	*TablesPerSpareRgn* · *TableDepthInPU* · *SUsPerPU* + *SpDepthPerRgnInSU*	Total SUs on one disk in one spare region
LastSpareOffsetInSU	See text	Disk offset of last spare region
NumCompleteSRs	*SUsPerDisk / SpRegionDepthInSUs*	Number of spare regions in array, not including last (partial) region.

Table A.2: Additional layout terms used in distributed sparing.

The initialization of *LastSpareOffsetInSU* is too complex to put in the above table. This term is necessary because, since the last full block design table may be partial, the start of the last region of spare space does not occur at the expected location. It is initialized by multiplying the number of spare regions per row by the depth of one spare region, and adding to this the number of extra tables per row times the depth of one block design table.

In addition to the above terms, the distributed-sparing layout uses two tables to map failed units to spare locations and vice versa. The *SpareTable* is a two-dimensional array of size *TablesPerSpareRgn* by *TuplesPerTable*. Each entry in this array is a structure containing two elements: *spareDisk* and *spareBlockOffsetInSU*. When the mapping code determines that a requested unit is failed, it uses the table ID within the spare region and the tuple ID within the table to index into the SpareTable. The *spareDisk* field within this entry indicates the disk to which this unit has been spared, and the *spareBlockOffsetInSU* field gives the offset into the spare block within this spare region on the indicated disk.

This table is initialized at the time a disk failure occurs by reading the corresponding sparemap computed according to the technique given in Chapter 5.

The second table used by the distributed sparing code is the *InverseSpareTable*, which is the exact inverse of the SpareTable. It is used by *MapPhysicalToStripeID* to compute the inverse mapping, and contains, for each spare unit in a spare region, the table ID within the spare region and the tuple ID within the indicated table for the failed unit that was remapped to the indicated spare unit.

The mapping code uses the function *remap_to_spare_space* to index into the spare table and find the corresponding spare unit. This function is described in Section A.3.4. Each of the three routines has the ability to optionally suppress the remapping of failed units to spare units, or vice versa in the case of *MapPhysicalToStripeID*, which is controlled by the *remap* argument. This is necessary in order for the array controller to discriminate between accesses that encounter a failed unit and those that do not.

A.3.1. *MapSector*

```
void MapSector(logicalSU, col, row, offset, remap)
    unsigned long logicalSU;                        /* input: logical stripe unit address */
    long *col, *row;                                           /* output: disk identifier */
    unsigned long *offset;                                 /* physical stripe unit offset */
    char remap;                /* whether or not we are allowed to remap to spare space */
{
    unsigned long fullTableID, fullTableOffset, tableID, tableOffset;
    unsigned long tupleID, tupleOffset, repIndex;
    unsigned long sus_per_fulltable = SUsPerFullTable;
    unsigned long fulltable_depth = FullTableDepthInPU * SUsPerPU;
    unsigned long base_suid = 0, outSU, spareRegion, spareSpace=0;

    adjust_params(&logicalSU, &sus_per_fulltable, &fulltable_depth, &base_suid);

    fullTableID     = logicalSU / sus_per_fulltable;
    *row            = fullTableID % numRow;
    fullTableID     /= numRow;

    spareRegion     = fullTableID / FTPerSpareRegion;
    spareSpace      = spareRegion * SpDepthPerRgnInSU;

    fullTableOffset= logicalSU % sus_per_fulltable;
    tableID         = fullTableOffset / SUsPerTable;
    tableOffset     = fullTableOffset % SUsPerTable;
    tupleID         = (tableOffset / PUsPerTuple) % TuplesPerTable;
    tupleOffset     = tableOffset % PUsPerTuple;
    repIndex        = PUsPerTuple - tableID;

    if (!NoRotate) tupleOffset += ((tupleOffset >= repIndex) ? 1 : 0);
    *col            = LayoutTable[tupleID][tupleOffset];

    /* remap to distributed spare space if indicated */
    if (remap) {
        remap_to_spare_space(*row, fullTableID, tableID, tupleID, (base_suid) ? 1 : 0,
                                                            spareRegion, col, &outSU);
    } else {

        outSU   = base_suid;
        outSU += fullTableID * fulltable_depth;
        outSU += spareSpace;
        outSU += tableID * TableDepthInPU * SUsPerPU;
        outSU += OffsetTable[tupleID][tupleOffset] * SUsPerPU;
    }
    outSU += tableOffset / (TuplesPerTable * PUsPerTuple);
    *offset = outSU;
}
```

A.3.2. *MapParity*

```
void MapParityDeclustered(logicalSU, col, row, offset, remap)
    unsigned long logicalSU;                    /* input: logical stripe unit address */
    long *col, *row;                                        /* output: disk identifier */
    unsigned long *offset;                              /* physical stripe unit offset */
    char remap;           /* whether or not we are allowed to remap to spare space */
{
    unsigned long fullTableID, fullTableOffset, tableID, tableOffset;
    unsigned long tupleID, tupleOffset, repIndex;
    unsigned long sus_per_fulltable = SUsPerFullTable;
    unsigned long fulltable_depth = FullTableDepthInPU * SUsPerPU;
    unsigned long base_suid = 0, outSU, spareRegion, spareSpace=0;

    adjust_params(&logicalSU, &sus_per_fulltable, &fulltable_depth, &base_suid);

    fullTableID     = logicalSU / sus_per_fulltable;
    *row            = fullTableID % NumRow;
    fullTableID     /= NumRow;

    spareRegion     = fullTableID / FTPerSpareRegion;
    spareSpace      = spareRegion * SpDepthPerRgnInSU;

    fullTableOffset= logicalSU % sus_per_fulltable;
    tableID         = fullTableOffset / SUsPerTable;
    tableOffset     = fullTableOffset % SUsPerTable;
    tupleID         = (tableOffset / PUsPerTuple) % TuplesPerTable;
    tupleOffset     = tableOffset % PUsPerTuple;
    repIndex        = (NoRotate) ? PUsPerTuple : PUsPerTuple - tableID;
    *col            = LayoutTable[tupleID][repIndex];

    if (remap) {
        remap_to_spare_space(*row, fullTableID, tableID, tupleID, (base_suid) ? 1 : 0,
                                                        spareRegion, col, &outSU);
    } else {
        outSU = base_suid;
        outSU += fullTableID * fulltable_depth;
        outSU += spareSpace;
        outSU += tableID * TableDepthInPU * SUsPerPU;
        outSU += OffsetTable[tupleID][repIndex] * SUsPerPU;
    }

    outSU += tableOffset / (TuplesPerTable * PUsPerTuple);
    *offset = outSU;
}
```

A.3.3. *MapPhysicalToStripeID*

```
void MapPhysicalDeclustered(col, row, offset, outStripeID, outSpare, remap)
    unsigned long col, row;                              /* input: disk identifier */
    unsigned long offset;                                /* input: offset into disk */
    unsigned long *outStripeID;                          /* output: stripe identifier */
    char *outSpare;         /* output: flag indicating whether or not mapped unit is spare */
    char remap;                  /* whether or not we are allowed to remap to spare space */
{
    unsigned long fullTableID, tableID, tupleID;
    unsigned long ft_limit = numCompleteFullTablesPerDisk;
    unsigned long ftDepthInSU = FullTableDepthInPU * SUsPerPU;
    unsigned long tDepthInSU = TableDepthInPU * SUsPerPU;
    unsigned long srDepthInSU, spRegion, spareBlockOffsetInSU;
    unsigned long tableInSpareRegion, tableOffset, ftOffset;

    *outSpare = 0;

    srDepthInSU = ftDepthInSU * FullTablesPerSpareRgn+SpDepthPerRgnInSU;
    spRegion = offset / srDepthInSU;

    /* check if unit we are trying to inverse-map is a spare unit */
    if ( (offset >= LastSpareOffsetInSU && offset < stripeUnitsPerDisk) ||
                (((spRegion+1)*srDepthInSU - SpDepthPerRgnInSU <= offset) &&
                ((spRegion+1)*srDepthInSU > offset))) {

        *outSpare = 1;
        if (!remap) return;                              /* don't waste any more time */

        /* the indicated unit is in a spare space region */
        if (offset >= LastSpareOffsetInSU) {
            spareBlockOffsetInSU = offset - LastSpareOffsetInSU;
        } else {
            spareBlockOffsetInSU = (offset % SpareRegionDepthInSU) -
                    TablesPerSpareRgn * (TableDepthInPU * SUsPerPU);
        }
        tableInSpareRegion =
                    InverseSpareTable[spareBlockOffsetInSU][col].tableInSpareRegion;
        tupleID        = InverseSpareTable[spareBlockOffsetInSU][col].offsetInTable;

        tableOffset    = spRegion * TablesPerSpareRgn + tableInSpareRegion;
        ftOffset       = tableOffset / NumParityReps;
        fullTableID    = ftOffset * NumRow + row;
        tableID        = tableInSpareRegion % NumParityReps;

    } else {

        /* not in a spare space region */
        offset         -= spRegion * SpDepthPerRgnInSU;
        fullTableID    = (offset / ftDepthInSU) * NumRow + row;
        tableID        = (offset % ftDepthInSU) / tDepthInSU;
        tupleID        = TupleTable[offset % tDepthInSU][col];
    }
    if (fullTableID >= ft_limit) {

        *outStripeID = ft_limit * NumParityReps * TuplesPerTable * SUsPerPU +
                                    row * (ExtraTablesPerDisk * TuplesPerTable) * SUsPerPU;

        *outStripeID += tableID * TuplesPerTable * SUsPerPU + tupleID;

    } else {

        *outStripeID = fullTableID * NumParityReps * TuplesPerTable * SUsPerPU +
                    tableID * TuplesPerTable * SUsPerPU + tupleID;
    }
}
```

A.3.4. *remap_to_spare_space*

```
void remap_to_spare_space(row, fullTableID, tableID, tupleID,
                                             base_suid, spareRegion, outCol, outSU)
unsigned long row;                                                    /* input */
unsigned long fullTableID, tableID, tupleID;                          /* input */
unsigned long base_suid, spareRegion;                                 /* input */
unsigned long *outCol, *outSU;              /* output: new column and disk offset */
{
    unsigned long ftID, spareTableStartSU, tableInSpareRegion;

    /* FullTableID may have gotten tweaked by adjust_params.
     * Detect this by noticing that base_suid is not 0.
    */
    ftID = (base_suid==0) ?   fullTableID :
                              LastFTOffsetInSU / (FullTableDepthInPU * SUsPerPU);
    tableInSpareRegion = (ftID * NumParityReps + tableID) % TablesPerSpareRgn;

    *outCol = SpareTable[tableInSpareRegion][tupleID].spareDisk;

    spareTableStartSU = (spareRegion == NumCompleteSRs) ?
        LastFTOffsetInSU + ExtraTablesPerRow * TableDepthInPU * SUsPerPU :
        (spareRegion+1) * SpRegionDepthInSU - SpDepthPerRgnInSU;

    *outSU = spareTableStartSU +
                        SpareTable[tableInSpareRegion][tupleID].spareBlockOffsetInSUs;
}
```

A.4. ***adjust_params***

```
void adjust_params(logicalSU, sus_per_fulltable, fulltable_depth, base_suid)
    unsigned long *logicalSU;                    /* input/output: logical stripe unit */
    unsigned long *sus_per_fulltable;                                /* input/output */
    unsigned long *fulltable_depth;                                  /* input/output */
    unsigned long *base_suid;                                        /* input/output */
{
    if (*logicalSU >= FullTableLimitInSU) {

        /* new full table size is size of last full table on disk */
        *sus_per_fulltable = ExtraTablesPerRow * SUsPerTable;

        /* new full table depth is corresponding depth */
        *fulltable_depth = ExtraTablesPerRow * TableDepthInPU * SUsPerPU;

        /* set up the new base offset */
        *base_suid = LastFTOffsetInSU;

        /* convert logical address to an offset into the last fulltable */
        *logicalSU -= FullTableLimitSUID;
    }
}
```

Appendix B: Block Designs

This chapter summarizes the block designs that are known to us. As discussed in Chapter 3, Hanani [Hanani75] gives a list of designs and a set of techniques which together allow the generation of a design with any k when $v \leq 43$. Section B.1 therefore summarizes known designs on $v > 43$. Section B.2 gives a chart indicating the designs that exist in the block design database mentioned in Chapter 3. Section B.3 gives the block designs on $v = 40$ that were used to generate the simulation results in Chapters 3 through 5.

B.1. Block designs on *v* > 43

Each of the designs in Table B.1 can be found in the tables of Hall [Hall86, pp. 404-423], Chee [Chee90], or Mathon [Mathon90]. Given a block design B with parameters b, v, and k, a *complementary* block design B' with parameters $b' = b$, $v' = v$, and $k' = v - k$ can be constructed by forming the set of b tuples such that tuple i in B' contains exactly those elements that do not appear in tuple i of B. Thus, the table presents only designs with $k \leq v/2$. There also exist a large number of general construct techniques that can be used when the parameters of the design meet certain criteria [Hall86, Hanani75], so this list is by no means exhaustive.

B.2. Designs in the database

Table B.2 lists the designs in the block design database used in this study. The database contains the trivial designs on $k = 2$, $k = v$-1, $k = v$-2, and $k = v$ for all $v \leq 40$. We are actively adding designs, so this list may not be exhaustive.

The database is available via anonymous ftp from the machine *ftp.cs.cmu.edu* (internet address 128.2.206.173) in the file *project/nectar-io/Declustering/BD_database.tar.Z*. This file was generated using the UNIX utilities *tar* and *compress*. The designs are given in expanded form, so none of the encoding rules described in subsequent sections are necessary. There are 846 designs in the database. The above-named file is about 2.8 MB com-

v	*k*	*v*	*k*	*v*	*k*
45	3, 5, 9, 11, 12	46	4, 6, 10, 16	47	23
49	3, 4, 7, 9, 16, 21	50	8, 15	51	3, 5, 6, 25
52	4, 13, 18	53	13, 14	55	3, 4, 5, 10, 27
56	6, 11, 12, 16	57	3, 7, 8	58	4
59	29	61	3, 4, 5, 6, 10, 15, 16, 25	63	3, 7, 31
64	4, 8, 16, 28	65	5	66	6, 11, 26
67	3, 11, 12, 33	69	3, 17	70	24
71	5, 7, 8, 15, 21, 35	73	3, 4, 9, 10	75	3, 5, 15, 37
76	4, 6, 16, 19	77	7	78	22
79	3, 13, 27, 39	81	3, 5, 6, 9, 15, 21, 27	83	41
85	4, 5, 7, 21	88	4	91	6, 7, 10
96	6, 20	97	4	100	4, 25
101	5, 25	105	5	106	6
109	4, 9, 28	111	6	112	4, 7
113	8	117	9	120	8
121	4, 5, 6, 11, 25, 40	124	4	125	5, 25
126	6	133	12, 33	136	6
141	5	145	5, 9, 25	151	6
153	9	156	6, 31	161	5
165	5	169	7, 13	175	30
181	6, 10	183	14	186	6, 31
217	7	223	37	232	8
256	16	273	17	288	8
289	17	307	18	361	19
381	20	496	16	529	23
553	24	625	25	651	26
729	27	757	28	841	29
871	30	961	31	993	32
1024	32	1057	33	1369	37
1407	38				

Table B.1: Known block designs on *v* > 43.

pressed, and 21.5 MB uncompressed. A README file in the database describes the block design file format.

B.3. Block designs used in the simulations

The designs are presented in compacted form, since many of them are quite large. Tuples in a design are always specified using angle brackets, for example, <0,1,2>. The notation *mod p* after a tuple indicates that all the possible residues modulo *p* should be added to every element in the tuple, and the results should be expressed modulo *p*. If a tuple contains the symbol ∞, that symbol should not be modified when adding the modulo values. For example, the notation <∞,1,2> *mod* 3 expands to the following set of tuples: <∞,1,2>, <∞,2,0>, and <∞,0,1>. A *mod p* clause that is marked “of period *x*” indicates that only the first *x* tuples generated by adding the residues modulo *p* should be used, and the

v	*k*	*v*	*k*
v <20	all 2 ≤ *k* ≤ *v*	20	2, 4-16, 18-20
21	2-21	22	2-22
23	2-23	24	2-3, 5, 7-17, 19, 21-24
25	2-25	26	2, 4-6, 8-18, 20-22, 24-26
27	2-27	28	2-3, 5-23, 25-28
29	2-29	30	2-3, 6-9, 11-19, 21-24, 27-30
31	2-31	32	2-3, 5-27, 29-32
33	2-3, 6-8, 10, 12-21, 23, 25-27, 30-33	34	2-3, 5-11, 13-21, 23-29, 31-34
35	2-2, 5-14, 16-19, 21-30, 33-35	36	2-3, 6-7, 9-27, 29-30, 33-36
37	2-37	38	2-2, 6-32, 36-38
39	2-3, 5-34, 36-39	40	2-4, 6-34, 36-40
41	2-41	42	10, 16-17, 25-26, 32
43	7, 21-22, 36	44	22
49	4, 7, 42, 45	51	6, 45
52	4, 48	56	11, 16, 40, 45
57	8, 49	61	4-5, 56-57
62	5,57	64	8, 56
66	6, 11, 55, 60	67	33-34
68	34	73	9, 64
76	6, 70	78	22, 56
79	4, 13, 66, 75	81	9, 72
91	6-7, 10, 81, 84-85	97	4
121	11, 110	133	12, 121
151	4	273	17, 256
289	17, 272	307	18, 289
361	19, 342	381	20, 361

Table B.2: Block designs in the database.

remaining tuples discarded.

Often the elements of the tuples are specified as ordered pairs, instead of as integers. For example, the notation <(0,1),(1,2),(0,2)> indicates a tuple containing three elements, each of which is an ordered pair. In this case, the *mod* addendum takes the form *mod (p,q)*, and indicates that all the residues modulo *p* should be added to the first number in each order pair of the tuple, and for each tuple so generated, all the residues modulo *q* should be added to the second number. For example, <(∞,1),(1,2)> mod (2,3) expands to following set of tuples: <(∞,1),(1,2)>, <(∞,2),(1,0)>, <(∞,0),(1,1)>, <(∞,1),(0,2)>, <(∞,2),(0,0)>, <(∞,0),(0,1)>. After the expansion is complete, each unique ordered pair should be replaced by a unique integer in the range [0, *v*-1] to produce the final design. In some designs, the *mod* suffix contains a dash, for example, *mod* (-,19). This indicates that no residues should be added to the first ordered pair, but residues should be added to the second. Equivalently, the dash can be interpreted as a “1”.

Some designs in this section are specified as "residual" or "derived" designs, which means that they are generated from other designs [Hall86]. Given a block design with $v = b$ (called a *symmetric* design), the residual design is formed by deleting one of the tuples T_i, and then deleting all the objects contained in T_i from all other tuples in the design. A derived design is generated from a symmetric design by selecting one tuple T_0 and constructing tuples T'_1 through T'_{b-1} where T'_i contains the λ_p objects common to T_0 and T_i.

B.3.1. Designs on *v* = 40

Block designs on $v = 40$ used in the simulations include $k = 2$, 3, 10, 20, 30, and 40. The only possible block design on v=40, k=2 is the complete design, which has b=780, r=39, and λ_p=1. This design consists of all possible 2-element subsets of the set of integers in [0, 39].

1. v=40, k=3, b=520, r=39, λ_p=2:

<∞,0,19> mod 39

<0,13,26> mod 39 of period 13

<0,1,12> mod 39	<0,2,11> mod 39	<0,3,10> mod 39	<0,4,9> mod 39
<0,5,8> mod 39	<0,6,7> mod 39	<0,2,20> mod 39	<0,4,21> mod 39
<0,6,22> mod 39	<0,8,23> mod 39	<0,10,24> mod 39	<0,12,25> mod 39

2. v=40, k=10, b=156, r=39, λ_p=9:

<(∞,∞) (1,1) (1,3) (1,9) (2,2) (2,8) (2,6) (2,11) (2,5) (2,7)> mod (3,13)

<(0,0) (0,4) (0,12) (0,10) (1,8) (1,11) (1,7) (2,4) (2,12) (2,10)> mod (3,13)

<(0,0) (0,4) (0,12) (0,10) (1,8) (1,11) (1,7) (2,4) (2,12) (2,10)> mod (3,13)

<(0,0) (0,1) (0,3) (0,9) (1,2) (1,6) (1,5) (2,1) (2,3) (2,9)> mod (3,13)

Note that the second block is used twice.

3. v=40, k=20, b=78, r=39, λ_p=19:

Construct the block design on v=39, k=19, λ_p=9 from the set of tuples given below. This yields a design B with objects 0 through 38. Construct the complement design B', which has parameters v=39, k=20, λ_p=10. Append to each tuple in B the object 39. Concatenate together the tuples of *B* and *B*' to form the required design.

The required design on *v*=39, *k*=19, λ_p=9 is generated from:

<(∞,∞) (0,1) (0,4) (0,16) (0,7) (0,9) (0,17) (0,11) (0,6) (0,5) (1,1) (1,4) (1,16) (1,7) (1,9)
(1,17) (1,11) (1,6) (1,5)> mod (-,19)

<(0,0) (0,2) (0,8) (0,13) (0,14) (0,18) (0,15) (0,3) (0,12) (0,10) (1,1) (1,4) (1,16) (1,7) (1,9)
(1,17) (1,11) (1,6) (1,5)> mod (-,19)

<(1,0) (1,1) (1,2) (1,4) (1,8) (1,16) (1,13) (1,7) (1,14) (1,9) (1,18) (1,17) (1,15) (1,11) (1,3)
(1,6) (1,12) (1,5) (1,10)>

4. *v*=40, *k*=30, *b*=156, *r*=117, λ_p=87:

This design is generated as the complement of the *v*=40, *k*=10, *b*=156, *r*=39, λ_p=9 design given above.

5. *v*=40, *k*=40:

In order to achieve criterion six, we use a special design whenever $v = k$. The design has $b = r = \lambda_p = v = k$, and is constructed by listing one tuple containing all the objects in order, duplicating it until there are *v* copies, and rotating each tuple one slot to the right with respect to the previous. By suppressing the rotation of parity in the mapping algorithms, this design implements the left-symmetric RAID Level 5 layout. Since the design on $v = 40$ is large, we demonstrate the structure of the design for $v = 10$ here.

```
0 1 2 3 4 5 6 7 8 9
9 0 1 2 3 4 5 6 7 8
8 9 0 1 2 3 4 5 6 7
7 8 9 0 1 2 3 4 5 6
6 7 8 9 0 1 2 3 4 5
5 6 7 8 9 0 1 2 3 4
4 5 6 7 8 9 0 1 2 3
3 4 5 6 7 8 9 0 1 2
2 3 4 5 6 7 8 9 0 1
1 2 3 4 5 6 7 8 9 0
```

B.3.2. Designs on *k* = 4

This section gives the designs on *k*=4 used in Section 5.5.2.

1. $v=10$, $k=4$, $b=15$, $r=6$, $\lambda_p=2$ is the residual design of $v=16$, $b=16$, $r=6$, $k=6$, $\lambda_p=2$:

   <(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (1,1,0,0), (0,0,1,1)> mod (2,2,2,2)

2. $v=20$, $b=95$, $r=19$, $k=4$, $\lambda_p=3$ (all tuples mod 19):

   <∞, 0, 1, 6> <0, 1, 3, 7> <0, 1, 8, 11> <0, 2, 5, 9> <0, 2, 6, 11>

3. $v=40$, $b=130$, $r=13$, $k=4$, $\lambda_p=1$ (all tuples mod 40):

   <0, 1, 26, 32> <0, 7, 19, 36> <0, 3, 16, 38>
   <0, 10, 20, 30> of period 10

4. $v=61$, $k=4$, $b=305$, $r=20$, $\lambda_p=1$ (all tuples mod 61):

   <0, 3, 18, 23> <0, 4, 6, 33> <0, 7, 8, 24> <0, 9, 19, 30> <0, 13, 25, 39>

5. $v=79$, $k=4$, $b=1027$, $r=52$, $\lambda_p=2$ (all tuples mod 79):

<0, 1, 23, 55>	<0, 3, 69, 7>	<0, 9, 49, 21>	<0, 27, 68, 63>
<0, 2, 46, 31>	<0, 6, 59, 14>	<0, 18, 19, 42>	<0, 54, 57, 47>
<0, 4, 13, 62>	<0, 12, 39, 28>	<0, 36, 38, 5>	<0, 29, 35, 15>
<0, 8, 26, 45>	<0, 24, 78, 56>	<0, 72, 76, 10>	<0, 58, 70, 30>
<0, 16, 52, 11>	<0, 48, 77, 33>	<0, 65, 73, 20>	<0, 37, 61, 60>
<0, 32, 25, 22>	<0, 17, 75, 66>	<0, 51, 67, 40>	<0, 74, 43, 41>
<0, 64, 50, 44>	<0, 34, 71, 53>		

6. $v=97$, $k=4$, $b=1552$, $r=64$, $\lambda_p=2$ (all tuples mod 97):

<0, 1, 35, 61>	<0, 5, 78, 14>	<0, 25, 2, 70>	<0, 28, 10, 59>
<0, 43, 50, 4>	<0, 21, 56, 20>	<0, 8, 86, 3>	<0, 40, 42, 15>
<0, 6, 16, 75>	<0, 30, 80, 84>	<0, 53, 12, 32>	<0, 71, 60, 63>
<0, 64, 9, 24>	<0, 29, 45, 23>	<0, 48, 31, 18>	<0, 46, 58, 90>
<0, 36, 96, 62>	<0, 83, 92, 19>	<0, 27, 72, 95>	<0, 38, 69, 87>
<0, 93, 54, 47>	<0, 77, 76, 41>	<0, 94, 89, 11>	<0, 82, 57, 55>
<0, 22, 91, 81>	<0, 13, 67, 17>	<0, 65, 44, 85>	<0, 34, 26, 37>
<0, 73, 33, 88>	<0, 74, 68, 52>	<0, 79, 49, 66>	<0, 7, 51, 39>

7. $v=151$, $k=4$, $b=3775$, $r=100$, $\lambda_p=2$ (all tuples mod 151):

<0, 1, 32, 118>	<0, 6, 41, 104>	<0, 36, 95, 20>	<0, 65, 117, 120>
<0, 88, 98, 116>	<0, 75, 135, 92>	<0, 148, 55, 99>	<0, 133, 28, 141>
<0, 43, 17, 91>	<0, 107, 102, 93>	<0, 38, 8, 105>	<0, 77, 48, 26>
<0, 9, 137, 5>	<0, 54, 67, 30>	<0, 22, 100, 29>	<0, 132, 147, 23>
<0, 37, 127, 138>	<0, 71, 7, 73>	<0, 124, 42, 136>	<0, 140, 101, 61>
<0, 85, 2, 64>	<0, 57, 12, 82>	<0, 40, 72, 39>	<0, 89, 130, 83>
<0, 81, 25, 45>	<0, 33, 150, 119>	<0, 47, 145, 110>	<0, 131, 115, 56>
<0, 31, 86, 34>	<0, 35, 63, 53>	<0, 59, 76, 16>	<0, 52, 3, 96>
<0, 10, 18, 123>	<0, 60, 108, 134>	<0, 58, 44, 49>	<0, 46, 113, 143>

<0, 125, 74, 103>	<0, 146, 142, 14>	<0, 121, 97, 84>	<0, 122, 129, 51>
<0, 128, 19, 4>	<0, 13, 114, 24>	<0, 78, 80, 144>	<0, 15, 27, 109>
<0, 90, 11, 50>	<0, 87, 66, 149>	<0, 69, 94, 139>	<0, 112, 111, 79>
<0, 68, 62, 21>	<0, 106, 70, 126>		

Appendix C: Simulation and Model Data

This appendix gives the raw data for each plot in the thesis. Recall that for fault-free and degraded-mode simulations, the simulation was not terminated until the 95% confidence interval on the average user response time had fallen to less than 5% of the mean. In reconstruction mode, the response-time confidence intervals are uniformly small because the simulation is continued until reconstruction is complete, and thus a very large number of accesses are run. For these reasons, the tables do not report confidence intervals on user response time.

Read Fraction	Workload Increase Factor
0.0	1.23
0.1	1.25
0.2	1.28
0.3	1.31
0.4	1.34
0.5	1.38
0.6	1.44
0.7	1.51
0.8	1.62
0.9	1.76
1.0	2.00

Table C.1: Workload increase factor data from Figure 2.2.

40/10 Declustered				4x 9+1 RAID Level 5			
90th percentile		average		90th percentile		average	
Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time
202.1	77.4	202.1	36.6	201.1	77.8	201.1	36.7
406.2	86.0	406.2	42.2	406.9	85.8	406.9	42.0
612.2	103.8	612.2	50.7	616.2	103.6	616.2	50.9
696.1	115.0	696.1	56.1	697.7	116.2	697.7	56.6
828.0	141.0	828.0	68.2	824.3	142.8	824.3	68.7
992.5	216.0	992.5	101.2	991.7	216.2	991.7	101.4

Table C.2: Response time data from Figure 3.5.

40/10 Declustered				4x 9+1 RAID Level 5			
90th percentile		average		90th percentile		average	
Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time
198.9	77.8	198.9	37.2	201.6	78.4	201.6	37.6
406.8	90.6	406.8	44.6	406.4	92.8	406.4	45.8
610.0	119.4	610.0	58.1	610.2	153.0	610.2	71.1
697.8	141.0	697.8	67.7	675.3	301.4	675.3	113.2
817.0	195.4	817.0	93.2				
908.2	279.6	908.2	130.1				

Table C.3: Response time data from Figure 3.6.

40/10 Declustered				4x 9+1 RAID Level 5			
90th percentile		average		90th percentile		average	
Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time	Acc Rate	Resp Time
201.1	90.4	201.1	45.0	201.5	82.0	201.5	40.5
405.0	108.8	405.0	55.2	401.0	98.8	401.0	49.0
606.9	139.6	606.9	71.4	600.6	153.2	600.6	71.6
806.5	212.4	806.5	105.0	679.6	337.8	679.6	123.5
882.4	284.0	882.4	136.2				

Table C.4: Response time data from Figure 3.7.

40/10 Declustered			4x 9+1 RAID Level 5		
Acc Rate	Recon Time	95% Conf	Acc Rate	Recon Time	95% Conf
200.0	210.2	0.7	200.0	355.2	4.7
400.0	218.6	0.4	400.0	716.8	5.4
600.0	260.0	2.3	600.0	2551.0	38.8
800.0	572.4	19.6	680.0	11610.6	785.1
1000.0	970.8	6.7			

Table C.5: Reconstruction time data from Figure 3.8.

40/10 Declustered			4x 9+1 RAID Level 5		
Acc Rate	P(Loss)	95% Conf	Acc Rate	P(Loss)	95% Conf
5.0	1.8e-4	1.0e-6	5.0	6.9e-5	1.0e-6
10.0	1.8e-4	0.0	10.0	1.4e-4	1.0e-6
15.0	2.2e-4	2.0e-6	15.0	5.0e-4	8.0e-6
20.0	4.8e-4	1.7e-5	17.0	2.3e-3	1.5e-4
25.0	8.2e-4	6.0e-6			

Table C.6: Reliability data from Figure 3.9a.

40/10 Declustered			4x 9+1 RAID Level 5		
Acc Rate	P(Loss)	95% Conf	Acc Rate	P(Loss)	95% Conf
5.0	3.5e-04	1.0e-06	5.0	1.4e-04	2.0e-06
10.0	3.7e-04	1.0e-06	10.0	2.8e-04	2.0e-06
15.0	4.4e-04	4.0e-06	15.0	9.9e-04	1.5e-05
20.0	9.7e-04	3.3e-05	17.0	4.5e-03	3.0e-04
25.0	1.6e-03	1.1e-05			

Table C.7: Reliability data from Figure 3.9b.

α	90%	average
0.03 (Mirroring)	82.6	37.5
0.03 (G=2)	80.6	35.3
0.05	93.4	46.6
0.23	99.8	48.7
0.49	99.2	48.5
0.74	99.6	48.7
1.00	100.2	48.9

Table C.8: Response time data from Figure 3.10.

α	90%	average
0.03 (Mirroring)	82.6	37.9
0.03 (G=2)	79.4	35.3
0.05	95.6	47.5
0.23	111.6	54.6
0.49	133.4	64.7
0.74	162.0	77.6
1.00	212.8	100.7

Table C.9: Response time data from Figure 3.11.

α	Recon Time	95% Conf
0.03 (Mirroring)	925.4	16.5
0.03 (*G*=2)	232.0	3.4
0.05	229.0	2.1
0.23	228.2	1.4
0.49	448.8	3.1
0.74	895.4	11.8
1.00	1757.8	32.0

Table C.10: Reconstruction time data from Figure 3.12.

α	5 Years		10 Years	
	P(Failure)	95% Conf	P(Failure)	95% Conf
0.03 (Mirroring)	2.0e-05	0.0e+00	4.0e-05	1.0e-06
0.03 (*G*=2)	2.0e-04	3.0e-06	3.9e-04	5.0e-06
0.05	1.9e-04	2.0e-06	3.9e-04	3.0e-06
0.23	1.9e-04	1.0e-06	3.9e-04	2.0e-06
0.49	3.8e-04	3.0e-06	7.6e-04	5.0e-06
0.74	7.5e-04	1.0e-05	1.5e-03	2.0e-05
1.00	1.5e-03	2.8e-05	3.0e-03	5.5e-05

Table C.11: Reliability data from Figure 3.13.

α	*w*=0.0	*w*=0.2	*w*=0.4	*w*=0.6	*w*=0.8	*w*=1.0
0.00	2.00	1.70	1.54	1.44	1.37	1.32
0.10	1.82	1.59	1.46	1.38	1.32	1.28
0.20	1.67	1.49	1.39	1.32	1.27	1.24
0.30	1.54	1.40	1.33	1.27	1.23	1.20
0.40	1.43	1.33	1.27	1.22	1.19	1.17
0.50	1.33	1.26	1.21	1.18	1.16	1.14
0.60	1.25	1.20	1.16	1.14	1.12	1.11
0.70	1.18	1.14	1.12	1.10	1.09	1.08
0.80	1.11	1.09	1.08	1.07	1.06	1.05
0.90	1.05	1.04	1.04	1.03	1.03	1.02
1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table C.12: Workload increase factor data from Figure 3.14.

α	Applied Load				Avg Surviving Disk Utilization			
	λ=14	λ=12	λ=10	λ=8	λ=14	λ=12	λ=10	λ=8
0.05	18.73	16.05	13.38	10.70	0.79	0.67	0.56	0.44
0.23	17.60	15.09	12.57	10.06	0.80	0.68	0.56	0.44
0.49	16.21	13.90	11.58	9.27	0.80	0.67	0.56	0.44
0.74	15.02	12.88	10.73	8.59	0.80	0.67	0.56	0.44
1.00	14.00	12.00	10.00	8.00	0.79	0.68	0.56	0.45

Table C.13: Data from Figure 3.15.

α	λ = 8	λ = 10	λ = 12	λ = 14
0.03	35.2	38.0	42.3	49.2
0.05	46.3	54.8	69.7	102.6
0.23	48.4	56.8	72.3	107.6
0.49	48.2	56.7	71.3	102.8
0.74	47.9	56.0	69.5	102.0
1.00	47.9	55.6	68.9	95.7

Table C.14: Response time data from Figure 3.16.

Access Size	α=0.05		α=0.23		α=0.49		α=0.74		α=1.0	
	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf
2^2	0.2	0.0	0.2	0.0	0.2	0.0	0.2	0.0	0.2	0.0
2^3	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0
2^4	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0
2^5	0.9	0.0	0.9	0.0	0.9	0.0	0.9	0.0	0.9	0.0
2^6	1.6	0.0	1.6	0.0	1.6	0.0	1.7	0.0	1.7	0.0
2^7	2.9	0.0	3.1	0.0	3.1	0.0	3.2	0.0	3.4	0.0
2^8	5.0	0.0	5.3	0.1	6.0	0.1	6.1	0.0	6.5	0.0
2^9	7.7	0.1	7.1	0.1	8.8	0.0	12.0	0.1	12.8	0.2
2^{10}	11.4	0.2	10.1	0.1	13.2	0.2	15.7	0.3	20.2	0.2
2^{11}	14.3	0.1	14.1	0.1	17.5	0.1	22.3	0.2	22.7	0.2
2^{12}	16.3	0.3	18.0	0.2	20.8	0.2	26.7	0.4	28.4	0.2

Table C.15: Transfer rate data from Figure 3.17a.

Access Size	α=0.05		α=0.23		α=0.49		α=0.74		α=1.0	
	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf
2^2	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0
2^3	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0
2^4	0.2	0.0	0.2	0.0	0.2	0.0	0.2	0.0	0.2	0.0
2^5	0.5	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0
2^6	0.8	0.0	0.6	0.0	0.6	0.0	0.6	0.0	0.6	0.0
2^7	1.9	0.0	1.4	0.0	1.2	0.0	1.2	0.0	1.2	0.0
2^8	3.1	0.0	2.6	0.0	2.5	0.0	2.7	0.0	2.5	0.0
2^9	5.8	0.1	5.0	0.0	4.8	0.0	4.5	0.0	5.2	0.1
2^{10}	8.9	0.1	8.3	0.1	9.1	0.1	9.5	0.1	9.0	0.2
2^{11}	11.1	0.2	12.6	0.1	14.8	0.2	14.9	0.1	16.2	0.3
2^{12}	13.2	0.2	16.3	0.1	18.6	0.2	22.0	0.1	24.4	0.1

Table C.16: Transfer rate data from Figure 3.17b.

Access Size	α=0.05		α=0.23		α=0.49		α=0.74		α=1.0	
	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf
2^2	0.03	0.00	0.03	0.00	0.02	0.00	0.03	0.00	0.02	0.00
2^3	0.03	0.00	0.02	0.00	0.03	0.00	0.03	0.00	0.03	0.00
2^4	0.02	0.00	0.03	0.00	0.03	0.00	0.03	0.00	0.03	0.00
2^5	0.04	0.00	0.04	0.00	0.04	0.00	0.04	0.00	0.04	0.00
2^6	0.06	0.00	0.07	0.00	0.07	0.00	0.07	0.00	0.07	0.00
2^7	0.11	0.00	0.12	0.00	0.12	0.00	0.13	0.00	0.13	0.00
2^8	0.18	0.00	0.19	0.00	0.23	0.00	0.23	0.00	0.25	0.00
2^9	0.27	0.00	0.25	0.00	0.32	0.00	0.44	0.00	0.48	0.00
2^{10}	0.36	0.01	0.33	0.00	0.43	0.01	0.52	0.00	0.71	0.00
2^{11}	0.41	0.00	0.41	0.00	0.51	0.01	0.65	0.00	0.68	0.00
2^{12}	0.42	0.01	0.47	0.00	0.54	0.01	0.70	0.00	0.74	0.00

Table C.17: Transfer rate data from Figure 3.18a.

Access Size	α=1.0	α=0.74	α=0.49	α=0.23	α=0.05
2^2	0.17	0.17	0.17	0.17	0.17
2^3	0.31	0.27	0.30	0.31	0.30
2^4	0.51	0.52	0.48	0.52	0.52
2^5	0.88	0.86	0.87	0.88	0.89
2^6	1.68	1.70	1.69	1.65	1.71
2^7	3.35	3.39	3.33	3.35	3.48
2^8	6.52	6.59	6.52	6.69	6.80
2^9	12.82	13.06	13.21	13.72	13.87
2^{10}	20.18	21.49	21.94	21.81	22.42
2^{11}	22.69	23.18	23.45	23.64	23.92
2^{12}	28.43	28.16	28.39	28.34	28.64

Table C.18: Normalized transfer rate data from Figure 18b.

α	ReconTime		90th percentile User Resp Time			Average User Resp Time		
	Time	95% Conf	Recon	Degraded	Fault-Free	Recon	Degraded	Fault-Free
0.11	507	57	280.80	263.40	255.20	147.29	129.70	128.66
0.21	523	21	270.60	249.20	231.80	148.98	127.13	120.97
0.47	1039	14	281.60	265.20	223.80	165.59	142.05	123.14
0.74	1899	40	294.80	277.00	213.80	179.65	152.37	122.69
1.00	3180	290	319.60	309.20	220.40	191.12	167.17	127.54

Table C.19: Data from Figure 3.19.

α	Reconstruction Time						Avg Response Time		
	1 Track		7 Tracks		14 Tracks		1 Track	7 Track	14 Track
	Time	95% Conf	Time	95% Conf	Time	95% Conf			
0.03	247.4	6.3	224.0	0.6	221.0	0.6	36.3	38.1	40.9
0.05	234.0	9.0	224.4	1.2	223.2	0.7	51.4	55.8	63.1
0.23	228.8	1.1	229.4	0.7	227.2	0.4	67.9	93.4	129.4
0.49	450.4	1.3	331.2	5.4	283.8	4.7	78.2	146.1	323.5
0.74	897.2	6.5	634.2	20.4			89.6	352.0	
1.00	1773.8	38.5					111.2		

Table C.20: Data from Figure 3.24.

α	1 Track	7 Tracks	14 Tracks
0.03	0.2	0.6	1.2
0.05	0.9	1.9	3.6
0.23	4.4	10.3	18.4
0.49	13.2	32.3	78.0
0.74	36.6	191.9	
1.00	110.6		

Table C.21: Cumulative degradation data from Figure 3.25.

Access Size	RAID5	G=3		G=10		G=20		G=30	
		Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt
2^2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
2^3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
2^4	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
2^5	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
2^6	1.7	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.7
2^7	3.4	2.9	2.9	3.0	3.1	3.1	3.1	3.1	3.2
2^8	6.5	5.1	5.0	5.4	5.3	5.9	6.0	5.9	6.1
2^9	12.8	7.8	7.7	8.0	7.1	9.6	8.8	11.1	12.0
2^{10}	20.2	12.2	11.4	12.3	10.1	15.2	13.2	15.4	15.7
2^{11}	22.7	16.8	14.3	17.0	14.1	20.7	17.5	21.9	22.3
2^{12}	28.4	21.2	16.3	22.1	18.0	26.6	20.8	26.4	26.7

Table C.22: Transfer rate data from Figure 3.33.

α	1-Thread		8-Thread		16-Thread		Disk-Oriented	
	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf
0.05	1393.6	8.0	278.2	0.7	262.2	2.5	229.0	2.1
0.23	2970.6	11.0	561.0	2.3	382.4	1.2	228.2	1.4
0.49	4763.2	23.3	1106.6	6.9	792.0	2.5	448.8	3.1
0.74	6704.6	36.0	1765.4	16.5	1345.0	6.9	895.4	11.8
1.00	9141.4	43.7	2684.4	14.4	2139.8	10.5	1757.8	32.0

Table C.23: Reconstruction time data from Figure 4.1a.

α	1-Thread		8-Thread		16-Thread		Disk-Oriented	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.05	43.8	0.1	46.7	0.1	46.7	0.1	51.6	0.2
0.23	50.5	0.0	56.5	0.1	59.5	0.2	67.6	0.4
0.49	57.0	0.1	63.4	0.1	66.0	0.1	78.1	0.3
0.74	64.0	0.2	70.4	0.2	72.9	0.2	90.1	0.6
1.00	72.2	0.1	78.6	0.2	81.9	0.1	111.9	1.5

Table C.24: Average response time data from Figure 4.1.

α	1-Thread		8-Thread		16-Thread		Disk-Oriented	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.05	88.0	0.0	94.4	0.4	94.2	0.4	102.4	0.7
0.23	105.0	0.0	117.4	0.4	122.4	0.4	131.4	0.7
0.49	121.0	0.0	133.0	0.0	136.6	0.4	152.0	0.6
0.74	138.2	0.4	148.8	0.7	151.8	0.7	177.8	1.3
1.00	157.8	0.4	167.2	0.4	170.4	0.4	223.6	3.0

Table C.25: 90th percentile response time data from Figure 4.1.

α	40+40 Buffers		40+80 Buffers		40+120 Buffers		40+500 Buffers	
	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf
0.05	297.6	13.6	248.0	9.9	229.0	2.1	232.2	0.7
0.23	248.0	1.7	230.2	0.7	228.2	1.4	236.4	0.7
0.49	520.6	2.4	465.8	5.1	448.8	3.1	439.6	5.7
0.74	1025.2	14.4	914.4	14.8	895.4	11.8	917.0	12.7
1.00	1971.6	34.5	1806.4	47.4	1757.8	32.0	1978.6	44.8

Table C.26: Reconstruction time data from Figure 4.2a.

α	40+40 Buffers		40+80 Buffers		40+120 Buffers		40+500 Buffers	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.05	50.7	0.2	51.1	0.3	51.6	0.2	51.4	0.1
0.23	66.8	0.3	67.6	0.4	67.6	0.4	66.9	0.3
0.49	76.5	0.2	77.8	0.4	78.1	0.3	79.5	0.4
0.74	87.9	0.3	89.7	0.4	90.1	0.6	93.9	0.7
1.00	107.5	1.2	110.7	1.5	111.9	1.5	123.0	1.0

Table C.27: Average response time data from Figure 4.2b.

α	40+40 Buffers		40+80 Buffers		40+120 Buffers		40+500 Buffers	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.05	100.8	0.4	102.0	0.6	102.4	0.7	102.0	0.0
0.23	130.0	0.6	131.6	0.9	131.4	0.7	130.6	0.4
0.49	149.8	0.7	151.2	0.9	152.0	0.6	154.8	1.2
0.74	174.0	1.0	176.8	0.9	177.8	1.3	186.6	1.5
1.00	215.4	2.5	221.2	3.1	223.6	3.0	250.2	2.2

Table C.28: 90th percentile response time data from Figure 4.2b.

α	RPW 000		RPW 100		RPW 010		RPW 001		RPW 111	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	26.5	0.1	26.1	0.0	26.5	0.1	26.4	0.0	26.0	0.1
0.05	51.9	0.3	49.9	0.1	51.1	0.2	51.4	0.1	49.7	0.1
0.23	68.5	0.3	60.6	0.2	66.0	0.2	67.9	0.2	59.0	0.1
0.49	78.1	0.4	69.4	0.2	78.4	0.2	77.8	0.3	68.5	0.2
0.74	91.0	0.9	77.7	0.7	91.2	0.7	90.5	0.7	78.3	0.5
1.00	112.2	1.6	90.5	1.4	113.6	2.1	113.7	1.2	91.8	4.1

Table C.29: Average response time data from Figure 4.4.

α	RPW 000		RPW 100		RPW 010		RPW 001		RPW 111	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	40.0	0.0	39.2	0.4	40.0	0.0	40.0	0.0	39.2	0.4
0.05	103.2	0.7	100.0	0.0	101.6	0.4	102.0	0.6	99.8	0.4
0.23	132.8	0.9	120.2	0.4	130.0	0.6	131.8	0.7	118.6	0.4
0.49	152.0	0.6	135.8	0.7	153.4	0.4	151.0	0.8	135.4	0.4
0.74	179.6	2.1	152.0	1.3	180.4	1.6	178.4	1.7	153.8	1.1
1.00	224.4	3.6	179.6	3.2	228.0	4.5	227.4	2.9	183.0	9.0

Table C.30: 90th percentile response time data from Figure 4.4.

α	RPW 000		RPW 100		RPW 010		RPW 001		RPW 111	
	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf
0.03	220.0	4.4	278.8	5.7	220.0	4.4	232.0	3.4	292.4	3.3
0.05	218.8	2.9	334.8	8.6	288.2	3.4	233.0	4.8	457.0	7.1
0.23	218.6	0.9	331.0	2.1	283.4	1.6	228.6	1.2	453.6	2.4
0.49	445.8	3.5	418.8	6.1	444.2	4.2	442.8	6.5	469.8	2.9
0.74	911.6	9.3	648.4	8.5	909.0	14.2	894.0	9.0	651.8	9.3
1.00	1801.6	43.9	1065.8	20.5	1838.2	53.5	1805.2	17.7	1060.0	62.8

Table C.31: Reconstruction time data from Figure 4.5.

α	Monitored		Constant		None	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	26.4	0.0	26.1	0.0	26.5	0.1
0.05	51.7	0.2	49.9	0.1	51.9	0.3
0.23	68.1	0.2	60.6	0.2	68.5	0.3
0.49	73.4	0.4	69.4	0.2	78.1	0.4
0.74	79.9	0.7	77.7	0.7	91.0	0.9
1.00	91.0	1.5	90.5	1.4	112.2	1.6

Table C.32: Average response time data from Figure 4.6.

α	Monitored		Constant		None	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	40.0	0.0	39.2	0.4	40.0	0.0
0.05	102.6	0.7	100.0	0.0	103.2	0.7
0.23	132.0	0.0	120.2	0.4	132.8	0.9
0.49	142.8	1.1	135.8	0.7	152.0	0.6
0.74	156.6	1.3	152.0	1.3	179.6	2.1
1.00	180.6	3.2	179.6	3.2	224.4	3.6

Table C.33: 90th percentile response time data from Figure 4.6.

α	Monitored		Constant		None	
	Recon Time	95% Conf	Recon Time	95% Conf	Recon Time	95% Conf
0.03	220.6	3.5	278.8	5.7	220.0	4.4
0.05	218.4	2.4	334.8	8.6	218.8	2.9
0.23	219.0	0.8	331.0	2.1	218.6	0.9
0.49	400.6	5.8	418.8	6.1	445.8	3.5
0.74	673.8	7.2	648.4	8.5	911.6	9.3
1.00	1077.2	23.6	1065.8	20.5	1801.6	43.9

Table C.34: Reconstruction time data from Figure 4.7.

α	No Following		Following	
	Recon Time	95% Conf	Recon Time	95% Conf
0.05	215.2	0.4	461.0	0.0
0.23	217.4	0.9	437.0	1.0
0.49	442.0	2.1	479.6	5.2
0.74	896.0	17.7	915.4	6.5
1.00	1808.4	45.4	1839.4	31.7

Table C.35: Reconstruction time data from Figure 4.8a.

α	Average				90th Percentile			
	No Following		Following		No Following		Following	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.05	52.0	0.0	48.7	0.0	103.4	0.7	97.0	0.0
0.23	68.2	0.0	63.1	0.0	132.4	0.9	124.6	0.4
0.49	78.8	0.0	78.7	0.0	152.8	0.7	152.2	0.9
0.74	90.9	0.0	89.7	0.0	178.8	1.8	176.2	0.9
1.00	113.1	0.0	114.1	0.0	226.4	2.9	228.0	3.2

Table C.36: Response time data from Figure 4.8b.

Access Time	Surviving		Replacement	
	No Following	Following	No Following	Following
14	0	2	1	0
15	0	2	0	0
16	189	14	11364	144
17	64	18	177	115
18	448	40	935	140
19	102	35	71	126
20	245	56	53	219
21	45	44	35	226
22	65	51	31	224
23	28	84	46	448
24	105	73	38	311
25	44	152	48	564
26	207	109	45	448
27	60	145	61	694
28	207	101	31	511
29	83	163	39	726
30	54	142	35	563
31	85	172	54	803
32	69	131	26	595
33	80	131	17	604
34	116	183	23	777
35	66	123	13	600
36	112	145	15	678
37	65	120	14	500
38	92	125	21	591
39	65	117	8	433
40	75	123	6	494
41	47	81	8	279
42	47	81	16	358
43	45	55	5	210
44	25	57	5	182
45	33	48	7	210
46	25	39	1	113
47	29	34	5	148
48	12	17	1	72
49	15	22	3	78
50	4	11	1	27
51	3	7	1	27
52	0	2	0	11
53	2	3	0	3

Table C.37: Access time histograms from Figure 4.9.

α	Dedicated		Distributed	
	Recon Time	95% Conf	Recon Time	95% Conf
0.03	277.6	6.4	34.6	0.4
0.05	233.0	4.8	65.8	0.9
0.08	231.8	3.7	79.0	1.0
0.13	256.2	20.9	122.8	3.1
0.23	228.6	1.2	208.0	2.4
0.49	442.8	6.5	471.6	3.5
0.74	894.0	9.0	938.4	14.7
0.97	1677.2	43.8	1723.8	32.1

Table C.38: Reconstruction time data from Figure 5.6a.

α	Dedicated				Distributed			
	90th percentile		Average		90th percentile		Average	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	58.2	0.4	34.7	0.1	75.6	0.7	49.4	0.4
0.05	102.0	0.6	51.4	0.1	121.0	1.0	63.5	0.4
0.08	111.6	0.4	55.8	0.1	126.8	1.1	65.7	0.5
0.13	117.0	1.1	58.9	0.7	129.4	0.7	67.2	0.4
0.23	131.8	0.7	67.9	0.2	135.0	0.0	70.0	0.2
0.49	151.0	0.8	77.8	0.3	151.8	0.4	78.3	0.1
0.74	178.4	1.7	90.5	0.7	178.8	1.6	91.0	0.6
0.97	219.8	4.7	109.9	2.0	219.0	3.4	110.2	1.4

Table C.39: Response time data from Figure 5.6b.

C	Recon Time		Avg Resp Time		90% Resp Time	
	Time	95% Conf	Time	95% Conf	Time	95% Conf
10	385.2	4.3	130.8	0.7	69.1	0.3
20	173.8	2.8	127.6	0.9	66.6	0.2
40	79.0	1.0	126.8	1.1	65.7	0.5
61	56.6	1.1	128.6	0.4	66.6	0.3
79	44.8	0.4	127.4	0.7	65.8	0.4
97	36.4	0.7	128.2	0.7	66.2	0.3
151	24.2	0.4	129.2	0.4	66.6	0.3

Table C.40: Data from Figure 5.7.

C	5-year		10-year	
	Fail Prob	95% Conf	Fail Prob	95% Conf
10	1.9e-05	2.0e-07	3.8e-05	4.1e-07
20	3.6e-05	5.6e-07	7.2e-05	1.1e-06
40	6.7e-05	7.2e-07	1.3e-04	1.4e-06
61	1.1e-04	1.7e-06	2.3e-04	3.4e-06
79	1.5e-04	9.0e-07	3.0e-04	1.8e-06
97	1.9e-04	2.8e-06	3.7e-04	5.5e-06
151	3.0e-04	4.7e-06	6.0e-04	9.3e-06

Table C.41: Reliability data from Figure 5.8.

Access Size	Reconfig, G=3		Reconfig, G=30		Fault-Free, G=3		Fault-Free, G=30	
	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf	Xfer Rate	95% Conf
2^2	0.17	0.00	0.17	0.00	0.17	0.00	0.17	0.00
2^3	0.31	0.00	0.30	0.00	0.31	0.00	0.30	0.00
2^4	0.52	0.00	0.52	0.01	0.51	0.01	0.51	0.01
2^5	0.88	0.01	0.88	0.01	0.87	0.01	0.87	0.02
2^6	1.60	0.01	1.61	0.03	1.59	0.01	1.61	0.01
2^7	2.92	0.06	3.00	0.05	2.93	0.04	3.02	0.02
2^8	5.01	0.04	5.80	0.11	5.08	0.08	5.87	0.09
2^9	7.65	0.10	10.61	0.22	7.88	0.12	11.07	0.17
2^{10}	11.88	0.17	14.73	0.15	12.12	0.17	15.21	0.23
2^{11}	16.35	0.17	20.51	0.43	17.19	0.35	21.75	0.24
2^{12}	20.42	0.26	24.89	0.32	21.15	0.24	26.15	0.28

Table C.42: Transfer rate data from Figure 5.9.

α	Redirection Off		Redirection On	
	Recon Time	95% Conf	Recon Time	95% Conf
0.03	34.2	0.7	34.4	0.7
0.05	64.6	0.4	65.0	1.6
0.23	205.2	1.8	191.8	0.9
0.49	473.2	2.6	393.6	1.6
0.74	943.4	10.7	683.4	6.3
0.97	1731.6	18.9	1083.4	19.7

Table C.43: Reconstruction time data from Figure 5.11a.

α	Average				90th Percentile			
	Redirection Off		Redirection On		Redirection Off		Redirection On	
	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf	Resp Time	95% Conf
0.03	48.1	0.3	48.0	0.2	73.4	0.4	73.0	0.6
0.05	62.9	0.3	62.9	0.3	120.2	0.7	120.6	0.9
0.23	69.4	0.2	67.2	0.2	133.8	0.7	129.6	0.4
0.49	77.6	0.1	71.7	0.2	150.4	0.4	139.0	0.6
0.74	89.6	0.6	78.3	0.5	176.0	1.6	152.6	1.2
0.97	106.3	0.9	87.9	0.9	210.2	2.1	172.6	1.9

Table C.44: Response time data from Figure 5.11b.