UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Improving disk array reliability and performance

Permalink

https://escholarship.org/uc/item/1kt0t7cg

Author

LeFevre, Jeffrey P.

Publication Date

2009

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Improving Disk Array Reliability and Performance

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Jeffrey P. LeFevre

Committee in charge:

Professor Walt Burkhard, Chair
Professor Amin Vahdat
Professor Geoffrey M. Voelker

2009

Copyright

Jeffrey P. LeFevre, 2009

All rights reserved.

The thesis of Jeffrey P. LeFevre is approved and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2009

TABLE OF CONTENTS

Table of Contents iv

List of Figures vi

List of Tables ix

Acknowledgements x

ABSTRACT OF THE THESIS xi

Chapter 1. Introduction 1
1.1. RAID Architecture 2
1.2. Additional RAID Configurations 3
1.3. RAID in Current Enterprise Disk Arrays 5
1.4. Disk Drive Design 5
1.5. Other Methods for Improving Performance 7
1.5.1. Additional Mirroring Systems 7
1.5.2. Active Reorganization Methods for Improving Performance . . 8

Chapter 2. The RAID1nr Data Layout
and Scheduling Policies 9
2.1. Physical Disk Model 10
2.2. Components of a Disk Access 11
2.3. The Data Layout for RAID1nr 12
2.4. Design of the RAID1nr Mapping Function 14
2.5. Layout Discussion 15
2.6. An Example Mapping Function for Fault-Free Mode 16
2.7. The RAID1nr Mapping Function 18
2.7.1. Scheduling Policies for Reads 19
2.7.2. Scheduling Policies for Writes 21
2.7.3. Load Balancing 22

Chapter 3. Implementation within RAIDframe 23
3.1. RAIDframe Design 23
3.2. Request Processing in the RAIDframe Simulator 24
3.3. RAID1nr Implementation 26
3.3.1. Read Scheduling 26
3.3.2. Load Balancing 30
3.3.3. Write Scheduling 33
3.4. RAID1nr Consistency in RAIDframe 34
3.4.1. Creating the Minimum Set and Remaining Set 34

3.5. Implementing Immediate and Eventual Consistency 36
3.5.1. User-observed Behavior 38
3.5.2. Integrity Check and Failure Recovery 39
3.5.3. Read Scheduler Modifications 39
3.6. Design of Primary and Delayed Write DAGs 40
3.7. Motivation 43
3.8. Determining M and Creating DAGs in Degraded Mode 43
3.8.1. Minimum Set and Remaining Set Write Implementation 45

Chapter 4. Experimental Results 55
4.1. Experimental Setup 55
4.2. Read Performance 57
4.2.1. Response Time 57
4.2.2. Throughput in IOs per Second 58
4.2.3. Seek Time 61
4.3. Write Performance 62
4.4. Mixed Workload 66
4.5. Load Balancing 70
4.6. Degraded Mode 72

Chapter 5. Discussion and Conclusion 76

Appendix A. Appendix 78

References 80

LIST OF FIGURES

Figure 2.1. Graphical representation of physical disk. 11
Figure 2.2. Example mapping of stripe units in a RAID1 redundancy group, showing groupings by chunk number. 13
Figure 2.3. Example mapping of stripe units in a RAID1nr redundancy group, showing groupings by chunk number. A different chunk is rotated onto the outer edge of each of n disks. 15
Figure 2.4. Example mapping of stripe units in a RAID1nr redundancy group with an alternative stripe unit layout where consecutive stripe units map to separate disks. 16

Figure 3.1. Example read and write DAGs for RAID1nr with n disks. 3.1a reads the stripe unit from one disk, but we may choose any disk in the redundancy group since all have identical copies. In 3.1b we must write D0 to all n disks in the redundancy group. 25
Figure 3.2. The primary read service area (lighter) for each disk is restricted to those stripe units that are nearest the disk's outer edge. 29
Figure 3.3. Read service area (lighter) for each remaining disk when Disk C has failed. 29
Figure 3.4. Read service area (lighter) for each remaining disk when Disk C has failed, using shortest queue load balancing policy. Each surviving disk serves the entire address space. 31
Figure 3.5. Read service area (lighter) for each remaining disk when Disk C has failed, using the preferred mirror load balancing policy. Each surviving disk serves an equal portion of the entire address space. 32
Figure 3.6. Read service area (lighter) for each remaining disk when Disk B and Disk C have failed, using the preferred mirror load balancing policy. In 3.6a the single surviving disk must serve the entire address space, but in 3.6b all surviving disks serve equal portions of the entire address space. 32
Figure 3.7. Disks in set M (lighter) for 3- and 4- disk RAID1nr redundancy groups with $|M| = 2$, for stripe unit D0. In (a) $M = \{DiskA, DiskC\}$, in (b) $M = \{DiskA, DiskD\}$ 35
Figure 3.8. Disks in set M (lighter) for 3- and 4- disk RAID1nr redundancy groups with $|M| = 2$, for stripe unit D4. 36
Figure 3.9. Write DAGs for the minimum set M and remaining set R. 41
Figure 3.10. Disks in set M (lighter) for stripe unit D0 in a 3-disk RAID1nr redundancy group with $|M| = 2$. 42
Figure 3.11. Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$. 42
Figure 3.12. Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$ and Disk D has failed. $M = \{DiskA, DiskC\}$, and $R = \{DiskB\}$. 44

Figure 3.13. Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$ and Disks C and D have failed, leaving $M = \{DiskA, DiskB\}$, and $R = \emptyset$. There are no disks in R since we always attempt to meet $|M|$ first. 45
Figure 3.14. The primary read service area (lighter) for each disk is restricted to those stripe units that are nearest the disk's outer edge. For writes, when $|M| = 2, n = 3$ the primary service area is expanded in order to write 2 copies of each stripe unit. 49
Figure 3.15. For $M = 2$, 3.15a shows the active service areas (lighter) for Disks A, C in NORMAL state, and Disk B in UPDATING state. 3.15b shows the service area (lighter) when all disks are in UPDATING state. 51

Figure 4.1. Response time for reads maintaining a constant workload of 10 users per disk. 57
Figure 4.2. IOPS for reads maintaining a constant workload of 10 users per disk. Improvement percentages are shown in Table 4.2. 58
Figure 4.3. IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1. 59
Figure 4.4. IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1. 60
Figure 4.5. IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1. 60
Figure 4.6. Average seek time for 64KB reads. 61
Figure 4.7. Seek time and Response time for 100% write workload with 10 users. 63
Figure 4.8. Seek distance from chunk 0 (C0) to chunk 1 (C1) for a 4 disk RAID1 and RAID1nr configuration. In (b) Disk B has a long seek due to the rotated layout. 64
Figure 4.9. Seek distance from chunk 1 (C1) to chunk 2 (C2) for a 4 disk RAID1 and RAID1nr configuration. In (b) Disk C has a long seek due to the rotated layout. 64
Figure 4.10. Response time for 64KB writes with exactly 10 users over 2-6 disks for Raid1 and Raid1nr using Method 1 (no batching) with $M = 1$ and $M = 2$ (a), and Method 2 (batching) with $M = 2$ with various batch sizes (b). 66
Figure 4.11. Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users on 2-6 disks showing Raid1 and Raid1nr with $M = \{1, 2\}$. . . . 67
Figure 4.12. Response time improvement percentage of Raid1nr configurations over a 2-6 disk Raid1 baseline, taken from figure 4.11. 67
Figure 4.13. Response time for 64KB mixed workload (80/20 read/write) on 3 disks with 1-10 users showing Raid1 and Raid1nr with $M = \{1, 2\}$. 68
Figure 4.14. Response time improvement percentage of Raid1nr configurations over Raid1 baseline, taken from figure 4.13. 69
Figure 4.15. Response time improvement percentage of several 3-disk configurations compared to a 2-disk Raid 1 baseline. Showing a 64KB mixed workload (80/20 read/write) on 3 disks for 1, 10, 100, and 1000 users. 70

Figure 4.16. Skew injected workload with load balancing. Response time for exactly 10 users on 2-6 disks with an 80/20 mixed workload, including Method 1 with $M = 2$ and load balancing (LB). 71
Figure 4.17. Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users, showing 0-2 disk disk failures, including Method 1 with $M = 2$ and load balancing (LB). 3 disk configurations. 72
Figure 4.18. Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users, showing 0-4 disk failures, including Method 1 with $M = 2$ and load balancing (LB). 5 disk configurations. 73

Figure A.1. IOs per second, 64KB reads, 10 - 1000 users. Each request is for a single stripe unit, of size 64KB. 78
Figure A.2. IOs per second, 1024KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1. Each request is for a single stripe unit, of size 1024KB. 79

LIST OF TABLES

Table 1.1. Commonly available RAID configurations for selected current enterprise storage arrays. 6

Table 2.1. Major components of disk access for several disk drives. Performance times in ms, and data transfer time for 64KB using published maximum sustained transfer rate. 12

Table 4.1. Response time improvement of RAID1nr over RAID 1, taken from figure 4.1. 57
Table 4.2. Throughput improvement (IOPS) of RAID1nr over RAID 1, taken from figure 4.2. 59
Table 4.3. Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1 configuration with disk 0 failed. 74
Table 4.4. Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1nr configuration with disk 0 failed. 74
Table 4.5. Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1nr configuration with disk 0 failed, using load balancing. 75

ACKNOWLEDGEMENTS

I would very much like to thank everyone who has contributed valuable input to this work, especially my primary advisor Walt Burkhard for his continued guidance and support.

ABSTRACT OF THE THESIS

Improving Disk Array Reliability and Performance

by

Jeffrey P. LeFevre

Master of Science in Computer Science

University of California, San Diego, 2009

Professor Walt Burkhard, Chair

In this work we present a new data layout and associated scheduling policies to improve RAID reliability and performance. Our implementation uses multiple mirrors, utilizing n disks in a redundancy group thus providing fault tolerance for $n - 1$ disk failures. As this is an extended form of RAID 1, we refer to this as RAID1nr, where n is the number of mirrors and r indicates the position of the data is rotated on each mirror. The rotated layout is such that a different $\frac{1}{n}$ of the data is located on the outer edge of each disk. The redundancy scheme is simple mirroring thus there is no added complexity introduced such as parity or other redundant encoding techniques. We then provide several scheduling policies for reads and writes that take advantage of the data layout. These policies can be set by the administrator for the desired level of performance and reliability. For example, all read requests for a particular block range may be serviced from the same disk. Writes may be scheduled to a subset of disks in order to improve performance using our immediate and eventual consistency policies. We also present load balancing policies for skewed workloads. While the RAID1nr system supports up

to $n - 1$ mirrors, we show that even adding a single extra mirror provides the increased reliability offered by an extra copy of the data, as well as a significant performance increase for read workloads, write workloads, mixed workloads, workloads with skewed distributions, and during degraded mode operation.

Chapter 1

Introduction

Redundant Arrays of Inexpensive Disks (RAID) provide encoding methods that offer several possibilities for tolerating disk failures as well as techniques for improving disk performance by paralleling and overlapping IOs to achieve higher throughput. (Patterson et al., 1988; Chen et al., 1994; Burkhard and Menon, 1993; Blaum et al., 1995; Alvarez et al., 1997). In this work we present an implementation of RAID that uses multiple mirrors, consisting of n disks in a redundancy group thus providing fault tolerance for $n - 1$ disk failures. As this is an extended form of RAID 1, we refer to this as RAID1nr, where n is the number of mirrors and r indicates the position of the data is rotated on each mirror. The rotated layout is such that a different $\frac{1}{n}$ of the data is located on the outer edge of each disk. The redundancy scheme is simple mirroring thus there is no added complexity introduced such as parity or other redundant encoding techniques. We then provide several scheduling policies for reads and writes that take advantage of the data layout. These policies can be set by the administrator as required for the workload and desired level of performance and reliability. While the RAID1nr system allows for up to $n - 1$ mirrors, we show that even adding a single extra mirror offers a significant performance increase for several types of workloads, along with the increased reliability offered by an extra copy of the data.

1.1 RAID Architecture

RAID disk arrays provide for various levels of fault tolerance and performance, and may be created using software or hardware controllers. An array failure is defined as single or multiple disk drive failure(s) leading to unrecoverable data loss. RAID arrays appear as a single logical address space to the operating system and the RAID controller handles the translation of OS requests to physical disk blocks in the array. Typically, larger storage arrays can be purchased in a cabinet/rack-style configuration with many disks and multiple array controllers that have redundant connections. The RAID controller maps the data onto groups of disks and schedules the IO requests based on the data layout while also computing and storing any parity calculations depending on the encoding system used. RAID systems divide the logical address space into smaller pieces called stripe units which contain the file data or parity information and are spread across the available disks. The stripe unit is the primary unit of data storage in a RAID array. While raw disk blocks contain 512 bytes of usable space, the stripe unit size is configurable by varying the number of disk blocks per stripe. By distributing stripe units across the disks, RAID configurations parallelize data transfer to increase performance. Data is encoded for redundancy by grouping one or more data stripe units and one or more parity stripe units together into a single "stripe". For a RAID 1 configuration, a stripe will consist of a data stripe unit which contains the data and its associated parity stripe unit which contains an exact copy of the data. This configuration is also called data mirroring. A RAID 5 stripe configuration will include several data stripe units together with a parity stripe unit, which is an XOR of the data stripe units. For both configurations, each stripe unit is stored on a separate disk. If any disk fails, the data is recoverable through reading the remaining units in the stripe and rebuilding the lost data.

The RAID controller is responsible for storing or retrieving the encoded data and parity for all read and write requests made by the operating system. Disk failures are handled by the RAID controller which can rebuild a failed disk's data from the

remaining disks while the array remains in operation. Many deployed systems will have an extra disk (or more) which is used by the controller to store the rebuilt data without user intervention. Depending on the size of the failed disk and the encoding scheme for parity stripes, rebuilding time can vary. For instance, a 300GB disk with a sustained (maximum) transfer rate of 100MB/sec will take at least 50 minutes to rebuild in an idle array, and over 100 minutes for 50MB/sec transfer rate. More likely, this will take several hours even in an idle array, where idleness is itself unlikely in a production-deployed system. This is known as recovery time. During this time window, a second disk failure can lead to data loss, depending on the RAID configuration. Also, the failed disk would need to be removed and replaced manually in order to keep a spare drive in the system, and again during this time window a RAID disk group may only tolerate a single failure. Any system that includes a spare stand-by drive has the added cost of that drive and/or an on-call administrator, and is susceptible to data loss during the recovery phase and spare drive replacement time.

1.2 Additional RAID Configurations

For high reliability needs, several RAID configurations have been introduced that can tolerate more than one disk failure. RAID 6 is an extension of RAID 5 that uses 2 different redundancy computations such as Reed-Solomon ECC, each value is a stripe unit. Each value is calculated and then the full stripe (data and parity units) is stored across all drives in a distributed fashion. RAID 6 can tolerate 2 disk failures, and the storage cost overhead is 2 extra drives per group. The additional IO overhead for computing and storing two separate redundancy units has a negative performance impact on data writes similar to, but worse than RAID 5. This impact is due to the need to keep the redundancy units updated with any new write. If a write occurs over less than an entire data stripe, one or more data and redundancy units in the original stripe must first be read and then the updated redundancy unit and new data unit(s) must all be written to their corresponding disks. The need for reading previous data before writing

new data is known as the small write problem which slows down the performance of RAID 5 and RAID 6.

RAID 10 (or 1+0) is a system which can tolerate up to two disk failures (in some cases) by combining 2 independent RAID 1 physical systems into a logical mirror pair. This results in essentially 2 groups of 2 disks, where each group contains a mirrored pair and can tolerate a single failure, but there cannot be two failures within the same group. This restriction is severe, as otherwise the system is only single failure tolerant.

Other disk array architectures have been introduced that can tolerate multiple disk failures. Systems such as (Holland and Gibson, 1992) and (Alvarez et al., 1998) use declustered parity in order to spread out the workload more evenly during disk failures and reconstruction (called degraded mode) so that a single disk does not become a bottleneck. Improving the reconstruction time reduces the window for a second failure leading to data loss in RAID architectures that can only tolerate a single failure. These techniques increase the amount of parity overhead in the system by using balanced incomplete or complete block designs. DATUM (Alvarez et al., 1997) is a declustered architecture that achieves a more even distribution of parity stripes which allows for graceful degradation during multiple disk failure. The PRIME (Alvarez et al., 1998) architecture provides almost optimal large read operations, requiring at most one extra disk read and has similar degraded performance. Both systems use the optimally minimum length of parity codes for the level of fault tolerance. The PDDL (Schwarz et al., 1999) architecture assumes a spare disk on stand-by and rather than keeping the spare empty, incorporates the amount of spare space into its data layout and is able to use the spare disk for data storage while distributing the spare space among all drives in the configuration. These architectures are all susceptible to the small write problem, and additionally finite field multiplication is required, which introduces some complexity for the parity computations.

EVENODD (Blaum et al., 1995) is a system which uses simple XOR parity encoding and two extra disks in order to tolerate up to two simultaneous disk failures. The number of data disks m, is chosen to be a small prime number and parity is computed

along a row, and then also diagonally across columns with the resulting parity necessarily being either even or odd. This is an extension of the RAID 4 architecture with one extra parity disk added, however the parity is distributed across all disks and so this architecture suffers from the small write problem as well.

$\hat{B}$ (Theodorides and Burkhard, 2006) is a system that uses MDS codes for computing and distributing parity, and provides $n - 2$ fault tolerance. Distributing optimal length redundant data across disks achieves better balance and performance for write updates and also during reconstruction in degraded mode. This system shows only a small performance impact for few failures and under a high number of failures performance degrades gracefully. Performance is shown to be much better than RAID 5 in single failure mode.

1.3 RAID in Current Enterprise Disk Arrays

Many current storage solutions offer only a subset of the these RAID configurations. RAID 1 and RAID 5 are widely used and in general, RAID 1 is typically selected for higher performance, while RAID 5 is selected for lower storage overhead cost. Both provide single disk failure tolerance, while RAID 6 is a typical option for 2 disk failure tolerance. Commonly available RAID configurations for selected current enterprise disk arrays are shown in Table 1.1. (IBM, 2009; Data Domain, 2009; Dell, 2009; NetApp, 2009; EMC, 2009; LSI, 2009) None of these disk array systems offer more that 2 fault tolerance, and additionally require at least one hot stand-by drive, thus have a hidden storage overhead cost in the spare disks.

1.4 Disk Drive Design

Because disk drives are mechanical devices containing spinning platters and moving heads, there is a minimal mechanical time required for each disk access. Accessing data first requires a movement of the head/arm assembly over the correct track,

Table 1.1: Commonly available RAID configurations for selected current enterprise storage arrays.

Mfg.	**Model**	**RAID configurations**
Data Domain	DDX Series	6
Dell	Enterprise Storage Array	5, 6, 10, 50
EMC	Symmetrix	0, 1, 5, 6, 10
IBM	DS8000	5, 10
LSI Logic	Engenio 7900	0, 1, 3, 5, 6 ,10
NetApp	FAS6000	RAID-DP (6)

this is the seek time. Once the heads are in position they must wait for the correct block to rotate under the head, this is the rotational latency time. Seek time and rotational latency are a function of the type of drive, enterprise vs. personal storage, and rotation speed. Disk drives have a request queue and disk scheduler which collect and reorder IO requests for optimal performance. Their goal is to minimize mechanical service time overhead and typical algorithms used are shortest seek time first (SSTF) and shortest positioning time first (SPTF). When accessing the data, the speed of the data transfer also depends on these same factors along with the areal density. Another factor affecting data transfer time is the location of the data. Because areal density is constant across the entire platter and the rotation speed is fixed, more data will pass under the head in a single revolution on the outer circumference of the drive than the inner. This effect is part of the design in zoned bit recording (ZBR). ZBR results in higher data transfer rates for blocks on the outer edges of the disk platters, and it has been observed that performance varies across the drive in this way (VanMeter, 1997) (Talagala et al., 2000) (Ghandeharizadeh et al., 1996). Our physical disk model is discussed further in Chapter 2. For additional information on disk drive design and modeling we refer the reader to (Ruemmler and Wilkes, 1994), (Anderson et al., 2003), (Anderson, 2003) and (Seagate, 2008).

1.5 Other Methods for Improving Performance

Methods for minimizing seek distance and improving physical locality to take advantage of higher transfer rates have been introduced. A common technique to improve database performance is "short-stroking" the disk. This refers to placing data only on the outer-most portion of the disk platters, while leaving the center area empty. Short-stroking has the benefit of reduced seek time since no full-stroke seeks are ever required, and also improved data transfer rates since all data is located only on the outer edge. A disk might be short-stroked such that it only uses the the first 40-75 % of the physical blocks on each disk (Storage Networking Industry Association, 2009) (Seagate, 2004b) and the remaining space is wasted. This administrative optimization is done by simply reducing the size of the primary partition on each disk when the partition is first created. Performance gains are stated to be up to $2\times$ in some cases, but at the cost of wasting disk space on every drive in the system. In (Schlosser et al., 2005), the system exposes the physical disk layout to the file system and application layer in order to allocate data sequentially, and to preserve the spatial locality of multidimensional data by placing it on adjacent tracks.

1.5.1 Additional Mirroring Systems

Distorted mirrors (Solworth and Orji, 1991) presents a logically but not physically identical mirror disk that provides write-anywhere semantics thus reducing seek distances and improving the performance of small writes. High performance disk mirroring (Morris, 2003) describes a pair of disks with mirrored data, and modifies the physical placement of data on each disk. The logical address space is divided into two sections. Section one is placed on the outer-most cylinders of one disk in the pair, and the inner-most cylinders of the other disk in the pair. Section two follows similarly. Though developed independently, our system is most similar to this work, however this work only considers two disks. Fractured mirrors (Ramamurthy et al., 2003) is presented for database systems whereby each disk in the mirrored pair stores the same data

but uses a different data storage model for the tables on each disk. Depending on the type of query, it is routed to the disk with the table layout that will likely perform the best.

1.5.2 Active Reorganization Methods for Improving Performance

Other methods attempt to actively reorganize data over time for optimal placement to match access patterns in order to reduce disk service times. Heat-based clustering systems such as ALIS (Hsu et al., 2005) and (Staelin and Garcia-Molina, 1990) attempt to improve read workloads by periodically relocating active files on disk to improve locality, reduce seek time, and minimize rotational latency. Other reorganization schemes such as initial contiguous layout, disk defragmentation, or file and block reorganization (Akyrek and Salem, 1995) move single entities (files, blocks, cylinders) but do not consider access patterns. Microsoft Windows XP and Vista (Microsoft, 2001) (Microsoft, 2004) operating systems also monitor file access patterns. Periodically, frequently accessed files are automatically relocated to the outer region of the disk storing them contiguously to reduce seek time and increase read throughput both by storing files sequentially and using the outer edge of the disk. This is maintained using the WINDOWS/Prefetch/Layout.ini file.

Dynamic reorganization schemes require additional, somewhat complex management software or layout-aware applications to monitor access patterns and relocate data along with dynamically redirecting accesses to current physical disk locations. The RAID1nr system requires no additional software layer or encoding methods, offers the fault tolerance of $n - 1$ disks, and moreover our scheduling policies increase both the read performance and write performance as more disks are added to a redundancy group.

Chapter 2

The RAID1nr Data Layout and Scheduling Policies

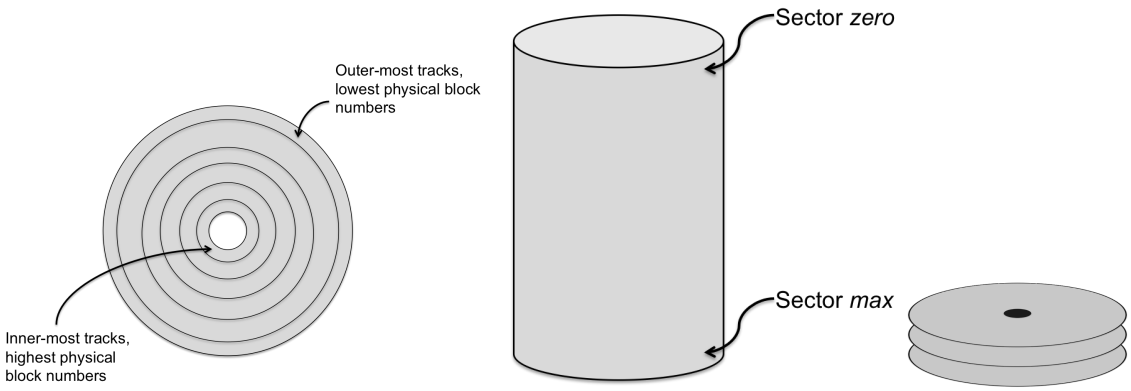
The RAID1nr system presents a data layout and associated scheduling policies which can provide both increased fault tolerance and increased IO performance for disk arrays. The system is based on a RAID1 configuration, which is normally composed of mirrored pairs of disks. Each disk in a RAID1 mirrored pair contains identical data. A standard RAID1 system provides fault tolerance for a single disk failure within a mirrored pair, or redundancy group. The RAID1nr system extends the number of disks in a redundancy group to n. Since all disks contain the same data, this provides for fault tolerance of $n-1$ disk failures within the redundancy group. The RAID1nr system then rotates the physical position of the data on each disk such that each stripe unit is located at a different physical offset on each disk in the redundancy group. This rotation of the data position allows us to take advantage of the mechanical characteristics of modern disk drives by designing scheduling policies that are aware of the rotated data layout. Read and write requests are scheduled such that each disk primarily services a particular range of the logical address space. These policies allow us to maintain all n copies of the data while providing fast response times for reads and writes. Additionally we present methods for load balancing uneven workloads so that all disks may bear an equal amount

of work when the array is in degraded mode or there are temporary hot-spots in the data request stream.

2.1 Physical Disk Model

Sectors appear on disk as 512 byte addressable blocks, numbered sequentially from block 0 to `max_block_number`. The `max_block_number` is equal to the addressable size of the disk divided by the sector size. The physical layout of the sectors on disk start with sector 0 on the outside edge, and then sequential sectors follow in a circular fashion around the disk platter. The sectors that comprise one full revolution are collectively called a track. Tracks form concentric circles and are numbered from the outer edge toward the center of the disk, so track number zero will be along the outer-most edge as shown in figure 2.1a. A cylinder describes the tracks that are aligned vertically when multiple platters are stacked in a disk housing (figure 2.1c). Modern disks may have more than one platter, and data may be stored on both surfaces of each platter (Seagate, 2008). Cylinders are also numbered from outer (lowest) to inner (highest). In all of our figures, we abstract this physical disk model as top to bottom for simplicity. The top of the disk picture represents physical disk block number 0, which resides along the outer edge of the disk platter. The bottom of the disk picture represents the highest numbered disk block, which resides at the inner most portion of the platter, nearest the center. We show this depiction in figure 2.1, and note that disk sector and disk block are synonymous. We design our system using this model of the physical disk block layout.

Figure 2.1 shows our representation of a disk drive for the purposes of this paper. Figure 2.1a shows a single disk platter layout illustrating just a few tracks (concentric circles) and physical block ordering. Blocks (sectors) are numbered sequentially from outer to inner. Following this, we use the convention of top to represent `sector_zero` which is actually on the outer edge of the platter, to bottom `sector_max` which is actually nearest the center or spindle of the disk drive as shown in 2.1b. Multiple platters



(a) Tracks on a platter (b) Sector number depiction (c) Stacked platters

Figure 2.1: Graphical representation of physical disk.

are stacked within the disk housing and in the center of the disks is the spindle as shown in 2.1c. Vertically aligned tracks from multiple platters form a cylinder. An offset into the disk is defined as a positive offset from `sector_zero`.

2.2 Components of a Disk Access

There are several components to a disk access to service a user request, divided into two distinct phases - positioning time and data transfer time. Because disk drives are mechanical, there are fixed costs to positioning the heads over the data. The main components of this cost are seek time and rotational latency. Additionally there may also be other fixed costs such as head switch time, track or cylinder switch time, and command processing which all have a much smaller but non-zero time component (Ruemmler and Wilkes, 1994), which is typically 0.2-0.3 ms for reads(IBM, 1999; Seagate, 2004a, 2008). After all of these mechanical positioning requirements are met, the second phase is the actual data transfer. The data transfer rate can depend on several factors including the bit density, rotational speed of the disk, and the track location (inner vs. outer). A few examples are shown in table 2.1. For smaller accesses the mechanical positioning components comprise a majority of the total access time. For larger accesses, the trans-

Table 2.1: Major components of disk access for several disk drives. Performance times in ms, and data transfer time for 64KB using published maximum sustained transfer rate.

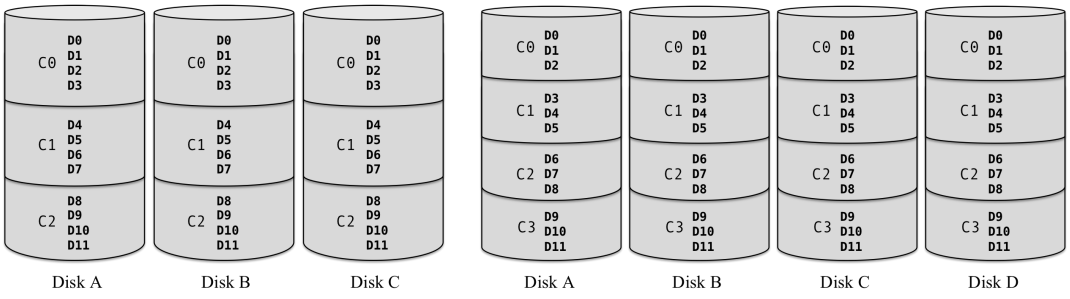
Model	RPM	Seek (read)	Rot.Latency	Xfer rate	Xfer time
IBM 18LZX	10k	4.9 ms	3.0 ms	44 MB/s	1.42 ms
Seagate 10k.7	10k	4.7 ms	3.0 ms	80 MB/s	0.78 ms
Seagate 15k.6	15k	3.5 ms	2.0 ms	125 MB/s	0.50 ms

fer time can dominate the total access time. It can be seen that for a 64KB disk access, the seek time and rotational latency required for positioning are much more costly than the transfer time. The transfer rate has improved significantly mainly due to increased bit density while the seek time has improved only slightly for the 10k disk drives, introduced in 1999 and 2004 respectively. The 15k drive (2008) has a lower seek time primarily due to smaller size platters (2.5 vs. 3.5 inch) and it only has a lower latency and higher transfer rate due to it's higher rotational speed.

2.3 The Data Layout for RAID1nr

RAID configurations address data in terms of stripe units, where a stripe unit consists of a group of logically sequential sectors. The logical address space is divided into equal-sized chunks or stripe units, whose size is set during initial array configuration. These stripe units may typically be 4KB, 8KB, 16KB, 32KB, etc. and are chosen by the system administrator. The stripe unit size is fixed for all disks within the same redundancy group. These logical stripe units are numbered sequentially and are then mapped to each disk. The map operation requires a logical to physical (stripe unit to disk block) address translation based on the RAID configuration. Each RAID configuration maintains its own unique mapping function that must be used to read or write data to the disk array. The result returned by the mapping function is a `disk_ number` and a `sector_number` (or physical offset) into the the disk.

We describe the mapping function over a single redundancy group of n disks, each containing the same data and has identical stripe units. A standard RAID1 group of n disks would map each stripe unit to the same physical offset on each disk. For a RAID1 configuration with n-way mirroring, we show the typical layout for consecutive stripe units in figure 2.2. The stripe units are numbered in order as Data Unit 0 (D0), Data Unit 1 (D1), etc. The number of stripe units per disk is a function of the disk size, we only show 12 stripe units here for simplicity. We also indicate a distinct grouping of consecutive stripe units we call chunks, which are themselves numbered Chunk 0 (C0), Chunk 1 (C1), etc.



(a) 3 disk RAID1

(b) 4 disk RAID1

Figure 2.2: Example mapping of stripe units in a RAID1 redundancy group, showing groupings by chunk number.

As seen in figure 2.2, stripe unit D1 is located at the same physical offset on each disk in the group. An n-disk RAID1 redundancy group will contain the logical address space of only 1 disk because each disk is a duplicate of the same address space. The storage overhead of the disks in the redundancy group is simply calculated as a percentage of the logical address space, shown in Equation 2.1:

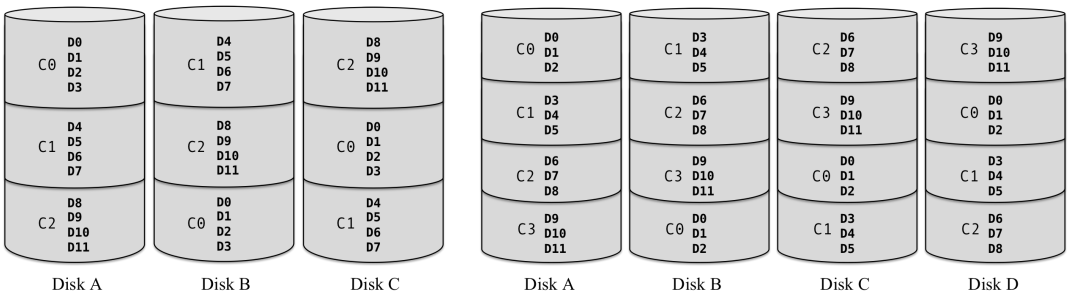
$$overhead = (n-1) \times 100\% \tag{2.1}$$

This storage overhead is costly at 100% per redundant disk added. However, mirroring provides the most benefit for transaction based systems which require high

availability and high performance. We present the performance increase for varying numbers of disks, users, and request sizes later in Chapter 4. The RAID1nr design target is high performance transactional database system (smaller request sizes) or an archival system, although we also show good performance for large, sequential reads and writes as well.

2.4 Design of the RAID1nr Mapping Function

While the logical address space and storage overheard of an n-disk RAID1nr configuration are the same as an n-disk RAID1 configuration, the RAID1nr mapping function will map each stripe unit to a different physical offset on each disk, in a rotated fashion. By storing a different set of stripe units nearest the outer edge of each disk in the redundancy group, our scheduler can take advantage of this layout to improve performance. The Raid1nr data layout divides the address space into n equal-sized chunks, and each chunk is stored on the outer edge of a different disk in the group. The total sectors of the redundancy group are equivalent to that of a single disk. We divide the total sectors by the stripe unit size, which gives us the number of stripe units. We then divide the number of stripe units by the number of disks, n. This creates n chunks of stripe units, each chunk with $\frac{1}{n}$ of the address space. An example RAID1nr configuration is shown in figure 2.3 for 12 stripe units and both 3- and 4-disk redundancy groups.



(a) 3 disk RAID1nr

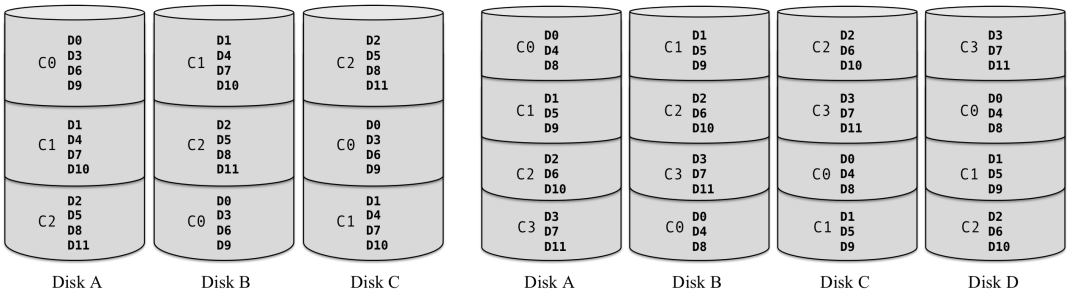
(b) 4 disk RAID1nr

Figure 2.3: Example mapping of stripe units in a RAID1nr redundancy group, showing groupings by chunk number. A different chunk is rotated onto the outer edge of each of n disks.

2.5 Layout Discussion

The RAID1nr data layout does not map consecutive stripe units to distinct disks. Most RAID configurations map consecutive stripe units to separate disks to gain parallelism when a user requests is composed of more than a single stripe unit. By reading several stripe units from multiple disks simultaneously, a large-sized user request can see a performance gain due to this parallelization of accesses. However, the RAID1nr data layout as described maps a group of consecutive stripe units (a chunk) to the same disk. The motivating factor for this decision is due to the mechanical nature of disks as described in 2.2. For smaller data access, a large part of the service time is due to mechanical positioning. Thus for a 64KB read request size, it may not make much sense to issue two 32KB accesses in parallel. While the data transfer time for each 32KB access will be exactly half as long as the 64KB access, both of the 32KB accesses will also incur exactly the same positioning time cost as the single 64KB access because those costs are fixed and much larger than the data transfer time for smaller accesses such as these. Because of this, a fair argument can be made that once the disk head arrives at the data position, it should transfer as much as possible. Our layout and scheduling policies attempt to access data on the outer edges of disks where the transfer rates are highest,

and this reasoning becomes even more relevant.



(a) 3 disk RAID1nr (alternative layout)

(b) 4 disk RAID1nr (alternative layout)

Figure 2.4: Example mapping of stripe units in a RAID1nr redundancy group with an alternative stripe unit layout where consecutive stripe units map to separate disks.

However, an alternative RAID1nr data layout that resembles a standard RAID stripe unit mapping is shown in figure 2.4. With this alternative layout, if a single user request touches two consecutive stripe units, they could be accessed in parallel on different disks. The alternative layout may be used for RAID1nr, but we do not use this for our tests and we refer to the layout of figure 2.3 for all future discussions in this thesis.

2.6 An Example Mapping Function for Fault-Free Mode

For illustrative purposes, we show a simple method to find the disk and offset for a stripe unit that maps it to the front of some disk in the redundancy group when the array is in fault-free mode. However, in the next section we show the more general method which is actually used by the `Map()` function and can be applied during fault-free or degraded mode. Equation 2.2 and 2.3 along with the following examples use divide and modulus operations which result in a disk number and stripe unit (*su*) offset. The `su_number` is the stripe unit number requested by the user, where each stripe unit in the redundancy group is numbered beginning with `zero`. A stripe unit is a grouping of consecutive logical blocks in the address space where the `su_size` is the size of a stripe unit, such as 4KB, 8KB, 16KB, etc. The `su_per_chunk` indicates the

number of stripe units contained in a chunk, and all chunks are the same size. The `user_block_address` is the logical block number requested by the user.

$$\textit{disk_number} = \texttt{su_number}/\texttt{su_per_chunk} \tag{2.2}$$

$$\textit{su_offset} = \texttt{su_number} \bmod \texttt{su_per_chunk} \tag{2.3}$$

The result generated is the disk where this stripe unit is nearest the front and the stripe unit offset into the disk. As an example, for data stripe unit 7 (D7) and with disks A, B, C, and D numbered as 0, 1, 2, 3, the computations for configurations in figures 2.3a and 2.3b are shown in the examples below.

From figure 2.3a using equations 2.2 and 2.3, a 3-disk redundancy group showing 4 stripe units (*su*) per chunk, D7 will map to the front of disk B, at an offset of 3 stripe units.:

```
disk_number = 7 / 4 = 1, or Disk B
su_offset = 7 mod 4 = 3
```

From figure 2.3b using equations 2.2 and 2.3, a 4-disk redundancy group showing 3 stripe units per chunk, D7 will map to the front of disk C, at an offset of 1 stripe units.:

```
disk_number = 7 / 3 = 2, or Disk C
su_offset = 7 mod 3 = 1
```

Now that the stripe unit mapping function is described, the entire process of mapping a user request from logical address to physical disk array address is as follows for the fault-free read case.

```
FaultFreeReadMap(user_block_address) {
/* Determine the stripe unit for the user request.*/
su_number = user_block_address / su_size

/* Determine the disk and stripe unit offset into
    the disk, based on the RAID1nr layout. */
disk_number = su_number / su_per_chunk
su_offset = su_number % su_per_chunk

/* Determine the physical block offset into
     the disk. */
block_offset = su_offset * su_size

return (disk_number, block_offset)
}
```

Thus, the fault-free read mapping function for each stripe unit requires 2 integer divisions, 1 modulus operation, and one integer multiplication. In the general case, we can compute the mapping function using even less complexity. Next we describe the actual `Map()` function used by RAID1nr.

2.7 The RAID1nr Mapping Function

A more general approach is used by the RAID1nr `Map()` function. This is used for both reads and writes in fault-free and degraded mode, regardless of how many disks have failed. `Map()` is called by the read and write scheduler which both loop over each non-dead disk and call `Map()` to compute the block offset for the user request. As implemented, the `Map()` function requires a few static values. During initial array configuration, the values `total_num_su` (total number of stripe units) and `su_per_chunk` (stripe units per chunk) are computed and stored as integer values.

The `Map()` function is very simple and takes two arguments as input, then computes the disk and the physical block offset for that disk. It does not consider if a disk is dead, this is done elsewhere.

```
Map(disk_number, user_block_address) {

 su_number = user_block_address / su_size
 su_offset = su_number + total_num_su
 su_offset = su_offset - (su_per_chunk * disk_number)
 su_offset = su_offset % total_num_su
 sector_number = su_offset * su_size
 return sector_number
}
```

The computation cost is minimal as it only requires one operation each of integer division, addition, subtraction, modulus, and multiplication. The computation for `su_per_chunk * disk_number` only needs to be computed once for each disk number, which can be done during array initialization and stored. For reads, we call `Map()` for each non-dead disk, and select the disk with the smallest block offset. For writes, we call `Map()` for all non-dead disks. The scheduling loops for read and write are described in more detail in Chapter 3. We note that if the user request composes more than one stripe unit, this process is repeated for each composite stripe unit.

2.7.1 Scheduling Policies for Reads

Based on this data layout, we can allow for several possible scheduling methods for reads and writes. Because reads may be retrieved from any disk in the redundancy group, we can choose among n disks to read from. There are several possible scheduling policies available to the user at run time that we have tested. One is random among all n disks. This policy performs slightly worse than the others, and has a higher variance in response times. Another is round-robin, which just schedules read requests to the next disk based on the previous request. Neither of these policies were very effective, so we do not discuss them further. The next scheduling policy selects the disk with the shortest queue length. The number of outstanding requests is known by the controller because the controller is responsible for all disk scheduling, and so there is no need to query the

disks for this information. The controller can just iterate though its outstanding counts for each disk. This policy is commonly used in RAID1. The next scheduling policy is called preferred mirror, which takes advantage of our data layout. We attempt to restrict the physical service area for each disk to only the outer edge area. As we have mentioned earlier, each disk has a different $\frac{1}{n}$ of the address space nearest its outer edge. By using our `Map()` function, we can determine the disk where the user request is nearest the outer edge, and our scheduler selects this disk to read from.

The preferred mirror scheduling policy results in each disk servicing a portion of the address space. The goal of this policy is to improve the response time for user requests. By servicing a smaller physical area, requests will require less mechanical seek distance, which can improve response time. Additionally, the transfer rate is higher for tracks along the outer edge of the disk than the inner, due in part to the constant angular velocity of the spinning platter and zoned bit recording (VanMeter, 1997). Because of these factors, during one revolution more sectors will pass under the mechanical heads on the outer tracks versus the inner tracks. Also since there are more sectors per track on the outer edge, less track boundaries will need to be crossed so there are potentially less track and head switches which both have a mechanical cost in time. By keeping requests more physically local to each other, this array scheduling policy also receives the most possible benefit from individual disk scheduling algorithms which reorder requests to reduce service time for the request stream. Disk drives will typically use some variant of shortest seek time first (SSTF) or shortest positioning time first (SPTF) to optimally order queued requests. Using several forms of tagged command queueing, many modern disk drives can store and reorder up to 32 requests in their internal request queues (T13 Technical Committee, 2004; Seagate, 2008). All of these factors help to reduce the service time per request and increase array throughput due to our layout and scheduling policies.

2.7.2 Scheduling Policies for Writes

For writes, we present several scheduling methods based on our data layout that can increase throughput. Since a write must go to all disks in the redundancy group, the response time is bound by the last disk to finish. Based on our preferred mirror scheduling policy for reads, it becomes clear that writes scheduled to the "preferred mirror" should also see improved performance. Our write scheduling policy allows the user to select the number of disks to be written synchronously, which we call the minimum set, denoted by M. The user selects a value for $|M|$, and we guarantee those disks will be written synchronously while the user waits. The other disks will be written asynchronously in the background within some interval. Disks in the set M are determined dynamically at the user request time. To determine the disks in M, we start with the preferred mirror, because that is the disk where the write should complete the fastest. The next disks we add to M are those where the user request is next nearest the outer edge. We expect these disks to be the ones where the write request will complete next-fastest. The remaining disks we call the remaining set, denoted by R, which will be those disks where the request is farthest from the outer edge, and thus outside of those disks' normal service area. We expect these disks to take longer to complete the write request than those in M. To facilitate this, we introduce *immediate* and *eventual* consistency for each write request. For a given request, those disks in M are made consistent synchronously, thus are immediately consistent. The disks in R are made consistent asynchronously within some interval, and are eventually consistent. The interval may be a time window, a count of outstanding requests, or until the next idle period. This consistency model and implementation are discussed in detail in Chapter 3. The goal of this policy is to improve write performance while still maintaining a minimum level of fault tolerance during the interval.

When writing a stripe unit, this policy create two distinct phases. A write is in the synchronous phase when it is being written to disks in M. It is in the asynchronous phase when it is being written to disks in R. Therefore, our write scheduling policies

result in a reliability scheme in which the fault tolerance for any stripe unit being written varies depending on the write phase. Because the write is not considered complete until the minimum set is synchronously updated, we describe the reliability after the first phase. Provided that $|M|$ disks have not failed, the user will have at least $|M|$ copies of the data physically written to stable storage after the synchronous phase. At that time, any stripe unit that is in the asynchronous phase waiting to be written can tolerate at least $|M|$ disk failures during the interval window, assuming $M < n$. This is due to the fact that even if all of the disks in M fail, an unwritten copy of the data is present either in the controller cache or in the physical disk cache of the disks in R. After the write has completed the asynchronous phase, n copies of the stripe unit have been updated and $n - 1$ disk failures can be tolerated without data loss.

2.7.3 Load Balancing

In a RAID1nr system, a load imbalance can occur in two situations. One is during a disk failure, the other is when some data is temporarily more popular than other data. If there is a disk failure, the preferred mirror scheduling policy will lead to an imbalanced workload distribution across the remaining disks in the redundancy group because all of the failed disk's workload will be scheduled onto an immediate neighboring disk. The other situation happens if user requests are not uniformly distributed over the logical address space due to a temporary "hot spot" in the data. To mitigate this, we introduce specific load balancing methods in Section 3.3.2, where the controller may offload a some of a disk's work to a neighboring disk based on their relative workloads. The load balancing policy attempts to preserve the preferred mirror scheduling concept as much as possible. To do this, it only considers the neighboring disk's workload where the request is next nearest the outer edge. Similarly, that disk may then get increasingly busy and may consider offloading some it's work to it's neighboring disk. This may help to maintain the intended goal of higher throughput per disk due to better physical locality even in the case of skewed workload distributions.

Chapter 3

Implementation within RAIDframe

3.1 RAIDframe Design

The RAIDframe simulator is designed for rapid prototyping of RAID configurations in disk array systems. It is made available by the Parallel Data Lab at Carnegie Mellon University (Gibson et al., 1996). RAIDframe provides a framework for simulating the performance of data layouts over groups of simulated disks. The simulator functions as an array controller with connections to all disks in the system. We have added our Raid1nr data layout and its associated scheduling policies to this framework for all of our simulated tests. Tests may be run with a configurable number and type of disks, stripe size, users, and request sizes and distributions. These values are all chosen at run time and are persistent throughout the entire test run. Test runs are specified for a finite time in seconds, and the simulator maintains an internal clock which is moved forward as new requests are generated and then completed. In simulator mode, individual users generate only a single request at a time, so the maximum number of in-flight requests at any time is equal to the number of users in the system.

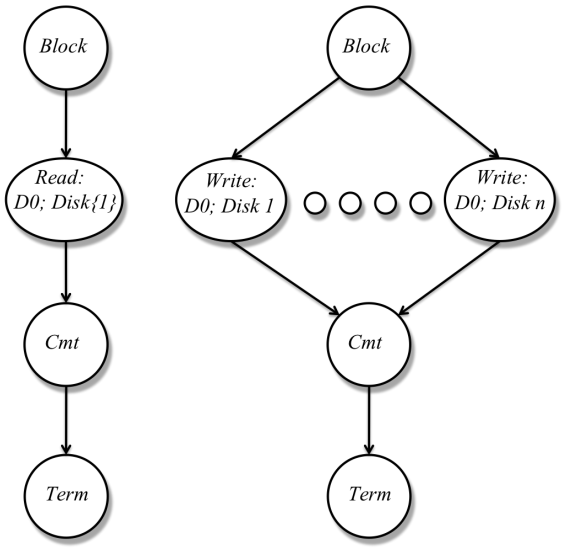
3.2 Request Processing in the RAIDframe Simulator

User requests are generated over the logical address space, and the layout policy determines the mapping of logical addresses to physical disk addresses. Data is separated into stripe units stored on disk according to the data layout for the particular RAID configuration being tested. Depending on the RAID configuration and the user request size, a single request may involve multiple stripe units, each of which may be retrieved from disks simultaneously in parallel. RAIDframe models requests as directed acyclic graphs (DAGs), where each node indicates a disk operation, redundancy computation, or a barrier. Two typical DAGs are shown in figure 3.1 Operations on stripe units become internal nodes in the graph. User requests start at the top with a source node, called *Block* (a no-op), which then connects to internal nodes representing stripe unit accesses. These internal nodes are marked as *Read* or *Write* operations on a disk, or parity calculations. During DAG execution by the controller, each of the internal nodes will fire and insert its action into the appropriate disk queues. All of these internal nodes are then connected to a Commit node designated by *(*Cmt) which blocks forward progress until all previous nodes complete. Optionally, next may be another series of Read/Write nodes with the same or different fan-out but that step is not required for RAID1nr. The Commit node is connected to another barrier node, Terminate (Term). After terminate, the user is notified the request has completed and the controller cleans up the DAG data structure.

The DAG is implemented as a finite state machine in RAIDframe, and it primarily has create, execute, and cleanup stages. States are incremented as the DAG is processed by the controller. During some states, the DAG may be suspended by the controller, such as when it is waiting to acquire stripe unit locks, or while it is waiting for all disks to complete during the execution state (Cmt node) since disks do not operate in lockstep. We take advantage of this when we suspend some DAGs temporarily in order to implement our write policies, which are discussed later in section 3.8.1. If a disk fails, DAG execution will fail and the DAG will be regenerated from the create state (Block

node) using the remaining disks. Re-creation of the internal nodes is attempted, until the number of failed disks exceeds the fault tolerance of the configuration. A RAID1nr DAG will be able to recreate internal nodes and complete as long as there is at least one disk remaining because we use a simple mirroring system.

The RAID1nr implementation creates a basic read DAG identical to a RAID1 read DAG, with Block, Read, Cmt, Term nodes. Reads are sent to a single disk unless we must obtain contiguous stripe units laying across chunk boundaries. A read and write DAG are shown in figure 3.1.



(a) Read DAG (b) Write DAG

Figure 3.1: Example read and write DAGs for RAID1nr with n disks. 3.1a reads the stripe unit from one disk, but we may choose any disk in the redundancy group since all have identical copies. In 3.1b we must write D0 to all n disks in the redundancy group.

For writes, the fan-out of the Write nodes is always equal to the number of disks in a RAID1nr configuration, as the stripe unit is mirrored on each disk in the redundancy group. Given a write request, the controller decomposes it into its component stripe units, maps each stripe unit to all disks in the redundancy group, and builds a DAG by creating a Write node for each disk as shown in figure 3.1b.

3.3 RAID1nr Implementation

We have made several additions to the simulator to support our RAID1nr configuration. Here we discuss the data layout addition followed by the read scheduling policies added. In the later sections 3.5 and 3.6, we discuss the write policies with additional queues and DAGs we have added and the new write scheduling policies that make use of them. The original RAID1 implementation in the simulator supported a single data copy and a single mirror copy. Our RAID1nr implementation extends this representation to a single data copy and $n-1$ mirror copies, where n is the total number of disks. For degraded mode, a RAID1nr configuration can tolerate $n-1$ faults, and it can operate in degraded mode until all disks fail or until the failed disk(s) are recovered. We have implemented our layout design as described in Chapter 2, which maps all logical addresses to physical addresses in the disk array. The map operation of our layout will convert a logical data address to a stripe unit number, and then map that stripe unit to a disk and a physical offset. At DAG node creation time, we call the `Map()` function for each internal node and include an operation, a disk number, and a physical offset into the node.

3.3.1 Read Scheduling

Referencing figure 3.1a, the internal Read nodes are where we offer several scheduling choices depending on the policy selected for reads. Since all n disks contain the same data, we can choose any disk from which to read a stripe unit. Based on the user request size, we determine the number of stripe units to be accessed, and create a Read node for each. We then build the Read node in several ways per user configuration and array state. First we convert the user address requested into its component stripe units. Next we determine the first `su_number` to be accessed, and the offset within that stripe unit. Then we create a DAG node to retrieve each stripe unit, and use `Map()` to fill in the (disk, offset) for each node based on the scheduling policy selected.

Shortest Queue Length Policy, Fault-Free Mode

This policy chooses the disk with the shortest queue length, which is the normal policy used for RAID1. For each stripe unit (`su_number`) requested, we call the Policy_Shortest_Q function and pass it the stripe unit. This function first checks all the disk queue lengths, selects the disk with the shortest length, then calls the Map function to determine the physical offset within that disk. This value is then installed in the DAG node. Since this all happens inside the array controller, it must have knowledge of the number of outstanding requests at each disk and does not need to query the disks themselves.

```
Policy_Shortest_Q(user_block_address)
min_queue_length = max_queue_length
For d in disks:
 if d.isDead(): /* not required for fault-free mode */
   continue
 if d.queue_length <= min_queue_length
   min_queue_length = d.queue_length
   disk_number = d
sector_number = Map(disk_number, user_block_address)
return disk_number, sector_number
```

The disk with the shortest queue length is chosen regardless of the data layout. This policy makes no attempt to consider the data layout when scheduling reads. The result is that a non-rotated layout and a rotated layout select the same disks to access and create identical DAGs with the only difference being the offset into the disk, since the stripe unit is at a different physical location on disk for each layout. The performance of this policy is identical for RAID1n and RAID1nr since all disks in the redundancy group contain the same data (although shifted around) and we are using the same scheduling policy.

Shortest Queue Length Policy, Degraded Mode

If a disk fails, then the disk array will continue to operate in degraded mode until zero disks are left in a redundancy group or all disks are recovered. The shortest

queue length policy will simply choose from among the remaining disks when creating the Read node. This is done in the mirror selection code for reads, where we check the queue lengths of all non-dead disks and select the shortest, as shown in the pseudo code above.

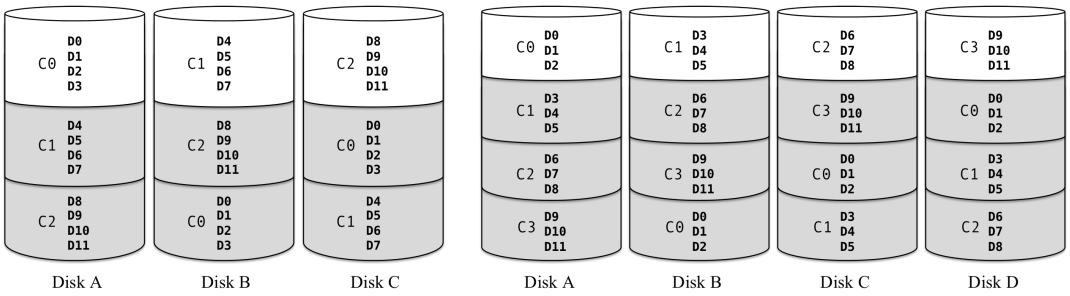
Preferred Mirror Policy, Fault-Free Mode

This Read scheduling policy applies knowledge of the data layout to select the disk from which to read the requested stripe unit. In a way, it follows the same logic as Policy_Shortest_Q, except here we are looking for the lowest disk sector number. A low disk sector will indicate that sector is nearest the outer edge, based on the sector layout format described in Chapter 2.

```
Policy_Preferred_Mirror(user_block_address)
min_sector_number = max_sector_number
For d in disks:
 if d.isDead():   /* not required for fault-free mode */
   continue
 sector_number = Map(d, user_block_address)
 if sector_number <= min_sector_number
   min_sector_number = sector_number
   preferred_disk = d
return preferred_disk, min_sector_number
```

Queue lengths are not tested here, as this policy just restricts the data range serviced by each disk, improving the physical locality of data requests. This policy results in a high throughput as each disk serves a smaller data range and thus physical area. It performs best when user requests are uniformly distributed, which results in each disk doing the same amount of work. We examine the case of non-uniform workloads due to disk failures or hot-spots later in section 3.3.2 “load balancing”. A read DAG using this policy will select the disk where the stripe unit resides nearest the outer edge and will not take queue lengths into consideration at all. Thus the read service area for each disk is specific, and restricted to stripe units on its outer-most sections as shown in

figure 3.2. For example, read requests for stripe units in chunk 0 will go to disk A, reads for stripe units in chunk 1 will go to disk B, etc.



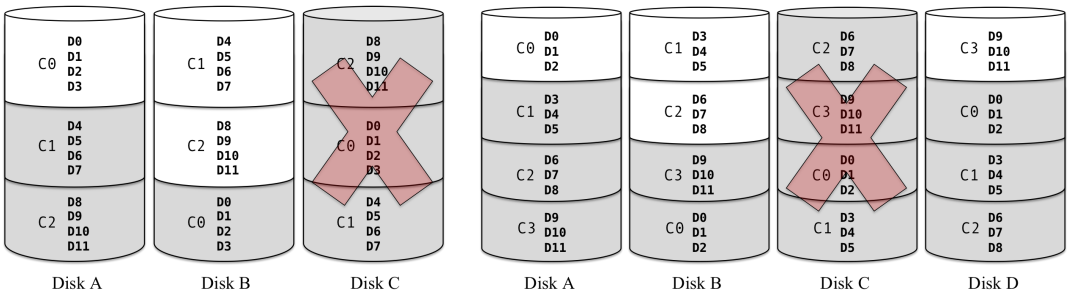
(a) 3 disk RAID1nr configuration

(b) 4 disk RAID1nr configuration

Figure 3.2: The primary read service area (lighter) for each disk is restricted to those stripe units that are nearest the disk's outer edge.

Preferred Mirror Policy, Degraded Mode

When considering scheduling read requests for a redundancy group with one or more disk failures, we can apply the same algorithm as in section 3.3.1. We compute the `disk_sector` for every non-dead disk, and again choose the lowest numbered sector. With one or more disk failures, this policy will result in an unbalanced workload distribution across the remaining disks. For instance, given disk C fails, and the RAID1nr layout in figure 3.2 is used, we show the remaining disks, and their corresponding service areas for reads:



(a) 3 disk RAID1nr configuration

(b) 4 disk RAID1nr configuration

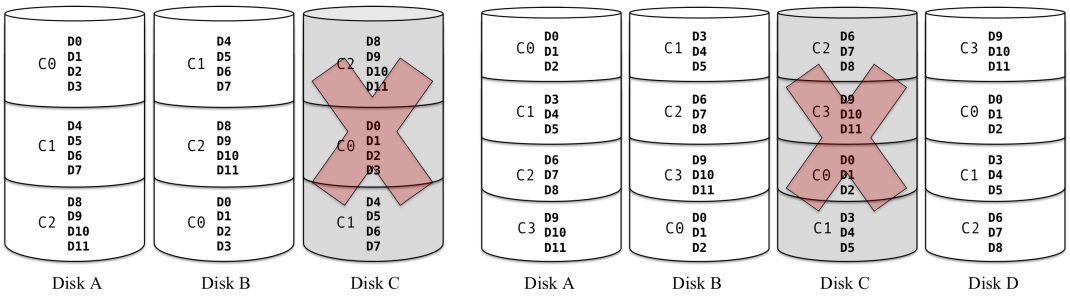
Figure 3.3: Read service area (lighter) for each remaining disk when Disk C has failed.

As shown in figure 3.3, Disk B increases its physical service area by $\frac{1}{n}$ which doubles its workload since it absorbs all of Disk C's read workload. The preferred mirror policy will schedule reads for chunk 2 (C2) to the disk where chunk 2 is next nearest the outer edge, causing a workload imbalance as shown. The preferred mirror policy will schedule the failed disk's workload on an immediate neighboring disk. The goal of this shifting is to preserve physical locality as much as possible by minimally increasing the service area of the remaining disks. However, this obviously causes an uneven share of the workload to be absorbed by the neighboring disk. All of the failed disk's read workload will be offloaded to its left-hand neighbor. In figure 3.3 the read service area for Disk B becomes larger by $\frac{1}{n}$ as indicated, which doubles its original workload. The resulting uneven workload distribution will create a high variance in response time for reads, as those scheduled for Disk B will take longer than those scheduled on the other remaining disk(s). We present two ways to mitigate this in the next section.

3.3.2 Load Balancing

Load balancing option is provided as an administrative option, in order to avoid bottlenecks in the system due to hot-spots or disk failures. In fault-free mode, the preferred mirror policy can actually create artificial hot spots under highly localized workloads because it schedules all reads for a given data range to be serviced by the same disk. During degraded mode, the rotated layout and preferred mirror scheduling policy cause the failed disk's workload to shift unevenly to an immediate neighboring disk as described in the previous section. We present two solutions in order to balance uneven workload distributions.

First we consider reverting to the default shortest queue policy during degraded mode. The read scheduler will just select the shortest queue length among all remaining disks. This expands each remaining disk's read service area to the full range of the disk as shown in figure 3.4.



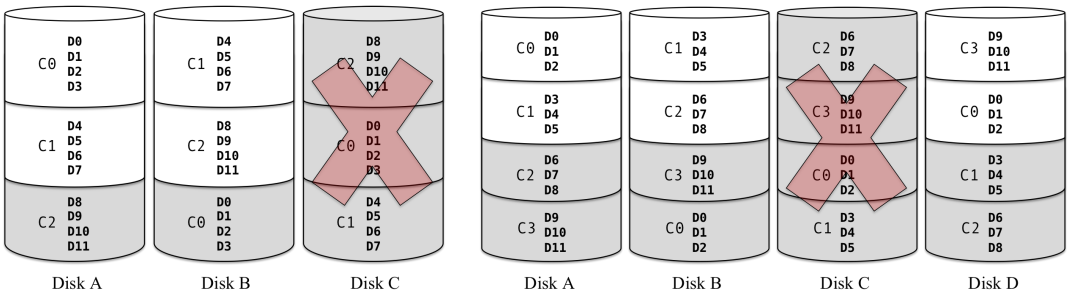
(a) 3 disk RAID1nr configuration

(b) 4 disk RAID1nr configuration

Figure 3.4: Read service area (lighter) for each remaining disk when Disk C has failed, using shortest queue load balancing policy. Each surviving disk serves the entire address space.

The performance will then be identical to a fault free disk array with shortest queue policy using fewer disks. This is because each disk has the same number of stripe units, although shifted around, and we simply choose the least busy disk. Given the workload is uniformly distributed, the amount of work done by each disk will be the same. We show this experimentally in Chapter 4. This method is simple and effective but because each disk services its entire area, it will incurr full-stroke seeks on each disk since it does not take advantage of our rotated layout or scheduling policies. Because full-stroke seeks are costly, throughput is reduced with this method.

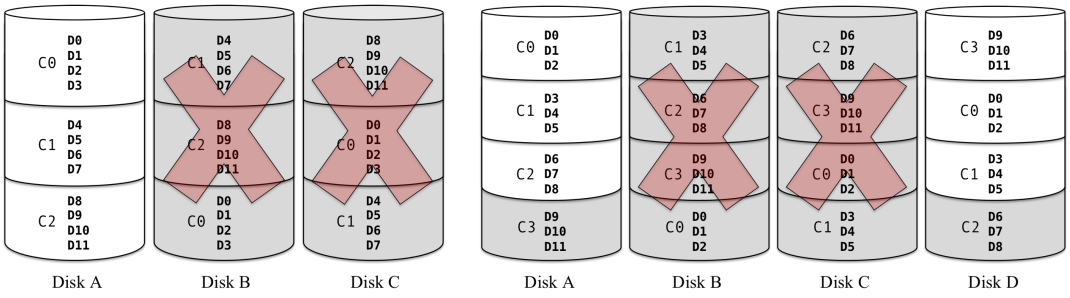
A second solution is a combination of preferred mirror with shortest queue. This allows us to minimally expand each disk's service area for reads, and attempts to eliminate full-stroke seeks. For a data stripe unit, we normally prefer to read it from the disk where it is nearest the outer edge. In the case of an uneven workload caused either by a hot-spot or a disk failure, we wish to schedule the read onto the disk where it is next nearest the outer edge. Each disk's new service area will still start at the outer edge but will go deeper into the center than it would during fault free mode. As long as more than one disk remains, this method avoids the full-stroke seeks that can happen with the shortest queue policy.



(a) 3 disk RAID1nr configuration (b) 4 disk RAID1nr configuration

Figure 3.5: Read service area (lighter) for each remaining disk when Disk C has failed, using the preferred mirror load balancing policy. Each surviving disk serves an equal portion of the entire address space.

During degraded mode, load balancing is turned on for all disks. This allows any disk to shift some of its workload to a neighboring disk if needed. A single read may be shifted by at most f disks, where f is the number of failed disks. Each remaining disk will now serve an extra $\frac{f}{n}$ of the address space in addition to its original $\frac{1}{n}$ during fault-free mode. Figures 3.5 and 3.6 show the expansion of the service area for each remaining disk is minimized as compared to figure 3.4 which uses the shortest queue policy.



(a) 3 disk RAID1nr configuration (b) 4 disk RAID1nr configuration

Figure 3.6: Read service area (lighter) for each remaining disk when Disk B and Disk C have failed, using the preferred mirror load balancing policy. In 3.6a the single surviving disk must serve the entire address space, but in 3.6b all surviving disks serve equal portions of the entire address space.

Additionally we use a scalar multiplier on the queue lengths of the disks being considered. If the ratio of queue lengths is above some threshold, we choose the shorter of the queue lengths. A value such as $\frac{2}{3}$ or $\frac{3}{4}$ might be considered for the scalar. This is a tunable parameter, whose purpose is to prevent one disk being overloaded with work. However, it is fractional because we still want to schedule a read to its preferred disk as much as possible to preserve the RAID1nr design goal of achieving higher throughput using a restricted service area. The same load balancing method can be used in fault-free mode, where we consider up to one neighboring disk for any user request. This allows shifting some requests for a hot spot to its nearest neighbor, which again distributes the workload while minimally expanding each disk's service area. If load balancing is turned on, it is always active and will respond immediately to uneven workload distributions, within the constraints of the scalar parameter.

3.3.3 Write Scheduling

The DAGs we have described for writes previously use what we have termed an *immediate consistency* model, meaning we write to all disks simultaneously while the user waits. This is the behavior in standard RAID1: both disks in the mirror pair are written to synchronously, guaranteeing two stable copies of each datum. Following this behavior, for an n-disk RAID1nr configuration, the immediate consistency model will issue one write update to each of n disks in parallel, the user waits for the DAG to complete, and the user is notified the write has been stored by the system. Thus, all mirrors are immediately consistent. As the number of disks in the redundancy group increases, waiting for every disk to complete the write operation can slow down performance observed by the user. To mitigate this we introduce another level of consistency, *eventual consistency*, which we describe next.

3.4 RAID1nr Consistency in RAIDframe

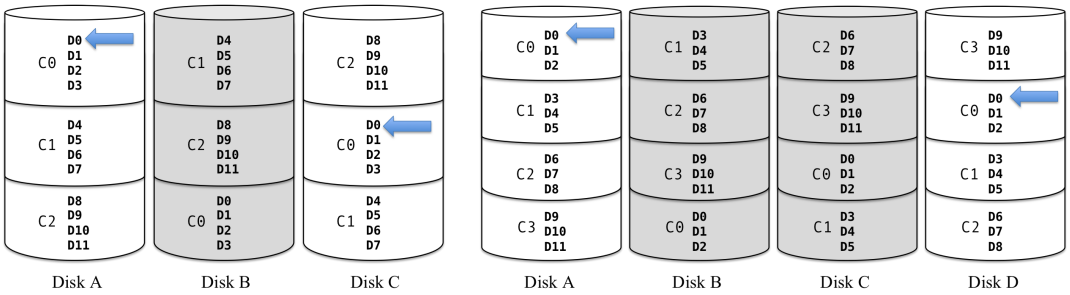
As described in Chapter 2, for an n-disk configuration, the IO performance of an individual write request is bound by the slowest response time of n disks. In an effort to improve performance in this situation, the RAID1nr system incorporates two models of consistency. The first is the *immediate consistency* model, where disks are written to synchronously. However, we only include some of the disks rather than all n mirrors. We call these disks the minimum set designated by M such that M is a subset of n disks in a RAID1nr redundancy group and $|M| > 0, |M| \leq n$. The remaining disks we call the remaining set, designated by R which has cardinality $n - |M|$ and we allow those to have *eventual consistency*, meaning they are written to asynchronously in the background and become consistent at a later time. Based on the required level of redundancy and performance, the administrator selects the cardinality of M at array configuration time. The RAID1nr system uses these two models of consistency together to improve performance while providing at least a minimum fault tolerance to the user.

3.4.1 Creating the Minimum Set and Remaining Set

When a write request is initiated and executed, it will return to the user as "completed" after it is written synchronously to $|M|$ of n disks (immediate consistency). The remaining disks, R, are issued asynchronous writes and become consistent at a later time (eventual consistency). The user sees an increased performance because the user no longer has to wait for the slowest of n disks to complete, she only has to wait for $|M|$ disks to complete. Notably, for a given stripe unit the scheduler will build the set M to consist of disks which will most likely be the fastest to complete, based on it's knowledge of the data layout. The minimum set M for a stripe unit consists of $|M|$ disks where that stripe unit physically resides nearest the outer edge. This is due to the rotated data layout - some disks will contain the stripe unit near the outer edge, and on other disks it will be nearest the center (spindle) as previously described and shown in figure 2.3. Because there may be one or more disk failures, the set M is dynamically

determined for every stripe unit at the time the write request is issued. The write scheduler uses a simple loop operation that compares the physical offset of the stripe unit for each non-dead disk in the redundancy group, and selects the $|M|$ disks with the lowest offset. This is exactly the same loop structure used by the read scheduler described in 3.3.1.

For example, given a 3-disk RAID1nr configuration with $n = 3, |M| = 2$, we determine the set M for stripe unit D0 by choosing the disks where D0 is nearest the outer edge as shown in figure 3.7.



(a) 3 disk RAID1nr

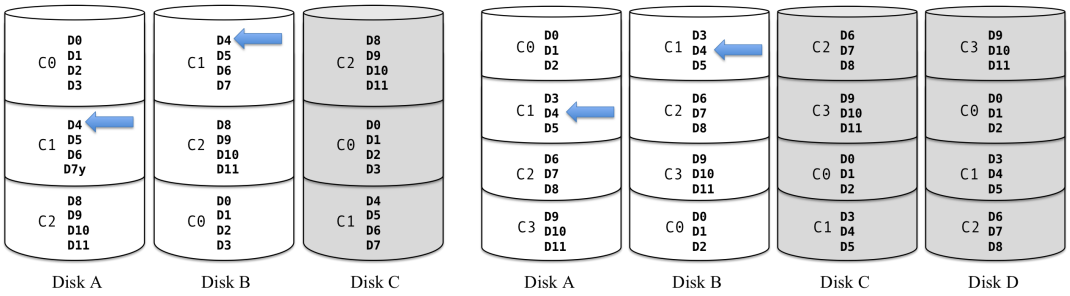
(b) 4 disk RAID1nr

Figure 3.7: Disks in set M (lighter) for 3- and 4- disk RAID1nr redundancy groups with $|M| = 2$, for stripe unit D0. In (a) $M = \{DiskA, DiskC\}$, in (b) $M = \{DiskA, DiskD\}$.

As indicated by the arrows, stripe unit D0 is nearest the front of disk A in 3.7a, and is next-nearest the front on disk C resulting in $M = \{DiskA, DiskC\}$, and $R = \{DiskB\}$. In the 4-disk example 3.7b stripe unit D0 is also nearest the front of disk A, and is next-nearest the front on disk D, resulting in $M = \{DiskA, DiskD\}$, and $R = \{DiskB, DiskC\}$.

To further illustrate that the disks in M depend on the specific stripe unit being considered, we show $M = \{diskA, diskB\}$ for stripe unit D4 in figure 3.8 to contrast with $M = \{diskA, diskC\}$ for stripe unit D0 in figure 3.7. These are examples of dynamically-determined consistency sets for a stripe unit. RAIDnr uses these policies when writing to any stripe unit, such that disks in M are immediately consistent, and

disks in R are eventually consistent.



(a) 3 disk RAID1nr

(b) 4 disk RAID1nr

Figure 3.8: Disks in set M (lighter) for 3- and 4- disk RAID1nr redundancy groups with $|M| = 2$, for stripe unit D4.

In 3.8a stripe unit D4 is nearest the front of disk B, and is next-nearest the front on disk A as indicated by the arrows, so $M = \{DiskA, DiskB\}$, and $R = \{DiskC\}$. The 4-disk example in 3.8b shows stripe unit D4 is nearest the front of disk A, and is next-nearest the front on disk B, so $M = \{DiskA, DiskB\}$, and $R = \{DiskC, DiskD\}$.

3.5 Implementing Immediate and Eventual Consistency

We have modified RAIDframe to produce up to 2 DAGs for each write operation. One DAG is generated for disks in M, and we call this the minimum set DAG, which we use to implement immediate consistency. The other DAG is generated for the remaining disks in R, and we call this the remaining set DAG, which is used to implement eventual consistency. This design creates an additional overhead in the RAID controller of maintaining 2 DAGs for each write. The additional overhead has two components. One is the small space and maintenance overhead of the extra DAG data structure. The second is the time the controller needs to maintain its copy of the actual write data. Even though we expect the minimum set DAG to complete quickly, the remaining set DAG will likely complete at a later time. To manage this, the controller needs to main-

tain its cached copy of the data to be written until both DAGs compete, or at least until it executes the remaining set DAG and sends the data and the write command to the appropriate disks.

When a write request is generated by the user, both DAGs are created dynamically whether in fault-free or degraded mode, in order to make a best effort to meet $|M|$ if there are any disk failures. The value of $|M|$ is always based on the configuration value supplied by the administrator at array initialization time. In normal RAIDframe operation the user generates a request and a single DAG is created, executed, and the user waits on the DAG to complete. However, because RAID1nr consistency allows the user to wait for only disks in the minimum set to complete, two separate DAGs are created. The minimum set DAG (for M) executes synchronously while the user waits as normal. The remaining set DAG (for R) executes asynchronously in the background and is unattached to the user, but is monitored as outstanding by the RAID controller.

We employ the standard mechanisms within RAIDframe for creating both DAGs, with two modifications. First we do not acquire locks on the stripe units for the remaining set DAG. Although this violates a standard consistency model, stripe unit locks are acquired for the minimum set DAG to ensure the user's view of the data is correct, just as it would be in standard RAIDframe operation. The system scheduler hides this process and the eventually consistent data from the user, and this policy allows both DAGs to be in-flight at the same time. The second modification is that the user only waits for the minimum set DAG to complete (immediate consistency), he does not wait on the remaining set DAG. As soon as the minimum set DAG finishes, the user is notified the write has completed and he is free to continue forward progress, which in simulator mode means the user can generate a new request. Because we build M dynamically whether or not the array is in fault free or degraded mode, this policy guarantees to the user that the system will always meet the configuration value for $|M|$ when possible. That is, if at least $|M|$ disks are not dead, the system will guarantee $|M|$ disks are immediately consistent when the user is notified the write has completed. If fewer than $|M|$ disks remain, the system will guarantee all non-dead disks are immediately consistent

when the the user is notified, because that is obviously the best it can do.

For the disks in the remaining set DAG (eventual consistency), the system also guarantees that it has stored the write data either in the controller cache or at the disks themselves, waiting in the disk's cache. We allow this remaining set DAG to complete at a later time, within some interval which is described in "Method 2" of Section 3.8.1. When the remaining set DAG does complete, only the controller is made aware of the completion since this is hidden from the user. The write completion time for the remaining set DAG does not get accumulated into the user's response time metrics, because this time is not visible to the user. At that point, all disks have become consistent for the write.

3.5.1 User-observed Behavior

Because our scheduler hides the eventually consistent disks from the user, the system always provides a consistent view of the data to the user: the read scheduling policies described next always provide the user with the latest copy of the data. Incoming reads for a stripe unit are served from disks which are in M for that given stripe unit. These are the disks that were made immediately consistent for that stripe unit and the write is on the physical platter. If for some reason the read were to be scheduled on a disk in R that is only eventually consistent, the read would be served from the controller's cache or that disk's cache, thus still returning the latest copy of the data to the user. This behavior is analogous to the standard behavior for any write-through or write-back caching system, where the minimum set can be thought of as write-through, and the remaining set can be thought of as write-back. Indeed, a typical user has no idea whether his write request has actually been stored on disk or not. The system cache or disk cache may be set to write-back mode, in which case the write data is only stored in cache, not on the actual disk. Generally, users have an implicit trust that their data has been permanently stored somewhere by the system when they write it. It is the responsibility of a system administrator to use either write-through caching, or write-back

caching with a battery back-up system for the disk arrays. A RAID1nr array would require such a battery back-up system in order for all disks to be consistent after a power failure. If a battery back-up is not used, a data integrity check functionality would need to be run upon array restart.

3.5.2 Integrity Check and Failure Recovery

Although not implemented within this thesis, the integrity check functionality can easily be added to RAID1nr. It would require a simple sequential scan of each stripe unit, comparing any disk in M to all disks in R for that stripe unit. This amounts to comparing the chunk at the front of each disk (any disk in M for those stripe units), to all disks which are not in M for that chunk, i.e. the disks in R for those stripe units. This would remedy the case where any disk in R was unable to write it's data before the power failed - unwritten data in the disk or controller cache was lost. Since read scans are sequential throughout a chunk, and since only a basic data comparison operation is required with no parity calculation, this integrity check / recovery procedure would run at nearly the full sequential scan bandwidth of the disks involved. Similarly, a complete rebuild operation would require sequentially reading the front chunk of each disk, and piecing those together to rebuild a spare disk (also sequentially) with current data. Essentially rebuild is a sequential copy operation where each disk contributes one chunk, except for the neighboring disk that contributes two chunks such as indicated by the active areas in figure 3.3b.

3.5.3 Read Scheduler Modifications

In order to maintain consistency from the user's perspective, we modify the Read scheduler slightly. When scheduling a read request for a given stripe unit, we only consider disks in M. Since we know any previous writes for this stripe unit were synchronously written to M and were immediately consistent, we are free to choose any disk in M to read from as these are all guaranteed to have the latest copy of the data.

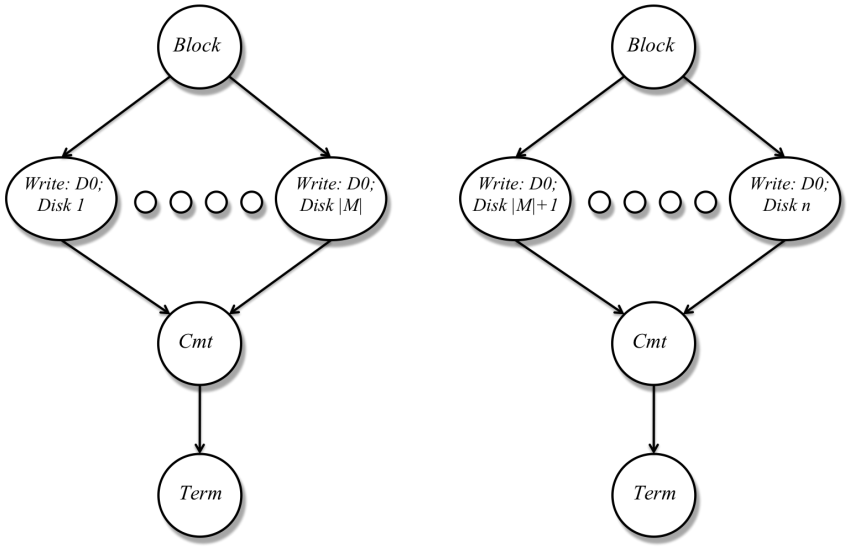
Reads are scheduled this way to avoid a read-after-write hazard on one of the disks not in M.

However, after a disk failure, a disk previously in R for a given stripe unit may now become part of M for that stripe unit. A potential consistency problem could arise since the read scheduler considers all disks in M to be consistent. Any in-progress writes on disks in R may not have been physically written yet when one of those disks becomes part of M for a given stripe unit. At that time, even though a disk may now be in M, the data may not be written on the physical platter. We offer three solutions for this issue. First, any incomplete write data will still be in the disk queue/cache and should be served from there. In RAID1nr, every disk (or the controller) has the latest copy of the data until written on all disks. We address this more specifically in section 3.8.1. Serving write data from the cache is the standard behavior of write-back cacheing available on modern disk drives. A second, stronger solution is to acquire stripe locks for the disks in R, but this is not done in our implementation due to simulator limitations. Although we create the M and R DAGs simultaneously, RAIDframe acquires stripe locks on a per-DAG basis within a single DAG's state machine. There is no current mechanism to acquire the locks in both DAGs, although this could be added to RAIDframe. A third solution is to require all disks to flush their queued write data when the controller detects a disk failure. There is a mechanism for this in the current ATA standard called the `FLUSH CACHE` (`0xE7h`) command (T13 Technical Committee, 2004) which will flush all data in the write cache to the physical media, and the disk will not accept new commands nor service queued reads until this is complete. Upon a disk failure the controller would send this signal to all surviving disks forcing them to become consistent immediately.

3.6 Design of Primary and Delayed Write DAGs

The minimum set DAG includes all operations on the disks in M. When $|M| = n$ all disks are updated synchronously and there is no remaining set DAG. The user waits while the minimum set DAG completes, and there is immediate consistency across all

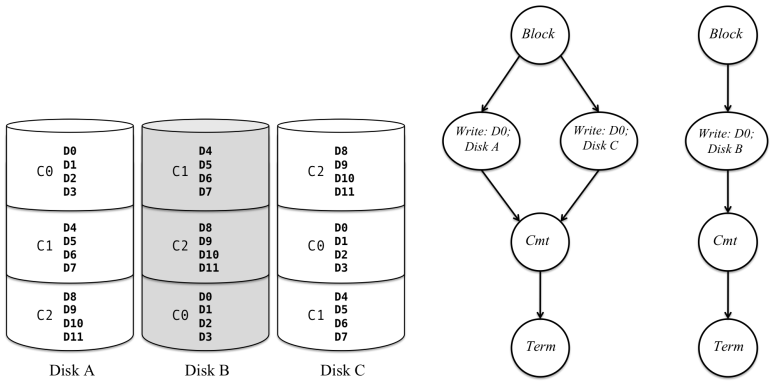
disks in the redundancy group. When $M < n$, both a minimum set DAG and a remaining set DAG must be generated. The minimum set DAG will include all disks in M while the remaining set DAG will include all disks in R. This is shown in figure 3.9.



(a) Write DAG for M (b) Write DAG for R

Figure 3.9: Write DAGs for the minimum set M and remaining set R.

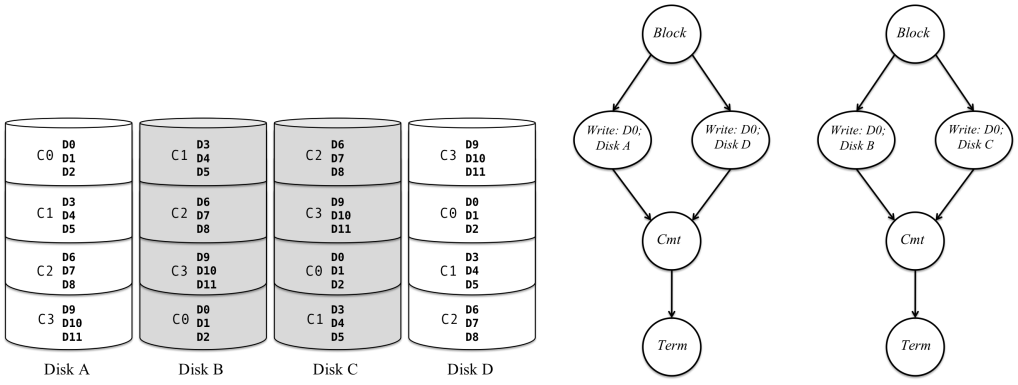
For figure 3.9a the number of write nodes is equal to $|M|$ since we must write to each disk in M. This DAG is used to implement immediate consistency. Figure 3.9b indicates the the number of write nodes is equal to $|R|$ since we must write to each disk in R. This DAG is used to implement eventual consistency. The number of disks in M is administrator configurable at array boot time, and may be any value from 1 to n. A more aggressive approach might set $|M| = 1$, and a very conservative approach would set $|M| = n$, with the trade off being better write performance observed by the user vs. higher immediate consistency and hence reliability. Specific DAG examples for $|M| = 2$ over both 3- and 4-disk configurations are shown in 3.10 and 3.11 . Examples for $|M| = 3$ over 3- and 4-disk configurations are shown in Appendix A



(a) 3-disk RAID1nr $|M| = 2$ (b) M DAG (c) R DAG

Figure 3.10: Disks in set M (lighter) for stripe unit D0 in a 3-disk RAID1nr redundancy group with $|M| = 2$.

In 3.10a $M = \{DiskA, DiskC\}$, and $R = \{DiskB\}$ because stripe unit D0 is nearest on the front of disk A and disk C, thus those are the disks selected for M. Figure 3.10b shows the minimum set DAG over M for figure 3.10a. Since $|M| = 2$ there are 2 write nodes. Figure 3.10c depicts the remaining set DAG over R for figure 3.10a. Since $|R| = 1$ there is 1 write node.



(a) 4-disk RAID1nr $|M| = 2$ (b) M DAG (c) R DAG

Figure 3.11: Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$.

In 3.11a $M = \{DiskA, DiskD\}$ because stripe unit D0 is nearest on the front

of disk A and disk D, and $R = \{DiskB, DiskC\}$. The minimum set DAG 3.11b over M has 2 write nodes since $|M| = 2$. Figure 3.11c depicts the remaining set DAG over R with 2 write nodes since $|R| = 2$.

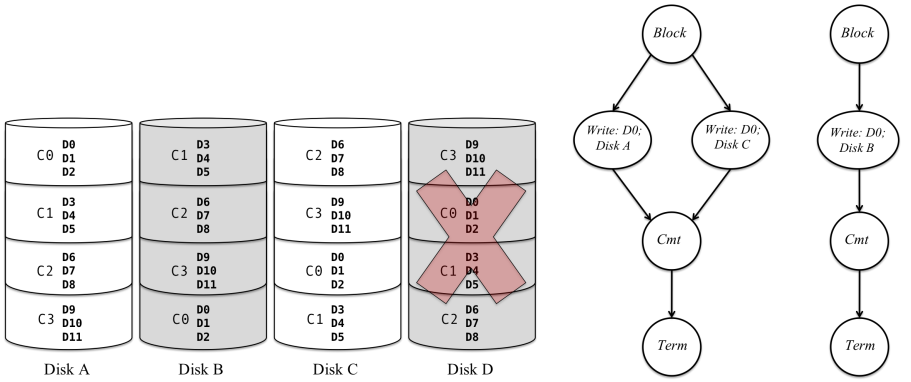
3.7 Motivation

Because the minimum set DAG only involves disks in M, the write will go to disks which have the stripe unit nearest their outer edge by design. As shown in 3.2, the read service area of each disk is already restricted to it's outer edge. When a disk receives a write from a minimum set DAG, that request will also be within that same service area or some minimally expanded service area (depending on n and $|M|$). Thus when writes are sent to disks in M where the stripe unit to be written is within or close to a disk's normal service area, we expect write request to complete faster than it would on disks in R. This should be the case if the disk scheduling algorithm is SSTF (or SPTF). This reasoning follows from the fact that disks in R will have the stripe unit farthest outside their normal service area. The user should see a relatively fast response time for his minimum set DAG, and the request will be considered completed from the point of view of the user.

3.8 Determining M and Creating DAGs in Degraded Mode

During degraded mode, M is determined by the same procedure describe previously in section 3.4.1. Because M is always determined dynamically at write request time, there is no special logic for degraded mode. That is, for a given stripe unit the disks in M will always be those non-dead disks where the stripe unit resides nearest the outer edge. Figure 3.12 shows the dynamic set M for stripe unit D0 in degraded mode when disk D has failed. We also show the resulting minimum set and remaining set DAGs. During degraded mode (just as fault-free mode), the number of Write nodes in each DAG is exactly the cardinality of M and R, less the failed disks. In this degraded

mode example, M is now $\{DiskA, DiskC\}$, and can be contrasted with the previous example in figure 3.11 where M was $\{DiskA, DiskD\}$ during fault-free mode. If a disk fails during DAG execution, the DAG will still complete for all non-failed disks as mentioned in 3.8.1.

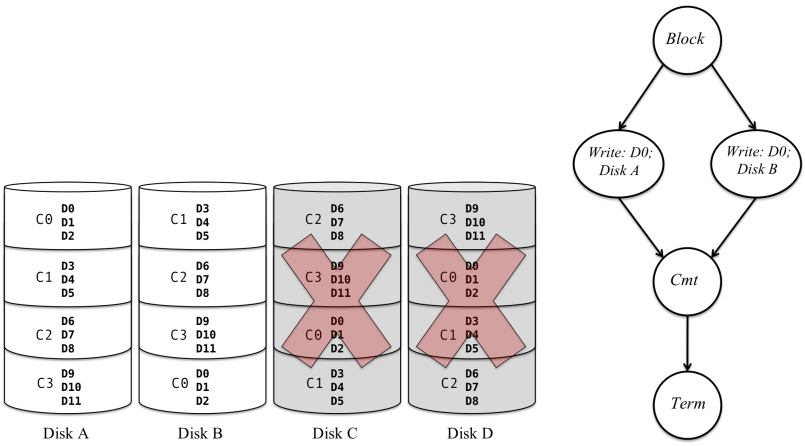


(a) 4-disk RAID1nr, 1 failure (b) M DAG (c) R DAG

Figure 3.12: Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$ and Disk D has failed. $M = \{DiskA, DiskC\}$, and $R = \{DiskB\}$.

Figure 3.12 shows $M = \{DiskA, DiskC\}$ now for stripe unit D0, since those are the non-dead disks where D0 is nearest the outer edge. This approach to determining M shows that disk B, the disk where D0 is farthest from the outer edge, remains in R. Determining M dynamically in this manner ensures that when writing a stripe unit, M excludes disks where the stripe unit is farthest from the outer edge whenever possible. The goal of this policy is to give the user the best response time possible for write operations by trying to avoid long seeks.

Similarly if disk C were to also fail, then the remaining disks A and B would be in M for stripe unit D0 as shown in 3.13. If the number of non-dead disks $\leq |M|$, there is no remaining set R and we do not need to generate a remaining set DAG.



(a) 4-disk RAID1nr, 2 failures (b) M DAG

Figure 3.13: Disks in set M (lighter) for stripe unit D0 in a 4-disk RAID1nr redundancy group with $|M| = 2$ and Disks C and D have failed, leaving $M = \{DiskA, DiskB\}$, and $R = \emptyset$. There are no disks in R since we always attempt to meet $|M|$ first.

3.8.1 Minimum Set and Remaining Set Write Implementation

We present 3 different methods for implementing these write DAGs. The first method is the most straightforward and executes both the minimum set and remaining set DAGs simultaneously, allowing them to insert their operations into all the disk queues for each DAG. We call this Method 1. Methods 2 and 3 will execute the minimum DAG immediately, and execute the remaining set DAG "later" within some administrator-defined interval as described below. Each disk services requests using some preset algorithm, such as elevator (SCAN/C-SCAN) or shortest seek time first (SSTF). For all methods, the user is notified as soon as the minimum set DAG finishes and may move forward with new requests while the remaining set DAG completes in the background.

Method 1

The motivation for Method 1 is the expectation that disks in M will complete a write request faster than those in R, since the request should be within or close-to their

normal service area as discussed in the motivation section 3.7. In this method, when a write request is initiated by the user both the minimum set and remaining set DAGs are created and executed simultaneously. During fault-free mode, this will insert the the write operations into all of the disk queues, since $|M| + |R| = n$. In degraded mode, the write operations for each DAG are inserted to all surviving disks. Since write requests are issued to all disks at the same time, every disk in the redundancy group will immediately have a copy of the data. Disks in the minimum set DAG begin servicing the write request as the user waits. At the same time, the disks in the remaining set DAG begin servicing the requests as well. We do not distinguish among which disks complete first, we must wait explicitly for the minimum set DAG to complete. Disks in R service the request as normal using the disk scheduling algorithm, but since the request is outside their normal service area by design, it is expected to take longer. When the minimum set DAG completes the user is notified the write is done. Simultaneously, the RAID controller is waiting on the remaining set DAG to finish and updating it's internal data structures, but the user is not notified when it does finish. While the disks in M have likely finished first and the user believes the write has completed, the write is likely still in-progress at each of the disks in R. The exact upper bound on completion time for the remaining set DAG to complete would be dependent on the starvation prevention mechanism of the disk scheduling algorithm for each disk in R. This method has shown good results which we present in Chapter 4.

Method 2

The motivation for method 2 is to take advantage of batch scheduling. Method 2 executes the minimum set DAG immediately just as in Method 1. However, Method 2 suspends the execution of all remaining set DAGs for some interval (discussed next). Eventually this results in a number of write requests waiting to be sent to each disk in the array. These writes are then batched together for a selected disk, and submitted to that disk all at once. Generally, batching is done to amortize the extra cost of mechanical disk

seek time when servicing individual requests in a random order. Batching IO requests together and re-ordering them for optimal performance is a widely practiced method for disk scheduling algorithms, as well as within the OS kernel. For example, by default Linux 2.6 uses what it calls an "anticipatory scheduler", which maintains 4 separate queues, and services reads and writes under a deadline using an elevator-type ordering (Bovet and Cesati, 2001). This OS queuing mechanism attempts to batch reads and writes to improve locality, which is coarsely defined as nearby (linear) block numbers. It gives reads service preference over writes, and each has a separate timeout that by default is set to 125 ms and 250 ms respectively. By issuing a batch of requests to the disk instead of a stream of individual requests, the policy takes advantage of the best scheduling algorithms present on modern disk drives. Because the disk will re-order a batch of requests to achieve the lowest overall service time, it will minimize seek time and rotational latency. These calculations are specific to the disk model, and the system does not need to be aware of them, it only needs to batch the requests before sending to disk, where the optimal ordering will be done.

In Method 2 the minimum set DAG is created and executed immediately while the user waits just as in Method 1. However, the remaining set DAG is implemented differently to accommodate batching. For Method 2 the remaining set DAG is created at request time, but is not executed immediately. It is suspended by the controller for some interval, during which it does not send those write requests to the corresponding disks. This interval may be a count of outstanding writes per disk, or a time window. We have currently implemented the interval as a count of outstanding writes per disk, and have varied the count from 0-100 for our tests, as presented in Chapter 4. These suspended DAGs reside in the controller along with their associated write data. At the end of the interval, the controller batches all the writes waiting for a given disk, and sends them to that disk all at once.

To do this we have implemented a simple batching scheme during the interval. If the interval is zero, Method 2 does not collect writes but immediately forwards them along to the corresponding disks involved in the remaining set DAG. A zero interval

results in effectively the same behavior as Method 1. If the interval is one or more, Method 2 allows all remaining set DAGs and their data to collect in a secondary queue within the controller during the interval period. Because the controller monitors this interval on a per-disk basis (for the purpose of batching), we have implemented an extra state variable for each disk, which the controller maintains. We have also implemented the "secondary" queues for each disk within the RAIDframe simulated controller. Both in the RAIDframe simulator and within a real implementation, this secondary queue exists inside the controller, is simply a list of pointers to actual write data, and requests accumulate in each queue separately.

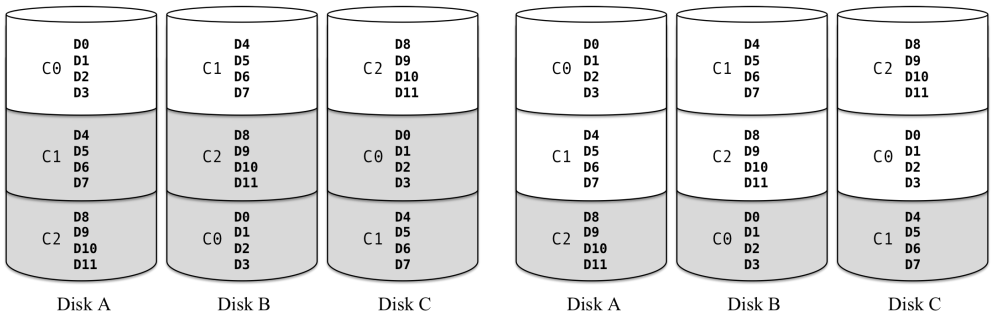
The state variable allows a disk to be in one of three states: NORMAL, DRAINING, UPDATING. The NORMAL state means the disk is servicing read and write requests normally. That is, a disk only services read requests for read DAGs, and write requests for minimum set DAGs. All remaining set DAG requests are retained in the secondary queue of the controller. In the RAIDframe simulator, a disk uses it's primary queue (i.e. the disk's internal queue) when in NORMAL or DRAINING state.

While a disk is in NORMAL state, the controller collects write updates for the remaining set DAGs into it's secondary queue. The controller keeps only a single copy of this write data, but it maintains a separate secondary queue for each disk. Periodically, it checks if the interval threshold has been crossed for any disk in the group. When this threshold value is crossed, the controller changes the state of that disk to DRAINING. While in the DRAINING state, a disk is completing all of the requests currently in its primary queue, and the controller does not schedule any new, incoming work to that disk. Reads that would normally be scheduled on this disk are serviced by sending them to a neighboring disk. This is done in the read scheduling loop simply by skipping over a disk that is not in NORMAL state, allowing read requests for that particular disk to be offloaded to another disk temporarily. In that case, the disk selected would be the disk where the stripe unit resides next-nearest it's outer edge in order to service the read as fast as possible. While incoming reads my be off-loaded, any incoming writes must not be moved to another disk (minimum set or remaining set DAGs) and

are collected into the controller's secondary queue for this disk and suspended. This is costly because it effectively blocks all minimum set DAGs involving the DRAINING disk. Alternatively, during the DRAINING state any new, incoming writes could be added to the accumulated batch that is waiting to be issued.

The purpose of the DRAINING state is to allow the disk to empty its current queue, satisfying all previous requests as normal. At that time, the disk state is set to UPDATING, and all the writes that have been collected into the secondary queue for this disk are sent to it in one batch. Within our RAIDframe simulator implementation, this is accomplished by setting the disk's queue to point to the secondary queue. Within an actual controller, the controller would simply forward to writes to disk.

Our design implies that requests from read DAGs and minimum set write DAGs will all be within some physical range of the outer-edge of a disk. We call this range the disk's normal read service area during fault-free mode as previously shown in 3.2. For clarity figure 3.14 shows both this normal service area for reads and the expanded service area to accommodate writes in the minimum set when $n = 3, |M| = 2$, that must be written to 2 disks while the user waits.



(a) 3 disk read service area

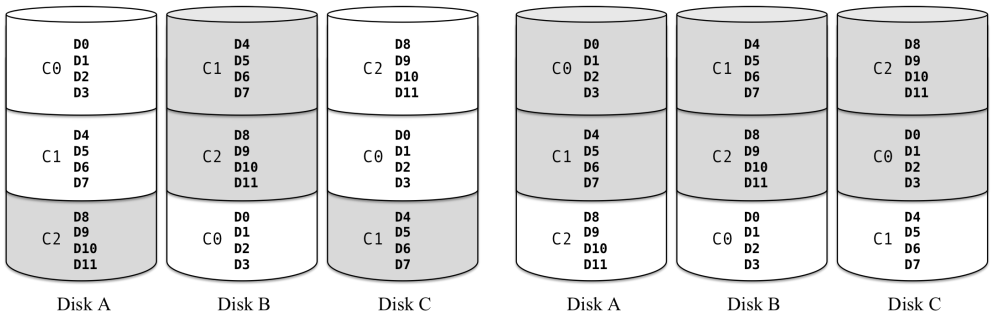
(b) 3 disk read and write service area for $M = 2$

Figure 3.14: The primary read service area (lighter) for each disk is restricted to those stripe units that are nearest the disk's outer edge. For writes, when $|M| = 2, n = 3$ the primary service area is expanded in order to write 2 copies of each stripe unit.

It can be seen from figure 3.14 that reads for all chunks (C0, C1, C2) can be

serviced by staying within the (lighter) service area. But with the expanded service area, minimum set writes for all chunks can also be serviced since $|M| < n$. This example for $|M| = 2$ shows C0, C1, and C2 can each be written to 2 disks while staying within the indicated physical service area. This policy helps disks to avoid long seeks while the user is waiting for reads and writes because the disk arm does not seek into the non-service area (lighter).

Because writes in the remaining set DAG are farthest from the outer edge of the disk (bottom of figures), they are outside a disk's normal service area by design, as the lighter chunks illustrate in figure 3.14. Therefore the intent of batching writes for remaining set DAGs is that the disk when the disk receives them, it will reorder them presumably with the mechanical arm moving from the outer-edge of the disk inward toward the center of the disk in one direction, servicing them optimally in one sweep. This would mean servicing the writes for the inner (lighter) area of the disks in figure 3.14 all at once. When a disk is in DRAINING or UPDATING state, it does not service any incoming requests, those are accumulated into the secondary queue. This is to prevent starvation of the suspended writes which are waiting to be serviced during the sweep, and to guarantee the disks become consistent after the interval. As an example, figure 3.15 shows the service area of each disk when Disk A and C are in NORMAL state, and Disk B is in UPDATING state.



(a) 3-disk RAID1nr service area, Disk B UPDATING

(b) 3-disk RAID1nr service area, all disks UPDATING

Figure 3.15: For $M = 2$, 3.15a shows the active service areas (lighter) for Disks A, C in NORMAL state, and Disk B in UPDATING state. 3.15b shows the service area (lighter) when all disks are in UPDATING state.

In figure 3.15a where $|M| = 2$, Disk B does not service any new, incoming reads or writes outside of the indicated service area while it is completing it's batched writes during UPDATING state. As shown by the indicated areas (lighter), all incoming read requests (in C0, C1, or C2) can be serviced by disks A and C. Figure 3.15b shows all disks in UPDATING state thus unable to service any new, incoming reads or writes. A mechanism to avoid this case is discussed in the following section on staggered updates.

Disks A and C will service incoming reads for stripe units in chunks C0, C1, and C2 while disk B is in UPDATING state as shown by their service area. During the UPDATING state, disk B is servicing batched writes. Incoming writes (only those in minimum set DAGs) for disks A and C will be serviced as normal, because those disks are in NORMAL state. As can be seen, this only includes writes for stripe units in C0, because that is the only chunk that can be written to 2 disks while staying within the indicated service area here. Writes for C1 and C2 must wait on disk B to return to NORMAL state because disk B is not servicing those areas while in UPDATING state. This results in forward progress being blocked for any incoming writes whose minimum set includes disk B, which is costly to user response time. However, no incoming reads are blocked because C0, C1, and C2 can all be serviced by disks A or C as can be seen

by their indicated service areas.

While a disk is in UPDATING state, the controller will accumulate incoming writes for that disk into the controller's internal queue. When the disk is finished UPDATING, the controller sets the disk's state to NORMAL, and then sends all accumulated requests from it's internal queue to that disk. Within our RAIDframe simulator implementation, this means the controller just points disk B back to its primary queue. Within a real implementation, the controller would actually forward the contents of the accumulated queue to that disk.

Staggered Updates

In order to minimize the impact of blocking new, incoming work, we only allow one disk at a time to be in DRAINING or UPDATING state. All other disks are in NORMAL state. This allows us to avoid the worst case scenario where all disks are in UPDATING state at the same time, and no new foreground read or write requests could be serviced. As mentioned previously and shown in figure 3.15, when only one disk is in UPDATING state, the others can service at least all incoming reads (C0, C1, C2), and some writes as well (those for C0 in this example).

To implement this in the controller, a single `update_token` is used, which the controller maintains and must assign to a disk before it can be set to DRAINING or UPDATING state. The controller passes the `update_token` round-robin to disks when they cross their interval threshold. If a disk crosses it's threshold but another disk owns the `update_token`, it will remain in NORMAL state while it is waiting. After the `update_token` is assigned and the disk completes all of it's work in DRAINING and UPDATING states, the controller sets the disk state to NORMAL, the `update_token` is marked available, and the controller sends that disk any new work. The accumulated work would only be minimum set writes that were waiting while the disk was in UPDATING state. After returning to NORMAL state, a disk resumes normal operation until the interval threshold is crossed again. At that time the controller will repeat the

process of assigning the `update_token`.

Method 3

Method 3 follows the same procedures as Method 2, but changes the location where the actual write data waits while it is stalled. While remaining set DAGs are suspended, Method 2 stores their data in a secondary queue within the controller. With Method 3, that data is sent to disk. This method would require disks that have the ability to separate their request queue into 2 segments, representing a primary queue and a secondary queue. We describe it here as a possible suggested implementation as we are not aware of any disk with this specialized logic. However, the performance results are expected to be nearly identical to Method 2, no matter where the actual write data is accumulated while it is waiting.

The disk would maintain this write data (belonging to remaining set DAGs) in a separate segment of its request queue until some interval threshold is met just as in Method 2. The disk would also maintain its own states internally. The controller would not be involved in state changes, so no `update_token` would be needed. The suggestion to allow the disks to maintain their own states is because a disk would know best when it is time to do its write updates based on its queue size, or other parameters. A similar mechanism exists in modern disks which may switch into background scrubbing mode on their own after some idle time period, in an attempt to identify and correct latent sector errors (Seagate, 2008). At that time, the disk could switch to its secondary queue and begin servicing requests from there. Alternatively, the disk could use a single queue with priority tags. Remaining set writes could be given a low priority and ignored until some condition is met, such as our interval threshold.

Implementation of this mechanism at the disk level would require some extra logic added to disk drives that equates to a normal priority for regular work (primary queue), and a low priority for other work (secondary queue). When the interval threshold is met, the disk simply switches from normal priority work to low priority work, thus

clearing out all of its low priority work at once, which should have the same mechanical arm “sweeping” effect described in Method 2.

Chapter 4

Experimental Results

We present the results for read-only (archival), write-only, and mixed workloads. We also show results for degraded mode and the affects of our load balancing methods. In the final chapter we offer observations and discussion.

4.1 Experimental Setup

All of our experiments were conducted using the RAIDframe simulator. We vary the number of disks in each redundancy group, and the number of concurrent users from 1 - 1000 to simulate light and heavy workloads. All of the disks in each group are identical IBM 18LZX disks. Their characteristics are 4.9 ms average seek time, 3.0 ms average rotational latency, 10,000 rpm, 18.3 GB capacity (IBM, 1999). We use the simulator's workload generator which creates a workload of equal-sized IOs uniformly randomly distributed across the logical address space. We have chosen a random workload to test our system as this is known to be the most difficult for disk drive performance due to the mechanical cost of random seeks. One of the goals of our testing is to show that our design performs well specifically for random workloads, without the use of complex access prediction algorithms, dynamic block reorganization, read-ahead buffering, or adding extra cache to the disk or controller. In the RAIDframe

simulator, there is no cache at the disk or controller level.

We vary the percentage of reads and writes, and present the results separately. For each configuration, we use several different stripe sizes as indicated, and we vary the IO size from small (4K) to large (1024K). Our current RAID1nr implementation limits the IO size to the stripe size or smaller. The reasons for this are to simplify our implementation, and to follow our design principle of not parallelizing a single read operation over multiple disks, but rather taking advantage of our data placement by transferring as much data as possible once we arrive at the data on a particular disk. Since the stripe unit size is a tunable parameter and IO performance is sensitive to the value, all requests in the following experiments are for a single stripe unit in order to clearly show only the disk performance.

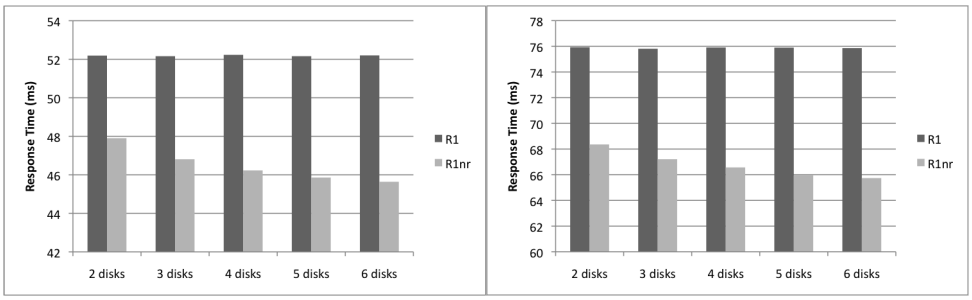
We have selected the shortest seek time first (SSTF) disk scheduling algorithm for each disk. The SSTF algorithm as implemented in RAIDframe reorders requests by computing a simple distance calculation from the current physical block position to the next physical block, where distance is measured in sequential blocks between the two positions. RAIDframe does not offer the SPTF algorithm.

For all performance measurements we compare our RAID1nr system to a standard 2-disk RAID1, as well as an n-disk RAID 1. The n-disk RAID 1 is a single RAID 1 configuration with n-1 mirrors. For the most straightforward direct comparison, we only consider a single redundancy group of each type. Because the disks, array controller, and the forward progress of time are all completely simulated within RAIDframe, the machine which runs the simulator has no affect on the results. Each data point represents the sample mean for 5 runs, each 200 seconds. Error bars indicate our sample variance, however, almost all of the variances were near zero so the error bars are not visible as they are contained within the data point marker.

4.2 Read Performance

4.2.1 Response Time

We begin with a fault-free read-only workload. Figures 4.1a and 4.1b show the response time for 4KB and 64KB reads respectively, while maintaining a constant workload of 10 users per disk. Specifically, for 2 disks the experiment is run with 20 users, for 3 disks there are 30 users, etc. Improvement percentages are given in table 4.1.



(a) 4KB reads (b) 64KB reads

Figure 4.1: Response time for reads maintaining a constant workload of 10 users per disk.

Table 4.1: Response time improvement of RAID1nr over RAID 1, taken from figure 4.1.

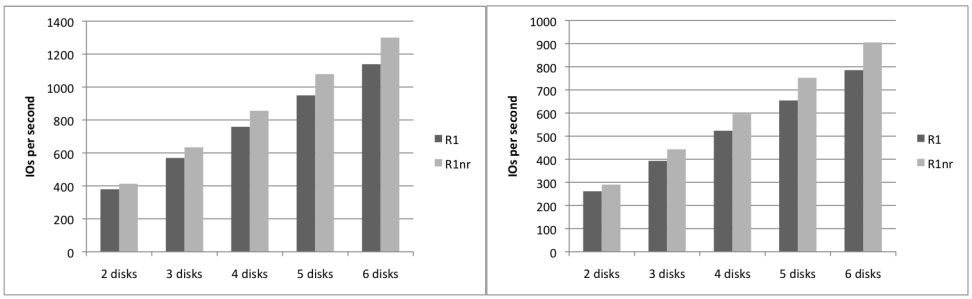
	4KB Reads	**64KB reads**
2 disks	8.2%	9.9%
3 disks	10.3%	11.5%
4 disks	11.5%	12.3%
5 disks	12.2%	13.1%
6 disks	12.6%	13.4%

Figure 4.1 illustrates an interesting aspect of our results. As more users and disks are added in equal proportions, the response time remains constant for RAID 1.

This is expected since each new disk is servicing the same amount of work, i.e. 10 new users, and the scheduler is using the shortest queue length algorithm. As indicated, the workload for each disk remains constant. However, the RAID1nr array shows an improvement in response time as more disks and users are added due to the RAID1nr read scheduling algorithm. This is precisely because the workload for each disk remains constant at 10 users, but disks in the RAID1nr configuration each only service $\frac{1}{n}$ of the address space. Thus, while the workload remains the same, the service area of each disk becomes smaller as more disks are added to the group. The seek time is reduced because each disk is servicing a smaller area. Moreover, the smaller service area also helps the disk scheduling algorithm provide a more optimal scheduling sequence. Additionally, there is a clear trend toward a plateau as more disks are added, showing the initial few extra mirror disks gain the most benefit. The diminishing benefit is due to the seek time improvement levelling-off, as we show later in Figure 4.6.

4.2.2 Throughput in IOs per Second

Figure 4.2 shows the throughput performance behavior, indicating the benefit increases as more disks are added. Improvement percentages are given in table 4.2.



(a) 4KB reads (b) 64KB reads

Figure 4.2: IOPS for reads maintaining a constant workload of 10 users per disk. Improvement percentages are shown in Table 4.2.

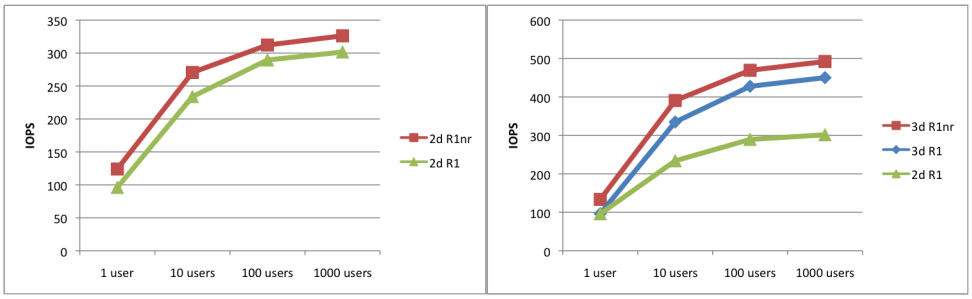
Figures 4.2 and 4.1 show that RAID1nr consistently outperforms a standard

Table 4.2: Throughput improvement (IOPS) of RAID1nr over RAID 1, taken from figure 4.2.

	4KB Reads	**64KB reads**
2 disks	8.9%	11.0%
3 disks	11.3%	12.7%
4 disks	12.8%	13.9%
5 disks	13.6%	15.0%
6 disks	14.2%	15.3%

RAID1 configuration with n mirrors, and the performance improvement increases as more users and disks are added while maintaining a constant workload. These throughput results show that a 3 disk RAID 1 is a 50% improvement over a 2 disk RAID 1, but a 3 disk RAID1nr yields a 67% improvement.

Next is the throughput performance as an increasingly heavy workload is added to a fixed number of disks. Figures 4.3 and 4.4 show the performance trend for 1 - 1000 users and 2 - 5 disks, each time also comparing to a standard 2-disk RAID 1 as a baseline.

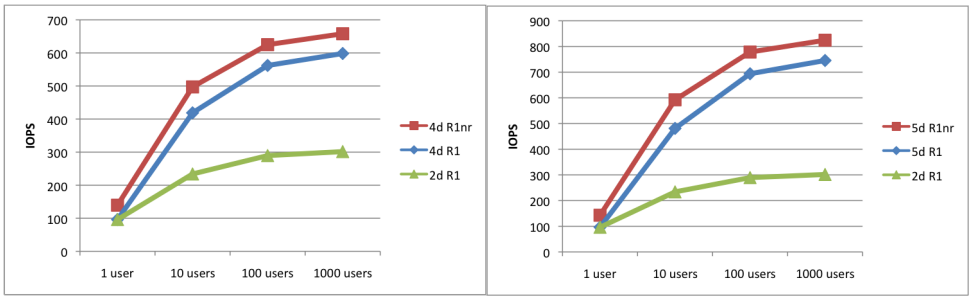


(a) 2 disks (b) 3 disks

Figure 4.3: IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1.

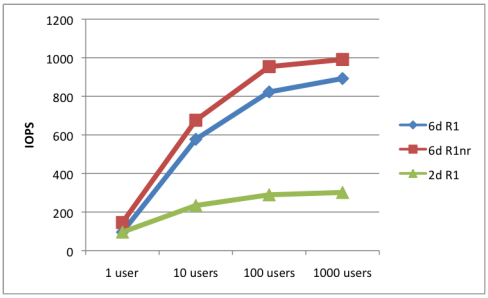
Figure 4.3a shows a 13% higher IOPS for 10 users on the 2-disk RAID1nr configuration over the standard 2-disk RAID1 configuration. Figure 4.3b shows the 3-disk

RAID1nr has 17% higher IOPS than a 3-disk RAID1, and 63% higher IOPS than the 2-disk RAID1 for 10 users. The same performance trends appear in figures 4.4a, 4.4b and 4.5 .



(a) 4 disks (b) 5 disks

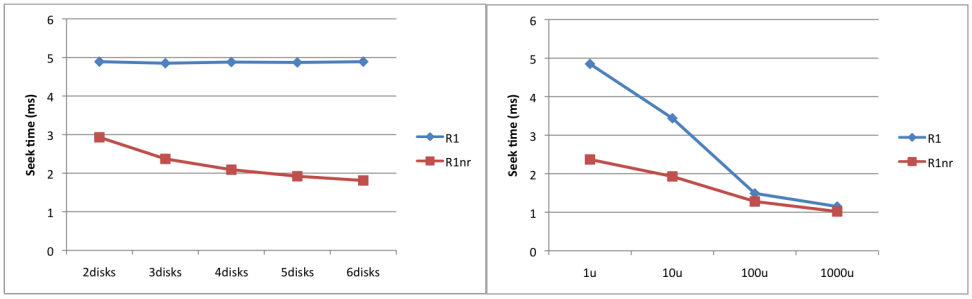
Figure 4.4: IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1.



(a) 6 disks

Figure 4.5: IOs per second, 64KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1.

For these smaller reads, our system takes advantage of the reduced seek time, but for larger reads where the transfer time dominates the access time, our system takes advantage of the higher transfer rates for the outer edges of the platters. Graphs showing 1024 KB reads for 1-1000 users can be found in Appendix A. Also in Appendix A we show 64KB reads for 10, 20, 30, ... 100, 200, etc. users to verify the behavior between the data points on these 1-1000 user graphs. Those graphs show that the behavior



(a) 64KB reads, 2-6 disks, 1 user (b) 64KB reads, 3 disks, 1-1000 users

Figure 4.6: Average seek time for 64KB reads.

follows the same trends between these data points.

4.2.3 Seek Time

In figure 4.6 seek times are shown when adding more users or more disks. As expected, these results indicate a plateau for seek time improvement. Because seek time is dependent on the mechanical properties of a disk drive, the minimum seek time is bound by the hardware.

Figure 4.6a strictly shows the affect of a smaller physical service area on the seek time. As more disks are added the RAID1nr configuration serves requests from a smaller physical area on each disk, the outer $\frac{1}{n}$, keeping the mechanical arm movement within the outer-most region of the disk platters. With more disks, the region becomes smaller and the seek time decreases but slowly begins to plateau. The affect of this plateau-like behavior on response time can be seen previously in Figure 4.1. Seek time for the RAID 1 configuration remains constant because a request may go to any disk, and may be located anywhere on the disk platter. Figure 4.6b shows two behaviors. First it can be seen that the seek time for the RAID1nr configuration is less than RAID1 even for an increasingly heavy workload. Next, the affect of more users on the disk scheduling algorithm shows that at about 100 users, there is little difference in seek time for either configuration. This is due to the density of the request stream on each disk

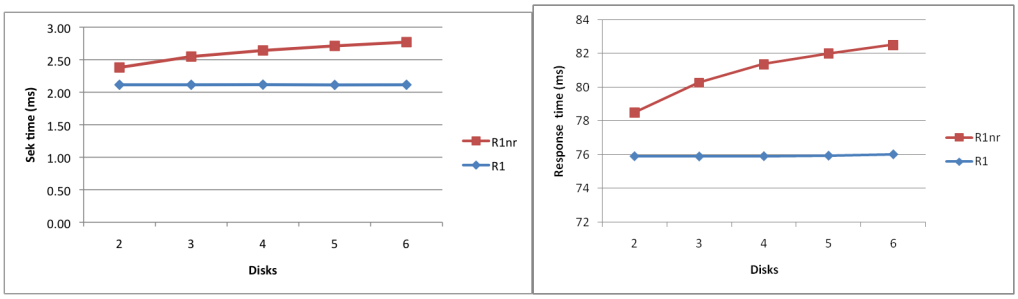
allowing the disk scheduling algorithm (SSTF) to provide the best possible sequence to minimize the disk arm movement. The requests become so close together in physical space that they require nearly the minimum possible seek movement and positioning time.

The initial value shown for 1 user is 4.9 ms which is exactly the average seek time reported on the data sheet (IBM, 1999). From 10-100 users, it shows the seek time improvement levels-off as the disk request queue gets increasingly long. These graphs illustrate how our design takes advantage of both a smaller physical space to serve data from (fig.4.6a) and an increased number of users (fig.4.6b) to improve performance. The smaller space requires less mechanical seek distance, while adding more users helps to maximize the disk scheduling algorithm's performance. Each factor contributes to the reduced response time and increase the IO throughput for requests in our system.

4.3 Write Performance

We present the results for a 100% write fault-free workload. Since performance is bound by the last mirror disk to complete the write, RAID 1 write performance does not change with more disks. The average response time for writes remains flat since each disk is servicing the same request stream in this 100% writes case. However, the RAID1nr configuration actually performs slightly worse than a standard RAID1 as the number of mirrors increases. This is due to the rotation of the chunks on each disk, and we explain this effect in detail next. We also present results for improving write performance by using Method 1 and Method 2 with sets of M disks implementing our immediate and eventual consistency policies as presented in Section 3.8.1.

Figure 4.7 shows the write performance for an n-disk RAID1 and RAID1nr configuration. Method 1 or 2 are not used in these examples in order to show the default behavior of RAID1nr for writes. As the number of disks increases, figure 4.7a shows the average seek time remains flat for the RAID1 configuration (since each disk is servicing the same request stream) , but actually gets increasingly worse for the RAID1nr



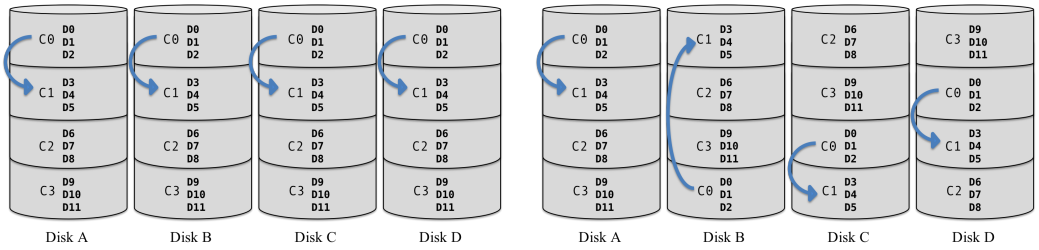
(a) Seek time for 64KB writes (b) Response time for 64KB writes

Figure 4.7: Seek time and Response time for 100% write workload with 10 users.

configuration. Figure 4.7b shows the impact this has on response time for writes. At 2 disks, the RAID1nr rotated layout is 3% slower than standard RAID 1. At 6 disks, it is nearly 8% slower. This effect is due to the behavior of writes in the rotated data layout which has a negative impact on average seek distance.

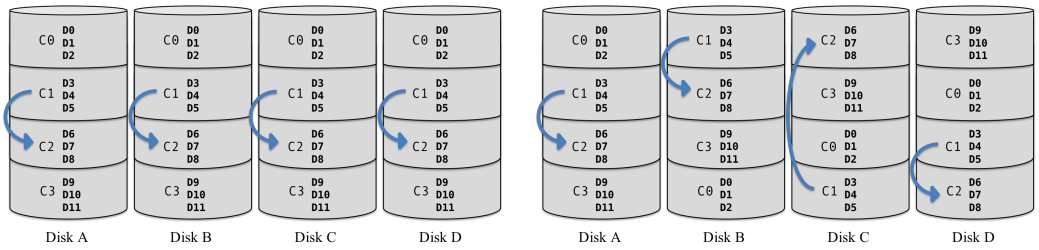
The reason Raid1nr has a slower response time for writes is due to the rotation of the chunks. This was not immediately obvious, but it can be seen from figures 4.8 and 4.9 that the seek distance from stripe unit D0 to D3 is the same for each disk in the RAID1 configuration, but not in the RAID1nr configuration. Using stripe units D0 and D3 as an example, the non-rotated layout has the same distance to seek when going from chunk 0 to chunk 1 (C0 to C1), which is basically seeking across one chunk. However, the RAID1nr rotated layout must seek farther for one of the disks as can be seen for disk B in 4.8. One disk will always incur the maximum chunk seek distance (top chunk to/from bottom chunk). In the RAID1 configuration, a long seek is only incurred between the first and last chunk. For RAID1nr, a seek from any chunk to a neighboring chunk on one disk will incur a long seek on one of the other disks. This occurs for any n-disk RAID1nr configuration. For n disks, each disk has n chunks and so there are $n \times n$ possible chunk-to-chunk seeks on each disk. In the non-rotated layout, there is only one long chunk-to-chunk seek: between the top chunk and the bottom chunk. This occurs in exactly 2 of the $n \times n$ cases. In contrast, one of the disks in the rotated layout will incur the long (top chunk to bottom chunk) seek in $2(n-1)$ of the $n \times n$ possible

cases. Because data must be written to all n disks, the response time will always be bound by the slowest disk to complete the write operation. In figure 4.8b the write is not considered complete until disk B is done, which has the longest seek. Therefore, even though some disks have a short seek and complete the write quickly in the rotated layout, it will still need to wait on one disk with a long seek more often than the non-rotated layout and this results in a slower response time for writes as seen in 4.7b.



(a) 4 disk RAID1 (b) 4 disk RAID1nr

Figure 4.8: Seek distance from chunk 0 (C0) to chunk 1 (C1) for a 4 disk RAID1 and RAID1nr configuration. In (b) Disk B has a long seek due to the rotated layout.



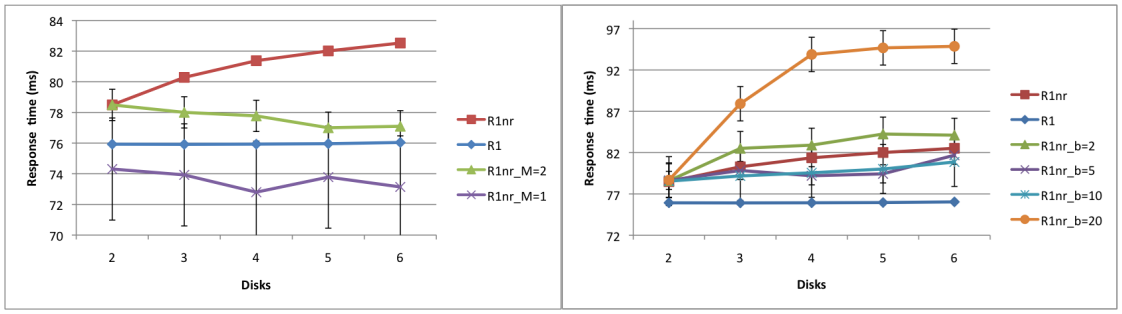
(a) 4 disk RAID1 (b) 4 disk RAID1nr

Figure 4.9: Seek distance from chunk 1 (C1) to chunk 2 (C2) for a 4 disk RAID1 and RAID1nr configuration. In (b) Disk C has a long seek due to the rotated layout.

By applying the methods described in section 3.8.1, next we show the improvement in the response time for writes. Figure 4.10a shows the performance for Method 1 when $M = \{1, 2\}$, while overlaying this onto figure 4.7b to show the changes. Method 1 (section 3.8.1) issues writes to all disks simultaneously but only waits on the disks in M to complete, it does not do batching of stalled writes. This shows how Method 1

helps to improve response time for writes in the Raid1nr system due to the rotated layout, since we never have to wait on the disk with the long seek, such as depicted for disk B of figure 4.9b. By always avoiding the worst case seek distance for the immediate consistency writes, while the eventual consistency writes complete asynchronously in the background, the user response time is decreased for both $M = 1$ and $M = 2$. With $M = 2$, we improve the response time over Raid1nr by 0% (2 disks) to 6.6%(6 disks). With $M = 1$, we improve the response time over Raid1nr, and it is even less than the Raid 1 layout since we are only writing to one disk immediately. This graph shows the effectiveness of the immediate and eventual consistency design for writes.

Figure 4.10b uses Method 2 from section 3.8.1. Method 2 issues writes to the disks in M immediately, and stalls the writes for the remaining disks (R), later issuing them in batches as each disk enters it's UPDATING state. Here we are using $M = 2$, and writes for the disks in R are issued later when the batch sizes for each disk reach $\{2, 5, 10, 20\}$. The metric for a disk entering UPDATING state in this example was batch size. As mentioned in "Method 2" of Section 3.8.1 a different metric for issuing an update batch could be elapsed time. Figure 4.10b shows the batching behavior is not very stable, and has a high variance for each result. This is likely due to the particular time a disk changes into UPDATING state, when it does not service any foreground work until it returns to NORMAL state. With a batch size of 2, the response time is slightly worse than RAID1nr, presumably due to the cost of blocking foreground work while updating, as well as a small batch size that is unable to benefit much from request reordering. Batch sizes of 5 and 10 show a slight but varying improvement, since larger batch sizes benefit a bit more from reordering. Although a batch size of 20 could benefit even more with reordering, it is clearly the worst performer. This is most likely due to the amount of time it remains in UPDATING state and is unable to service any foreground work. Additionally, because this takes some time, it is probable that 2 or more disks will be in UPDATING state simultaneously, thus potentially preventing foreground work from progressing on more than one disk at a time. It is clear from figure 4.10b that Method 2's batching behavior is a bit unstable with high variance,



(a) Method 1 with $M = \{1, 2\}$ (b) Method 2 with $M = 2$, batch sizes {2,5,10,20}

Figure 4.10: Response time for 64KB writes with exactly 10 users over 2-6 disks for Raid1 and Raid1nr using Method 1 (no batching) with $M = 1$ and $M = 2$ (a), and Method 2 (batching) with $M = 2$ with various batch sizes (b).

while figure 4.10a shows Method 1 using $M = 1$ or $M = 2$ performs consistently better than RAID1nr and has more stable behavior.

4.4 Mixed Workload

We present the results for an example mixed workload of 80% reads, and 20% writes using 64KB IOs. The first experiment uses Method 1 with $M = 1$ and $M = 2$ for 2 - 6 disks with a workload of exactly 10 users. Figure 4.11 shows that the n-disk Raid 1 has the slowest response time. As expected, Raid1nr shows definite improvement over Raid1, and Raid1nr with $M = 1$ and $M = 2$ both consistently outperform the standard Raid1nr. The exact improvement percentages are show in figure 4.12.

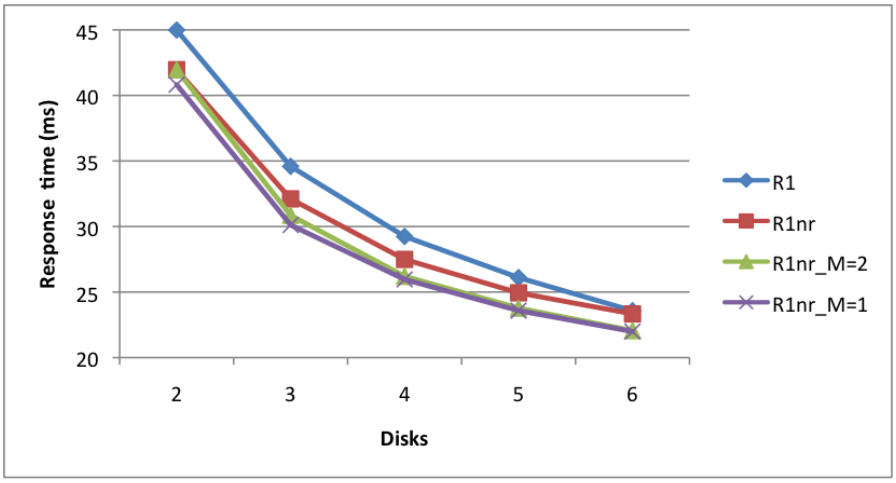


Figure 4.11: Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users on 2-6 disks showing Raid1 and Raid1nr with $M = \{1, 2\}$.

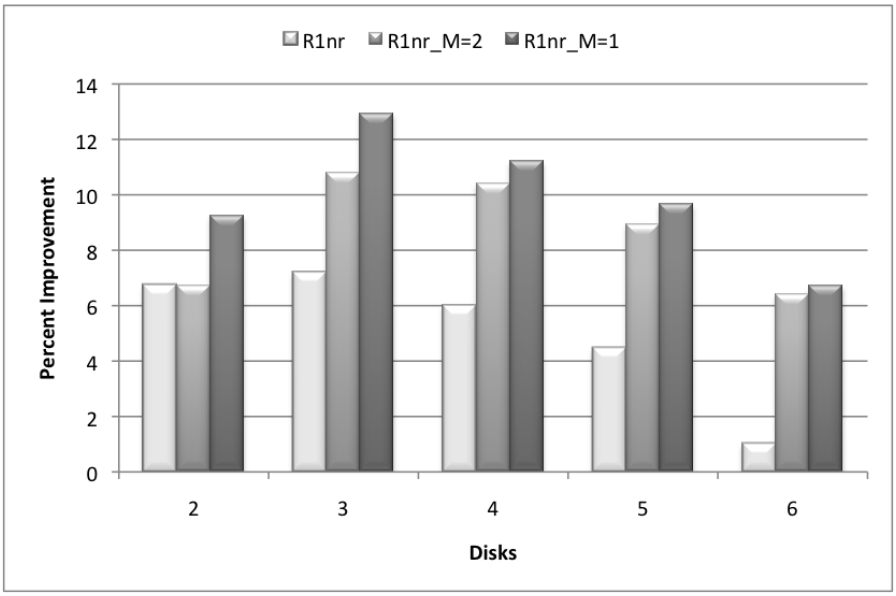


Figure 4.12: Response time improvement percentage of Raid1nr configurations over a 2-6 disk Raid1 baseline, taken from figure 4.11.

Figure 4.13 shows the performance of 3-disk configurations as the workload increases from 1-10 users. It shows that Raid1nr consistently outperforms Raid 1 with one exception. Notably it also shows that as more users are added, the relative performance

of Raid1nr as compared to Raid 1 gets increasingly better. This benefit comes from our data layout and the preferred disk scheduling policy. As more users are added, each Raid1nr disk is serving more requests from within the same smaller physical space than a Raid 1 disk, thus the SSTF disk scheduling algorithm is able to achieve better request reordering. Although we show up to 10 users here, as seen previously in 4.6b the seek time reduction from reordering begins to show diminishing returns.

In the case where Raid1nr is slower, this happens when the number of users is equal to the number of disks. This is because it is possible that one disk has extra work while another is idle. Due to the preferred disk scheduling policy, if each of the users issues a request for data within the same chunk, all of those requests will be scheduled to the preferred disk for that chunk while the other disks remain idle. This does not happen in Raid 1 as it uses the shortest queue scheduling policy. When the number of disks is equal to the number of users, each disk in Raid 1 is servicing 1 request all of the time and no disk will be idle or overloaded. Thus the Raid 1 performance is slightly better than Raid1nr in single exception. The response time improvement percentages for each point in figure 4.13 are shown in figure 4.14.

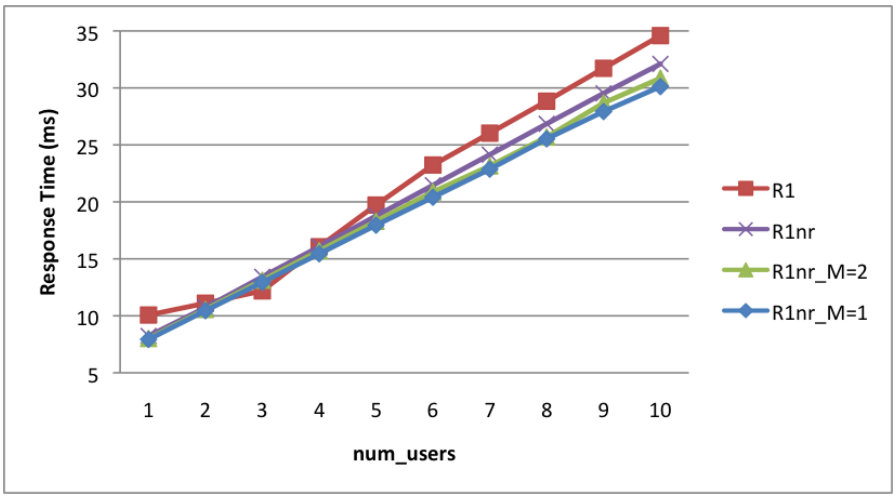


Figure 4.13: Response time for 64KB mixed workload (80/20 read/write) on 3 disks with 1-10 users showing Raid1 and Raid1nr with $M = \{1, 2\}$.

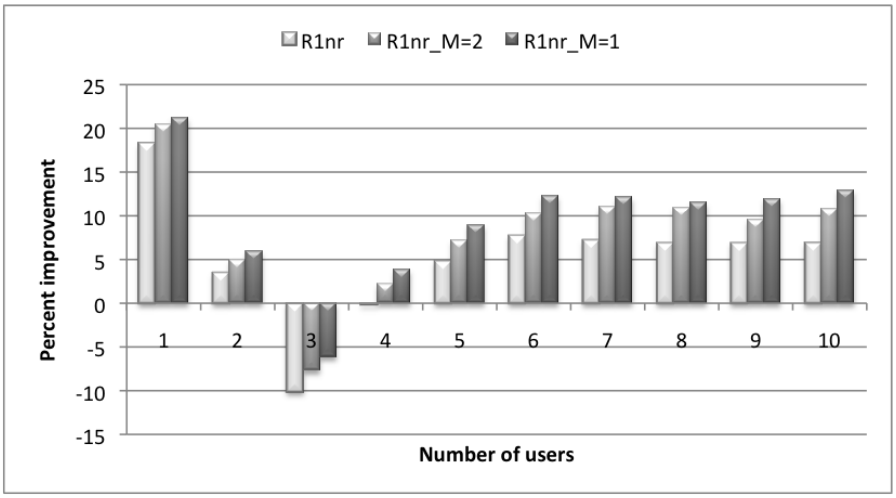


Figure 4.14: Response time improvement percentage of Raid1nr configurations over Raid1 baseline, taken from figure 4.13.

Lastly, figure 4.15 shows the performance improvement of several 3-disk configurations to a standard 2-disk Raid 1. The column for '3d R1' shows the performance benefit of adding a single extra mirror disk to a standard 2-disk Raid 1. The columns for '3d R1nr' show the benefit of using the Raid1nr data layout and scheduling policies along with our immediate and eventual consistency policy using $M = 1$ and $M = 2$. At 10 users, the 3-disk Raid 1 gives a 23.1% improvement, while the 3-disk Raid1nr with $M = 1$ gives a 33.1% improvement. This figure shows similar behavior for lightly loaded systems of 1-10 users, and heavily loaded systems of 100-1000 users.

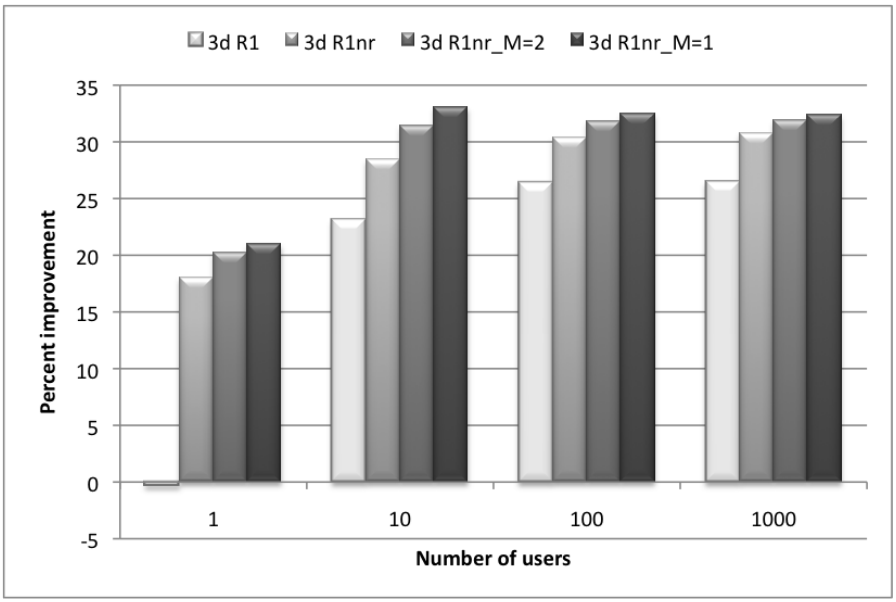


Figure 4.15: Response time improvement percentage of several 3-disk configurations compared to a 2-disk Raid 1 baseline. Showing a 64KB mixed workload (80/20 read/write) on 3 disks for 1, 10, 100, and 1000 users.

4.5 Load Balancing

Load balancing results are shown for 10 users on 2 - 6 disks. This experiment uses 64KB IOs in a mixed 80/20 read/write workload, with a non-uniform workload. Here we configure 10% of the logical address space as "hot", beginning with logical block 0. Then we force 30% of the workload generated to be within this hot spot.

Figure 4.16 shows the load balancer improves the response time when a workload has a skewed distribution due to a hot spot in the logical address space. We are only able to test a single hot spot here since the RAIDframe simulator does not allow us to specify multiple simultaneous hot spots. The goal of the load balancer is to prevent any disk from having a disproportionally long queue compared to its immediate neighboring disk. Here, a scalar setting of $\frac{2}{3}$ is used to compare the preferred disk's queue length with the "next-best" non-dead disk during read scheduling. If the neighboring disk's

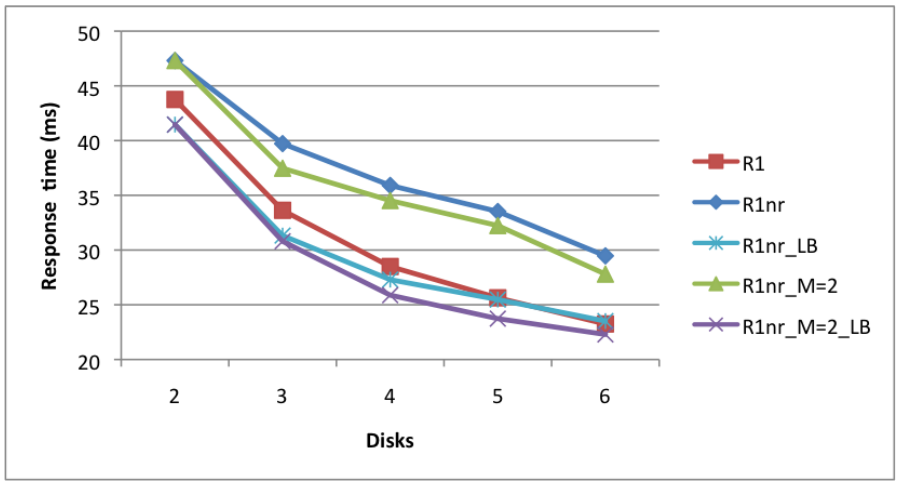


Figure 4.16: Skew injected workload with load balancing. Response time for exactly 10 users on 2-6 disks with an 80/20 mixed workload, including Method 1 with $M = 2$ and load balancing (LB).

queue length is less than $\frac{2}{3}$ of the preferred disk's queue length, the read is scheduled to the neighboring disk. The scalar fraction is a tunable parameter. At most, one non-dead neighboring disk is considered when offloading work, in order to help preserve the smaller read service area of each disk.

It is clear that Raid1nr response time is worse when there is a skewed workload as compared to Raid 1. Here, Raid1nr is the worst performer. Using Method 1 with $M = 2$ improves the response time only slightly. However, with load balancing the response time is greatly improved. At 3 disks, the Raid1nr with load balancing shows a 6.9% improvement in response time over a 3-disk Raid 1, while Raid1nr with load balancing and $M = 2$ shows an 8.4% improvement over a 3-disk Raid1. We only show Method 1 here since in general it performs as well or better than Method 2 and is more stable.

A workload imbalance due to hot spots from skewed workloads or disk failures is an affect of the RAID1nr system design. Due to the preferred disk scheduling policy, a hot spot within a given chunk will increase the service workload of the preferred disk serving the chunk. Similarly, in degraded mode the uneven shifting of work from the

failed disk's preferred chunk to it's neighboring disk is depicted previously in figure 3.3. The correcting effect of the load balancer is then depicted in 3.5, which shows how the the load balancer helps to shift the workload slightly so that the disks maintain a minimally expanded service area when there is a workload imbalance. Results showing the effect of the load balancer during degraded mode are presented in the next section.

4.6 Degraded Mode

For degraded mode, we present results for Raid1 with Raid1nr with and without load balancing using Method 1. Figure 4.17 shows a 3 disk configuration while failing multiple disks, and figure 4.18 shows a 5 disk configuration while failing multiple disks. Both examples show that Raid1nr configurations without the load balancing have worse response time than Raid1. However, Raid1nr with load balancing performs better than Raid1 in all cases until only a single disk remains, at which time the performance for all configurations is the same.

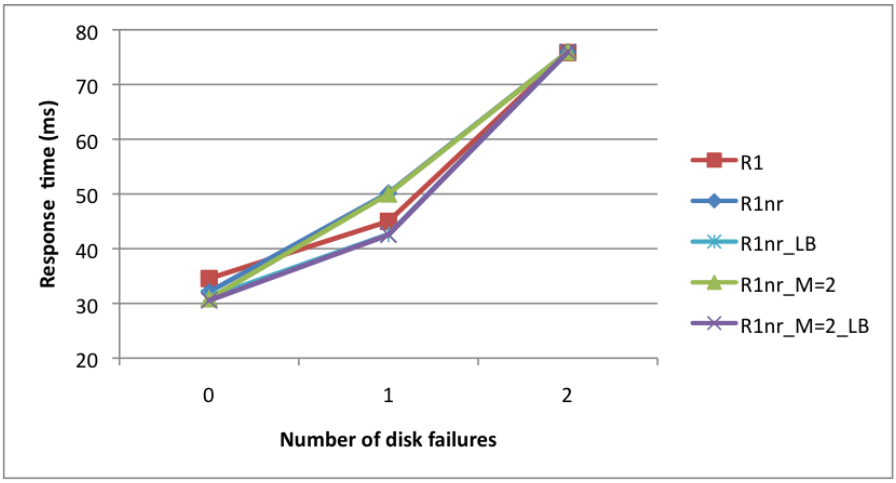


Figure 4.17: Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users, showing 0-2 disk disk failures, including Method 1 with $M = 2$ and load balancing (LB). 3 disk configurations.

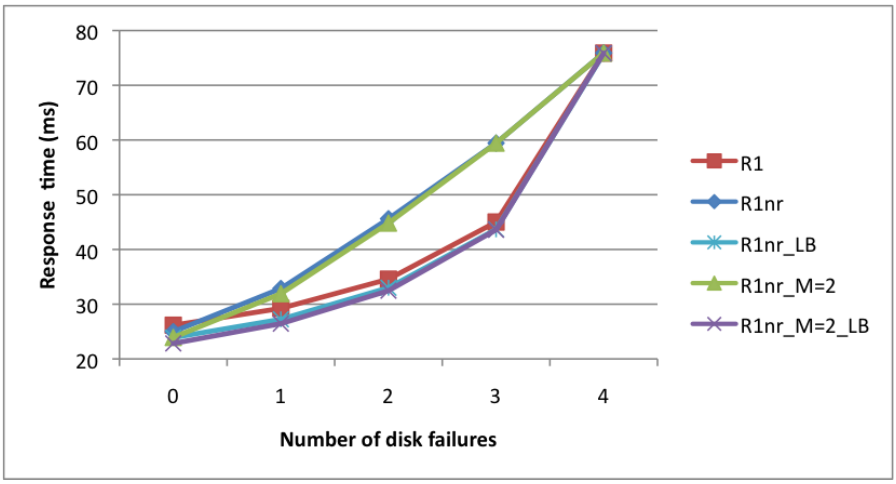


Figure 4.18: Response time for 64KB mixed workload (80/20 read/write) with exactly 10 users, showing 0-4 disk failures, including Method 1 with $M = 2$ and load balancing (LB). 5 disk configurations.

Tables 4.3, 4.4, and 4.5 show the workload distribution per disk for Raid1, Raid1nr, and Raid1nr with load balancing when a disk fails. Table 4.3 shows Raid1 evenly distributes the read workload across each remaining disk during failure due to it's shortest queue scheduling policy. Table 4.4 shows the Raid1nr has a 12.4% worse response time than Raid1. The preferred disk scheduling policy offloads all of the failed disk's workload onto the neighboring disk where the chunk is next-nearest the outer edge. In this case disk 4 doubles its read workload from 16% to 32% since it absorbs all of the reads for failed disk 0.

The preferred disk scheduling policy shifts all of a failed disk(s) workload to it's nearest non-dead neighboring disk, increasing it's read service area from $\frac{1}{n}$ to $\frac{2}{n}$ of the logical address space resulting in the workload distribution imbalance depicted previously in figure 3.3. A disk will absorb at most $\frac{f}{n}$ of the workload, where f is the number of failed neighboring disks.

The load balancer mitigates this imbalance very effectively and provides better performance as well. Table 4.5 shows how the load balancer distributes the workload and improves the performance in Raid1nr to 6.8% better than Raid1. As mentioned

previously, for any given disk, the load balancer will consider the queue length of the next-best non-dead disk when scheduling a read for a given chunk. Again we apply the scalar $\frac{2}{3}$ to the neighboring queue length here. Because of the workload imbalance due to disk 0 failing, table 4.5 shows the cascading effect of distributing the workload to a neighboring disk, where disk 4's workload is decreased from 32% to about 23%. Similarly, it can be seen disk 4 shifts some of it's read workload to disk 3, and so on.

Table 4.3: Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1 configuration with disk 0 failed.

	Read %	**Write %**
Disk 0	0.0	0.0
Disk 1	19.82	20.20
Disk 2	19.80	20.20
Disk 3	20.17	20.20
Disk 4	20.01	20.20
Avg. response time = 29.24 ms		

Table 4.4: Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1nr configuration with disk 0 failed.

	Read %	**Write %**
Disk 0	0.0	0.0
Disk 1	16.04	19.87
Disk 2	15.97	19.87
Disk 3	16.00	19.87
Disk 4	32.12	19.87
Avg. response time = 32.87 ms		

Table 4.5: Workload distribution per disk for mixed workload (80/20 read/write) during degraded mode. A 5-disk Raid1nr configuration with disk 0 failed, using load balancing.

	Read %	**Write %**
Disk 0	0.0	0.0
Disk 1	17.45	19.88
Disk 2	18.12	19.88
Disk 3	20.93	19.88
Disk 4	23.61	19.88
Avg. response time = 27.24 ms		

Chapter 5

Discussion and Conclusion

This thesis has presented a data layout for an n-way mirroring system with associated scheduling policies for reads and writes. The results show our Raid1nr system outperforming a similar Raid 1 system with n-way mirroring when $n \leq 6$ for various types of workloads with 1-1000 users. We have shown these performance gains for read only workloads, write only workloads, mixed workloads, workloads with skewed distributions, and workloads during degraded mode failing up to $n - 1$ disks. The system is very flexible and provides administrative options for the amount of fault tolerance desired by increasing the number of mirrors. We also provide increased performance by allowing a very small window of slightly-less fault tolerance by using our immediate and eventual consistency policies for writes, while guaranteeing a minimum level of fault tolerance during this window. This minimum may be chosen by the administrator. We have shown that Raid1nr degrades gracefully for up to $n - 1$ disk failures, using load balancing to distribute the workload for high throughput among the surviving disks. Though not presented here, additional experiments were done for larger values of n showing similar performance gains, giving us no reason to believe the improvements are eliminated as n increases.

Our system design and IO scheduler take advantage of smaller service areas for reads to reduce seek time, use the outer edge of the disk to achieve higher data transfer

rates, and exploit disk scheduling policies to improve the benefits from request reordering. Our immediate and eventual consistency scheduling policies help to improve read and write performance even as more disks are added. Additionally, the load balancer improves performance during skewed workloads and during degraded mode. We have also shown that the rotated data layout alone can improve performance somewhat over Raid 1, but by adding our scheduling, load balancing, and consistency policies Raid1nr can achieve even better results.

Appendix A

Appendix

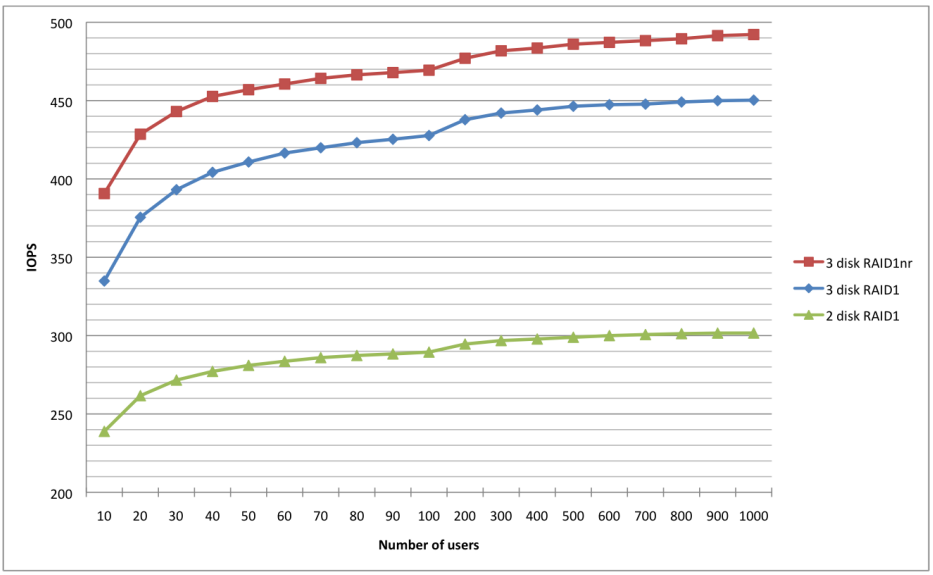


Figure A.1: IOs per second, 64KB reads, 10 - 1000 users. Each request is for a single stripe unit, of size 64KB.

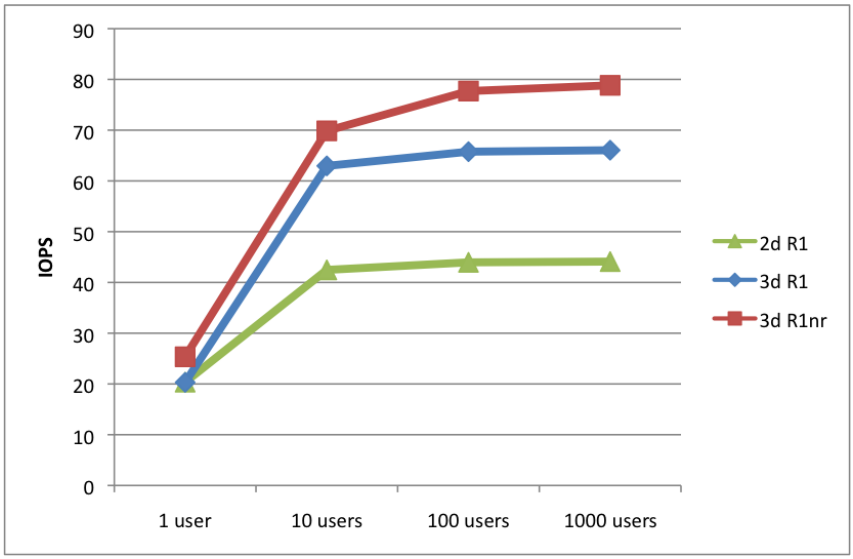


Figure A.2: IOs per second, 1024KB reads, 1 - 1000 users. RAID1nr compared to standard RAID 1. Each request is for a single stripe unit, of size 1024KB.

References

Akyrek, S., and Salem, K., 1995: Adaptive block rearrangement. *ACM Transactions on Computer Systems*, **13(2)**, 89–121.

Alvarez, G., Burkhard, W., and Christian, F., 1997: Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Int'l Symposium on Computer Architecture*, 62–72. ACM.

Alvarez, G., Burkhard, W., Stockmeyer, L. J., and Christian, F., 1998: Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Computer Architecture*. ACM/IEEE.

Anderson, D., 2003: You dont know jack about disks. *ACM Queue*, **1**(4).

Anderson, D., Dykes, J., and Riedel, E., 2003: SCSI vs. ATA - More than an interface. In *Proceedings of the 2nd USENIX Conference on File and Storage*.

Bitton, D., and Gray, J., 1988: Disk shadowing. In *Proceedings of the 14th VLDB Conference*, 331–338.

Blaum, M., Brady, J., Bruck, J., and Menon, J., 1995: EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, **44(2)**, 192–202.

Bovet, D., and Cesati, M., 2001: *Understanding the Linux Kernel*. O'Reilly and Associates, 3rd edition.

Burkhard, W., and Menon, J., 1993: Disk array storage system reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, 432–441.

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A., 1994: RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, **26(2)**, 145–185.

Data Domain, 2009: DDX Series Arrays data sheet. http://www.datadomain.com/pdf/DataDomain-DDXArraySeries-Datasheet.pdf.

Dell, 2009: EqualLogic PS6000E Spec Sheet. http://www.dell.com/downloads/global/products/pvaul/en/equallogic_ps6000e_specsheet.pdf.

EMC, 2009: Symmetrix DMX series enterprise data sheet. http://www.emc.com/collateral/hardware/data-sheet/c1005-dmx-series-ds.pdf.

Ghandeharizadeh, S., Ierardi, D., and Kim, D., 1996: Placement of data in multi-zone disk drives. In *Proceedings of the Second International Baltic Workshop on Databases and Information Systems*.

Gibson, G., Holland, M., and Zelenka, J., 1996: Raidframe: rapid prototyping for disk arrays. In *1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*.

Holland, M., and Gibson, G., 1992: Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 23–35. ACM.

Hsu, W., Smith, A., and Young, H., 2001: I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Transactions on Database Systems*, **26**(1), 96–143.

Hsu, W., Smith, A., and Young, H., 2005: The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, **23(4)**, 424–473.

IBM, 1999: Datasheet 18LZX disk drive. http://www.hitachigst.com/tech/techlib.nsf/techdocs/85256AB8006A31E587256A79006CA249/$file/dmvs_ds.pdf.

IBM, 2009: DS8000 data sheet and configuration manual. www.ftp://service.boulder.ibm.com/storage/isv/LogicalConfigConceptsDS8000.pdf.

Intel, 1997: Intel architechture optimizations manual. http://download.intel.com/design/PentiumII/manuals/24281603.PDF.

Kandaswamy, M., and Knighten, R., 2000: IO phase characterization of TPC-H query operations. In *Proceedings of Computer Performance and Dependability Symposium*, 81–90. IEEE.

LSI, 2009: Engenio 7900 Storage System. http://www.lsilogic.com/DistributionSystem/AssetDocument/ESG_7900_PB_040609.pdf.

Microsoft, 2001: Benchmarking on Windows XP. http://www.microsoft.com/whdc/archive/benchmark.mspx.

Microsoft, 2004: Fast boot / fast resume design. http://www.microsoft.com/whdc/system/sysperf/fastboot/default.mspx.

Morris, J. M., 2003: High performance disk mirroring. US Patent 6,654,862 B2.

NetApp, 2009: FAS6000 data sheet. http://www.netapp.com/us/products/storage-systems/fas6000/fas6000-tech-specs.html.

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

Pinheiro, E., and Bianchini, R., 2004: Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*.

Ramamurthy, R., DeWitt, D., and Su, Q., 2003: A case for fractured mirrors. *The VLDB Journal - The International Journal on Very Large Data Bases*, **12**(2), 89–101.

Ruemmler, C., and Wilkes, J., 1994: An introduction to disk drive modeling. *IEEE Computer*, **27**(3), 17–29.

Savage, S., and Wilkes, J., 1996: AFRAID: A frequently redundant array of independent disks. In *Proceedings of* USENIX *1996 Annual Technical Conference*.

Schindler, J., Griffin, J., Lumb, C., and Ganger, G., 2002: Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*.

Schlosser, S., Schindler, J., Papadomanolakis, S., Shao, M., Ailamaki, A., Faloustsos, C., and Ganger, G., 2005: On multidimensional data and modern disks. In *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST)*.

Schwarz, T., Steinberg, J., and Burkhard, W. A., 1999: Permutation development data layout (PDDL). In *Proceedings of the 5th Int'l Symposium on High-Performance Computer Architecture*, 214–217.

Seagate, 2004a: Datasheet 10k.7 disk drive. www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah10k.7.pdf.

Seagate, 2004b: The economies of speed. White paper, http://www.seagate.com/docs/pdf/whitepaper/economies_capacity_spd_tp.pdf.

Seagate, 2008: Datasheet 15k.6 disk drive. http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_6.pdf.

Solworth, J., and Orji, C., 1991: Distorted mirrors. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*.

Staelin, C., and Garcia-Molina, H., 1990: Clustering active disk data to improve disk performance. Technical Report CS-TR-283-9, Princeton University.

Storage Networking Industry Association, 2009: Optimizing disk drive performance in storage networks. http://www.snia.org/education/storage_networking_primer/stor_devices/optimizing.

T13 Technical Committee, 2004: AT Attachment with Packet Interface - 7 Volume 1 (ATAPI). http://www.t13.org/Standards/Default.aspx?DocumentType=3.

Talagala, N., Arpaci-Dusseau, R., and Patterson, D., 2000: Microbenchmark-based extraction of local and global disk characteristics. Technical report, University of California, Berkeley.

Theodorides, B., and Burkhard, W., 2006: $\hat{B}$: Disk array data layout tolerating multiple failures. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 21–32.

TPC, 2009: Transaction processing performance council. http://www.tpc.org/information/about/about.asp.

VanMeter, R., 1997: Observing the effects of multizone disks. In *Proceedings of USENIX 1997 Technical Conference*.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T., 1991: The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 96–108. ACM Press.