RAIDframe: A Rapid Prototyping Tool for RAID Systems

William V. Courtright II, Garth Gibson, Mark Holland,
LeAnn Neal Reilly, Jim Zelenka

June 4, 1997
CMU-CS-97-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Redundant disk arrays provide highly-available, high-performance disk storage to a wide variety of applications. Because these applications often have distinct cost, performance, capacity, and availability requirements, researchers continue to develop new array architectures. RAIDframe was developed to assist researchers in the implementation and evaluation of these new architectures. It was designed specifically to reduce the burden of implementation by restricting code changes to mapping, algorithms and other functions that are known to be specific to an array architecture. Algorithms are executed using a general mechanism which automates the recovery from device errors, such as a failed disk read. RAIDframe enables a single implementation to be evaluated in a self-contained simulator, or against real disks as either a user process or a functional device driver.

This research is supported in part by the National Science Foundation through the Data Storage Systems Center, an NSF engineering research center, under grant number ECD-8907068 and an AT&T fellowship. It is also supported in part by industry members of the Parallel DataConsortium, including: Hewlett-Packard, Data General Corporation, Digital Equipment Corporation, International Business Machines, Seagate Technology, Storage Technology, and Symbios Logic.

Keywords: disk array, storage, architecture, simulation, directed acyclic graph, software.

INTRODUCTION

The Importance of RAIDframe to the Research and Development Communities

The demand for high-capacity, high-performance, and highly available data storage has increased as information systems have grown to critical importance in business operations. Given how rapidly the market for Redundant Arrays of Independent Disks (RAID) [Patterson88] is growing [DISK/TREND94], these architectures are clearly the storage technology of choice for meeting this demand.

The increasing importance of RAID systems has led to a number of proposals for new architectures and algorithms, for example, designs emphasizing improved write performance [Menon92, Mogi94, Polyzois93, Solworth91, Stodolsky94]. While many of these proposals are promising, they have been largely evaluated by simulation or analytic modeling. To understand the advantages and limitations of these new designs, it is essential for RAID architects to experiment with concrete implementations.

However, evaluating new designs by introducing them into the marketplace is expensive, slow, and too often unenlightening. Using traditional approaches, implementing redundant disk arrays has been a difficult, manual process. This is evidenced by an inabilty to generate code which is reusable, extensible, and easily verifiable as correct. While these problems prevent RAID researchers and developers from exploring the design space, they also lead to long development times and uncertain product reliability for RAID vendors.

In developing RAIDframe, our primary goal was to decrease design-cycle time by simplifying the process of implementation without sacrificing performance (measured in terms of storage access and response time). We developed a simple programming abstraction from which distinct RAID operations (and therefore, architectures) may be easily implemented in RAIDframe. Once the basic instructions (fewer than a dozen) are implemented, the time required to implement a new RAID operation is simply the time required to write a new program. Error recovery is then mechanized without diminishing performance or increasing overhead—in contrast to traditional approaches which were manual and prone to error [Courtright94].

The programming abstraction RAIDframe uses is based on directed acyclic graphs (DAGs). A designer wishing to introduce a new architecture or optimize an existing architecture will be able to achieve this goal by modifying the library of graphs and graph-invocation rules implemented in RAIDframe. While graphs and the binding of graphs to requests varies widely, the majority of the code in RAIDframe is found in the unchanging DAG interpretation-engine. In this way, designers are encouraged to experiment with and extend various RAID architectures because they can ignore the majority of the code, which is devoted to device-manipulation details.

A particularly powerful feature of RAIDframe is that it separates error recovery from array architecture. The mechanism used to recover from failed primitive operations (such as a disk read) during the execution of an array operation is a part of RAIDframe's internal infrastructure. To do this, RAIDframe uses a two-phase approach to error recovery which we call *roll-away error recovery.* RAIDframe's architecture-independent DAG interpreter handles errors by identifying those nodes in a DAG which commit data to disk and by specifying the direction of recovery based on when errors occur in relation to this commit point.

Specifically, if an error occurs before any data has been committed to disk, then the system rolls back, releasing resources, and retries the operation with a more appropriate graph. On the other hand, if an error occurs after data has been committed, the system rolls forward through the remainder of the graph, giving later requests the impression that this graph completed instantaneously before the error. In either case, this process is hidden from the user and performed without regard to array architecture. Graph commit points can be specified so that roll-back is inexpensive (that is, it does not induce additional device work in preparing for or executing roll-back) and so that roll-forward does not need to execute any device operation not already coded in the in-progress graph. By eliminating the need for architecture-specific code for handling errors, roll-away error recovery further simplifies the process of building new RAID architectures: there is no need to create or alter thousands of lines of error-recovery code.

Currently, RAIDframe acts as a software-only RAID controller for Alpha-based OSF/1 machines. To emphasize our intent to enable real designers to experiment with and use RAIDframe, we have implemented the software so that it can be configured to execute as an event-driven simulator, as a stand-alone application managing disks through the UNIX raw-disk-interface, or as an OSF/1 device driver through which standard UNIX file systems can be mounted and accessed.

RAIDframe's library of architectures includes RAID levels 0 (nonredundant), 1 (mirroring with shortest-queue selection), 4 (centralized parity), 5 (rotated parity), 6 (Reed-Solomon double-failure protection), declustered parity, interleaved declustering, and chained declustering; additionally, variants of some of these support distributed, on-line spare-disk capacity. Preliminary performance analysis shows that RAIDframe's RAID level 0 can keep an array as busy as a much-more-limited direct implementation of disk striping without substantially increasing response time, although RAIDframe requires more processing power to achieve this goal [Gibson, 1995]. Moreover, beginning with the RAID level 0 graphs in its library, well over 90% and frequently 99% of the lines of code in RAIDframe are unchanged by the modifications necessary to implement the architectures listed above. Finally, the roll-away error recovery is fully functional, requiring only that a graph’s commit nodes be marked.

This content in this document can be roughly divided into two categories: *background* and *using RAIDframe*. Background chapters are Chapter One: Redundant Disk Arrays; Chapter Two: Theory of Operation; and Chapter Three: RAIDframe: A Framework for Implementing New Designs. Together, these chapters provide a basic understanding of RAID technology, explain the programmatic abstraction RAIDframe uses for modeling RAID operations, and detail RAIDframe’s features, internal architecture, and supporting libraries. The remaining chapters are Chapter Four: Installing, Configuring, and Using RAIDframe; and Chapter Five: Extending RAIDframe. These last two chapters help provide designers and developers with the necessary information for using RAIDframe.

This document, along with the RAIDframe code, will be continually revised and updated. These updates will be made available on the Parallel Data Laboratory Web pages at the URL http://www.cs.cmu.edu/afs/cs/project/pdl/WWW/Index.html. To be notified when updates are made available, send mail to pdl-webmaster@cs.cmu.edu.

CHAPTER 1

Redundant Disk Arrays: A Brief Overview

In Chapter 1, we will present a brief overview of redundant disk arrays. The text for this chapter was excerpted from Chapter 2 of Mark Holland's thesis, "On-line Data Reconstruction in Redundant Disk Arrays," published in 1994 by Carnegie Mellon University. The text has been edited and updated in minor ways to allow it to fit into the RAIDframe documentation. For a more thorough description of RAID technology, we recommend *The RAIDbook: A Source Book for Disk Array Technology* [RAID96].

1.1 The Need for Improved Availability in the Storage Subsystem

There exist several trends in the computer industry that are driving the design of storage subsystems toward higher levels of parallelism. This means that current and future systems will achieve better I/O performance by increasing the number, rather than the performance, of the individual disks used [Patterson88, Gibson92]. This distinction is important in that, as will be seen, it implies directly the need for improved data availability. This section briefly describes these trends (Sections 1.1.1 through 1.1.3), and shows why they lead to the need for improved availability in the storage subsystem (Section 1.1.4).

1.1.1 The Widening Access Gap

First and foremost, processors are increasing in performance at a much faster rate than disks. Microprocessors are increasing in computational power at between 25 and 30% per year [Myers86, Gelsinger89], and projections for future performance increases range even higher. Gelsinger et. al. [Gelsinger 89] predicts that the huge transistor budgets projected for microprocessors in the 1990s will allow on-chip multiprocessing, yielding a further 20% annual growth rate for microprocessors. Bell [Bell89] projects supercomputer growth rates of about 150% per year.

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

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

1.1.2 The Downsizing Trend in Disk Drives

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

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

These advantages, coupled with very aggressive development efforts necessitated by the highly competitive personal computer market, have caused the gradual demise of the larger drives. In 1994, the best price/performance ratio was achieved using 3-1/2-inch disks, and the 14-inch form factor has all but disappeared. The trend is toward even

[1] Seek time, rotational latency, and transfer rate are defined in Section 1.2.1.

smaller form factors: 2-1/2-inch drives are common in laptop computers [ST9096], and 1.3-inch drives are available [HPC3013]. One-inch-diameter disks should appear on the market by 1995 and should be common by about 1998. At a (conservative) projected recording density in excess of 1-2 GB per square inch [Wood93], one such disk should hold well over 2 GB of data.

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

To summarize, the inherent advantages of small disks, coupled with their ability to provide very high I/O performance through disk-array technology, leads to the conclusion that storage subsystems are, and will continue to be, constructed from a large number of small disks, rather than from a small number of powerful disks. Many trends in the storage industry substantiate this claim. For example, DISK/TREND predicts that the redundant-disk-array market will exceed thirteen billion dollars by 1997 [DISK/TREND94]. Storage Technology Corporation, traditionally a maker of large-form-factor IBM-compatible disk drives, has stopped developing disks altogether and is replacing this product line by one based on disk arrays [Rudeseal92].

1.1.3 The Advent of New, I/O-Intensive Applications

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

1.1.4 Why These Trends Necessitate Higher Availability

The preceding discussion demonstrated that higher degrees of I/O parallelism (an increased number of disks in a storage subsystem) are increasingly necessary to meet the storage demands of current and future systems. The discussion deliberately avoided identifying the specific organizations to be used in future storage systems but made the case that such systems will be composed of a relatively large number of independent disks. However, constructing a storage subsystem from a large number of disks has one significant drawback: the reliability of such a system will be worse than that of a system constructed from a small number of disks because the disk array has a much higher component count.

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

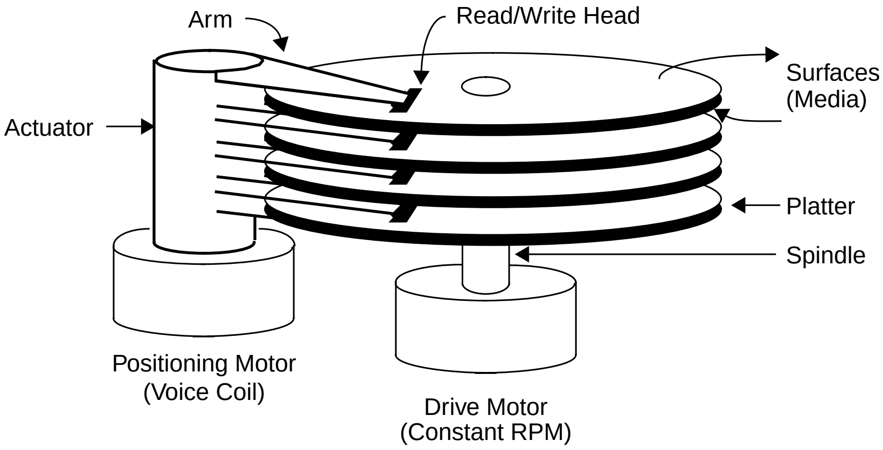
Disk arrays typically incorporate some form of redundancy in order to protect against data loss when these failures occur. This is generally achieved either by *disk mirroring* [Katzman77, Bitton88, Copeland89, Hsiao91], or by *parity encoding* [Arulpragasam80, Kim86, Park86, Patterson88, Gibson93]. In the former, one or more duplicate copies of each user data unit are stored on separate disks. In the latter, commonly known as Redundant Arrays of Inexpensive[1] Disks (RAID) [Patterson88], a portion of the array’s physical capacity is used to store an error-correcting code computed over the data stored in the array. Section 1.2.2 describes both of these approaches in detail. Studies have shown that, due to superior performance on small read and write operations, a mirrored array, also known as RAID Level 1, may deliver higher performance to many important workloads than can a parity-based array [Chen90a, Gray90]. Unfortunately, mirroring is substantially more expensive—its storage overhead for redundancy is 100%, whereas the overhead in a parity-encoded array is generally less than 25% and may be less than 10%. Furthermore, several recent studies [Rosenblum91, Menon92a, Stodolsky94] demonstrated techniques that allow the small-write performance of parity-based arrays to approach and sometimes exceed that of mirroring.

1.2 Technology Background

This section describes the structure and organization of modern disk drives and disk arrays; the subsection on disk technology has been kept to a minimum. Product manuals such as Digital Equipment Corporation’s *Mass Storage Handbook* [DEC86] provide more thorough descriptions of disk-drive technology. This section describes disk-array structure and functionality in more detail because this information is essential to understanding the RAIDframe prototyping tool.

[1] Because of industrial interest in using the RAID acronym and because of their concerns about the restrictiveness of its “Inexpensive” component, RAID is often reported as an acronym for Redundant Arrays of Independent Disks [RAID96].

FIGURE 1 Physical Components of a Disk Drive

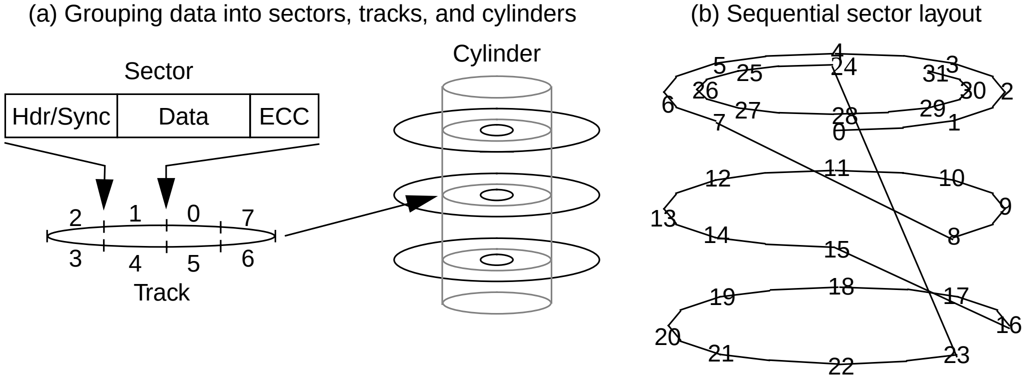


1.2.1 Disk Technology

Figure 1 shows the primary components of a typical disk drive. A disk consists of a stack of platters coated with magnetic media with data stored on all surfaces. The platters rotate on a common spindle at constant velocity past the read/write heads (one per surface), each of which is fixed on the end of a disk arm. The arms are connected to a common shaft called an actuator. Applying a directional current to a positioning motor causes the actuator to rotate small distances in either direction. Rotating the actuator causes the disk heads to move, in unison, radially along the platters, thereby allowing access to a band spanning most of the coated surface of each platter.

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

FIGURE 2 Data Layout on a Disk Drive



(a) Grouping data into sectors, tracks, and cylinders

(b) Sequential sector layout

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

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

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

Figure 2b illustrates the assignment of sequential data to sectors, tracks, and cylinders. Nearly all disks read or write only one head at time, that is, they do not access multiple heads in parallel,[1] and so sequential user data is sequential in any given sector. Thus, as shown in the figure, sequential data starts at sector zero, proceeds around to the end of the track, moves to the next track (which is actually on the underside of the first platter),

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

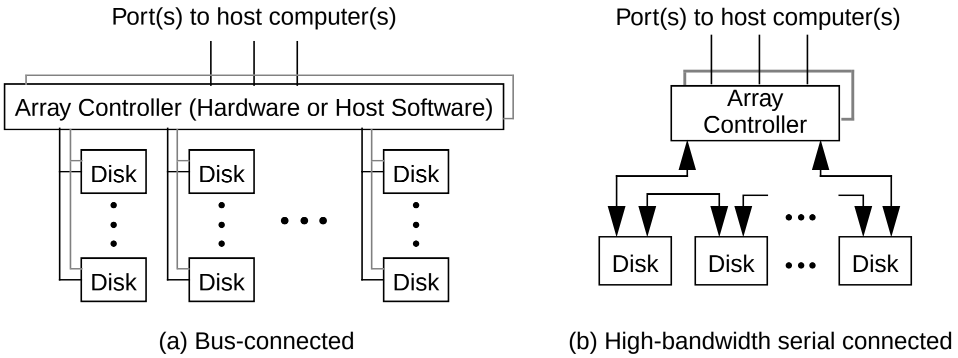
continues this way to the end of the cylinder, and then moves to the next cylinder and starts again. Note that in this example a rotational distance equal to one sector is skipped upon crossing a track boundary (moving from sector 7 to 8), and two sectors are skipped upon crossing a cylinder boundary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

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

1.2.2 Disk-Array Technology

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

FIGURE 3 Disk-Array Architectures


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

(a) Bus-connected

(b) High-bandwidth serial connected

1.2.2.1 Disk-Array Architecture

Figure 3 illustrates two possible disk-array-subsystem architectures. Today’s systems use the architecture of Figure 3a in which the disks are connected via inexpensive, low-bandwidth (e.g., SCSI [ANSI86]) links to an array controller, which is connected via one or more high-bandwidth parallel buses (e.g., HIPPI [ANSI91]) to one or more host computers. Array controllers and disk buses are often duplicated (indicated by the dotted lines in the figure) so that they do not represent a single point of failure [Katzman77, Menon93]. The controller functionality can also be distributed amongst the disks of the array [Cao93].

As disks get smaller [Gibson92], the large cables used by SCSI and other bus interfaces become increasingly unattractive. The system sketched in Figure 3b offers an alternative. It uses high-bandwidth, bidirectional serial links for disk interconnection. This architecture scales to large arrays more easily because it eliminates the need for the array controller to incorporate a large number of string controllers. Further, by making each link bidirectional, it provides two paths to each disk without duplicating buses. Standards for serial-interface disks have emerged (P1394 [IEEE93], Fibre Channel Fibre91], DQDB [IEEE89]) and Seagate has begun shipping drives with serial interfaces. As the cost of high-bandwidth serial connectivity is reduced, architectures similar to that of Figure 3b may supplant today’s short, parallel bus-based arrays.

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

1.2.2.2 Defining the RAID Levels: Data Layout and ECC

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

Fundamental to all disk arrays is the concept of *striping* consecutive units of user data across the disks of the array [Kim86, Livny87, Patterson88, Gibson92, Merchant92]. Striping is defined as breaking up the linear address space exported by the array controller into blocks of some size and assigning the consecutive blocks to consecutive disks rather than filling each disk with consecutive data before switching to the next. The *striping unit* (or *stripe unit*) [Chen90b] is the maximum amount of consecutive data assigned to a single disk. The array controller has the freedom to set the striping unit arbitrarily; the unit can be as small as a single bit or byte, or as large as an entire disk. Striping has two benefits: automatic load balancing in concurrent workloads and high bandwidth for large sequential transfers by a single process.

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

Arrays achieve high data rates in low-concurrency workloads by striping at a finer grain, for example, one byte or one sector. Such arrays are used when the expected workload is a single process requesting data in very large blocks. Fine-grain striping assures that each access uses all the disks in the array, which maximizes performance when the workload concurrency (number of processes) is one[1]. After the initial seek and rotational delay penalties associated with each access, a fine-grain-striped array transfers data to or from the CPU at *N* times the rate of a single disk. Therefore, a fine-grain-striped array can service only one I/O at any one time but is capable of reading or writing the data at a very high rate.

Patterson, Gibson, and Katz [Patterson88] classified redundant disk arrays into five types, called RAID Levels 1 through 5, based on the organization of redundant information and the layout of user data on the disks. This terminology has gained wide acceptance [RAID93] and is used throughout this dissertation. The term “RAID Level 0” has since entered common usage to indicate a non-redundant array. Figure 4 illustrates the layout of data and redundant information for the six RAID levels. The remainder of this section briefly introduces each of the levels, and subsequent sections provide additional details.[2]

RAID Level 1, also called *mirroring* or *shadowing*, is the standard technique used to achieve fault-tolerance in traditional data-storage subsystems [Katzman77, Bitton88]. The disks are grouped into mirror pairs, and one copy of each data block is stored on each of the disks in the pair. To unify the taxonomy, RAID Level 1 defines the user data to be block-striped across the mirror pairs, but traditional mirrored systems instead fill each disk with consecutive user data before switching to the next. This can be thought of as setting the stripe unit to the size of one disk. RAID Level 1 is a highly reliable organization since the system can tolerate multiple disk failures (up to *N*/2) without losing data, so long as no two disks in a mirror pair fail. It can be generalized to provide multiple-failure tolerance by maintaining more than two copies of each data unit. Its drawback is that its cost per megabyte of storage is at least double that of RAID Level 0.

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

[2] Editor's Note: Mark Holland's thesis did not include a description of RAID Level 6, a level which offers protection from two disk failures.

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

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

In RAID Level 3, user data is bit- or byte-striped across the data disks, and a simple parity code is used to protect against data loss. A single check disk (called the *parity disk*) stores the parity (cumulative exclusive-or) over the corresponding bits on the data disks. This reduces the capacity overhead for redundancy to 1/*N*. When the controller identifies a disk as failed, it can recover any unit of lost data by reading the corresponding units from all the surviving disks, including the parity disk and XORing them together. To see this, assume that disk 2 in the RAID Level 3 diagram within Figure 4 has failed, and note that

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

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

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

FIGURE 4 Data and Redundancy Organization in RAID Levels 0 through 5

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

RAID Level 0: Nonredundant

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

RAID Level 1: Mirroring

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

RAID Level 2: Hamming-Code ECC

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

RAID Level 3: Byte-Interleaved Parity

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

FIGURE 4 Cont. Data and Redundancy Organization in RAID Levels 0 through 5

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

RAID Level 4: Block-Interleaved Parity

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

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

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

loop circuitry to synchronize the rotation of the spindles of the disks comprising the array. Many disks currently on the market support this spindle synchronization.

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

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

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

In RAID Level 5, there are a variety of ways to lay out data and parity such that parity is evenly distributed over the disks [Lee91]. The structure shown in Figure 4 is called the *left-symmetric* organization and is formed by first placing the parity units along the diagonal and then placing the consecutive user data units on consecutive disks at the lowest available offset on each disk. This method for assigning data units to disks assures that, if there are any accesses in the workload large enough to span many stripe units, the maximum possible number of disks will be used to service them.

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

1.2.2.3 Reading and Writing Data in the Different RAID Levels

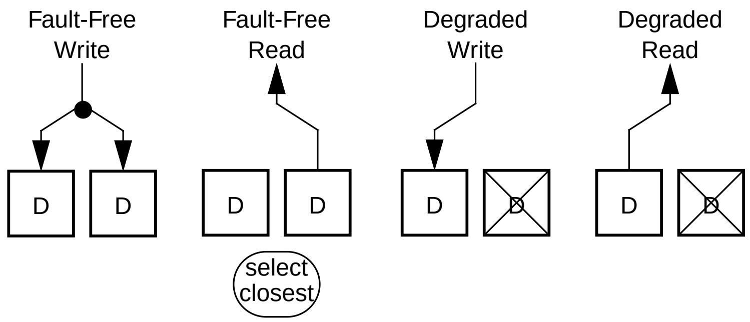
This section describes the techniques used to read and write data in the different RAID levels, both when the array is fault-free (“fault-free mode”) and when it contains a single failed disk (“degraded mode”). The focus is on the techniques used to maintain parity and to continue operation in the presence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

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

1.2.2.3.1 RAID Level 1

Figure 5 illustrates the different read and write operations in RAID Level 1. In fault-free mode, the controller must send user write operations to both disks. This reduces the maximum possible write throughput to 50% of that of RAID Level 0. The two write operations can, in general, occur concurrently, but some systems perform them sequentially in order to guarantee that the old data will be recoverable should the first write fail.

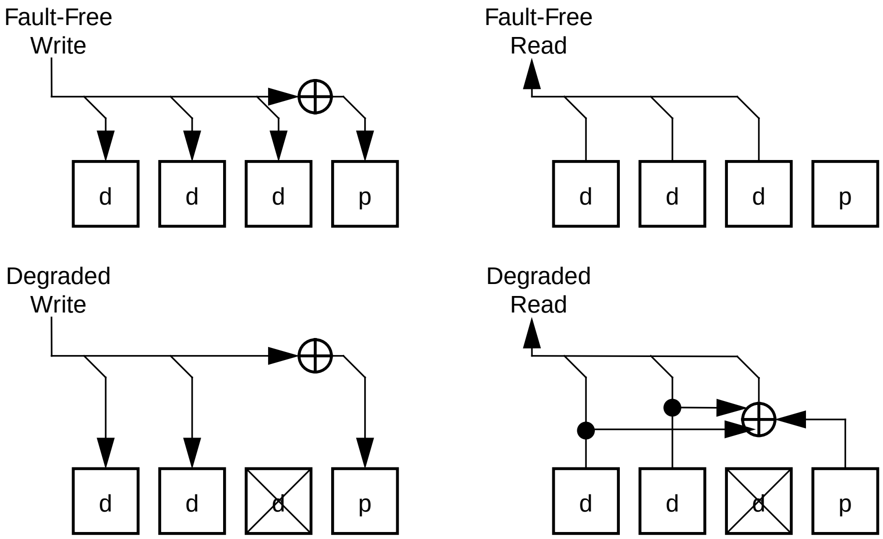
FIGURE 5 Read and Write Operations in RAID Level 1 (mirroring)



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

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

FIGURE 6 Read and Write Operations in RAID Level 3 (bit-interleaved parity)



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

1.2.2.3.2 RAID Level 3

Figure 6 illustrates reads and writes in RAID Level 3. The following discussion assumes that each user access is some multiple of *(N-1)·S* in size, where *N* is the number of disks in the array and *S* is the number of bytes in a sector (almost always 512). This is because each access uses all data disks, and the minimum sized unit that can be read from or written to a disk is one sector. If the array is to support accesses that are not a multiple of this size, the controller must handle any partial-sector updates via read-modify-write operations, which can degrade write performance.

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

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

1.2.2.3.3 RAID Level 5

Figure 7 illustrates the various translations of user accesses to disk accesses in RAID Level 5. User write operations in fault-free mode are handled in one of three ways, depending on the number of units being updated. In all cases, the update mechanisms are designed to guarantee the property that after the write completes, the parity unit holds the cumulative XOR over the corresponding data units, or

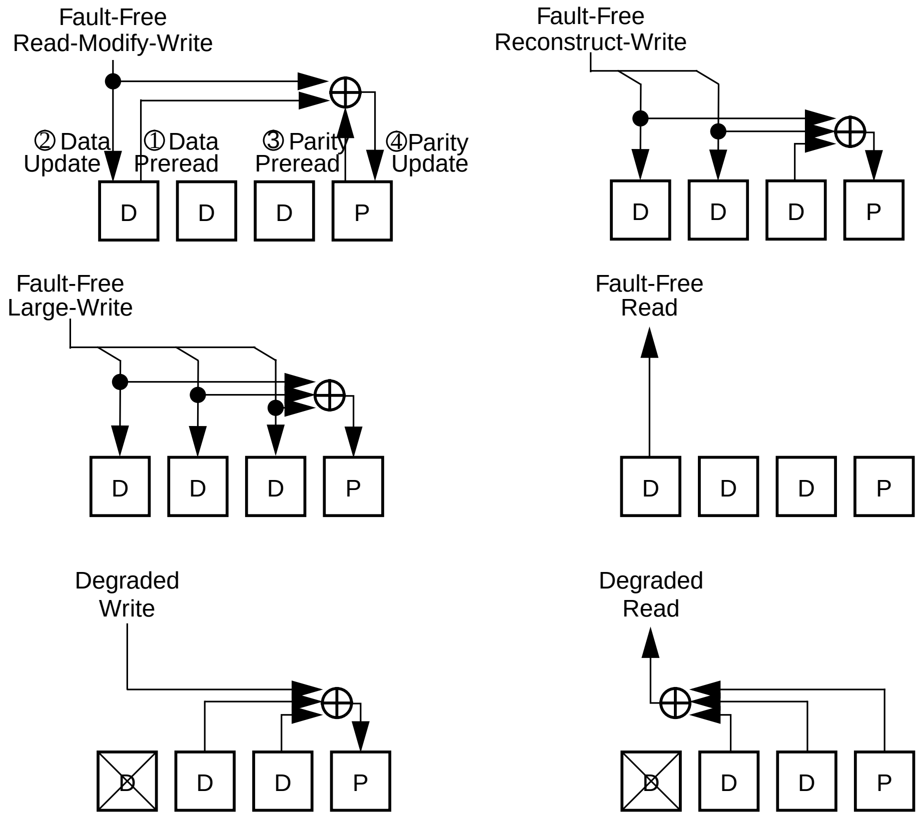
$$P_{new} = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_{N-1}$$

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

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

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

FIGURE 7 Read and Write Operations in RAID Level 5 (rotated parity)



The preread-and-then-write operation performed on the data unit is typically done atomically to minimize the positioning overhead incurred by the access [Stodolsky94]. This is also true for the parity unit. Since the old data must be available to perform the parity update, the data preread-and-write is typically allowed to complete (atomically) before the parity preread-and-write is started.

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

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

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

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

1.2.2.4 Comparing the Performance of the RAID Levels

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

1.2.2.5 On-line Reconstruction

The preceding has shown how a disk array operates, and how it may continue to operate in the presence of a single disk failure. The next step to take is that the array should have the ability to *recover* from the failure, that is, restore itself to the fault-free state.[1] Further, a disk array should be able to effect this recovery without taking the system off-line. This is implemented by maintaining one or more on-line spare disks in the array. When a disk fails, the array switches to degraded mode as described above but also invokes a *background reconstruction process* to recover from the failure. This process successively reconstructs the data and parity units that were lost when the disk failed and stores them on the spare disk. The mechanism by which this is accomplished is called the *reconstruction algorithm*. Once all the units have been recovered, the array

[1] Editor’s Note: The term “recovery” traditionally encompasses more than the process of the array restoring itself to the fault-free state following a single disk failure: it also includes the process by which the array controller handles software errors during operation. Mark Holland limited the term here, however, to the specific case of reconstructing data lost on a failed disk. To clarify this distinction further: recovering from the physical loss of a disk can take the array anywhere from several minutes to several hours. Handling errors, on the other hand, will take the array milliseconds, occurring transparently to the host or user. Automating error recovery is central to our design of RAIDframe and is covered in greater detail in Chapter 2. To lessen confusion, we will use the term “recovery” in its broader sense throughout the rest of the document.

returns to normal performance and is once again single-failure tolerant, and so the recovery is complete.

TABLE 1. First-Order Comparison Between the RAID Levels for an *N*-disk Array

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

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

1.2.2.6 Related Work: Variations on These Organizations

This section summarizes industrial and academic research on disk arrays. It defines nine categories of investigation and presents brief summaries of some papers in each. These studies serve as background in the area of redundant disk arrays.

1.2.2.6.1 Multiple-Failure Toleration

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

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

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

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

1.2.2.6.2 Addressing the Small-Write Problem

Recall from Section 1.2.2.3.3 that small write operations in RAID Level 5 incur up to four disk operations: data preread, data write, parity preread, and parity write. This degrades the performance of small write operations by a factor of four when compared to RAID Level 0. Several organizations have been proposed to address this problem.

Menon and Kasson [Menon89, Menon92a] proposed a technique based on *floating* the data and/or parity units to different disk locations upon each update. Normally, the controller services a small write operation by pre-reading the old data, waiting for the disk to spin through one revolution, writing the new data back to the original location, and then repeating this process for the parity unit. In the floating data/parity scheme, the controller reserves (leaves unoccupied) some number of data units on each track of each disk. After each preread operation, the array controller writes the new data to a rotationally convenient free location rather than writing it in place. This saves up to one full rotation (10-17 milliseconds of disk time) per preread-write pair. An analytical model in the paper shows that a free unit can typically be found within about two units of the location of the old data. This makes each preread/write pair take only slightly longer than a single access and thus can potentially nearly double the small-write performance of the array. Menon and Kasson concluded that the best capacity-performance tradeoff is achieved by applying this floating only to the parity unit rather than to both data and parity. A potential problem with this approach is that the array controller must be intimately familiar with the geometry and performance characteristics of the component disks as well as the latencies involved in communicating with them. This requires a high degree of predictability from the disks and makes the design difficult to verify, tune, and maintain.

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

Stodolsky et. al. [Stodolsky94] adapted the ideas behind LFS to the problem of parity maintenance and proposed an approach based on logging the parity changes generated by each write operation rather than immediately updating the parity upon each user write. In this scheme, the controller reads the old data (or acquires it from the buffer cache) and writes the new data as before. It then XORs together the old and new data to produce a *parity-update record*, which it appends to a write-only buffer rather XORing it with the old parity. The controller spills the entire buffer to disk when it becomes full. No parity operations are performed for each user write, but some of the array’s capacity (about one disks’ worth) must be reserved to hold the parity update logs. Eventually the log space in the array becomes full, at which time the controller empties it by reading the log records and the corresponding parity units, XORing them together, and writing the result back out to the parity locations. Note that the controller buffers only parity information and so is not vulnerable to data loss due to power failure. While in RAID Level 5 parity is updated using a large number of small, random accesses, in parity logging it is updated using a smaller number of large, sequential accesses. The paper showed simulation results indicating that this technique can allow the performance of RAID Level 5 arrays to approach, or under certain conditions even exceed, that of mirroring.

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

1.2.2.6.3 Spare-Space Organizations

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

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

1.2.2.6.4 Distributing the Functionality of the Array Controller

The existence of a centralized array controller in both of the architectures shown in Figure 2 has two disadvantages: it constitutes either a single point of failure or an expensive system resource that must be duplicated, and its performance and connectivity limit the scalability of the array to larger numbers of disks. Cao et. al. [Cao93] described a disk-array architecture they call *TickerTAIP* that distributes the controller functionality amongst several loosely coupled controller nodes. Each node controls a relatively small set of disks (one SCSI string, for example) and communicates with the other nodes via a small, dedicated interconnect network. Under the direction of the distributed controllers, data and parity units as well as control information pass through the interconnect to effect the RAID read and write algorithms. The paper demonstrated the elimination of several performance bottlenecks through the use of the distributed-control architecture.

1.2.2.6.5 Striping Studies

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

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

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

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

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

Lee and Katz [Lee91] described several different strategies for placing the parity units amongst the striped data units. They found that the most significant performance effect of varying parity placement was the number of disks used for large reads and writes; some placement strategies caused fewer than the maximum number of possible disks to be used on large accesses, and these suffered in performance. The left-symmetric parity placement illustrated in the RAID Level 5 case of Figure 4 was among the best of the options.

Merchant and Yu [Merchant92] noted that it is common for a database workload to consist of two components: transactions and ad hoc, read-only queries into the database. Transactions generate small, randomly distributed accesses into the array, whereas the ad hoc queries often scan significant portions of the database. To efficiently handle this workload combination, they proposed a dual striping strategy for mirrored arrays where the size of the stripe unit is small in one copy (4 KB) and large in the other (32 KB). The authors note that using a large stripe unit is efficient for relatively large accesses because it reduces the number of actuators used, but under a small-access model it can cause workload imbalance amongst the disks. They assert that the converse is true as well: a small stripe unit achieves good workload balance but causes too many actuators to be used per large access. Thus they service the transactions using the small-stripe-unit copy of the data and the ad hoc queries with the large-stripe-unit copy. Merchant and Yu evaluated this organization, using both analytical modeling and simulation, with a synthetically generated workload that adhered to the assumptions made in designing the striping strategy. They found substantial benefits to this approach.

1.2.2.6.6 Disk-Array Performance Evaluation

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

1.2.2.6.7 Reliability Modeling

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

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

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

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

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

1.2.2.6.8 Improving the Write-Performance of RAID Level 1

As shown in Table 1, mirrored systems achieve only 50% of the write performance of nonredundant arrays because each write must be sent to two disks. This section describes several studies intended to improve this performance. Most of the ideas here relate to caching and deferring updates and so apply to parity-encoded arrays as well.

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

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

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

1.2.2.6.9 Network File Systems Based on RAID

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

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

CHAPTER 2

Managing the Complexity of Array Software

In Chapter 1, we described the need for improved availability in the storage subsystem due to the widening access gap, the downsizing trend in disk drives, and the advent of new I/O-intensive applications. We discussed the structure and operation of disk arrays in some detail, explaining the different data layouts and fault tolerance for each of the original RAID levels. We also summarized some of the related work done on variations of these RAID organizations, most of which looks at improving performance by identifying the best techniques for laying out and writing data.

What should be clear from our description of disk arrays in Chapter 1 is the complexity of the array software used to control the disks in the array. What may not be clear from our discussion is that most of the related work has approached the task of managing this complexity on a case-by-case basis. What we mean by this is that researchers have looked at specific contexts for using redundant arrays and have proposed ways to optimize the software based on the specific needs of expected workloads. This ad hoc approach to designing and implementing array software means that there is little code reused between RAID organizations. It also means that each architecture handles any errors that occur during operation in a specific, limited way, adding to the complexity of the array software.

Our goal is to simplify the process of designing and implementing array software that performs optimally for a particular situation. To do this, we have aimed to increase the amount of code reused between RAID designs, to enable a means for verifying the correctness of designs before they are implemented, to generalize an error-recovery mechanism, and to provide a mechanism for reconstructing data on-line when a disk fails. Achieving these four things, we believe, will lead to shorter design-cycle times, software that performs as it was designed to do, mechanized error recovery, and highly available and reliable systems.

In this chapter, we introduce a structured method for implementing array software, based on a graphical programming abstraction, which allows many RAID operations to be composed quickly from a relatively small set of primitive operations. We begin by looking in more detail at traditional approaches to managing array software in Section 2.1, then move to the concept underlying our own structured approach in Section 2.2: that RAID operations can be viewed as software programs. Next, we describe how to compose these RAID operations, or programs, with graphs in Section 2.3 before discussing how to execute them in Section 2.4. Finally, in Section 2.5 we discuss an algorithm for reconstructing data when a disk fails.

2.1 Traditional Approaches in Managing Array Software are Suboptimal

As we have already said, redundant arrays have typically been designed in an ad hoc fashion, each organization developed to address particular needs and each customized to handle specific error conditions. It is this customized error recovery that has particularly added to the complexity of array software and that has contributed to the difficulty in managing array software. Traditionally, array designers have adopted one of two approaches to error recovery: forward error recovery and backward error recovery.

Briefly, forward error recovery requires anticipating all possible errors and manually coding actions for completing operations once an error has occurred. This approach requires hundreds of thousands of lines of code with the possibility of overlooked errors. While custom-designed code from a complete understanding of all error vectors allows the software to achieve near-optimal performance, the error-recovery code must be re-written to handle a new set of error vectors if the code is to be reused for a similar but distinct application. As long as the set of vectors is relatively small, this task is not too difficult and, in fact, this approach is the dominant method of error recovery in general-production software.

Of course, many of the error vectors may be consolidated and treated similarly, reducing the number of unique cases which must be handled. For example, if parity has failed in the middle of a large write operation, the remaining data writes may continue unaffected, regardless of their current disk state (old or new). However, this does not eliminate the problem of extending existing code to support new array operations. This is because the remaining error vectors are still a function of error context and as new array operations are introduced, that context will change, thereby requiring changes in error-recovery code.

Finally, verifying code constructed in this fashion can be tedious and prone to mistakes. To demonstrate that it is correctly implemented, each RAID operation must satisfy a set of *invariants*, rules which are always true for a consistent array. Ensuring correctness requires identifying each error scenario and demonstrating that the code correctly handles each error vector. Automating this process is possible if the code structure is well defined, perhaps in the form of a state machine [Clarke82, Clarke94]. However, because of the ad hoc nature of code using forward error recovery, hand analysis is required.

In an ideal world, redundant-disk-array software would be constructed without regard for the context in which errors occur. This implies that when, for example, a disk read fails, only the very general process of recording the fact that a disk has failed would need to be implemented. The implications of the error (e.g., failure to read non-overwrite data during a “reconstruct write”) would be irrelevant, making the software completely independent of array architecture.

Database systems have achieved this simplicity, allowing programmers to create new transactions with little regard for error recovery. This is accomplished by guaranteeing that the operations which compose the transaction are atomic and undoable. When an error occurs which causes an atomic operation to fail, the programmer is presented with the illusion that the operation never occurred. Furthermore, the *system* undoes the effects of the previously completed operations, completely removing all effects of the failed transaction. With the burden of detecting and recovering from errors delegated to the underlying system, the programmer is left with the relatively straightforward task of creating transactions which begin in a consistent system and commit only consistent state changes to the system.

The approach used to achieve this simplicity, backward error recovery, requires a durable log which records the effects of operations as they complete. When it is determined that a transaction has failed, the contents of the log are used to undo the previously completed operations. Unfortunately, maintaining this log may be expensive—in addition to the resources required to store the log, additional work may be required to create the information which is stored in the log.

For example, consider a large write operation in a RAID level 5 array which overwrites data and parity with new information. To guarantee that each of these write operations is undoable, the previous contents of the data and parity must be stored in the log. Instead of just overwriting each one, each disk operation must now read and write data and parity, doubling the total workload of the disks and decreasing the response time and throughput of the system. If a disk operation fails, then the saved state is restored; and, while the system restores state, processing stops.

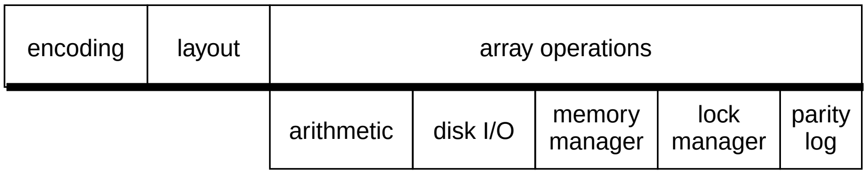
Our strategy is to address the limitations of both forward error recovery and backward error recovery and to provide criteria for using each, thereby enabling error recovery to be automated, transparent, and verifiably correct. Specifically, forward error recovery is easy if no case analysis is required; backward error recovery is easy if there is no state to save and restore. We call our approach *roll-away error recovery* because it is a hybrid approach. We will describe how roll-away error recovery works in more detail in Section 2.4.3 on page 53.

2.2 Treating RAID Operations as Programs

As we discussed in Section 1.2.2.3 on page 26, the array controller—however implemented—maps user read and write operations (such as *small write* and *degraded-mode read*) to a relatively small set of corresponding disk operations. These operations, which we will refer to as *primitive operations* throughout the remainder of this document, include operations for disk access (such as *disk read* and *exclusive-or*), redundancy computation, and resource allocation (such as memory buffers). Because primitive operations are the basic actions used by the array software to control disks, they can be thought of as instructions or steps, and when constraints upon their sequencing are imposed, they can be used to construct RAID operations in a programmatic fashion.

By treating RAID operations as programs, we are able to minimize the amount of code changes required to extend the software. The best-known method to do this is to create modular code which isolates functions that are known to change orthogonally with architecture [Meyers78].

FIGURE 8 Isolating Common Infrastructure



Infrastructure code, which provides the primitive operations from which array operations are implemented, appears in the lower half. Architecture-specific code, such as data encoding, appears in the upper half. When a new architecture is implemented, the infrastructure is unchanged, restricting changes to modules which contain array-specific code.

As Figure 8 shows, the most obvious functions which vary with array architecture are data encoding, information layout, and operation structure. For example, recall from Chapter One that the only difference between RAID levels 4 and 5 is the manner in which information is distributed across the disks in the array. By isolating device-specific code from the code which defines the array architecture and by requiring that the device software handle all device-specific errors, we are able to provide an infrastructure which allows array designers to build a variety of architectures without thinking about the underlying device actions.

In order to understand how primitive operations can be used to compose RAID operations, we will first look at the set of primitive operations most commonly used. In Section 2.2.1 through Section 2.2.2 we will then describe how to create pass-fall primitives and how to create RAID operations from primitive operations.

2.2.0.1 Primitive Operations Commonly Used in Redundant Disk Arrays

In addition to disk drives, the most popular devices used to construct arrays being sold today include: memory managers, lock managers, arithmetic units, and parity logs. Table 2 summarizes the primitive operations provided by these devices and their effects.

These devices may be constructed from either hardware, software, or some combination.

TABLE 2. Common Device Operations

Device	Primitive Operation	Effect
disk	disk read	copy data from disk to buffer
disk	disk write	copy data from buffer to disk
disk	Rd	copy data from disk to buffer
disk	Wr	copy data from buffer to disk
memory manager	MemA	acquire a buffer
memory manager	MemD	release a buffer
lock manager	Lock	acquire a lock
lock manager	Unlock	release a lock
arithmetic	XOR	xor contents of buffers
arithmetic	Q	generate a Reed-Solomon code
arithmetic	$\overline{\text{Q}}$	Reed-Solomon decode
read cache	probe	if hit, return, shared lock and pointer
read cache	copy	copy data from cache to a buffer

A memory manager is used to negotiate the use of a buffers from a shared pool. Similarly, the lock manager maintains a set of locks, granting either shared or exclusive ownership to competing processes. The arithmetic unit provides operations which perform data encoding and decoding functions, such as bitwise exclusive-or which is used in parity encodings and nonbinary polynomial multiplication which is used in Reed-Solomon encodings. The parity log is an append-only log used to accumulate either parity-update or parity-overwrite records.

2.2.1 Creating Pass-Fail Primitive Operations

Before we can automate array error recovery transparently, it is necessary for us to distinguish between errors at the device level and those at the array level. Isolating device-specific recovery from array-specific error recovery enables us to create RAID operations without regard for the internal details of the devices. To do this, we abstract primitive operations with a wrapper that is responsible for creating the illusion of pass-fail devices in which *pass* implies successful completion and *fail* implies the presence of a permanent fault [Courtright94].

By allowing primitive operations to return *fail* only when an unrecoverable device fault is detected, we are further able to restrict the class of errors observable by RAID operations to those which require handling at the array level. Otherwise, primitive operations return *pass*, completely hiding from RAID operations the effects of any device faults which may have been detected. When primitive operations do fail, we want them to fail atomically (i.e., all-or-nothing state changes), but we don’t require it. We will defer discussing how nonatomic failure is handled until Section 2.4.3, which describes error recovery.

To ensure that *pass* implies that a primitive operation has successfully completed, we allow primitive operations to commit only those state changes which are consistent with their behavior. For example, a disk write is required to write the correct ECC information to disk when writing data to a sector.

2.2.2 Constructing RAID Operations from a Set of Primitive Operations

As we have already said, RAID operations are composed from a relatively small set of primitive operations; the order in which primitive operations are executed is solely a function of the data and control dependencies which exist between them. Therefore, it is important for the array designer to know the location of necessary dependencies which exist between primitive operations in order to design RAID operations well. Omitting dependencies will result in erroneous behavior while extra dependencies may reduce concurrency and unnecessarily degrade performance. Table 3 lists the four basic types of dependencies which may exist between primitive operations.

TABLE 3. Ordering Constraints Imposed on Sequences of Primitive Operations

Dependence	Explanation
True	read after write data dependence
Anti	write after read data dependence
Output	write after write data dependence
Control	dependence of a primitive operation upon the completion of another

2.2.3 Summary

Defining an array operation is a straightforward process: our primary concern is to abstract device-specific operation from the array-specific operation, which is the external interface of the operation. To do this, we have required that primitive operations be responsible for detecting all faults and for tolerating those faults which are specified to be tolerable by the device fault model. Primitive operations which complete successfully, either by avoiding or tolerating a device fault, return *pass* to indicate success. Primitive operations return *fail* only when they are unable to recover from a fault. To compose RAID operations, the array designer must know where dependencies exist between primitive operations.

2.3 Representing RAID Operations as Graphs

Creating storage operations from a set of primitive operations is a technique which has been used for more than twenty years. The best-known example of this is the *channel-program* approach used in the IBM System/370 architecture [Brown72]. At the time it was introduced, much of the internal workings of a disk drive were exposed to the system, requiring external control of arm positioning, sector searching, and data transfer. Channel programs isolated these details from users by providing an abstract interface which was closer to that found in today's SCSI drives [ANSI91]. The programs are represented as a linear array of primitive operations which is parsed sequentially.

Similar methods for abstracting the details of storage operations were recently proposed in the distributed, redundant-disk-array architecture called TickerTAIP [Cao94]. In TickerTAIP, the work required to maintain valid data encodings is performed by *workers* which are distributed throughout the array. To simplify managing simultaneous primitive operations occurring across the array, TickerTAIP uses a centralized table in which each entry contains a list of operations for a worker to execute. Once an array operation is initiated, each worker is responsible for sequencing its own activities. Unlike channel programs, TickerTAIP achieves parallelism within an array operation because multiple workers may execute primitive operations concurrently.

These two examples clearly show that it is possible to construct RAID operations from a set of primitive operations using tables. However, we believe that there is better approach based upon *directed, acyclic graphs (DAGs)* which will allow designers to reason about the ordering of primitive operations. Because we have decided to treat RAID operations as programs, we are able to use DAGs to model primitive operations and the ordering constraints which bind them together—the visual information supplied by DAGs is intuitive and aids in analyzing the design of RAID operations. The following subsection describes how DAGs are created.

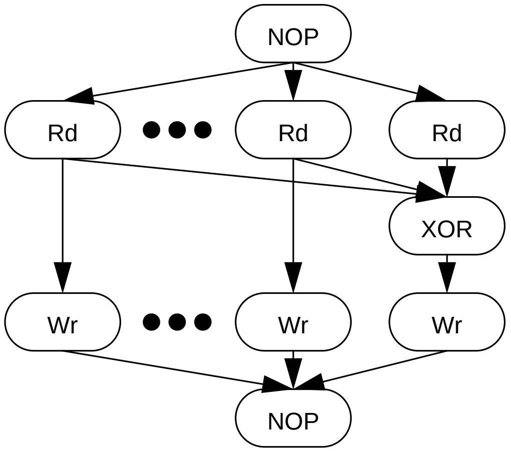
2.3.1 Directed, Acyclic Graphs (DAGs)

When using DAGs to model RAID operations, the primitive operations described in Table 2 on page 45 are represented as the nodes of the graph. Figure 9 illustrates a small write operation represented as a directed acyclic graph. Each primitive operation is represented by a single node and therefore the properties of a node (e.g., atomic failure) are inherited from the defining properties of the primitive operations.

Notice that the nodes in the graph of Figure 9 do not convey the context (e.g., “read old parity”) of each primitive operation. This is because the context is known only by the designer of the graph. Section 2.4.3 shows how we capitalize upon this independence of context to achieve mechanized execution.

As we already said in Section 2.2.2 on page 46, executing primitive operations within an array operation is constrained by the presence of control and data dependencies. Dependencies are represented in a DAG by the directed arcs which connect the nodes of the DAG. An arc is drawn from a parent node to a child node if executing the child is dependent upon the parent node. Because the type of dependence represented by the arcs will not be used to control execution, the arcs are left unlabeled. Furthermore, a single arc may represent the presence of one or more data or control dependencies. We defer discussing further the rules for executing DAGs until Section 2.4 on page 51.

FIGURE 9 RAID Level 4/5 Small-Write Graph



This illustration presents the small write operation. The nodes of the graph are pass-fail actions and the arcs represent the presence of control or data dependencies.

In this graph, the Rd-XOR-Wr chain on the far right performs the read-modify-write of parity. The Rd-Wr chains represent the reading of old data and the overwriting of new data. The fact that parity is computed from the old data is represented by the presence of the Rd-XOR arcs (true data dependencies). The Rd-Wr arcs represent anti (read after write) data dependencies. Finally, a NOP (no-operation) node has been added to simplify the structure of the graph, guaranteeing a single sink (tail) node.

2.3.2 Simplifying Constraints for DAGs

There are a number of constraints which we have imposed on DAGs to simplify executing them. First, a node that is a direct descendent of a predicate node may have no parents other than the predicate node. Second, because DAGs are by definition acyclic, there cannot be any cycles in RAID operations; eliminating cycles does not eliminate predicate nodes and conditional execution.[1] An array designer can include a node which selectively enables one or more branches for execution. Finally, all DAGs must be *rooted graphs*, meaning that all graphs begin with a single *root* or *source node*. The source node has the property that it has no parents. Similarly, all DAGs must have a single sink node, a node which has no children. If a graph does not contain a single source or sink node, a **NOP** (no operation) node can be inserted. Adding an extra **NOP** node to create a single source or sink does not have any effect upon the array operation represented by the graph.

[1] The current release of RAIDframe does not contain predicate nodes or support their processing.

Besides modeling RAID operations, we have also incorporated automated roll-away error recovery into the DAG structure. The following section describes the added requirements for structuring DAGs to enable them to handle errors when the array operates.

2.3.3 Incorporating Roll-Away Error Recovery Within DAGs

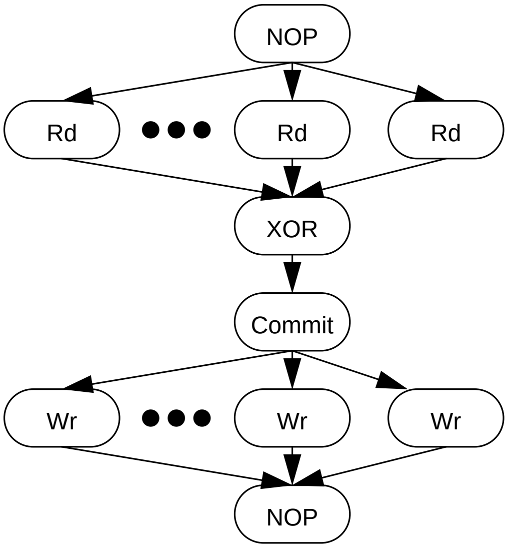
As we said earlier, roll-away error recovery is a hybrid approach: when appropriate, it uses forward error recovery without accounting for all possible error scenarios; when necessary, it uses backward error recovery without the cost of logging state information. A more detailed discussion of roll-away error recovery can be found in William V. Courtright II's dissertation, which is currently in progress. Here we will explain the basic method for mechanizing error recovery through the structure and composition of DAGs.

To understand how roll-away error recovery works, it is important first to recall that redundant arrays encode data to survive disk faults (See "Why These Trends Necessitate Higher Availability" on page 15). Codewords are composed of two types of symbols: one for data, the other for a check, for example parity. In order for redundant arrays to tolerate faults—meaning the loss of one or more symbols without losing information, the set of valid codewords is constrained. Primitive operations change data symbols, for example, they write new data; this in turn requires modifying the corresponding check symbols, that is, they must then write new parity. If a primitive operation fails before it has completed—that is, one or more symbols have been modified on disk—the codeword can be left in one of a large number of states.

Because the direction of error recovery depends upon when a primitive operation fails, it is essential to determine where in the RAID operation all modified symbols can be safely committed to disk. To establish this place, which we call the *commit barrier*, we have divided RAID operations into two phases in which codewords are modified only in phase two. Within the DAG structure, we add a Commit node to distinguish between these two phases.

In the first phase no existing codewords can be modified; here, nodes within a DAG represent primitive operations that can generally be undone easily, such as *disk read* or *XOR*. Obviously, the second phase of a RAID operation is where we place those primitive operations that modify symbols—however, not all RAID operations have two phases. For example, because a read operation does not modify any codewords, it does not have a phase two. On the other hand, a write operation (shown in Figure 10) clearly modifies codewords; in order for the write operation to progress to phase two, all symbols which are to be updated must be available. Section 2.4.3 on page 53, which follows a discussion of how DAGs are executed, explains how the error-recovery mechanism automatically executes when an error is detected.

FIGURE 10 RAID Level 4/5 Small-Write Graph with Commit Node



A Commit node was inserted to prevent writes of new data from proceeding until all reads of old data and the computation parity have been completed.

To establish the commit barrier when constructing a DAG for a RAID operation, the array designer must first identify all those nodes which modify a symbol. Next, the designer must create control dependencies from the nodes' parents to the nodes themselves. This will guarantee that no symbols will be modified until all modified symbols can be safely committed to disk. In short, commit nodes are generally the sink node of read operations and the parent of all symbol update actions which are found in write operations.

2.3.4 Verifying the Correctness of DAGs

Because we model RAID operations as well-structured graphs, correctness verification—that is, the process of demonstrating that an array's behavior is consistent with its specified behavior—is greatly simplified. Furthermore, automating this task is now possible. Given that DAGs consist of well-defined primitives, it is possible to think of them as state machines. Through model checking, used to verify the correctness of state machines [Clarke82, Clarke94], RAID designs can be verified immediately, long before actual implementation begins [Wing96].

Verifying that RAID operations are correctly implemented requires that graphs meet three criteria. First, primitive operations must be valid. Second, valid codewords for

RAID operations must be maintained; for example, to maintain valid parity for RAID Level 5, the sum of the parity bits must always equal 0. And third, graphs must recover from errors using roll-away handling, which we describe briefly in Section 2.4.3.

2.4 Executing RAID Operations

Array operations modeled as DAGs may be executed directly without first being translated into an intermediate form. More importantly, modeling with graphs has enabled us to simplify and automate error recovery. To do this, we employ an undo-redo error recovery scheme, similar to the one used in the System R recovery manager [Gray81]. In our approach, if a primitive operation fails at any time during the execution of a graph, the execution mechanism will automatically undo the effects of the previously completed primitives.

In this section, we describe node states and their transitions, how to execute a graph, and how to structure graphs to incorporate roll-away error recovery. To guarantee correct operation in the first two subsections, we assume that all primitive operations are atomic and undoable. We relax these requirements in Section 2.4.3 on error recovery, which allows much of the overhead (both performance and storage) required to achieve undoable atomic primitives to be eliminated.

2.4.1 Node States and Transitions

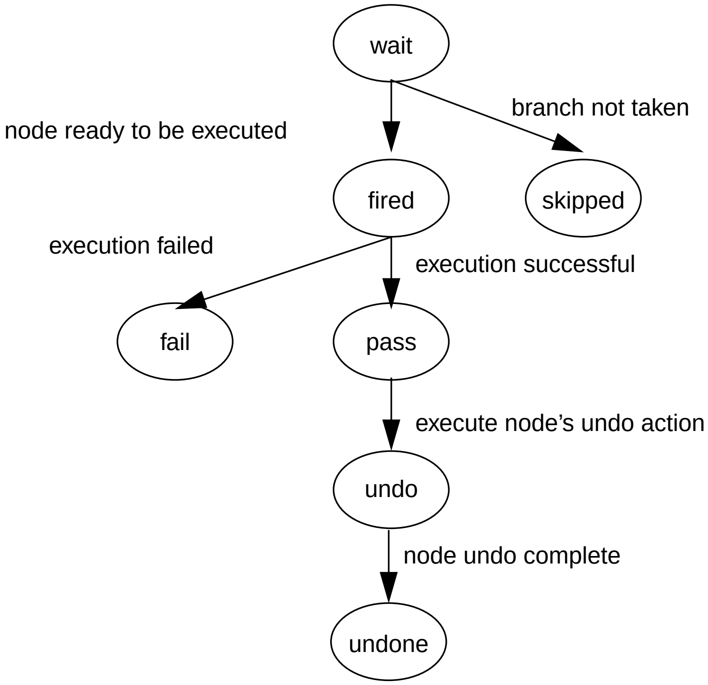
In addition to a primitive, each node in a graph has three other fields, summarized in Table 4: *do action*, *undo action* and *state*. The *do action* is used during normal execution and the *undo action* is used during error recovery. Each of these fields contains the name and parameters of an action.

TABLE 4. Node Fields

Node Field	Description
do action	function executed during normal processing
undo action	function which removes the effects of the do action
state	current state of the node

Each node in a graph may be in one of the seven states summarized in Table 5. The allowable transitions between these states are illustrated in Figure 11.

FIGURE 11 Node State Transitions



All nodes in a graph begin in the wait state. When a graph successfully completes execution, all nodes are in either the pass or skipped states. The error recovery and undone states, described later in Section 2.5, are reached only if the operation fails.

When a graph is initially submitted for execution, all nodes are in the wait state. A node enters the skip state if its parent is a predicate node which determines that the branch which contains the node will not be executed. Once entered, a node will never leave the skip state.

TABLE 5. Node States

Node State	**Description**
wait	blocked, waiting on parents to complete
fired	execution of do action in progress
pass	execution of do action completed successfully
fail	execution of do action failed
skip	node will not be executed
error recovery	execution of undo action in progress
undone	previously executed node has since been undone

The fired state is entered if at least one of its parents is in the pass state and the remainder of its parents are in either the skip or pass states. When a node enters the fired state, its *do action* is executed. The node remains in the fired state until the *do action* completes. The node then enters either the pass or fail state, depending upon the outcome of this execution. If a completed node must be undone, the node first enters the error recovery state which indicates that the node's *undo action* is being executed. Once the *undo action* completes, the node enters the undone state. The error-recovery procedure, which is responsible for moving nodes to the undone state, is described in further detail in Section 2.4.3.

2.4.2 Executing DAGs Without Errors

Executing a graph, for example the graph shown in Figure 9 on page 48, begins with the *source* (head) node and completes with the *sink* (tail) node. This direction of execution, from source to sink, is referred to as *forward execution* throughout the remainder of this document. The source node is executed and, assuming it completes successfully (that is, it returns *pass*), the node enters the pass state.

If the graph does not contain predicate nodes (which is the case with the current RAIDframe release), any node can be executed (i.e., enter the fired state) once all of its parents have reached the pass state. Assuming all nodes complete successfully, this process continues until the sink node enters the pass state; at this point, the execution of the graph is complete and the RAID operation is declared to be successful.

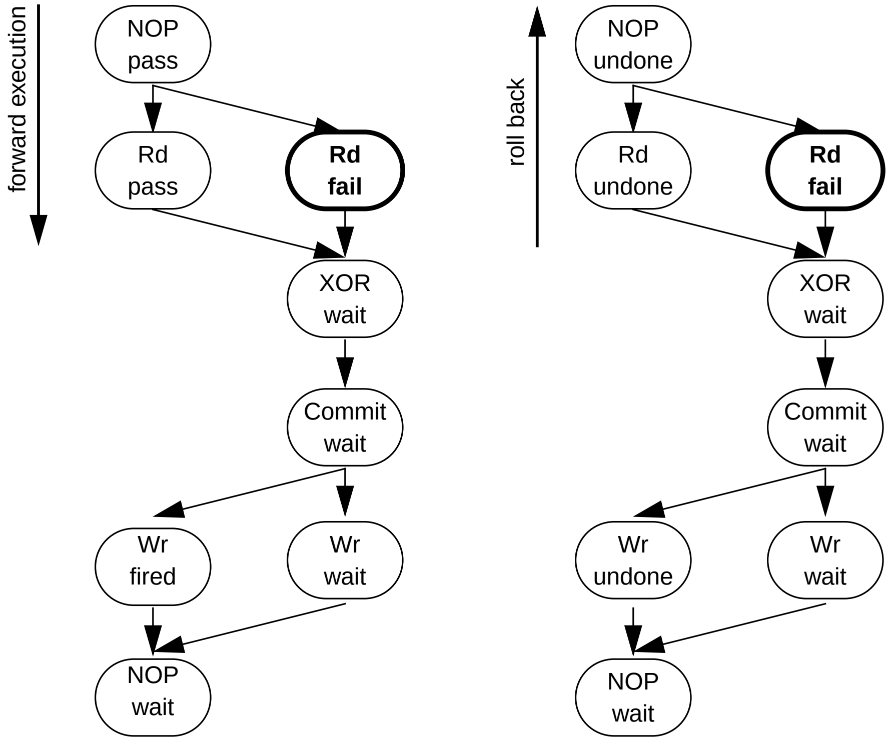
2.4.3 Handling Errors When Executing DAGs

Because device-specific error recovery is removed from the structure of the graph, we were able to define a general execution mechanism which automates handling of errors due to failed primitives. This mechanism, together with a library of RAID operations, will allow array architectures to be implemented rapidly.

As we explained in Section 2.3.3, we have divided RAID operations into two phases to determine the direction of roll-away error recovery. If an error occurs during phase one of a RAID operation, as shown in Figure 12, the error-recovery mechanism rolls backward, releasing resources. At this point, the system substitutes a new graph for the failed graph and retries the operation. If an error is detected during phase two, as shown in Figure 13, the error-recovery mechanism completes the RAID operation—when this happens, all symbols are simultaneously updated. To an outside observer, it would appear as if the failure(s) occured after the RAID operation completed.

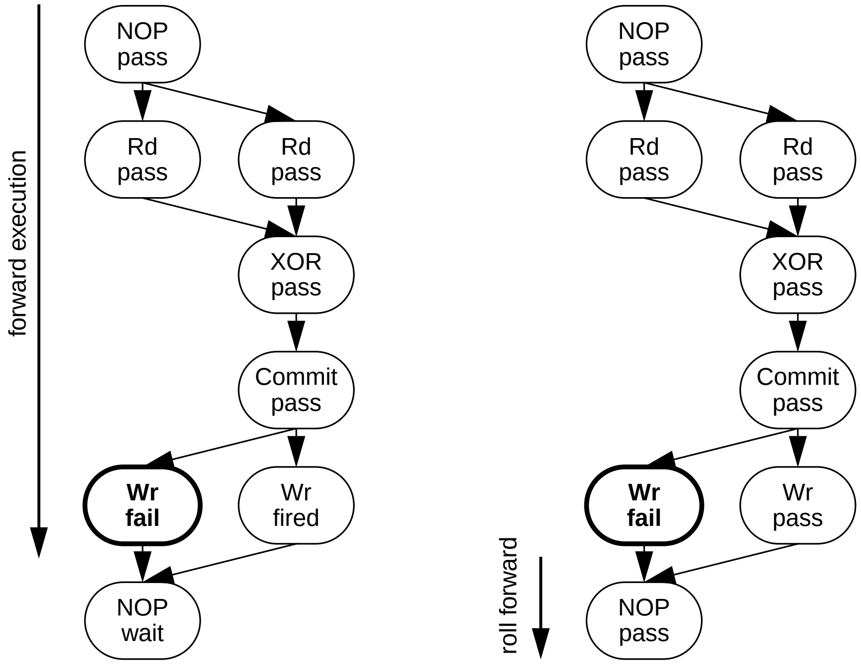
In the next section, we describe the mechanism we have developed which the array uses to recover from a disk failure. We present the library of DAGs provided in the current release of RAIDframe, the prototyping framework which incorporates our approach to modeling and executing RAID operations, in the Appendix.

FIGURE 12 Handling Errors Prior to Commit Point



The failure of the Rd node (indicated in bold) occurs prior to the commit point. This causes forward execution to halt and roll back to begin. Roll back works backward through the graph from the point of failure, undoing the previously completed nodes. If a failure occurs prior to the commit point, the system appears as if the graph never executed.

FIGURE 13 Handling Errors After Commit Point



Because the leftmost Wr node failed after the commit point had been reached, forward execution continues. The rightmost Wr node completes successfully as does the sink (NOP) node. If a failure occurs after a commit point, the sink node is always reached and teh system appears as if the successful completion of the graph was followed by a failure.

2.5 Reconstructing Data On-line When a Disk Fails

In Chapter One we introduced the need for a process in which the array restores itself to the fault-free state following a disk failure. In this section, we provide a brief description of a disk-oriented algorithm (taken from [Holland94]) for reconstructing lost data into spare disk space. For a more complete discussion of reconstruction algorithms, including performance evaluations and optimizations of the disk-oriented algorithm, please refer to Chapter Four in [Holland94].

2.5.1 Disk-Oriented Reconstruction

Not only must a single-fault-tolerant disk array recover from the loss of a disk, it should be able to effect this recovery without taking the system off-line. This is implemented by maintaining one or more on-line spare disks in the array. When a disk fails, the array switches to degraded mode as described in Chapter One; at the same time, it also invokes a background reconstruction process to recover from the failure. This process successively reconstructs the data and parity units that were lost when the disk failed and stores them on the spare disk. The mechanism by which this is accomplished is called the reconstruction algorithm. Once all the units have been recovered, the array returns to normal performance and is once again single-failure tolerant, and so the recovery is complete. Prior to Mark Holland's thesis work, the default algorithm for reconstructing data from a failed disk was *stripe-oriented*; in his thesis, Mark demonstrated a *disk-oriented* algorithm which performs substantially better than the stripe-oriented one. The disk-oriented algorithm creates *C* reconstruction processes, where C represents the number of disks in the array not including the spare. Each of the *C*-1 processes associated with a surviving disk execute the following loop:

```
repeat
1. Find lowest-numbered unit on this disk that is needed for reconstruction.
2. Issue a low-priority request to read the indicated unit into a buffer.
3. Wait for the read to complete.
4. Submit the unit's data to a centralized buffer manager for XOR or block the process
   if buffer manager has no memory to accept the unit.
until (all necessary units have been read)
```

The process associated with the replacement disk executes:

```
repeat
1. Request the next sequential full buffer from the buffer manager.
2. Block the process if none are available.
3. Issue a low-priority write of the buffer to the replacement disk.
4. Wait for the write to complete.
until (the failed disk has been reconstructed)
```

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

The advantage of this approach is that it is able to maintain one low-priority request in the queue for each disk at all times, which means that it will absorb a significant portion of the array's bandwidth that is not absorbed by users. This approach yields substantially faster reconstruction than alternative approaches.

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

2.5.2 Buffer Memory Management

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

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

Forcing reconstruction processes to stall when there are no available free buffers causes the corresponding disks to idle respecting reconstruction. For our purposes, a relatively small number of free buffers suffices to achieve good reconstruction performance. There should be at least as many free buffers as there are surviving disks, so that in the worst case each reconstruction process can have one access in progress and one buffer submitted to the buffer manager.

2.5.3 Interaction with Writes in the Normal Workload

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

We handle this problem by beginning a conflicting user write only after the desired stripe's reconstruction is complete. This approach is memory-efficient and does not waste disk bandwidth but if it is implemented as stated, a user write may experience a very long latency when it is forced to wait for a number of low-priority accesses to complete. The disk-oriented algorithm overcomes this drawback by expediting the reconstruction of a parity stripe containing the data unit that is about to be written by the user.

When the algorithm detects a user write to a data unit in a parity stripe that is currently under reconstruction, it elevates all pending accesses for that reconstruction to the priority of user accesses. If there are any reconstruction accesses for the indicated parity stripe that have not yet been issued, the algorithm issues them immediately, at regular priority rather than low priority. The user write triggering the re-prioritization stalls until the expedited reconstruction is complete, and the algorithm allows it to proceed normally.

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

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

2.5.4 Summary

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

CHAPTER 3

RAIDframe: A Framework for Implementing New Designs

We now describe RAIDframe, a framework for implementing RAID designs, intended for use in researching, verifying, testing and producing RAID systems. This chapter presents an overview of RAIDframe features and the RAID architectures implemented in the current release, then describes its internal architecture and the accompanying reconstruction architecture, and concludes by briefly describing the suite of test applications packaged with the RAIDframe release which can be used to create a variety of workloads for controlled testing.

3.1 Features

RAIDframe has a number of features which support experimenting and verifying advanced disk-array designs, including:

- extensibility
- correctness verification
- mechanized error recovery
- disk-oriented reconstruction
- applications for controlled testing of workloads
- synthetic workload generation
- trace playback
- performance monitoring
- debugging facilities
- multiple front ends for the user level

Array architectures implemented in RAIDframe can be evaluated in three distinct execution environments: a stand-alone application controlling UNIX “raw” disks, an event-driven simulator, and a Digital UNIX device driver capable of performing block and character operations (and thus, capable of mounting a standard file system on a set of disks). In all three environments, the code unique to a disk-array architecture (mapping, caching, DAGs, primitive operations, and disk queueing) is reused without change. In the following sections, we describe each of these environments, the types of uses each one is intended to support, and the limitations of each. Then we list the RAID architectures currently implemented in the RAIDframe. We end the section with a figure showing a case study of the performance of microbenchmarks in RAIDframe.

3.1.1 RAIDframe as a Stand-Alone User Application

As a stand-alone user application, RAIDframe is a process which accesses real disks through the UNIX “raw” device interface. RAIDframe itself is provided as a library, `libraidframe.a`. Applications may link this library into their address space and treat RAIDframe as a flat, addressable storage space (much like a single, large file). This enables users to verify and benchmark their work without modifying the kernel, which can greatly reduce time for developing and evaluating new RAID architectures, disk-queueing policies, DAG constructs, et cetera.

There are several front ends to this user-level library available with RAIDframe. One such (`driver`) can accept either a synthetic workload from a workload generator or a trace file of I/O activities. Because the parameters of the synthetic workload are precisely controllable, array architects can investigate specific array-performance effects. This front end also provides various debugging and stress tests for architectures and policies, including forced reconstruction, constant workload, and layout checking.

The stand-alone user application shares another front end, `rf_genplot`, with the event-driven simulator (described in the next section). The `rf_genplot` front end provides array architects with a means for comparing how different RAID architectures perform running a simulated workload: it runs workload scripts against various RAID configurations and outputs results into a file. Additionally, options allow users to graph the results, either from a current stand-alone run or using results from a previous run to generate graphs in multiuser mode.

Developing, testing, and instrumenting a RAID architecture at the user level enhances portability and extensibility. Moreover, as shown in Figure 14, there is almost no difference in the measurements between in-kernel and stand-alone user-level RAIDframe performance [Gibson95]—which means that array designers unable or unwilling to port RAIDframe’s in-kernel implementation to their operating system can be confident of the validity of user-level performance results. The main drawback of running RAIDframe as a stand-alone user application is that only a single application may be run against the disk array, and in doing so, may not have an access pattern identical to what it would be if it were running through a file system (and, thus, potentially performing additional meta-data accesses).

3.1.2 RAIDframe as an Event-Driven Simulator

The RAIDframe simulator exists to support analyses of configurations for which the user has no hardware (for example, a new disk) or no interest in building (for example, hundreds of disks in an array). The RAIDframe simulator is built on top of the Berkeley

RaidSim simulator [Chen90b, Lee91], which was further modified at CMU. In the simulator, the low-level disk operations are simulated by a configurable disk-geometry model instead of being executed by a real disk; the geometry model is configurable to a wide range of disks. The simulator, like the stand-alone application, uses either a synthetic workload generator or a trace file for replay. Because it runs a synthetic workload against simulated disks, the simulator provides results quickly—more quickly than the versions running against real disks.

The simulator runs as a single-threaded, event-driven program which tracks disk-I/O time. However, there are several disadvantages in using it. First, it is more difficult to run an application against this simulator because it does not actually transfer data, and its event-driven nature causes “virtual time” to pass more quickly than “wall time.” Next, while the geometry model provides seek, rotate, and transfer information for each SCSI I/O sent to any drive, it does not account for bus overhead or disk caching. Also, support for verifying data correctness is not provided. The lack of support for bus overhead and data verification can have significant impact on user results.

Like the previous configuration, the simulator provides its functionality in a library (`libraidframe_sim.a`) which applications may link against. This enables many of the same front ends to the real-disk user-level configuration to be used with this simulator (with the caveat that the simulator is single-threaded and its routines are not reentrant; therefore, multithreaded tests are not supported).

3.1.3 RAIDframe as a Device Driver in the Kernel

RAIDframe also runs as a Digital Unix device driver capable of mounting a standard file system on a set of disks (and supports standard file system operations, such as *newfs).* This allows RAIDframe users to measure the performance of a disk array when it is running a real workload (as opposed to the trace-driven or synthetic versions at the user level). At this level, RAIDframe represents disks as either a raw or block device.

Because the device driver must be compiled in the kernel, any unstable code—such as a bad memory access—can cause a machine crash. Therefore, it is recommended that new disk-array architectures be developed in user mode before being installed in the kernel.

3.1.4 RAID Architectures Implemented in RAIDframe

RAIDframe is released with a variety of disk-array architectures which include not only the basic RAID architectures which are in production today but also a number of experimental architectures which are proposed by the research community. Table 6 lists the architectures that have been implemented in RAIDframe. See Chapter 1, “Redundant Arrays: A Brief Overview,” for descriptions of these architectures.

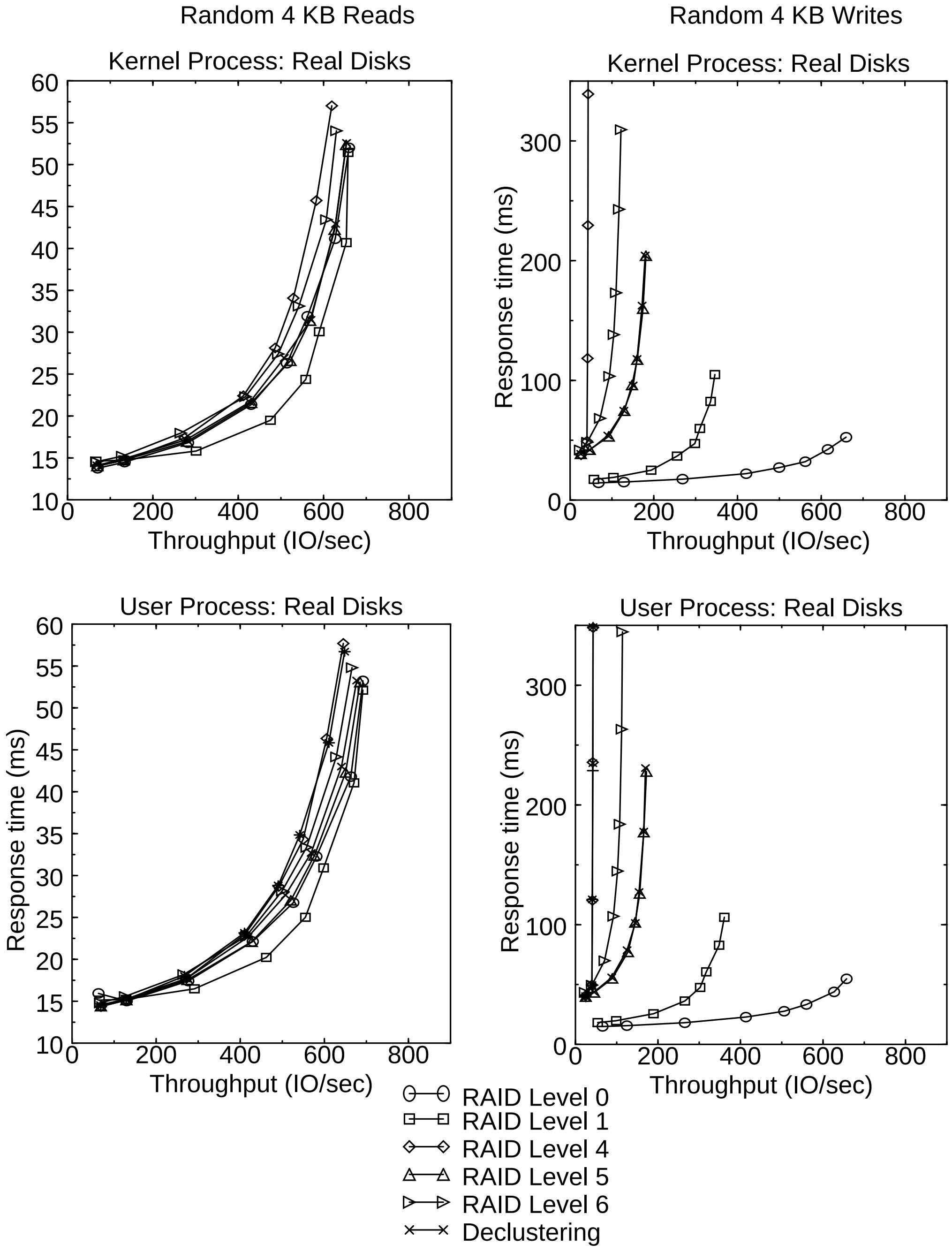
TABLE 6. RAID Architectures Currently Supported by RAIDframe

Architecture	Support Level
RAID level 0	[Fill in reconstruction ability, etc.]
RAID level 1	
RAID level 4	

TABLE 6. RAID Architectures Currently Supported by RAIDframe

Architecture	Support Level
RAID level 5	
Parity declustering	
Distributed sparing	
Parity declustering + Distributed sparing	
Chained declustering	
Interleaved declustering	

FIGURE 14 Case-Study Peformance of Microbenchmarks in RAIDframe

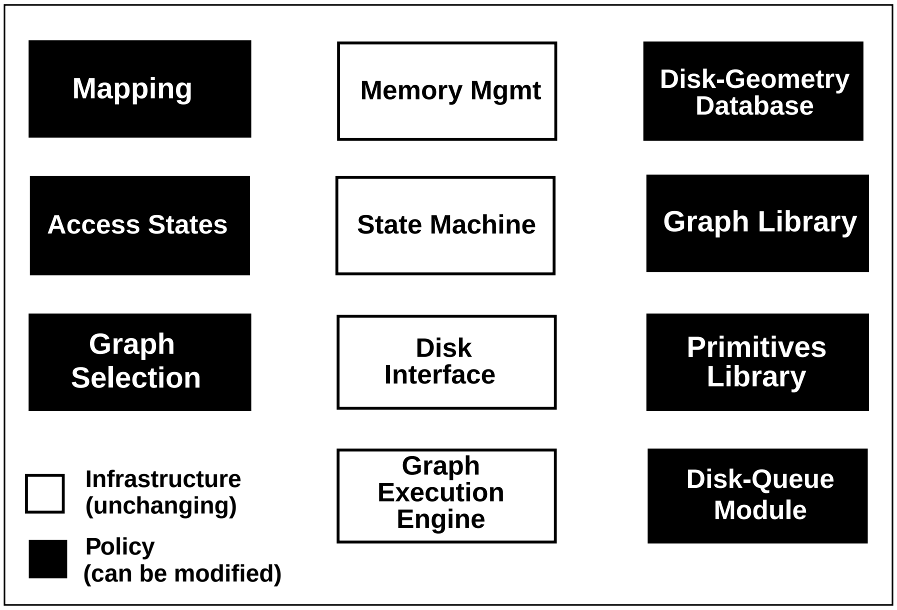


3.2 Internal Architecture

RAIDframe's internal architecture is partitioned into a relatively small set of modules; it separates infrastructure which does not change from libraries which users can modify to create and test new disk-array architectures.

As Figure 15 illustrates, RAIDframe is composed of eleven independent modules, seven of which may be modified to support new architectures.

FIGURE 15 RAIDframe Modules



3.2.1 RAIDframe Infrastructure

This section describes modules which we consider to be infrastructure and do not intend to be modified.

3.2.1.1 State Machine

User requests are processed by a central state machine which is responsible for creating graphs, submitting them for execution, et cetera. While the state machine is configurable, most architectures use a machine similar to the one illustrated in Figure 16 on page 66. The following table lists the access states controlled by the state machine.

TABLE 7. Access States and Their Function

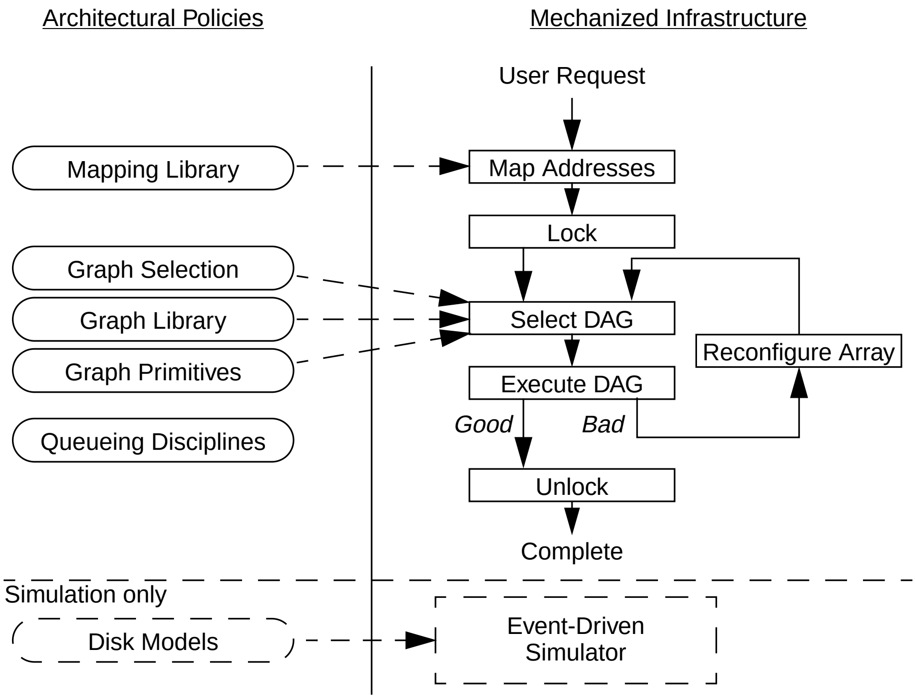
State	Function
`rf_MapState`	map user access
`rf_LockState`	acquire stripe locks
`rf_CreatDAGState`	select and create DAG(s)

TABLE 7. Access States and Their Function

State	Function
`rf_ExecuteDAGState`	execute DAG(s) which are ready
`rf_ProcessDAGState`	postprocess completed DAGs
`rf_CleanupDAGState`	free a graph and stripe locks
`rf_LastState`	null state (indicates end of sequence)
`rf_IncrAccessCountState`	increase count of graphs in flight
`rf_DecrAccessCountState`	decrease count of graphs in flight
`rf_QuiesceState`	wait for the array to quiesce (no graphs in flight)

It is important to note that RAIDframe performs stripe locking and memory allocation prior to creating a DAG.

FIGURE 16 RAIDframe Control Flow



In this example, when a request arrives in the system, it is first sent to the mapping module to compute the set of physical disk locations affected by the access. This produces a data structure describing, for each stripe touched by the access, the mapping of addresses in the RAID address space to physical disk units within each stripe. Next, stripes containing parity information are locked to assure that concurrent writes to the same stripe do not conflict in their parity updates. The access is then converted to a graph and submitted for execution. If a failure occurs while a graph is processing, recovery local to the failed graph leads to creating a graph appropriate for avoiding the failure, if possible.

3.2.1.2 Graph Execution Engine

The primary infrastructure module is the graph execution engine. This engine is responsible only for fully exploiting the allowable concurrency within a DAG; that is, the engine has no knowledge of the architecture embodied in the graph. Figure 15 illustrates the structure of RAIDframe.

RAIDframe’s engine also incorporates a simple and uniform mechanism for handling error conditions in the array. When any error condition occurs prior to the commit node, the engine rolls back, undoing previous state changes. The engine then creates a new graph and retries the operation. If an error occurs after the commit node, the engine rolls forward and finishes executing the graph.

3.2.1.3 Disk Interface

The disk interface module organizes pending disk requests according to queuing disciplines specified at the time of configuration; this allows users to optimize disk use as needed.

[Additional text to come from Jim.]

3.2.2 Configurable RAIDframe Modules

The following sections describe the default implementations of RAIDframe's configurable modules. Please see Chapter 5, "Extending RAIDframe," for more information about reconfiguring the modules.

3.2.2.1 Disk-Queue Module

In the current version, disk requests can be queued in RAIDframe or at the disk. The number of requests allowed for queuing at the disk is configurable. Within RAIDframe, multiple queueing policies are available, including FIFO, SSTF, SCAN, CSCAN and CVSCAN. FIFO is First Come First Serve—requests are serviced in arrival order. Shortest Seek Time First (SSTF) queueing specifies that the next request dispatched is the one closest geographically to the previous request. SCAN specifies that the disk arm traverses the disk from one end to another and back (two-way elevator algorithm), while CSCAN specifies one-way disk sweeps (one-way elevator algorithm). CVSCAN is a discipline that uses two parameters to queue disk requests. With CVSCAN, adding new queuing disciplines can be achieved simply by assigning new values to the two parameters. New disciplines can also be added to the disk-queue switch by specifying new function calls for *create*, *enqueue*, *dequeue*, *promote*, and *peek*.

TABLE 8 Disk-Queue Scheduling Algorithms

Name	Algorithm
`fifo`	First In, First Out
`cvscan`	CVSCAN*
`sstf`	Shortest Seek Time First
`scan`	Two-way Elevator
`cscan`	One-way Elevator

*For more information about CVScan, please refer to [Geist87].

3.2.2.2 Disk-Geometry Database

This database contains disk specifications used by the simulator. These specifications include layout parameters (tracks per cylinder, number of zones, etc.) as well as performance parameters (rpm, seek times, etc.).

3.2.2.3 Mapping

All accesses in RAIDframe go through a mapping module prior to locking the block ranges in the disk array. The framework for the mapping is general to all architectures and invokes architecture-specific mapping routines. The routines are typically short (for example, 5 lines of C code). Each routine provides the ability for the mapping code to:

- map individual sectors and parity units for a given RAID address
- identify a stripe for a given RAID address
- identify a parity-stripe ID for a given data-stripe ID

The mapping module maps an access in the RAID address space to the corresponding set of physical disk addresses. The result is returned as a list of Access Stripe Map (ASM) structures, one per stripe accessed. Each ASM structure contains a pointer to a list of physical-disk-address structures which describe the physical locations touched by the user access.

Note that this first-level mapping routine returns only static mapping information, that is, the list of physical locations that will actually be read or written. Additional remapping to physical location can be done at later stages of the access.

The mapping module also maps the parity. The physical-disk location returned always indicates the entire parity unit, even when only a subset of it is being accessed. This is because an access that is not stripe-unit aligned but spans a stripe-unit boundary may require access to two distinct portions of the parity unit. At this point, however, the system cannot determine which portion(s) of the parity unit will be needed. Instead, the algorithm-selection code decides what subset of the parity unit to access.

3.2.2.4 Graph Selection

A graph-selection algorithm is required for each architecture. This algorithm, implemented as a C routine, determines which graph from the graph library is to be used to execute a specific user request (type, layout map), given the current state of the array. By default, RAIDframe attempts to create one graph for each ASM (in other words, parity stripe). However, if this is not possible, graphs are then selected on a per data unit (or even per sector) basis.

3.2.2.5 Graph Library

The graph library contains the routines, such as `CreateSmallWriteDAG()`, which are capable of creating graphs if called by graph selection. Each routine receives type and physical mapping information and returns a pointer to a graph which is tailored for that request. Adding new graphs requires installing new or extending existing creation functions. The graphs which can be created by the graph-selection algorithm are shown in the Appendix.

3.2.2.6 Primitive-Operations Library

The primitive-operations library contains the functions which abstract single device operations (for example, `XOR`, `DISKRD`, etc.) from which graphs are created. Primitives delineate the failure domains that RAIDframe accommodates; that is, when a node fails, the device associated with it is considered failed as well. Primitives are required to independently detect and recover from soft errors.

TABLE 9. Primitive Operations Provided by RAIDframe

Operation	Function
`DiskRead`	read from disk
`DiskReadMirror`	issue disk read to disk with the shortest queue (RAID Level 1)
`DiskWrite`	write to disk
`XOR`	compute bit-wise exclusive-or
`NOP`	no operation
`Q`	compute Reed-Solomon encoding
`Q′`	decode Reed-Solomon code

3.3 Reconstruction Architecture

In Chapter 2, we described the disk-oriented algorithm which Mark Holland implemented and evaluated prior to RAIDframe's development (See "Reconstructing Data On-line When a Disk Fails" on page 55). We have incorporated this reconstruction algorithm into the current RAIDframe package to allow RAID designers to simulate a disk failure so that they can evaluate the performance of their systems while undergoing reconstruction; [May or may not be true:] RAIDframe currently supports reconstruction for all RAID architectures released with version one except distributed sparing. Planned future releases of RAIDframe will support reconstruction on real disks and distributed sparing.

In this section, we describe the reconstruction architecture currently implemented in RAIDframe.

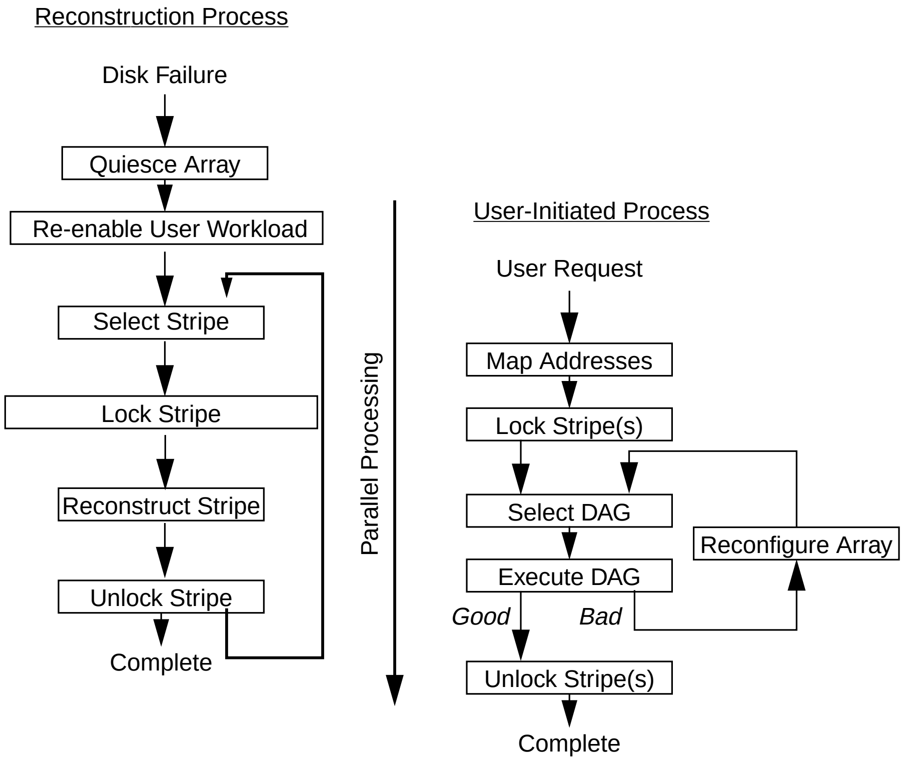
3.3.1 Reconstruction State Machine

The state machine in Section 3.2.1.1 controls the processing of user-initiated disk accesses. However, when a disk fails, a separate state machine which is responsible for reconstructing the lost data initiates a reconstruction thread and then processes reconstruction requests in parallel with the user workload. Reconstruction requests, of course, are lower priority than user-initiated ones; the reconstruction thread simply dispatches disk accesses in batches until all data on the failed disk has been restored. The disk-oriented algorithm allows reconstruction to keep one low-priority disk request in the queue for each physical disk at all times, maximizing the efficiency of reconstruction without significantly penalizing response time for the system user.

3.3.2 Reconstruction States

When invoked, the reconstruction thread issues, through the locking and DAG layers, a low-priority read request for the next unit on each disk required for reconstruction. As each reconstruct read completes, its data is XORed into the accumulating "sum" for the indicated stripe, and the next read request for that disk is issued. When the last unit associated with a particular stripe has been read and summed, the reconstruction thread issues a low-priority request for the now reconstructed data to be written to a replacement or spare disk.

FIGURE 17 RAIDframe Reconstruction Control Flow



The reconstruction thread starts by quiescing the array. The thread sets up its internal state, queues one access request per surviving disk, and then re-enables the user workload. From this time on, reconstruction proceeds in parallel with the applied user workload. A new request is submitted to a surviving disk after a previous read request is completed. Reconstruction completes when the reconstruction read requests associated with all surviving disks have completed (i.e., they have submitted their last stripe unit to the buffer manager).

3.4 Suite of Test Applications

In this section, we introduce the test suite we have included with the RAIDframe code which will allow implementers to test their systems at the user level.

An important method for testing the operation of an array is by actually using it; however, placing the system within a real workload environment in order to test it is obviously not ideal. Therefore, the stand-alone application and simulator versions of

RAIDframe receive I/O requests from a workload file which can contain either 1) a script that is interpreted by a synthetic workload generator or 2) traces of actual disk I/Os. For both of these versions of RAIDframe, the workload file is mandatory for operating (the stand-alone application currently does not accept live user workloads although it can be tested against real disks). The script test runs the workload file.

Six other tests—single-access, loop, degraded-mode read, random read or write, file write-read, and reconstruction—verify the data and redundancy of the array by accessing its disks in different ways. The layout test verifies that the mapping of data and redundancy between the array software (that is, logical location) and actual disk locations (that is, physical locations) is correct. We briefly describe how to use these tests in Chapter 4.

TABLE 10 Tests for Verifying Data, Redundancy, and Layout in RAIDframe

Test	Operation
single-access test	writes, reads, and verifies a single location in the RAID address space
loop test	writes, reads, and verifies multiple locations concurrently in the array
degraded-mode read test	tests read activity with the array in a faulted state
random read or write test	allows a user to read and write to multiple locations in the array in fault-free and degraded mode
file write-read test	writes, reads, and verifies the contents of a file
reconstruction test	runs the loop test while forcing reconstruction to occur at the same time
*script test	runs the workload file which contains either a script for generating a synthetic workload or actual I/O traces
*layout test	verifies the 1-to-1 mapping properties (that is, RAID address to physical locations) of a given architecture

*Because it runs without threading, the simulator version runs only these two tests.

CHAPTER 4 Installing, Configuring, and Using RAIDframe

RAIDframe may be installed as either a stand-alone application, a simulator, or a device driver in the kernel. Installed as a stand-alone application, RAIDframe runs against real disks using either a synthetically generated workload or replaying traces of actual workloads. As a simulator, RAIDframe uses a disk-geometry model to simulate various configurations of hardware; the workload for the simulator, as with the stand-alone application, can be either a synthetically generated one or traces of I/O from actual workloads. In the kernel, RAIDframe runs as a device driver against real disks and upon which a real file system can be mounted. All three versions currently run on DEC Alphas running versions 2.0 and 3.2c of the Digital UNIX operating system.

We begin this chapter by describing the contents of the first RAIDframe code release, then explain how to install and configure each version. Then we briefly describe how to test RAIDframe's operation by verifying data, redundancy, and mapping and how to generate workloads for RAIDframe. Finally, we end this chapter by describing how to access RAIDframe's built-in performance tracing and by listing some of the options for debugging implementations.

4.1 Installing RAIDframe

Before installing any of the RAIDframe versions, you will need to decompress and detar the distribution file. [Filename?]

4.1.1 Creating Executables for the Stand-Alone Application and Simulator

You can create executables for the user-level and simulator versions of RAIDframe without taking any machine-specific steps. Both user-level versions of RAIDframe, the stand-alone application and the simulator, have a number of options for installing them at compile time, which are listed below in Table 11.

TABLE 11. Compiling the Stand-Alone Application and Simulator

Command	**Resulting Action**
`make user`	creates stand-alone executable named "`driver`"
`make uo2`	creates optimized (`-O2`) stand-alone executable named "`driver`"
`make sim`	creates simulator executable named "`driver`" with -g compiler option
`make so2`	creates optimized (`-O2`) simulator named "`driver`"
`make depend`	updates file dependency lists in `Makefile.{s,u}`
`make clean`	removes the executable and all `.o` files
`make tags`	creates an emacs TAGS file
`make othertests`	compiles additional front-end applications for the user-level driver
`make othertestso2`	compiles additional optimized front ends for the user-level driver
`make sothertests`	compiles additional front-end applications for the simulator
`make utils`	compiles support utilities for all configurations

4.1.2 Installing the Device Driver

RAIDframe provides both block and character (UNIX "raw") device interfaces. To configure them into your kernel, you must add appropriate stanzas to the block and character device switches. This will require selecting a major device number. We recommend choosing 51. For further discussion about assigning major device numbers, see *DEC OSF/1 Writing Device Drivers, Volume 1: Tutorial*. [check publication info]

To compile RAIDframe in the kernel, you will need to take the following steps:

1. Add the RAIDframe option to your kernel's configuration file (`/sys/MACHINENAME` in a binary-only tree, `src/kernel/conf/alpha/CONFIGNAME` in a complete source tree). This entry looks like:

   ```
   pseudo-device raidframe <Number of arrays to support>
   options RAIDFRAME_RECON
   ```
   [1]

   The number of arrays to support must be an integer greater than 0.

2. Add an entry for RAIDframe to the device switch tables found in `conf.c` (if you're compiling from a complete source tree, this is `src/kernel/io/common/conf.c`; if you're compiling in a kernel binary tree, this is `/sys/io/common/conf.c`). To do this, type the following lines exactly:

[1] Removing this option disables in-kernel reconstruction but reduces code size.

```
#include <raidframe.h>
#if NRAIDFRAME > 0
int rf_open(), rf_close(), rf_strategy(), rf_read();
int rf_write(), rf_ioctl(), rf_size();
#else /* NRAIDFRAME > 0 */
#define rf_open       nodev
#define rf_close      nodev
#define rf_strategy   nodev
#define rf_read       nodev
#define rf_write      nodev
#define rf_ioctl      nodev
#define rf_size       nodev
#endif /*NRAIDFRAME > 0 */
```

3. Select the major number for the device. (*Note: OSF/1 requires that the number in the comment match the number in the entry table*).

   To do this, first look for the block device (`bdevsw`) table in the `conf.c` file; this is where you set the major number for the RAIDframe pseudo device. Type these lines into it with line breaks only for the start of each comment:

   ```
   {rf_open, rf_close, rf_strategy, nodev, /*51*/
   rf_size, 0, rf_ioctl, DEV_FUNNEL_NULL},
   ```

   Next, look for the character device switch (`cdevsw`) table in the same file; this is where you select the major number for RAIDframe. Type these lines into `cdevsw`:

   ```
   {rf_open, rf_close, rf_read, rf_write, /*51*/
   rf_ioctl, nodev, nulldev, 0,
   asyncsel, nodev, DEV_FUNNEL_NULL, NULL, NULL},
   ```

4. Copy the RAIDframe directory into the source directory of the kernel tree, then update the `files` file with the new modules. You can do so by appending the contents of the file `kfiles` included in the RAIDframe distribution.
5. Rebuild the kernel. Once RAIDframe has been configured in the kernel, a file system can be mounted.

4.2 Configuring RAIDframe

While all three versions of RAIDframe share the same configuration file, the kernel version is configured at the same time that it is compiled in the kernel. For those users who want to configure a device driver after it has been installed, we have included two control programs for doing so; we describe these control programs in Section 4.2.2 on page 79.

4.2.1 RAIDframe’s Configuration File

The configuration file is divided into sections marked by `START <section_name>`. Comments are supported in the configuration file; they must be preceded by a pound sign (#). Of the seven sections in the configuration file, four are mandatory: *array, disks*, *layout*, and *queue*; these are denoted with an **(m)** in the following paragraphs. See Figure 4.2 for a sample configuration file.

Each of the following sections describes how to enter specifications for the stand-alone application and the simulator.

4.2.1.1 Array (m)

This section is used to specify in integers the number of rows, columns, and spare drives in the array. Enter these specifications into the configuration file in this order:

```
<numRow> <numCol> <numSpare>
```

4.2.1.2 Disks (m)

This section lists the pathnames to the device files corresponding to physical disks for the kernel and user-level versions of RAIDframe; each item in the list is a string ending with the device filename. Enter pathnames in this format:

```
/dev/...
/dev/...
```

The simulator, on the other hand, uses a set of disk names that it will instantiate from the `disk.db` database file. Enter disk names as

```
<Disk name>
<Disk name>
```

where each item is a string containing the name of an actual disk drive.

4.2.1.3 Spare

This section may include the device files of spare disks (if they exist). Pathnames are entered in the same format as the **Disk** section.

For the simulator, the **Disks** and **Spare** sections must contain names of actual disk drives instead of listing the pathnames to the device files (that is, `/dev/...`). If a pathname is specified instead of an actual disk drive, the simulator version of RAIDframe will default to the Hewlett-Packard HP2247 disk drive.

4.2.1.4 Layout (m)

This section includes general layout parameters: *sectors per stripe unit*, *stripe unit per parity unit*, and *stripe units per reconstruction unit*. It also contains a parity configuration label (which is a single character) to specify the RAID architecture to use. The parameters are detailed in the following table.

TABLE 12. Layout Parameters for the RAIDframe Configuration File

Parameter	Explanation
`numRow`	number of rows of disks, each row a distinct parity group
`numCol`	number of columns of disks in each row
`sectPerSU`	number of sectors in a stripe unit
`parityConfig`	parity layout based on RAID level
`SUsPerPU`	number of stripe units per parity unit
`SUsPerRU`	number of stripe units per reconstruction unit

When specifiying SUsPerRU, set the number to 1 unless you are specifically implementing reconstruction under party declustering; if so, you should read through the reconstruction code first.

For the parity-configuration layout, there are nine single-character labels that correspond to the RAID architectures currently implemented (see Table 6 on page 61 for a complete list of architectures and their support levels).

TABLE 13 Parity Configurations

parityConfig	Architecture	Must be followed by
0	RAID level 0	
1	RAID level 1	
4	RAID level 4	
5	RAID level 5	
Q	RAID level 6	
T	Parity declustering	data layout file
D	Declustering + distributed sparing	data layout file
R	RAID level 5 + distributed sparing	
C	Chained declustering	[?]
I	Interleaved declustering	[?]

The details for specifying new parity-configuration parameters are given in Chapter 5, “Extending RAIDframe.” Enter layout specifications into the configuration file in this order:

```
<sectPerSU> <SUsPerPU> <SUsPerRU> <parityConfig>
```

where the items are integers. Depending on the value of the parity configuration, you can add a number of needed parameters that are specific to an architecture. In this event, the pathname for layout-specific parameters will follow the general ones in the **Layout** section (Table 12).

4.2.1.5 Queue (m)

This section contains generic parameters for the queue of disk I/O requests: queue type (FIFO, CVSCAN, etc.) and the number of concurrent requests that can be sent to disk. Enter queue specifications into the configuration file in the format:

```
<queue type> <numConcurrentrequests>
```

where the *queue type* is a string and the *number of concurrent requests* is an integer. Where necessary, queue-specific parameters will follow the general ones in the **Queue** section (Figure 4.2).

4.2.1.6 Debug

This section lists a number of user-configurable debug options. Enter these options into the configuration file in the format:

```
   <debug variable><value>
```

where the *debug variable* is a string and the *value* is an integer (a partial list of debug options and their variables is given in Section 4.6 on page 87). Some debugging options have only on/off settings—for these, zero is off, non-zero is on. Others can accept a range of integral values.

FIGURE 18 RAIDframe’s Configuration File

```
START array
# parameters are: numRow numCol numSpare
1 4 1

START disks
# a list of device files corresponding to physical disks
/dev/rrz17c
/dev/rrz19c
/dev/rrz20c
/dev/rrz21c

START spare
# a list of device files corresponding to spare physical disks
# spare device goes here
/dev/rrz117c

START layout
#general layout parameters: sectPerSU SUsPerParityUnit SUsPerReconUnit pari-
tyConfig
64 1 1 T

# layout-type specific parameters for 'T' layout: bd_file_name
/afs/cs/project/pdl/Reconstruction/lib/bds/4.4.bd

START queue
# generic queue parameters:  queue type, number
# concurrent requests that can be sent to a disk
FIFO 1
# queue-specific configuration lines:
# (none for FIFO)

START debug
accessDebug 1
mapDebug 1
dagDebug 1
testDebug 1
```

RAIDframe’s configuration file has seven sections: array, disks, spare, layout, queue, and debug; *array*, *disks*, *layout*, and *queue* must be specified. All sections begin with START and all comments are denoted with a #.

4.2.2 Configuring the Device Driver Using Control Programs

Once the RAIDframe device driver has been installed, you can configure it using either the command-line options of `rf_setconfig` or the menu-driven `rf_ctrl`—an OSF menu-driven program located in the RAIDframe directory. `rf_ctrl` is a simple front end to a set of I/O controls (ioctls) which are listed in Table 14; in addition, these ioctls can be used by other applications. Using both programs is explained in the following sections.

4.2.2.1 rf_setconfig

To run `rf_setconfig`, type:

```
rf_setconfig config<n>
```

where `<n>` is an integer for the each device you configure. `rf_setconfig` copies and saves `config0` into `/dev/.rfconfig0`.

To unconfigure the device, type:

```
rf_setconfig -s
```

4.2.2.2 rf_ctrl

To run `rf_ctrl`, type

```
rf_ctrl <device_file>
```

and select the desired ioctl from the menu..

TABLE 14 Ioctls Supplied with RAIDframe

Control Option	Syntax
Configure the driver; takes a struct `rf_configuration`	`RAIDFRAME_CONFIGURE:`
Unconfigure the array; takes no arguments	`RAIDFRAME_SHUTDOWN:`
Takes a struct `rf_test_acc`	`RAIDFRAME_TEST_ACC`
“Fail” a disk (for testing reconstruction); takes a struct `rf_recon_req`	`RAIDFRAME_FAIL_DISK`
Get reconstruction percentage complete on a row; takes and returns an integer	`RAIDFRAME_CHECKRECON`
Copy reconstructed data back to replaced disk	`RAIDFRAME_COPYBACK`
Start tracing accesses (DFStrace)	`RAIDFRAME_START_ATRACE`
Stop tracing accesses (DFStrace)	`RAIDFRAME_STOP_ATRACE`
Get the size of the device (number of sectors); yields an integer	`RAIDFRAME_GET_SIZE:`
Get basic configuration information (not the same as `rf_configuration`); yields struct `rf_device_config`	`RAIDFRAME_GET_INFO`
Reset `AccTrace` totals on the device	`RAIDFRAME_RESET_ACCTOTALS`
Retrieve `AccTrace` totals for a device; yields `RF_AccTotals`	`RAIDFRAME_GET_ACCTOTALS`
Turn `AccTrace` on if integer is nonzero (off otherwise); takes an integer	`RAIDFRAME_KEEP_ACCTOTALS`

4.3 Testing RAIDframe Operation

As we mentioned in Chapter 3, there are eight test applications for verifying the data, redundancy and layout for RAIDframe implementations at the user level (that is, the stand-alone application and the simulator). Because the simulator runs only against simulated disks, only the script and layout tests are available in that mode.

4.3.1 Running the Test Applications

In the following subsections, we will show you sample interactions with the menu-driven test applications. In many cases, we also comment on the options and interactions to give you a better idea about how to use them.

TABLE 15. Options for Tests

Test	Option
single-access test	`s`
loop test	`l`
degraded-mode read test	`d`
random read or write test	`r`
file write-read test	`f`
reconstruction test	`R`
*script test	`S`
*layout test	`L`

* The simulator runs only these two test options.

4.3.1.1 Single-Access Test

```
Pick a test: s
enter -1 for the RAID address to quit
Starting RAID address [0-82176]? 4032
number of blocks? 219
Input row id of disk to mark failed (-1 for none): -1
```

Entering 0 for the `input row id` will cause the system to prompt you for the column number of the disk to be failed.

4.3.1.2 Loop Test

```
Pick a test: l
How many parallel threads? 2
How many I/Os per thread? 10
Same seed or different seeds in each thread [s/d]? d
Degraded mode? [n=none, c=constant, a=asynchronously,
r=async, init recon] n
```

The mode option `n` is the fault-free test in which no disks are failed. Mode option `c` fails a disk before beginning the loop test; you must specify which disk just as you would in order to run the single-access test. Mode option `a` lets RAIDframe randomly fail a disk during the run; option `r` is same as option `a` but initiates reconstruction after failing the specified disk.

4.3.1.3 Random Read or Write Test

```
Pick a test: r
How many parallel threads? [0 is ok] 1
Reads or Writes [r/w]? r
Degraded mode: none, constant, constant double
degraded,async, async+init recon
 [n/c/2/a/r]? n
Random or sequential I/Os? [r/s] r
How many I/Os per thread? 2
```

4.3.1.4 File Write-Read Test

```
Pick a test: f
File name? foo
```

The only parameter RAIDframe requests is the file name.

4.3.1.5 Reconstruction Test

```
Pick a test: R
How many parallel threads? [0 is ok] 1
Degraded-mode: none, const, async, async+recon, reconfig,
recon+copyback?
 [n/c/a/r/R/C] n
Perform the painful test? [y/n] n
```

4.3.1.6 Script Test

```
Pick a test: S
Trace or script file name? foo
```

You must specify either a script or trace file. See Section 4.3.2 on setting up a workload file.

4.3.1.7 Layout Test

```
Pick a test: L
```

There are no parameters for this test.

4.3.2 Setting Up the Workload File For the Script Test

It may be necessary to test how the array operates under a simulated workload. In RAIDframe, the stand-alone user application and simulator versions receive I/O requests from a synthetic-workload generator or replay traces of actual disk I/Os. The following sections describe how to create a script file for the workload generator and the parameters for a trace file of actual disk I/Os.

4.3.2.1 Synthetically Generated Workloads

The synthetic-workload generator conforms its load to a script containing a variable number of access profiles with individual occurrence probabilities. Each profile defines a deterministic or exponentially distributed access size with a given mean and alignment. Access addresses are randomly generated throughout the entire address space, or with a given probability, within a single locality specified with each profile. Access types are either read, write or sequential (the same as the last access with its address advanced).

The script file contains a description of the workload that you want to run, including probability, I/O request type, size, alignment, distribution, and local region (Table 16).

TABLE 16 Parameters for Writing a Script to Generate a Workload

Parameter	What is Specified
`<probability>`	the fraction of the total workload (given as an integer between 0-100) that this script describes
`<reqType>`	the type of I/O request using an *r* ,*w*, or *s* for a read, write or save
`<size>`	the access size in KB (given as an integer)
`<align>`	the access alignment in KB (given as an integer)
`<distr>`	a character describing the access-size distribution: *d* means deterministic (this is always equal to `<size>`); *e* means exponentially distributed with mean `<size>`
`<lprob>`	the probability (given as an integer between 0-100) that this access is within the local region
`<lfrac>`	the fraction of the array’s data space (given as an integer between 0-100) defining the local region
`<loffs>`	the offset into the array of the start of the local region (given as an integer between 0-100)

The `<lfrac>` and `<loffs>` parameters allow you to define the local region of the disk array where you want to generate accesses.

The lines are in the format:

```
<probability> <reqType> <size> <align> [<distr> [<lprob>
<lfrac> <loffs>]]
```

where only the first four parameters, `<probability> <reqType> <size> <align>`, are mandatory in the script file. Or, they can be in the format:

```
<probability> s
```

If the script file contains a line in the second form (`<probability> s`), it means that with probability `<probability>` the next access selected by any given process will be sequential with respect to the previous access, whatever it happened to be. There can be only one such line in any given script file.

The following is an example of a script file that specifies running a 50/50 read/write workload using random 8k accesses that are 8k aligned:

```
50 r 8 8
50 w 8 8
```

4.3.2.2 Trace-Driven Workloads

The trace file contains actual I/O traces that have been collected from another application instead of synthetic traces that have been generated from a script. The trace file must contain a *header* and *trace records*. The header contains the number of independent processes in the trace, the number of traces for each process, and the file offsets for each trace. Traces understood by RAIDframe must contain an explicit sequence of tuples: (thread id, delay time before issuing this request, read or write, block address, number of blocks, and a requester-waits/requester-does-not-wait flag). Table 17 shows the parameters for trace records.

TABLE 17 Records for a File with Actual Workload Traces

Parameters	What is Specified
`<long blkno>`	RAID address (given as an integer)
`<long size>`	number of blocks (given as an integer)
`<double delay>`	number of seconds (given as an integer)
`<short pid>`	process identification number (given as an integer)
`<char op>`	character operation with an *r* or *w* for a read or write
`<char async_flag>`	character asynchronous flag; set to 1 if the I/O requests are asynchronous

For the parameters, *long* equals 4 bytes; *double* equals 8 bytes; *short* equals two bytes; and *char* equals 1 byte. These traces are stored in binary format as opposed to ASCII.

Each trace record has the following format:

```
<long blkno> <long size> <double delay> <short pid> <char
op> <char async_flag>
```

4.4 Comparing How RAID Architectures Perform

Because it is valuable to compare how different RAID architectures perform relative to one another when implemented in RAIDframe, we have included a front end for doing so at the user level called `rf_genplot`. A key benefit of `rf_genplot` is that it enables users to test throughput versus response time for various RAID architectures and configurations. As we explained in Section 3.1.1 on page 60, `rf_genplot` runs workload scripts from a work file against various RAID architectures and outputs results into a file.

4.4.1 Preparing to Run the rf_genplot Front End

rf_genplot requires three arguments in order to run: configlistfile, worklistfile, and outfilebase. It reads the files named configlistfile and worklistfile and writes files named outfilebase.out, outfilebase.ps, and outfilebase.mif.

The first four lines of configlistfile provide parameters for graphing the results of the workload scripts. Specifically, the first line lists the graph title, the second is the graph subtitle, the third defines x- and y-axis ranges, and the fourth defines major and minor tick marks for both the x and y axes. After that, the configlistfile lists configurations to use and names for them, separated by colons. The filename of the configuration file must appear before the colon; after the colon is the name of the configuration which will appear on the graph. Here's an example of a configlistfile:

```
Random 4KB Reads
RAID level 1 Vs. RAID level 5
0 900 10 60
200 100 5 2.5
/usr20/config/config1.user:Raid 1
/usr20/config/config5.user:Raid 5
```

The worklistfile simply lists scripts for rf_genplot to run; here's an example:

```
/usr20/data/randblock/randblock.1.Read.10disk.A.rst
/usr20/data/randblock/randblock.2.Read.10disk.A.rst
/usr20/data/randblock/randblock.5.Read.10disk.A.rst
/usr20/data/randblock/randblock.10.Read.10disk.A.rst
/usr20/data/randblock/randblock.15.Read.10disk.A.rst
/usr20/data/randblock/randblock.20.Read.10disk.A.rst
/usr20/data/randblock/randblock.30.Read.10disk.A.rst
/usr20/data/randblock/randblock.40.Read.10disk.A.rst
```

4.4.2 Running the rf_genplot Front End

rf_genplot runs each of the scripts listed in the worklistfile against each RAID configuration given in the configlistfile and outputs the results to the outfilebase.out file. Results are given as throughput and response time pairs for each architecture with blank lines between configurations.

If given the -o option before the filenames, rf_genplot will generate the xmgr batch file and run xmgr to produce outfilebase.ps and outfilebase.mif files in addition to running the workload scripts against the RAID configurations. The outfilebase.ps and outfilebase.mif files contain graphs of throughput versus response time for all the architectures listed in the configlistfile.

If given the `-p` option, `rf_genplot` will produce `outfilebase.ps` and `outfilebase.mif` files using the `outfilebase.out` file created from a previous run. This option allows you to run `rf_genplot` first as a stand-alone application then in a multi-user environment to generate graphs.

4.5 Accessing Built-in Performance Tracing

RAIDframe provides a mechanism for timing and tracing eleven predefined system events (Table 18). The codepath for each event is delineated by a of set macros that make calls to a built-in timer mechanism, which in turn relies on a cycle-counter register of the DEC Alpha architecture [Digital92]. An assembly module in the timer reads the cycle counter and evaluates the number of s elapsed. Once the built-in tracing mechanism is turned on, it gathers timer records and saves them in a file.

TABLE 18 RAIDframe System Events and Their Codepaths

Event Timed	**Codepath**
User I/O	Average Access Time
Graph suspend	Suspend Ovhd
Call to complete access stripe map (ASM)	Mapping
Acquiring stripe-lock ranges	Locking
Graph creation	DAG Creations
Graph retry	DAG Retry
Freeing graph structures and return to user	Cleanup
Execute full graph	DAG Execution
Request pending in disk queue	diskwait
Reconstruction	recon
Exclusive-or computation	Xor eval

To turn on tracing at the user level, set `accessTraceBufSize` to a value greater than 0 in the **Debug** section of the RAIDframe configuration file (see Section 4.2.1 for more details); this determines the number of trace entries to accumulate in memory before flushing them to disk where they are saved in the file `trace.dat` (an example of a trace file is given in Figure 19 below). `trace.dat` is accessed using a utility called `rf_tracestats` whose command line argument is in the form:

```
rf_tracestats [-v] [-p] trace.dat
```

where `-v` is verbose mode and `-p` prints formatted trace records on-screen. If no filename is given, `rf_tracestats` expects a trace to be fed in from `stdin`.

Traces can also be extracted from the kernel with `rf_tracestats` by running it with the `-k` argument and specifying the name of the device to extract traces from. For example:

```
rf_tracestats -k /dev/rraidframe_c
```

Section 5.1.9 on page 101 explains how to extend built-in performance tracing by adding new codepaths.

FIGURE 19 Parity Logging Execution Profile

```
Average Access Time: 24652.32 us
Suspend Ovhd       :     3.38 us ( 0.0 %)
Mapping            :    55.70 us ( 0.2 %)
Locking            :    46.03 us ( 0.2 %)
DAG Creation       :   136.50 us ( 0.6 %)
DAG Retry          :     0.00 us ( 0.0 %)
Cleanup            :    10.47 us ( 0.0 %)
DAG Execution      : 24342.59 us (98.7 %)

     ******** DAG Execution Profile********
     Total Xor Time   :   131.24 us ( 0.5 %)
     Total Log Time   :   169.22 us ( 0.7 %)
     Total Disk Queue :  5227.00 us (21.2 %)
     Total Disk Phys  : 17840.21 us (72.4 %)
     ******* summary disk statistics*******
     Avg num phys IOs :     1.50
     Avg queueing time:  3461.59 us (14.0 %)
     Avg physical time: 11814.71 us (47.9%)
     Avg total time   : 15276.30 us (62.0 %)
```

4.6 Debugging RAIDframe Installations

Here is a partial list of the currently implemented debug options and their effects; we have chosen to list the options which are most likely to be used frequently. A complete list of debug options may be found in the source file `rf_optnames.h`.

TABLE 19. Debug Options and Their Effects

Option	Effect
`accessdebug`	Prints out details of each user request.
`accSizeKB  n`	The “loop test” generates a synthetic workload of random I/Os. This debug variable can force the size of the I/Os to be n KB. If n=0, the size of the I/Os are not fixed. Default is n=0.

TABLE 19. Debug Options and Their Effects

Option	Effect
`accessTraceBufSize n`	Specifies the number of trace records which will be buffered before writing to the file trace.dat If n=0, tracing (execution profiling) is disabled. Default is n=0.
`alignAccesses  n`	The "loop test" generates a synthetic workload of random I/Os. This debug variable forces the I/Os to be aligned if n=1. Default is n=0.
`dagDebug`	This variable prints out the type of each DAG when created.
`degDagDebug`	This variable prints additional information about degraded-mode DAGs.
`demoMode  n`	This debug variable enables demo mode if n=1. In demo mode, most data and redundancy verification is disabled and meters are generated to display response time and throughput. Default is n=0.
`diskDebug`	This variable prints information about each disk at configuration time.
`doDebug`	This variable prints each disk operation as it begins and ends (user driver only).
`dtDebug`	This variable prints disk-thread status (user driver only).
`engineDebug`	This variable prints information about engine-thread and node processing.
`maxRandomSizeKB`	The "loop test" generates a synthetic workload of random I/Os. This debug variable can force the size ofthe I/Os to be no greater than n KB. If n=0, max size is unlimited. Default is n=0.
`maxTraceRunTimeSec n`	n = the amount of time in seconds a script file should drive I/Os into RAIDframe. If n=0, max time is unlimited.

TABLE 19. Debug Options and Their Effects

Option	Effect
`memDebug  n`	This variable is useful for debugging memory leaks and buffer overruns. Enabled when n=1. When n=2, this debug variable also prints the address range of every buffer as it is allocated and freed.
`printDagsDebug  n`	If n=1, each DAG (graph) is printed after creation. Default is n=0.
`printStatesDebug  n`	If n=1, the state machine prints state information. Default is n=0.
`queueDebug`	This variable prints disk-queue operations as they happen (policy-independent layer).
`rewriteParityStripes n`	If n=1, parity is rewritten prior to start of test. This is useful when tests which verify parity are run on an uninitialized array. Default is n=0.
`shutdownDebug`	This variable prints shutdown activities as they occur.
`sizePercentage  n`	n is an integer which represents what fraction of the total available disk space will be used. Useful for limiting the duration of reconstruction testing and array initialization. If n=0, 100% of the array is used. Default is n=0.
`validateDAGDebug  n`	If n=1, integrity of each DAG (graph) is verified prior to execution. Default is n=0.

CHAPTER 5 Extending RAIDframe

This chapter is intended to give you a head-start in understanding how to enhance the existing RAIDframe package; we expect that, in order to understand thoroughly how to extend RAIDframe, you will first have to become familiar with the code itself. The following sections briefly describe key RAIDframe subsystems, and provide a how-to guide for certain common extensions.

5.1 RAIDframe fundamentals

5.1.1 Types and Conventions

Most RAIDframe types are defined in `rf_types.h`. These definitions are intended both to make code more easily readable and more easily portable. For instance, a sector number is of type `RF_SectorNum_t`. This is defined as type `RF_uint64`, which is in turn the system-independent definition of a 64-bit unsigned integer. Thus, porting RAIDframe to a new system type requires the correct definition of `RF_uint64` on that platform, but does not require redefinition of `RF_StripeNum_t`, much less changes to the code using values of this type.

Here are some commonly used RAIDframe types and what they represent:

TABLE 20. Common RAIDframe Types

RAIDframe Type	Type Representation
`RF_SectorNum_t`	the number of an individual sector (e.g., *sector #37 of an array*)
`RF_SectorCount_t`	a number of sectors (e.g., *read 100 sectors*)

TABLE 20. Common RAIDframe Types

RAIDframe Type	Type Representation
`RF_StripeNum_t`	the number of an individual stripe (e.g., *stripe number three*)
`RF_StripeCount_t`	a number of stripes (e.g., *30 stripes*)
`RF_IoType_t`	kind of I/O (RF_IO_TYPE_READ, RF_IO_TYPE_WRITE, or RF_IO_TYPE_NOP)
`RF_Raid_t`	entire in-core state of an array

When new type and structures are introduced, the header files in which they are defined are given. It is often the case that a structure is defined in that header file, but the C language `typedef` which is used to refer to it is defined in the file `rf_types.h`. The convention is that a `struct RF_SomeName_s` will be defined in an appropriate header file, which `RF_SomeName_t` is defined in `rf_types.h` as

```
typedef struct RF_SomeName_s RF_SomeName_t;
```

In the future, this document will refer to “`RF_SomeName_t`, defined in `some_file.h`” even though the actual `typedef` of `RF_SomeName_t` is in `rf_types.h`, and `some_file.h` contains the definition of `struct RF_SomeName_s`.

5.1.2 Return Codes

Most RAIDframe operations return type `int`. This is a descriptive error code with `0` being defined as success and a non-zero value being a value defined in `sys/errno.h`, which is appropriate for providing to a calling process to identify the nature of a failure.

5.1.3 Memory Allocation

Memory allocation is different for different systems, and vastly different inside and outside the kernel. For this reason, RAIDframe provides an internal abstraction of memory allocation operations to avoid cluttering code with special cases for various environments and platforms. The following macros, which are defined in `rf_debugMem.h`, should suffice for most simple memory allocation and deallocation operations:

```
RF_Malloc(ptr,size,cast)
RF_Calloc(ptr,nelements,element_size,cast)
RF_Free(ptr,size)
```

In the user environment, these perform the following operations, respectively:

```
ptr = cast malloc(size)
ptr = cast calloc(nelements, element_size)
free(ptr)
```

Thus, to allocate an array of five integers, you might:

```
int *i;
RF_Malloc(i,5*sizeof(int),(int *));
```

or

```
RF_Calloc(i,5,sizeof(int),(int *));
```

And deallocate it with:

```
RF_Free(i,5*sizeof(int));
```

While the size argument to `RF_Free` is not used at the user level, it should be set correctly, because in-kernel memory deallocation does require this field.

5.1.4 Memory-Allocation Lists

Tracking memory allocations can be difficult. In addition, most allocation should be done at start-of-day and deallocated at end-of-day. To ease the programmer burden of tracking allocations, RAIDframe provides *allocation lists* (of type `RF_AllocListElem_t`, defined in `rf_alloclist.h`). In addition, two new memory-allocation operations are defined in `rf_debugMem.h`:

```
RF_MallocAndAdd(ptr,size,cast,alloc_list)
RF_CallocAndAdd(ptr,nelements,element_size,cast,alloc_list)
```

These behave the same as `RF_Malloc` and `RF_Calloc`, respectively, with the additional semantic that the operations are noted in the allocation list `alloc_list`. When `alloc_list` is destroyed, the memory will be freed automatically. Allocation lists are generally provided to start-of-day configuration routines to simplify cleanup and shutdown and are destroyed after all end-of-day activities are complete. In addition, individual I/Os have associated allocation lists which are used by some DAGs to track temporary buffers for parity computation.

5.1.5 Shutdown Lists

Another way in which RAIDframe simplifies the cleanup process is with the use of shutdown lists (`RF_ShutdownList_t`, defined in `rf_shutdown.h`). Start-of-day configuration routines are provided as a pointer to the head of a shutdown list, so they may add entries. The shutdown list is invoked to deconfigure and clean up any configured systems.

A shutdown list is a linked list of elements containing a void function pointer, and an argument to be passed to that function. When an item is added to a shutdown list, it is prepended. When a shutdown list is invoked, the functions in it are called in order from beginning to end and are passed their associated arguments. Thus, the last item added to a shutdown list is the first item called when the shutdown list is invoked. This is to ensure correctness when dealing with dependent modules where one module requires another to be configured and operational to function correctly. (For instance, module B must operate upon module A at creation and clean-up time; therefore, module A must be configured before module B and must not be unconfigured before module B is unconfigured).

Entries are added to a shutdown list by calling `rf_ShutdownCreate`, which is defined as:

```
int rf_ShutdownCreate(RF_ShutdownList_t **listp,
  void (*func)(void *arg), void *arg)
```

When the shutdown list pointed to by `listp` is executed, the function `func` will be called, and the argument `arg` will be passed to it. If `rf_ShutdownCreate()` returns non-zero, it was unable to add an entry to the shutdown list, and the caller should behave accordingly (and is guaranteed that `func()` has not been called, nor will it be called when the contents `listp` are invoked).

It is a RAIDframe convention that a failing configuration operation must provide for complete cleanup at its point of failure. That is, if a configuration operation returns unsuccessfully (see above), any memory it has allocated must be listed in an allocation list it was provided, or be already freed. Likewise, any necessary cleanup operations must be entered into the shutdown list provided, or must be invoked before the error is returned. To simplify the coding of such creation and configuration operations, a programmer may wish to add multiple entries to a shutdown list for a single configuration operation.

5.1.6 Threads

Thread support is provided by a variety of macros and functions found in `rf_threadstuff.[ch]`. These macros hide various porting issues, as well as user/kernel/simulator differences.

5.1.6.1 Thread Types

Threads in RAIDframe are represented by handles, which are of type `RF_Thread_t`. When a thread is created, it is passed a single pointer-sized argument of type `RF_ThreadArg_t`. Pointers may be explicitly cast to and from this type. Because synchronization primitives must be declared very differently in the kernel than at the user-level, and they do not exist at all in the simulator, there are no explicit mutex and condition types. Instead, several macros exist to declare mutexes and conditions.

TABLE 21 Mutex and Condition Declaration Macros

Macro name	Declaration type
`RF_DECLARE_MUTEX`	Declare a mutex with no special keywords
`RF_DECLARE_STATIC_MUTEX`	Declare a mutex with the `static` C keyword
`RF_DECLARE_EXTERN_MUTEX`	Declare a mutex with the `extern` C keyword
`RF_DECLARE_COND`	Declare a condition variable with no special keywords
`RF_DECLARE_STATIC_COND`	Declare a condition variable with the `static` C keyword
`RF_DECLARE_EXTERN_COND`	Declare a condition variable with the `extern` C keyword

These macros are invoked with a single argument, which is the name of the mutex or condition variable to declare. For example:

```
RF_DECLARE_MUTEX(rf_new_lock)
```

declares a global mutex named `rf_new_lock`. Note the lack of trailing semicolon on the line above; the declaration macros add semicolons as necessary.

5.1.6.2 Using mutex variables

Before they may be used, mutexes must be initialized with the function `rf_mutex_init()`. After they are no longer needed, they must be destroyed with the function `rf_mutex_destroy()`. Each of these functions takes a pointer to a mutex variable, and returns a value of type `int`. A zero-valued return indicates success; anything else indicates an error of some sort. To initialize and destroy `rf_new_lock`, from our example above:

```
int rc;
/* ... */
rc = rf_mutex_init(&rf_new_lock);
if (rc) {
  printf(“ERROR: cannot initialize rf_new_lock\n”);
  return(rc);
}

/* ... */
rc = rf_mutex_destroy(&rf_new_lock);
if (rc) {
  printf(“ERROR: cannot destroy rf_new_lock\n”);
}
```

To simplify the destruction of mutexes when necessary, an entry can be automatically added to a shutdown list to destroy a mutex. Rather than initializing a mutex with `rf_mutex_init()`, the function `rf_create_managed_mutex()` may be used instead. The first argument to this function is of type `RF_ShutdownList_t **`, and the second is a pointer to the mutex, just like `rf_mutex_init()`. This also returns an `int`, with a value of `0` indicating success. In this case, success indicates that not only was the mutex initialized correctly, but an entry has been added to the shutdown list which will destroy the mutex when necessary.

As their names imply, the macros `RF_LOCK_MUTEX` and `RF_UNLOCK_MUTEX` respectively lock and unlock mutexes. These macros each take a single argument, which is the name of the mutex to operate upon. Previously, we gave an example defining a mutex named `rf_new_lock`. Now we shall lock and unlock it, to provide a critical section for some new code:

```
RF_LOCK_MUTEX(rf_new_lock);
/* Your critical section here. */
RF_UNLOCK_MUTEX(rf_new_lock);
```

5.1.6.3 Using condition variables

Before a condition variable may be used, it must be initialized with `rf_cond_init()`. When a condition variable is no longer needed, it must be destroyed with `rf_cond_destroy()`. Like the corresponding mutex operations, these functions take as their only argument a pointer to the condition variable to be initialized, and return an `int`, with `0` indicating success. Similarly, `rf_create_managed_cond()` takes an `RF_ShutdownList **`, and a pointer to a condition variable, and returns success to indicate that not only has the condition variable been successfully initialized, but an entry has been added to the shutdown list which will automatically destroy the condition variable.

RAIDframe provides simple macros for accessing the functionality of condition variables, in the form of macros named `RF_WAIT_COND`, `RF_SIGNAL_COND`, and `RF_BROADCAST_COND`. The wait operation takes two arguments, a conditon variable to wait for an event on, and a mutex to atomically unlock before waiting, and lock after waiting. The signal and broadcast operations both take a condition variable upon which to generate a wakeup event. The signal operation attempts to wake at most one thread, which broadcast awakens all threads awaiting an event. For implementation reasons, it is important that threads waiting for events re-check their wakeup conditions upon exiting the wait operation to be sure that a bogus wakeup event has not been generated. Here is an example of what a consumer thread in a standard producer-consumer might look like:

```
while (1) {
  RF_DECLARE_EXTERN_MUTEX(rf_new_wrkr_mutex)
  RF_DECLARE_EXTERN_COND(rf_new_wrkr_cond)
  RF_LOCK_MUTEX(rf_new_wrkr_mutex)
  while (rf_new_wrkr_queue == NULL) {
   RF_WAIT_COND(rf_new_wrkr_cond, rf_new_wrkr_mutex);
   if (rf_new_wrkr_shutdown) {
    /* something wants us to quit */
    RF_UNLOCK_MUTEX(rf_new_wrkr_mutex);
    return;
   }
  }
  /* queue now locked and unempty, dequeue something */
  RF_UNLOCK_MUTEX(rf_new_wrkr_mutex)
  /* queue now unlocked, dispatch op */
}
```

5.1.6.4 Creating threads

The macro `RF_CREATE_THREAD` is used to create threads. This macro evaluates to a return code of type `int`, with `0` indicating success, and nonzero indicating that an error occurred (and the thread could not be created). For example:

```
static void showmyname_thread(arg)
  RF_ThreadArg_t arg;
```

```
{
  char *name = (char *)arg;

  printf(“My name is \”%s\”\n”, name);
  RF_EXIT_THREAD(0);
}

void run_name_threads()
{
  RF_ThreadArg_t a;
  RF_Thread_t th;
  char name[100];
  int i, rc;

  for(i=0;i<10;i++) {
    a = (RF_ThreadArg_t)name;
    rc = RF_CREATE_THREAD(th, showmyname_thread, a);
    if (rc) {
      printf(“ERROR: could not create thread %d\n”, i);
    }
  }
}
```

The above example also uses the macro RF_EXIT_THREAD, which a thread calls when it wishes to cease executing. This macro takes as an argument an integer exit status.

5.1.6.5 Managing threads

One problem with the code in the above example is that the loop which creates the threads does not know when the threads have been created or when they exit. In many cases, threads will be created for the purpose of dispatching various events. In these cases, the creator of the thread will want to know when the thread has begun execution, and is ready to accept events. Likewise, during a cleanup phase, end-of-day routines will want to know when a thread has received notification of system teardown, so resources which the thread might otherwise check or use in its normal operation (for instance, work queues, mutex and condition variables, et cetera) can be deallocated. To address this problem, RAIDframe provides “thread group” management, which can be used to determine when one or a group of threads have been created and are ready to execute events, and when they are no longer executing.

A thread group is of type RF_ThreadGroup_t. This must be initialized one of two ways. One is by calling rf_init_threadgroup(), which takes as its sole argument a pointer to an RF_ThreadGroup_t to initialize. The other is to call rf_init_managed_threadgroup(), which takes as its first argument an RF_ShutdownList_t** and an RF_ThreadGroup_t* as its second argument.

In the case of the former, the thread group must be deallocated when it is no longer needed by calling `rf_destroy_threadgroup()` with a pointer to the thread group as its sole argument (in the case of the latter, the deallocation action is queued on the shutdown list).

Several macros, described in the table below, are key to thread group operation:

TABLE 22 Thread Group Operations

Macro name	Caller	When called
RF_THREADGROUP_STARTED	Creator	After successfully creating a member thread
RF_THREADGROUP_RUNNING	Member thread	Once running
RF_THREADGROUP_DONE	Member thread	When ready to exit
RF_THREADGROUP_WAIT_START	Creator	Waiting for member threads to successfully begin running
RF_THREADGROUP_WAIT_STOP	Creator	Waiting for member threads to stop running

Rewritten to use a thread group, the previous example might look like:

```
static RF_ThreadGroup_t group;
int threads_should_run = 0;

static void showmyname_thread(arg)
  RF_ThreadArg_t arg;
{
  char *name = (char *)arg;

  printf(“My name is \”%s\”\n”, name);
  /* other local initialization */
  RF_THREADGROUP_RUNNING(&group);
  while(threads_should_run && (...)) {
   /* dispatch loop */
  }
  RF_THREADGROUP_DONE(&group);
  RF_EXIT_THREAD(0);
}

void run_name_threads()
```

```
{
  RF_ThreadArg_t a;
  RF_Thread_t th;
  char name[100];
  int i, rc;

  rc = rf_init_threadgroup(&group);
  if (rc) {
    printf(“ERROR: cannot create thread group\n”);
    return;
  }

  threads_should_run = 1;

  for(i=0;i<10;i++) {
    a = (RF_ThreadArg_t)name;
    rc = RF_CREATE_THREAD(th, showmyname_thread, a);
    if (rc) {
      printf(“ERROR: could not create thread %d\n”, i);
    }
    else {
      RF_THREADGROUP_STARTED(&group);
    }
  }
  RF_THREADGROUP_WAIT_START(&group);
  printf(“All threads running\n”);
  /* potentially do something here */
  threads_should_run = 0;
  RF_THREADGROUP_WAIT_STOP(&group);
  printf(“All threads done\n”);
  rc = rf_destroy_threadgroup(&group);
  if (rc) {
    printf(“WARNING: error destroying thread group\n”);
  }
}
```

If RF_THREADGROUP_WAIT_STOP is called on a thread group before RF_THREADGROUP_WAIT_START, the results may not be what is desired.

5.1.6.6 Threads in the simulator

The simulator does not support threads. In this environment, all mutex and condition operations become no-ops, and thread creation is disallowed. Architectures and modules which require a separate stream of execution should instead maintain timed event queues when compiled for simulation.

5.1.7 Creating New Debug Options

Debug options are of type `long`. To add a debug option, add an entry of the form:

```
RF_DBG_OPTION(<Name>,<Val>)
```

to `rf_optnames.h` where the `<Name>` is the name of your debugging variable and `<Val>` is the (long) value that it should default to. To use your debug variable, put the line `#include "rf_options.h"` at the top of `rf_optnames.h` and reference the variable as `rf_<Name>`. For example, say we want to add a debug variable named `newDebugVar`, with a default value of zero. The following line would be added to `rf_optnames.h`:

```
RF_DBG_OPTION(newDebugVar,0) /* our new entry */
```

Note that it is important to preserve the lack of whitespace between the parenthesis when adding new entries to rf_optnames.h. Code which uses this variable might look like:

```
if (rf_newDebugVar) {
  printf(“foo is now %d\n”, foo);
  if (rf_newDebugVar > 1) {
    /* print detailed info */
    printf(“bar is now %d, baz is %lu\n”, bar,
      (u_long)baz));
  }
}
```

5.1.8 Timing

RAIDframe provides a platform- and environment-independent timing mechanism which can be used both for microbenchmarking individual codepaths, and for collecting statistics about how time is being spent in the system overall. This generic timing mechanism is used, among other ways, to generate the elements of RAIDframe trace records (see **Built-in Tracing of RAIDframe Performance**, below).

A timer is of type `RF_Etimer_t`, which is defined in a platform-dependent manner in `rf_etimer.h`. Timers require no special initialization to be used, and are fully copyable. The macro `RF_ETIMER_START` takes as its only argument the timer to start. Likewise, `RF_ETIMER_STOP` also takes a timer as its sole argument. To find out how long a timer has been running, the difference between the start time and the stop time must be computed. Because this computation time might affect other timing results, it is invoked separately with the macro `RF_ETIMER_EVAL`, which computes the time elapsed between `RF_ETIMER_START` and `RF_ETIMER_STOP` for that timer. To access this result, the macro RF_ETIMER_VAL_US takes as its argument a timer, and returns the number of microseconds that RF_ETIMER_EVAL computed as the elapsed time. RF_ETIMER_VAL_MS likewise returns the number of elapsed milliseconds.

This example demonstrates how timers can be used to compute the amount of time that elapses between different points in a codepath. It takes advantage of the copyability of timers to snapshot a running timer at different points to obtain intermediate timing results. Evaluation of elapsed time is deferred until all events being timed have completed, to avoid timing the computation of elapsed time.

```
RF_Etimer_t timer, t1, t2;

RF_ETIMER_START(timer);
/* do some computation (A) here */
t1 = timer;
RF_ETIMER_STOP(t1);
/* do some computation (B) here */
t2 = timer;
RF_ETIMER_STOP(t2);
/* perform some set of operations (C) here */
RF_ETIMER_STOP(timer);
RF_ETIMER_EVAL(timer);
RF_ETIMER_EVAL(t1);
RF_ETIMER_EVAL(t2);
printf(“Operation A took %lu microseconds\n”,
  (unsigned long)RF_ETIMER_VAL_US(t1));
printf(“%lu ms elapsed before operation C started\n”,
  (unsigned long)RF_ETIMER_VAL_MS(t2));
printf(“Together, A, B, and C took %d:%06d\n”,
  (int)RF_ETIMER_VAL_US(timer)/1000000,
  (int)RF_ETIMER_VAL_US(timer)%1000000);
```

5.1.9 Built-in Tracing of RAIDframe Performance

RAIDframe has several predefined codepaths that it will evaluate once the tracing option is turned on in the **Debug** section of the RAIDframe configuration file. To turn on tracing, set accessTraceBufSize to a value greater than 0. Table 23 shows the source files used in timing and tracing and what their functions are.

TABLE 23. Source Files for RAIDframe's Timer and Trace Mechanism

Source File	Function
rf_etimer.h	Times codepaths
rf_readcc.s	Platform-specific assistance for rf_etimer.h
rf_acctrace.[ch]	Gathers timer records efficiently
rf_tracestats.c	Processes the records

To add a trace record to the trace file, you must call rf_LogTracRec().The tracing module accumulates records until it is shut down, or its tracing buffers fill (it uses the number of buffers specified by accessTraceBufSize). At this time, the accumulated buffers are flushed into the trace.dat file. rf_LogTraceRec() takes two arguments. The first is a pointer to an RF_Raid_t, which is the array for which an event has occurred. The second is a pointer to the trace record itself. Trace records are of type RF_AccTraceEntry_t, which is defined in rf_acctrace.h.

To read trace.dat, use rf_tracestats. The command line argument is in the form:

```
rf_tracestats [-v] [-p] trace_dat
```

where -v is verbose mode and -p prints formatted trace records on-screen (without arguments, rf_tracestats displays only summary information for an entire tracefile).

5.2 Installing a New RAID Architecture

A central switch table in the module rf_layout.c specifies the routines which each array architecture relies on for functions such as graph selection, mapping, and reconstruction. The first step in adding a new architecture is to create a new entry in this table, called mapsw in the code.

This is the mapsw entry for RAID level 5. Note that portions of the table appear within the RF_NK2 and RF_NU macros. These macros are used in mapsw entries to remove unnecessary parts of the table in certain environments. (For instance, the in-kernel portion of RAIDframe does not parse configuration files itself, but instead relies on a utility program (rf_setconfig or rf_ctrl) to do so. Likewise, this utility program has no need to actually perform RAID operations such as sector-mapping.)

```
    /* RAID level 5 */
    {‘5’, “RAID Level 5”,
RF_NK2(rf_MakeLayoutSpecificNULL, NULL)
RF_NU(
rf_ConfigureRAID5,
rf_MapSectorRAID5, rf_MapParityRAID5, NULL,
rf_IdentifyStripeRAID5,
rf_RaidFiveDagSelect,
rf_MapSIDToPSIDRAID5,
rf_GetDefaultHeadSepLimitRAID5,
rf_GetDefaultNumFloatingReconBuffersRAID5,
NULL, NULL,
rf_SubmitReconBufferBasic,
rf_VerifyParityBasic,
1,
DefaultStates,
0)
},
```

5.2.1 parityConfig, configName

The first entry is of type RF_ParityConfig_t. This is a single-character identifier of the RAID architecture. Every entry in this table should have a unique value for its RF_ParityConfig_t. This is the character identifier used in the RAIDframe configuration files to identify the RAID architecture. The second entry is of type char*, and is a string identifying the RAID architecture. For instance, “RAID Level 5” above. There is no limit on the length of this string, but it should be reasonably short, and not contain newlines, tabs, or any special characters.

5.2.2 MakeLayoutSpecific, makeLayoutSpecificArg

The next two entries are for parsing layout-specific information from the user’s RAIDframe configuration file. The first is a function returning int, which is used to parse the relevant portion of the configuration file. The second, MakeLayoutSpecificArg, is an extra argument to this function, to make it easier to use the same parsing function with different parameters for different RAID architectures.

The function has a declaration of the form:

```
int MakeLayoutSpecific(FILE *fp, RF_Config_t *cfgPtr,
  void *arg);
```

The first argument is a regular file pointer, which has advanced to the beginning of the layout-specific section of the configuration file (note that this section may begin with one or more blank lines). The second argument is the configuration which is currently being parsed (RF_Config_t is defined in rf_configure.h). The final argument, arg, is the aforementioned MakeLayoutSpecificArg.

The MakeLayoutSpecific function should perform all necessary parsing and computation, and allocate memory to store its results (as necessary). The number of bytes allocated for this purpose should be stored in cfgPtr->layoutSpecificSize, and a pointer to this memory should be stored in cfgPtr->layoutSpecific. This should be a single, contiguous block of memory that is fully copyable (that is, contains no pointer to other regions of memory). This can later be retrieved by other layout-specific functions.

Upon success, the MakeLayoutSpecific operation should return 0. Otherwise, it should return a meaningful error value from sys/errno.h.

5.2.3 Configure

The Configure operation is called at start-of-day to initialize any layout- and array-specific information, and to allocate any extra resources the RAID architecture may require. It has the form:

```
int Configure(RF_ShutdownList_t **shutdownListp,
  RF_Raid_t *raidPtr, RF_Config_t *cfgPtr);
```

The shutdown list is provided so that any necessary shutdown and cleanup activities may be registered at this configuration time. In addition, raidPtr->cleanupList is of type RF_ShutdownList_t*. The contents of raidPtr->cleanupList are deallocated after the array is quiesced and shut down. The array which is being configured is raidPtr, and the user's configuration file is described fully by cfgPtr.

On success, the Configure routine should return 0. On failure, it should return a descriptive, nonzero error code. Additionally, all memory which the Configure routine allocated should either be deallocated or enqueued on raidPtr->cleanupList. Likewise, any necessary cleanup activities should be performed immediately before returning a failure, or enqueued on shutdownList.

The Configure routine may use the field raidPtr->Layout.layoutSpecificInfo, which is of type void*, to store any array-specific information which it desires. It should also initialize raidPtr->totalSectors to the number of data sectors the array is capable of storing (note that this does not include the number of sectors which have been allocated to redundancy data). Additionally, there are several fields in the raidPtr->Layout structure (of type RF_RaidLayout_t, defined in rf_layout.h) which this routine is required to initialize. They are as follows:

TABLE 24 RF_RaidLayout_t fields to be filled in by Configure

Layout Field	Contents
numStripe	number of stripes in the array
dataSectorsPerStripe	number of data sectors in each stripe

Layout Field	Contents
`bytesPerStripeUnit`	number of bytes in each stripe unit
`numDataCol`	number of data columns in each stripe
`numParityCol`	number of parity columns in each stripe

5.2.4 MapSector, MapParity, MapQ

The `MapSector`, `MapParity`, and `MapQ` routines provide basic array-layout information. They are declared as:

```
void MapSector(RF_Raid_t *raidPtr,
  RF_RaidAddr_t raidSector, RF_RowCol_t *row,
  RF_RowCol_t *col, RF_SectorNum_t *diskSector,
  int remap);
void MapParity(RF_Raid_t *raidPtr,
  RF_RaidAddr_t raidSector, RF_RowCol_t *row,
  RF_RowCol_t *col, RF_SectorNum_t *diskSector,
  int remap);
void MapQ(RF_Raid_t *raidPtr, RF_RaidAddr_t raidSector,
  RF_RowCol_t *row, RF_RowCol_t *col,
  RF_SectorNum_t *diskSector, int remap);
```

Each of these functions is called to determine the location of a single sector in the array. The array is indicated by `raidPtr`. The sector is indicated by `raidSector`, which is the sector number of the array to be mapped. The function assigns `*row` and `*col` to indicate which disk the sector resides on, and `*diskSector` is the sector number on that disk which the mapping has yielded.

The `MapSector` routine is used to map data sectors to physical disk sectors. All array architectures must provide this routine. This should yield a unique mapping for every sector in the array.

The `MapParity` routine is like `MapSector`, except that the resulting sector is not the corresponding physical data sector, but rather the corresponding physical parity sector. In most architectures, many data sectors will map to the same parity sector. In non-fault-tolerant architectures, this routine may be `NULL`.

The `MapQ` routine is similar to `MapParity`, except it is used to map an additional redundancy unit. This is provided by dual-fault-tolerant architectures, such as Even-Odd and Raid Level 6.

If the `remap` argument has the value `RF_REMAP`, the mapping should be to the spare sector corresponding to the sector to which the mapping function would otherwise yield.

5.2.5 IdentifyStripe

The IdentifyStripe routine is used to determine which physical disks contain sectors that share a stripe with a particular sector. This routine has the declaration:

```
void IdentifyStripe(RF_Raid_t *raidPtr,
  RF_RaidAddr_t addr, RF_RowCol_t **diskids,
  RF_RowCol_t *outrow);
```

The first argument, raidPtr, is the array in which the mapping is to be performed. The second argument, addr, is the sector in said array for which IdentifyStripe is to determine the disks of its fellow stripe members. This function should assign to *diskids an array of (raidPtr->Layout.numDataCol + raidPtr->Layout.numParityCol) RF_RowCol_t elements. These are the column numbers of the disks. The row of disks which the stripe occupies should be assigned to *outrow.

When reading the extant RAIDframe code, one may note that some architectures actually generate an ordered list of disks in the stripe. This is not necessary; rather, this is a historic convention used to make debugging easier.

5.2.6 SelectionFunc

When an I/O request enters the system, it is passed through rf_SelectAlgorithm() in rf_aselect.c. This routine uses the layout-specific DAG selection routine to choose a DAG creation function for a particular access. This routine, SelectionFunc, is declared as:

```
void SelectionFunc(RF_Raid_t *raidPtr, RF_IoType_t type,
  RF_AccessStripeMap_t *asmap,
  RF_VoidFuncPtr *createFunc);
```

This routine is used to determine what DAG creation function a particular access to the array indicated by raidPtr should use. RF_IO_TYPE_READ and RF_IO_TYPE_WRITE are the only legal values for the type argument, which indicates the direction of the access. The asmap argument (of type RF_AccessStripeMap_t, found inrf_layout.h) describes the access in its entirety, including physical disk mappings for data and parity, ranges accessed, and the presence of disk failures which may affect the access. The SelectionFunc routine should take these failures into account when determining the creation function to use, potentially determining that an access should be performed in degraded mode, rather than fault-free. If a unit to be accessed has failed, but is already reconstructed, the SelectionFunc routine should also take this into account, and alter the physical mappings in asmap to reflect the fact that the data has been reconstructed. This is especially important when the access is a write, because without this remapping, a reconstructed data or parity unit will not be updated to reflect the new contents of the stripe.

A pointer to the DAG creation function should be assigned to *createFunc. A later section details DAG creation operations, and how this function should behave. Assigning a value of NULL to *createFunc indicates that a DAG cannot be created for this access. rf_SelectAlgorithm() will initially attempt to create one graph for each parity stripe in the access's codeword. If this creation is unsuccessful, rf_SelectAlgorithm() will then try to create a set of graphs for each stripe unit within that parity stripe. If graphs cannot be generated for each stripe unit, rf_SelectAlgorithm() will attempt to create a DAG for each sector in each stripe unit in the codeword. Finally, if this fails, rf_SelectAlgorithm() declares failure, and the access is failed.

5.2.7 MapSIDToPSID

The MapSIDToPSID routine is used by architectures for which the relationship between data stripes and parity stripes is not an equivalence. For instance, parity declustering allows multiple stripes to be packed into a single parity stripe, to increase the size of the reconstruction unit without affecting the size of the stripe unit. This routine has the declaration:

```
void MapSIDToPSID(RF_RaidLayout_t *layoutPtr,
  RF_StripeNum_t stripeID, RF_StripeNum_t *psID,
  RF_ReconUnitNum_t *which_ru);
```

The layout of the array in which this mapping is to be performed is described by layoutPtr. The stripe number of the stripe to be mapped is stripeID, and the resulting parity stripe is stored by MapSIDToPSID in *psID. This routine also stores the reconstruction unit of the stripe in *which_ru. The identity mapping is most common here; that is:

```
*psID = stripeID;
*which_ru = 0;
```

This is performed automatically if the MapSIDToPSID routine for an architecture is NULL, or if the number of stripe units per parity unit for a layout is 1.

5.2.8 GetDefaultHeadSepLimit

The disk-directed reconstruction code has the ability to keep disk arms synchronized with one another when sweeping surviving columns. This is controlled by the head separation limit for the array, which is assigned at start-of-day by calling the GetDefaultHeadSepLimit routine, which is declared as:

```
RF_HeadSepLimit_t GetDefaultHeadSepLimit(
  RF_Raid_t *raidPtr);
```

This function takes as its sole argument the array in question, and returns how many sectors ahead of the slowest disk the fastest disk is allowed to be. That is to say, it returns the maximal difference in sector number between the lowest-numbered-sector currently being read by the disk-directed reconstruction code, and the highest-numbered-sector currently being read by the disk-directed reconstruction code (neglecting stripes being read for forced reconstruction). If this routine is NULL, a value of (-1) is assumed. (-1) indicates that this separation is unlimited. Note that (-1) is the only legal value less than 1.

5.2.9 GetDefaultNumFloatingReconBuffers

The disk-directed reconstruction module maintains a pool of “floating” reconstruction buffers, which are not assigned to any particular disk, but are instead used to store the results of additional I/Os to disks which would otherwise be idle. An architecture may specify a minimum number of these buffers to keep for each array by providing a `GetDefaultNumFloatingReconBuffers` routine, which has the following form:

```
int GetDefaultNumFloatingReconBuffers(
  RF_Raid_t *raidPtr);
```

This routine is called at start-of-day on the array, and should return the minimum number of floating reconstruction buffers to maintain for the array.

5.2.10 GetNumSparePUs

Architectures which support distributed sparing tell the system how many spare reconstruction units there are on each disk with the `GetNumSparePUs` routine, which has the form:

```
RF_ReconUnitCount_t GetNumSparePUs(RF_Raid_t *raidPtr);
```

Given an array `raidPtr`, this routine returns the number of spare reconstruction units there are on each disk.

5.2.11 InstallSpareTable

Distributed-sparing architectures which have dynamic sparing mappings may need to compute a new sparing table when reconstruction begins for a disk. To do so, these architectures provide an `InstallSpareTable` routine with the following declaration type:

```
int InstallSpareTable(RF_Raid_t *raidPtr,
  RF_RowCol_t frow, RF_RowCol_t fcol);
```

The arguments indicate the array to determine the mapping for (`raidPtr`), and the row and column (`frow` and `fcol`, respectively) of the failed disk to be reconstructed to spare space. On success, this routine returns `0`. On failure, it returns a descriptive nonzero error code.

5.2.12 SubmitReconBuffer

When the disk-directed reconstruction code finishes reading a buffer, it must either use it to compute the contents of a failed unit, or save it until it has enough other information from the stripe from which the buffer originated to do so. When a read of a buffer from a surviving disk completes, an architecture’s `SubmitReconBuffer` routine is called. This routine is declared as:

```
int SubmitReconBuffer(RF_ReconBuffer_t *rbuf,
  int keep_it, int use_committed);
```

The buffer which has just been read is rbuf (the array from which it was read is rbuf->raidPtr). If keep_it is nonzero, the SubmitReconBuffer routine may hold the buffer, even if it cannot immediately use its contents. If keep_it is 0, the SubmitReconBuffer routine must either immediately use or copy the contents of rbuf. If use_committed is nonzero, this routine must consume a buffer off the committedRbufs list of the row's reconstruction control unit, even if such a buffer is not needed (in the case where the buffer is not needed, it may immediately be released with rf_ReleaseFloatingReconBuffer()). In turn, the SubmitReconBuffer routine should should call rf_CheckForFullRbuf() when a target RF_ReconBuffer_t contains the reconstructed data for the failed unit in the stripe.

If the SubmitReconBuffer routine for an architecture is NULL, the architecture cannot reconstruct failed units.

5.2.13 VerifyParity

RAIDframe has a built-in parity verification and correction mechanism (which is also used to format arrays with correct parity, and can be used in various tests for debugging purposes to determine that parity is correct for an access). This relies on the VerifyParity routine which an architecture must provide to check and correct (if requested) the redundancy information for a stripe. This routine has the form:

```
int VerifyParity(RF_Raid_t *raidPtr,
  RF_RaidAddr_t raidAddr, RF_PhysDiskAddr_t *parityPDA,
  int correct_it, RF_RaidAccessFlags_t flags);
```

The array in which redundancy information is to be verified is raidPtr. The stripe for which this information is to be checked is the one containing sector number raidAddr. To improve performance, and ease the coding of VerifyParity, the parityPDA argument provides the already-complete mapping of the redundancy information to physical addresses for this stripe. If correct_it is nonzero, and the redundancy information is not correct, new redundancy information should be computed and written for this stripe. Finally, any RAID accesses that must be performed should use the flags given as the last parameter to the VerifyParity routine.

When reading existing data in the stripe, or writing new redundancy information, the VerifyParity routine should create trivial DAGs which do so. The function rf_MakeSimpleDAG() in rf_parityscan.c assists in this task.

The VerifyParity routine returns a status value indicating the current correctness of the parity before and after execution. The following values, defined in rf_parityscan.h, are the legal returns for this routine:

TABLE 25 Return Values for the VerifyParity Operation

Value	Meaning
RF_PARITY_OKAY	redundancy information is correct
RF_PARITY_CORRECTED	redundancy information was incorrect, but `correct_it` was nonzero, and it is now correct
RF_PARITY_BAD	redundancy information is not correct, and `correct_it` was `0`
RF_PARITY_COULD_NOT_CORRECT	redundancy information is not correct, `correct_it` was nonzero, and correct redundancy information could not be computed or could not be written
RF_PARITY_COULD_NOT_VERIFY	redundancy information could not be verified, either current data or redundancy could not be read, or correct redundancy information could not be computed

5.2.14 faultsTolerated

The `faultsTolerated` field of the mapsw entry for a RAID architecture indicates the minimum number of faults that an array can tolerate without data loss. For example, Raid Level 4 can tolerate exactly one disk failure, so its `faultsTolerated` is 1. Raid Level 0 cannot tolerate any failures, so its `faultsTolerated` is 0. Raid Level 1 (mirroring) can potentially survive several faults; however, if both members of a mirror pair fail, data is lost; thus, its `faultsTolerated` is 1, because that is the minimum number of failures which it can guarantee surviving.

5.2.15 states

The `states` field lists the order in which an access to this array architecture passes through the access state machine. This field is an array of elements of type `RF_AccessState_t`. The last element in this array must be `rf_LastState`, which indicates that the access is complete. Most architectures will wish to use the value `DefaultStates` in this field, which is a standard ordering of states.

5.2.16 flags

The final field of a `mapsw` entry is flags, which are a set of flags ORd together to indicate that the architecture has certain standard properties. Some architectures will wish to provide a `0` in this field (indicating that none of these flags apply). Legal values include:

TABLE 26 RF_LayoutSW_t Flag Values

Value	Meaning
RF_DISTRIBUTE_SPARE	architecture supports distributed sparing
RF_BD_DECLUSTERED	this is a declustered architecture which requires externally generated block-design tables

5.3 Implementing New RAID Operations

5.3.1 DAG Creation

As discussed in Section 5.2.6 on page 106, RAIDframe graph-creation functions must at least be able to create graphs for accessing single blocks at a time for accesses to be successfully generated. RAIDframe will currrently never attempt to create graphs for an access which spans more than a single parity stripe (such accesses are broken up into sets of single-parity-stripe accesses, which are executed concurrently).

The appropriate graph creation routine for an access or portion of an access is determined by an architecture's `SelectionFunc`. The `SelectionFunc` provides a `void` function pointer. This function should have the form:

```
void DagCreationFunc(RF_Raid_t *raidPtr,
  RF_AccessStripeMap_t *asmap, RF_DagHeader_t *dag_h,
  void *bp, RF_RaidAccessFlags_t flags,
  RF_AllocListElem_t *allocList);
```

The array and parameterization of the access are described by `raidPtr` and `asmap`, respectively. The DAG creation function should fill in the empty DAG header `dag_h`. At the time the DAG creation function is called, `dag_h` is initialized as an enabled DAG with no nodes. In the RAIDframe kernel environment, `bp` is a `struct buf*` which represents the access’s target buffer (most DAG creation functions will not need this information at all. Some may choose to operate differently for kernel-internal or user accesses, so this information is available). Outside the kernel, `bp` is generally ignored. The `flags` variable is a bitwise OR of values from `rf_dagflags.h`. Many of these flags are not applicable to the DAG creation function, but again, they are provided for those few cases where the DAG creation function wishes to do something different as a result. Finally, a per-access memory allocation list, `allocList`, is provided for any temporary storage which may need to be allocated. This not only includes extra buffers for computing redundancy information before storing it, but also includes the storage required to hold the actual nodes of the DAG themselves.

5.3.2 Creating New Primitive Operations

The most important rule to follow when creating primitive operations is that they must be nonblocking. Primitives such as `disk read` employ call-back functions—the disk read is scheduled, the primitive returns, and the call-back routine is later called when the disk read actually completes. If a primitive is allowed to block, RAIDframe will not be able to properly schedule its workload (and may deadlock).

In its current release, RAIDframe provides a variety of primitive operations which may be reused by architectures that you later implement.

5.4 Adding a New Disk-Queueing Policy

RAIDframe supports multiple queueing disciplines for pending disk I/Os. The following section explains how to add a new queueing policy.

A queueing policy must maintain a set of pending I/Os for a single disk. Although an array may have many disks, a queueing policy is only aware of disks on an individual basis. Therefore, it only needs to support a limited number of simple operations: *create*, *enqueue*, *dequeue*, *peek*, and *promote*.

To add a queueing policy, you must register it with the disk queue manager. This is done by modifying the `diskqueuesw` structure in `rf_diskqueue.c`. Entries in this structure are of type `RF_DiskQueueSW_t` (defined in `rf_diskqueue.h`), and look like:

```
{"fifo", /* FIFO */
rf_FifoCreate,
rf_FifoEnqueue,
rf_FifoDequeue,
rf_FifoPeek,
rf_FifoPromote},
```

The first entry is the `queueType` (`RF_DiskQueueType_t`) and is a string which is used to identify the queueing discipline. RAIDframe configuration files will use this string to request this queueing policy. The remainder of the entries are function entry points, described in the sections below. You should add new policies to the end of the `diskqueuesw` array. The first entry in this array (`FIFO`) is the default policy (which is used when the configuration parser cannot recognize the requested queueing policy as specified in the RAIDframe configuration file).

5.4.1 Create Operation

Your creation function should have a declaration of the form:

```
void *rf_PolicynameCreate(RF_SectorCount_t
sectors_per_disk, RF_AllocListElem_t *cl_list,
RF_ShutdownList_t **listp)
```

This function is called to create and initialize a disk queue. It returns a generic (`void *`) pointer, which will be used later to identify the individual queue to your queueing module. (One disk queue will be created for each disk). The size of each disk in sectors is passed by the value in `sectors_per_disk`. An allocation list is passed as `cl_list`. Any memory which your queueing policy allocates should be registered with this allocation list by using RF_CallocAndAdd or RF_MallocAndAdd to allocate the memory. If any special operations need to be performed to shut down the queue, these should be resgistered with the shutdown list listp.

5.4.2 Enqueue Operation

Your enqueue function should have a declaration of the form:

```
void rf_PolicynameEnqueue(void *qptr, RF_DiskQueueData_t
*req, int priority)
```

This function is called to add a request to the disk's queue. The queue is uniquely identified by qptr, which is the returned value from the queue creation function. The request is pointed to by req and is of type RF_DiskQueueData_t (defined in rf_diskqueue.h). The priority is either of type RF_IO_NORMAL_PRIORITY or RF_IO_LOW_PRIORITY. When dequeueing, you should always give preference to dequeueing I/Os of NORMAL priority over I/Os of LOW priority. The RF_DiskQueueData_t structure contains two pointers, *next* and *prev*, both of type RF_DiskQueueData_t *, which may be used by this queueing code to maintain lists of pending I/Os.

The Enqueue, Dequeue, Peek, and Promote operations need not be protected internally with locks; the discipline-independent disk-queueing code in rf_diskqueue.c will do this automatically.

5.4.3 Dequeue Operation

Your dequeue function should have a declaration of the form:

```
RF_DiskQueueData_t *rf_PolicynameDequeue(void *qptr)
```

This function is called to remove a request from the disk's queue. The queue is uniquely identified by qptr, which is the returned value from the queue creation function. If an I/O of priority RF_IO_NORMAL_PRIORITY is in the queue, it should be returned. If there is more than one such I/O, the queueing module should select one and return it (for instance, FIFO queueing will return the first such I/O to be enqueued). If no I/O of NORMAL priority is awaiting dispatch in this queue, an I/O of priority RF_IO_LOW_PRIORITY may be returned. If there are no I/Os of any priority in the queue, this operation should return NULL. Before returning a valid pending I/O, it should be removed from the queue.

5.4.4 Peek Operation

Your peek function should have a declaration of the form:

```
RF_DiskQueueData_t *rf_PolicynamePeek(void *qptr)
```

This function should behave identically to the dequeue function, except that it should not remove the I/O from the list of pending I/Os for this disk. Additionally, if the Peek operation is called, and there are no subsequent Enqueue, Dequeue or Promote operations, another Peek or Dequeue operation should return the same I/O (that is, a queue should be deterministic for its contents at any given time, and its choice of which I/O to execute next should be affected only by a change of its contents).

5.4.5 Promote Operation

Your promote function should have a declaration of the form:

```
int rf_PolicynamePromote(void *qptr, RF_StripeNum_t par-
ityStripeID, RF_ReconUnitNum_t which_ru)
```

This operation should search the queue for entries for which the `parityStripeID` and `which_ru` fields of the `RF_DiskQueueData_t` structure match those which are passed as arguments to this function, and which have a priority field valued at `RF_IO_LOW_PRIORITY. Each such I/O should be re-marked as having priority RF_IO_NORMAL_PRIORITY`, and any necessary rearrangements of the queueing policy's data should be performed at this time. This function should return the number of such I/Os it has found and promoted to `NORMAL` priority or zero if none such were found.

5.5 Porting RAIDframe to Other Systems

Currently all three versions of RAIDframe—stand-alone user application, event-driven simulator, and in-kernel device driver—run on DEC Alphas running pre-4.0 versions of the Digital UNIX operating system. Additionally, the simulator runs on IBM RS/6000s running AIX. This section is intended as an aid in porting RAIDframe to new platforms.

5.5.1 Basic Types

The first step is to define a set of basic types in `rf_types.h`. You must provide various sizes of signed and unsigned integers for your system. Table 27 lists the types you must define, and what they must be defined to.

TABLE 27 Basic RAIDframe integer types

RAIDframe type	Meaning
RF_int8	signed 8-bit integer
RF_uint8	unsigned 8-bit integer
RF_int16	signed 16-bit integer
RF_uint16	unsigned 16-bit integer
RF_int32	signed 32-bit integer
RF_uint32	unsigned 32-bit integer
RF_int64	signed 64-bit integer
RF_uint64	unsigned 64-bit integer

5.5.2 Byte Ordering

If the target platform is big-endian, the macro `RF_IS_BIG_ENDIAN` must be set to `1` in `rf_types.h`. If it is not, `RF_IS_BIG_ENDIAN` must be set to `0`.

5.5.3 Word Size

The file `rf_dagfuncs.c` contains several optimized XOR routines. These routines require that the macro `LONGSHIFT` be defined in this file. LONGSHIFT should be defined to the $\mathtt{log_2(sizeof(long))}$ for your system (for example, on a system with 64-bit `long`s, this would be `3`, on a system with 32-bit longs, this would be `2`).

5.5.4 Timing

Section 5.1.8 describes various timing macros defined in `rf_etimer.h` which provide precision timing. These are architecture-dependent. Ideally, these functions provide microsecond-accurate timing with little or no overhead. When porting to a new platform, the nature of the precision/overhead tradeoff must be characterized, and an appropriate implementation provided. Some architectures need assembly-language assistance; this should be added to `rf_readcc.s`.

5.5.5 SCSI Operations

SCSI operations are isolated within rf_camlayer.c. Ports of more than just the simulator should provide code in this file for such operations as SCSI Read Capacity.

5.5.6 Threads

Section 5.1.6 details the thread operations defined in `rf_threadstuff.c` and `rf_threadstuff.h`. Ports which provide user-level-driver or kernel functionality must provide appropriate platform-dependent thread operations here. A Pthreads implementation is already provided for the user-level driver; architectures for which a compliant Pthreads implementation is available should be able to re-use this.

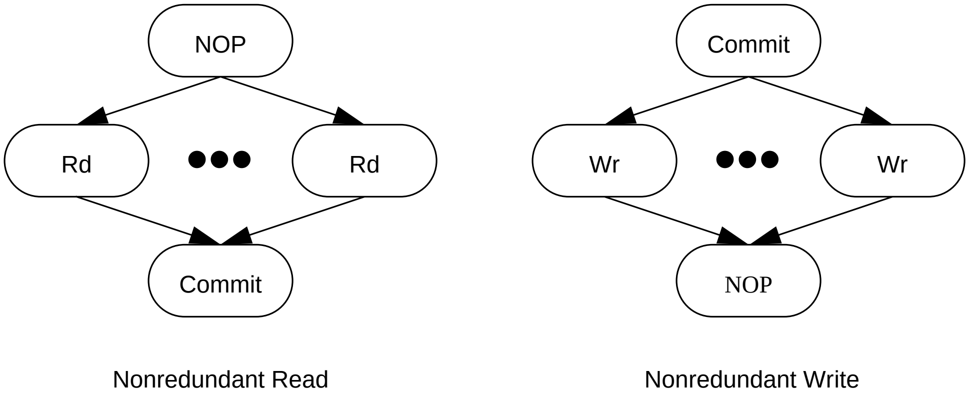
Appendix: Graph Library

The graphs necessary for implementing the RAID architectures listed in Table 6 in Chapter 3 are available for reuse in the graph library and they are shown in the following section. We have categorized the graphs implemented in RAIDframe by the particular architecture for which they were designed; in some cases, graphs are reused among several different RAID levels.

RAID Level 0

As we already explained in Chapter 1, RAID level 0 arrays do not encode data; therefore, a RAID level 0 array is not fault-tolerant. Because of this, only nonredundant operations are available for use. Figure 20 illustrates the structure of nonredundant read and write operations. The NOP operations guarantee that each DAG has single source and sink nodes. Each graph is capable of supporting one or more simultaneous primitive operations, allowing the graph to scale with the size of the user request.

FIGURE 20 Nonredundant Graphs



RAID Level 1, Chained Declustering, Interleaved Declustering

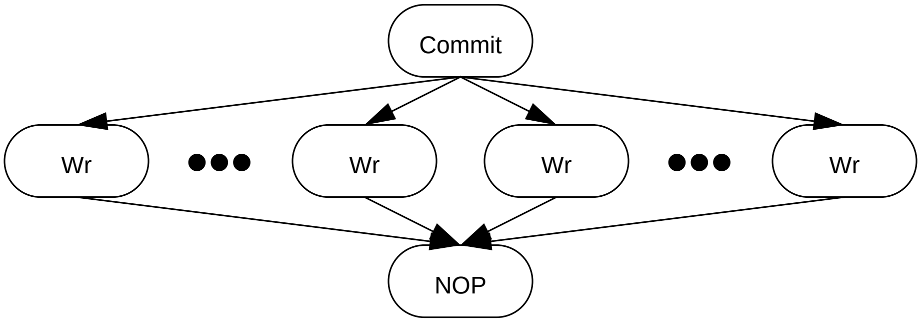
RAID level 1 arrays are fault tolerant and employ copy-based redundancy to survive single disk faults without loss of service. This means that operations are defined to service both fault-free and degraded read and write requests. Table 28 specifies which operations are used to service a request given the state of the disks.

TABLE 28. RAID Level 1 Graph Selection

Request	**Disk Faults**	**Graph**
read	none, single disk	nonredundant read
write	none	mirrored write
write	single disk	nonredundant write

In addition to the nonredundant graphs described in Figure 20, RAID level 1 arrays require an additional write operation, the *mirrored write*, which is responsible for maintaining copy-based redundancy in a fault-free array. This operation, illustrated in Figure 21, contains twice the number of write operations as a nonredundant write operation because a copy of each symbol is written to both a primary and a secondary disk.

FIGURE 21 Mirrored-Write Graph



RAID level 1 arrays use copy-based encoding to survive disk faults and require that data must be written to two independent disks. In this graph, the write operations on the left represent writes to a primary disk(s) and write operations on the right represent writes of data to secondary disk(s). The NOP source node of the nonredundant write graph is replaced by a Commit node.

RAID Level 4, RAID Level 5, Parity Declustering

RAID levels 4 and 5 tolerate disk faults through the use of parity encoding. As expected, the operations used to satisfy read and write requests are largely the same; however, because it is possible to write only a fraction of a codeword, additional write operations are required. Namely, the *small write* operation (Figure 22) which is used to write data to less than half of a codeword and the *reconstruct write* operation (Figure 23 on page 121) which is used to write data to more than half, but less than a full, codeword. Table 29 breaks down graph selection for RAID level 4 and 5 arrays. Because these two arrays differ only in mapping, the same table applies to both architectures.

TABLE 29. RAID Levels 4 and 5 Graph Slection

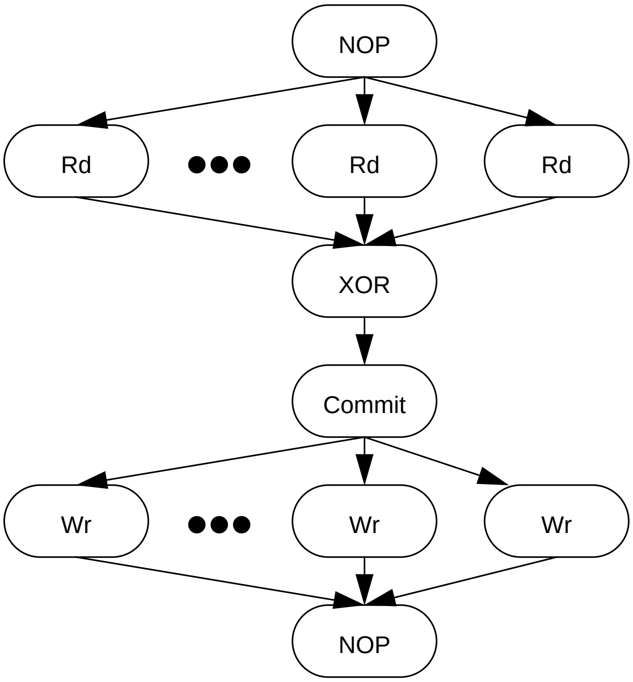
Request	Disk Faults	Graph
read	none	nonredundant read
read	data disk	degraded read
read	parity disk	nonredundant write
write < 50% of codeword	none	small write
write > 50% and < 100%	none	reconstruct write
write entire codeword	none	large write
write	data disk	reconstruct write
write	parity disk	nonredundant write

The small write operation, illustrated in Figure 22 on page 120, writes both data and parity to disk. Parity is computed as:

$$Parity_{new} = Parity_{old} \oplus Data_{old} \oplus Data_{new} \qquad \textbf{(EQ 1)}$$

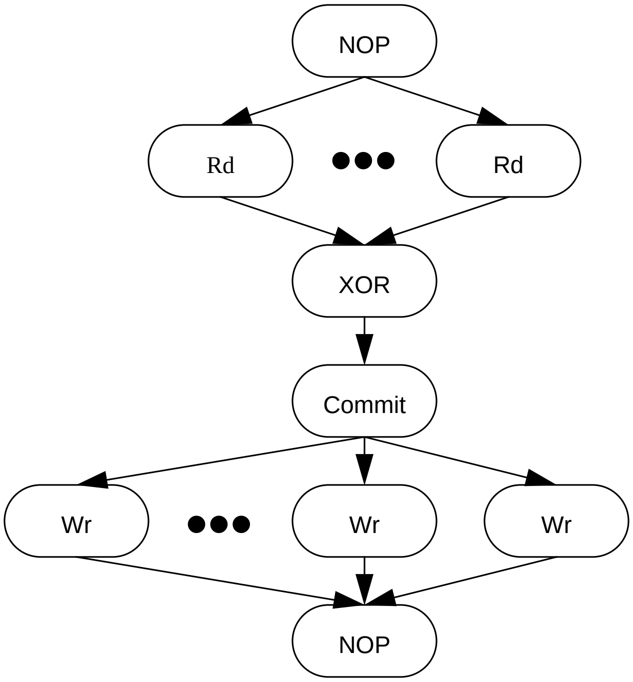
The cluster of read operations on the left side of the graph represent the read of old data and the single read operation on the right represents the read of old parity. Once parity has been computed, the new data and parity symbols are written to the array.

FIGURE 22 Small-Write Graph



In the reconstruct write operation, illustrated in Figure 23, parity is computed from all symbols in the codeword. The Rd operations collect data symbols which are not being overwritten. Once all data symbols are collected, parity is computed and the new data and parity symbols are written to disk

FIGURE 23 Reconstruct-Write Graph



The Rd operations read the data symbols which are not being overwritten. The left-most Wr operations overwrite data symbols and the Wr operation on the right over-writes parity.

RAID Level 6

In addition to parity, RAID level 6 arrays employ a second check symbol to allow them to survive two simultaneous disk failures. We refer to this second symbol as "Q." The graphs used by this architecture are summarized in Table 30.

TABLE 30. RAID Level 6 Graph Selection

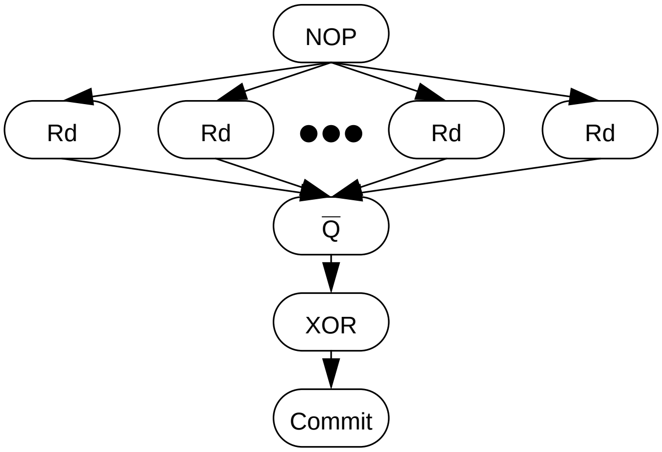
Request	Disk Faults	Graph
read	none	nonredundant read
read	single data disk	degraded read
read	parity disk	nonredundant read
read	Q disk	nonredundant read
read	two data disks	PQ double-degraded read
read	data + parity disks	PQ degraded-DP read
read	data + Q disks	degraded read

TABLE 30. RAID Level 6 Graph Selection

Request	Disk Faults	Graph
read	parity + Q disks	nonredundant read
write < 50% of codeword	none	PQ small write
write < 50% of codeword	parity	PQ small write, P omitted
write < 50% of codeword	Q	small write
write > 50% and < 100%	none	PQ reconstruct write
write > 50% and < 100%	parity	PQ reconstruct, P omitted
write > 50% and < 100%	Q	reconstruct write
write 100%	none	PQ large write
write 100%	parity	PQ large write, P omitted
write 100%	Q	large write
write	one data disk	PQ reconstruct write
write	two data disks	PQ double-degraded write
write	data + parity disks	PQ reconstruct, P omitted
write	data + Q disks	reconstruct write
write	parity + Q disks	nonredundant write

Read operations to fault-free or single-fault arrays are handled in much the same manner as RAID level 5. When an attempt is made to read a codeword with two missing data symbols, a *PQ double-degraded-read* operation, illustrated in Figure 24, is used.

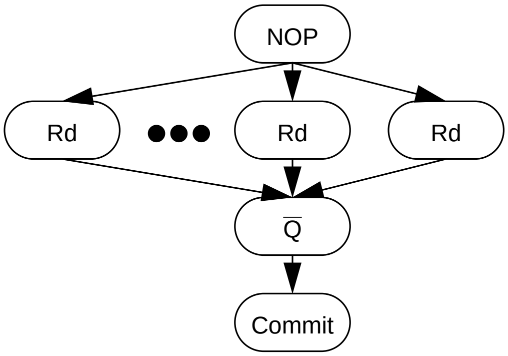
FIGURE 24 PQ Double-Degraded-Read Graph



This operation is used when two data units are missing from the codeword. The left-most Rd operation reads the old value of parity and the right-most operation reads the old value of Q. The center Rd operations read all surviving data in the codeword. The Q operation regenerates a single missing data symbol and the XOR node regenerates the other missing symbol.

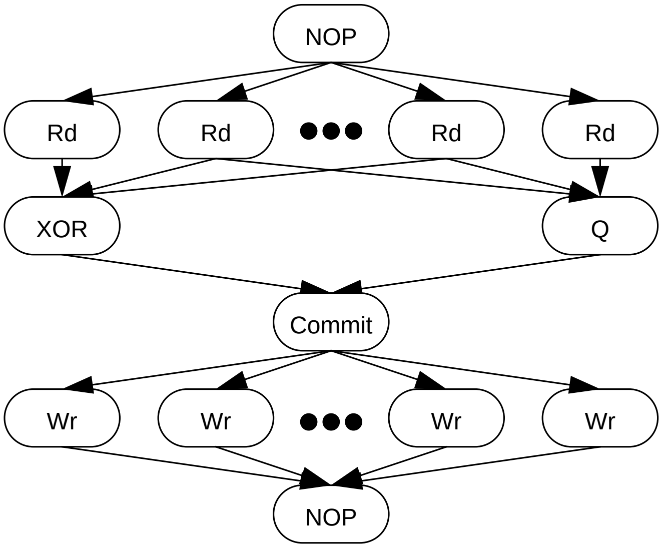
Reading data from a codeword in which both a data symbol and parity are missing requires the use of the "Q" symbol to reconstruct the missing data. The operation to do this, the *PQ degraded-DP-read* operations is illustrated in Figure 25.

FIGURE 25 PQ Degraded-DP-Read Graph



Similar to RAID level 5 arrays, writing less than half of a codeword to a RAID level 6 array is best done using a read-modify-write algorithm. The *PQ small write operation*, illustrated in Figure 26, writes new data symbols and computes new values of parity and "Q" using Equation 1 on page 120. If either the parity or Q disks fail, this same graph is used but the chains which would normally update the now-failed check symbol are omitted.

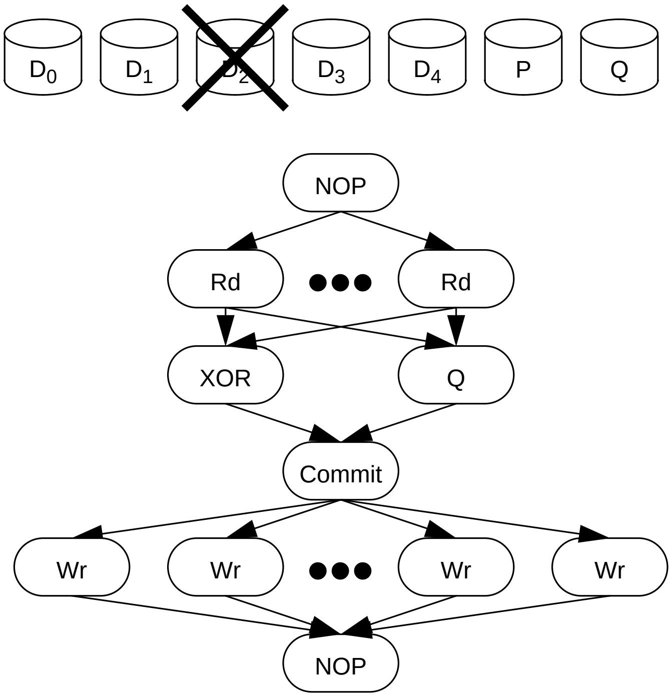
FIGURE 26 PQ Small-Write Graph



This graph is similar to the small-write graph (Figure 22) but with an extra chain added to update the "Q" disk. The Commit node blocks all writes from initiating until all new symbols (data, parity, and Q) have been computed.

Writing over half, but less than an entire, codeword is best done by a reconstruct write, similar to the one used in RAID level 5. Illustrated in Figure 27, the *PQ reconstruct-write* operation reads the data symbols not overwritten, meaning that the entire (new) codeword is held in memory. Parity and Q are then computed and the new data, parity, and Q are then written to disk. This operation is also used when data is being written to an array in which a single data disk has failed and a fault-free disk is being written.

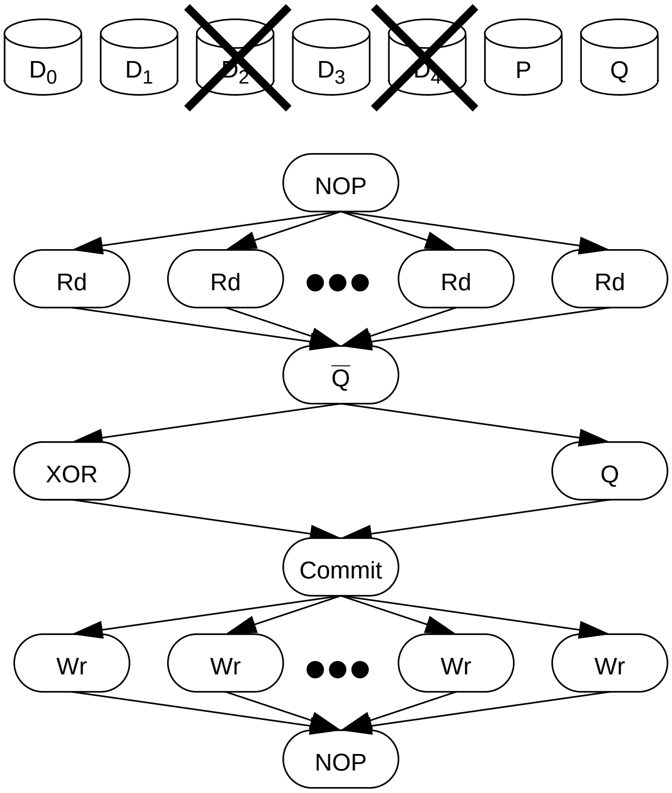
FIGURE 27 PQ Reconstruct-WriteGraph



This graph is similar to the reconstruct-write graph (Figure 23) but with an extra chain added to update the "Q" disk. In this example, assume that D1 and D2 are to be written. The Rd operations read old data ($D_{0,}$ D_3 and D_4). New values of P and Q are then computed and the writes of D_1, P, and Q are initiated.. The Commit node blocks all Wr nodes from executing until all new symbols have been computed.

If two data disks have failed and data is written to at least one, but not both, of the failed disks, the *PQ double-degraded write* operation, illustrated in Figure 28, is used. This graph employs an algorithm similar to the one used in the PQ degradedwrite operation but must reconstruct the failed data which is not overwritten.

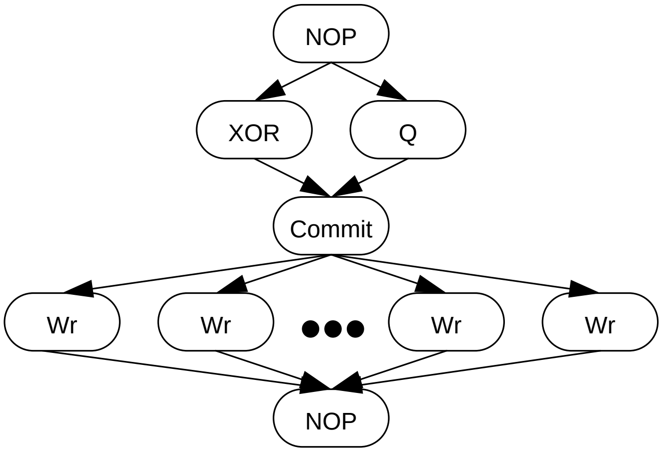
FIGURE 28 PQ Double-Degraded-Write Graph



Assume that D_1 and D_2 are to be overwritten. Because D_4 is missing, the PQ reconstruct operation cannot be used. This operation completes the requests by reconstructing D_4 and then using the reconstruct-write algorithm. First all surviving symbols are read. The Rd actions in the center read the read of data (e.g., D_0 D_1 and D_3), the Rd operations on the ends read old P and Q. The Q operation reconstructs D_4. At this point, the entire codeword is known and the computation and writing of parity, Q and data can begin. The Commit node was added to prevent Wr operations from executing before the XOR and Q nodes have completed.

Finally, writing data to the entire codeword is simply performed using the *PQ large-write* operation. Illustrated in Figure 29, the operation overwrites every symbol in the codeword.

FIGURE 29 PQ Large-Write Graph



Instead of allowing new data to be written concurrently while the parity overwrite record is computed, the Commit node blocks the writes of new data until the XOR and Q nodes have executed completely.

Bibliography

[Aho88] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-esley Publishing Company (March, 1988).

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

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

[ATC90] Array Technology Corporation, *RAID+ Series Model RX*, Boulder, CO, 1990. Product description.

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

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

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

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

[Blaum94] Blaum, M., Brady, J., Bruck, J., and Menon, J., “Evenodd: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures,” *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1994, pp. 245-54.

[Brown72] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesly, 1987.

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

[Buzen87] Buzen, J.P. and A.W. Shum, “A Unified Operational Treatment of RPS Reconnect Delays,” *Performance Evaluation Review*, Vol. 15, No. 1, 1987.

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

[Cao93] Cao. P, Lim, S.B., Venkataraman, S., and J. Wilkes, “The TickerTAIP Parallel RAID Architecture,” *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1993, pp. 52-63.

[Cao94] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. “The TickerTAIP parallel RAID architecture.” *ACM Transactions on Computer Systems*, Vol. 12, No. 3. August 1994, pp. 236-269.

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

[Chen90b] Chen, P. and D. Patterson, “Maximizing Performance in a Striped Disk Array,” *Proceedings of International Symposium on Computer Architecture*, 1990, pp. 322-331.

[Clark82] Clarke, E. and Emerson, E. A. "Synthesis of synchronization skeletons for branching time temporal logic." *Proc. of the Workshop on Logic of Programs*, May 1981, Yorktown Heights, NY. Published as *Lecture Notes in Computer Science*, Vol. 131. Wein, Austria: Springer-Verlag, 1982, pp. 52-71.

[Clark94] Clarke, E., Grumberg, O., and Long, D. “Model checking.” *Proc. of the International Summer School on Deductive Program Design*. Marktoberdorf, Germany. July 26 - August 27, 1994.

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

[Courtright94] Courtright, W.V. and G. Gibson, “Backward Error Recovery in Redundant Disk Arrays,” *Proceedings of the 1994 Computer Measurement Group (CMG) Conference*, 1994, pp.63-74.

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

[DISK/TREND94] DISK/TREND, Inc., *1994 DISK/TREND Report: Disk Drive Arrays*. 1925 Landings Drive, Mountain View, CA, SUM-3.

[Drapeau94] Drapeau, A., Shirriff, K., Hartman, J., Miller, E., Seshan, S., Katz, R., Patterson, D., Lee, E., Chen, P., and G. Gibson, “RAID-II: A High-Bandwidth Network File Server,” *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994, pp. 234-44.

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

[Fujitsu2360] Fujitsu Corporation, Model M2360A product information.

[Geist87] Geist, R., Reynolds, R., and E. Pittard, “Disk Scheduling in System V,” ACM..

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

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

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

[Gibson93] Gibson, G. and D. Patterson, “Designing Disk Arrays for High Data Reliability,” *Journal of Parallel and Distributed Computing*, Vol. 17, 1993, pp. 4-27.

[Gibson95] Gibson, G. A., Courtright, W.V., Holland, M., and J. Zelenka, “RAIDframe: Rapid Prototyping for Disk Arrays,” CMU-CS-95-200, Carnegie Mellon University, 1995.

[Gray81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. “The recovery manager of the System R database manager.” *Computing Surveys*, Vol. 13, No. 2. June 1981, pp. 223-242.

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

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

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

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

[Holland94] Holland, M. *On-line Data Reconstruction in Redundant Disk Arrays*, Carnegie Mellon University, 1994.

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

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

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

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

[IBM3390] IBM Corporation, *IBM 3390 Direct Access Storage Introduction*, Manual GC26-4573-0, 1989.

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

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

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

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

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

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

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

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

[Long94] Long, D., Montague, B., and L.-F. Cabrera, “Swift/RAID: A Distributed Computing System,” *Computing Systems*, Vol 3., No. 7, pp. 333-359, 1994.

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

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

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

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

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

[Menon92b] Menon, J. and D. Mattson, “Comparison of Sparing Alternatives for Disk Arrays,” *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1992, pp. 318-329.

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

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

[Merchant92] Merchant, A. and P. Yu, “Performance Analysis of A Dual Striping Strategy for Replicated Disk Arrays,” *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, 1992.

[Meyers78] Meyers, G. J. *Composite/Structured Design*. New York: Nav Nostrand Reinhold Co., 1978.

[Mogi94] Mogi, K. and M. Kitsuregawa. “Dynamic Parity Stripe Reorganizations for RAID5 Disk Arrays,” *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, IEEE Computer Society Press, September 1994, pp. 17-26.

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

[Myers86] Myers, G.J., Yu, A.Y.C., and D.L. House, “Microprocessor Technology Trends,” *Proceedings of the IEEE*, Vol. 74, No. 12, 1986.

[Ng92] Ng, S. and R. Mattson, “Maintaining Good Performance in Disk Arrays During Failure Via Uniform Parity Group Distribution,” *Proceedings of the First International Symposium on High-Performance Distributed Computing*, 1992, pp. 260-269.

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

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

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

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

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

[RAID96] RAID Advisory Board, *The RAIDBook: A Source Book for RAID Technology*, 5th Ed. St. Peter, Minnesota, 1996.

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

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

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

[Rosenblum92] Rosenblum, M. and J. K. Ousterhout. “The Design and Implementation of a Log-Structured File System.” *ACM Transactions on Computer Systems*, Vol. 10, No. 1., February 1992, pp. 26-52.

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

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

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

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

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

[STC94] Storage Technology Corporation, *Iceberg 9200 Storage System: Introduction*, STK Part Number 307406101, Storage Technology Corporation, Corporate Technical Publications, 2270 South 88th Street, Louisville, CO 80028.

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

[Stodolsky94] Stodolsky, D., Gibson, G., Courtright, W.V., and M. Holland, “A Redundant Disk Array Architecture for Efficient Small Writes,” Technical Report No. CMU-CS-94-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, July 1994.

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

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

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

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

[Wing96] Wing, J. and M. Vaziri-Farahani, “Model Checking a Controller Algorithm for the RAID Level 5 System,” unpublished paper.

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