A Transactional Approach to Redundant Disk Array Implementation

William V. Courtright II

15 May 1997
CMU-CS-97-141

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A Dissertation submitted to the
Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Thesis Committee:
Garth A. Gibson
Martin Francis
Jim Gray
Daniel P. Siewiorek
Jeannette Wing

This work was supported in part by Data General, Digital Equipment Corporation, Hewlett-Packard, International Business Machines, Seagate, Storage Technology, and Symbios Logic. Additional support was also provided by a Symbios Logic graduate fellowship. Carnegie Mellon's Data Storage Systems Center also provided additional funding from the National Science Foundation under grant number ECD-8907068. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of sponsoring companies or the government of the United States of America.

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

Dedicated to the memory of my grandfathers,

Charles Richard Courtright and Jappour Joseph

Abstract

Redundant disk arrays are a popular method of improving the dependability and performance of disk storage and an ever-increasing number of array architectures are being proposed to balance cost, performance, and dependability. Despite their differences, there is a great deal of commonality between these architectures; unfortunately, it appears that current implementations are not able to effectively exploit this commonality due to their ad hoc approach to error recovery. Such techniques rely upon a case-by-case analysis of errors, a manual process that is tedious and prone to mistakes. For each distinct error scenario, a unique procedure is implemented to remove the effects of the error and complete the affected operation. Unfortunately, this form of recovery is not easily extended because the analysis must be repeated as new array operations and architectures are introduced.

Transaction-processing systems utilize logging techniques to mechanize the process of recovering from errors. However, the expense of guaranteeing that all operations can be undone from any point in their execution is too expensive to satisfy the performance and resource requirements of redundant disk arrays.

This dissertation describes a novel programming abstraction and execution mechanism based upon transactions that simplifies implementation. Disk array algorithms are modeled as directed acyclic graphs: the nodes are actions such as "XOR" and the arcs represent data and control dependencies between them. Using this abstraction, we implemented eight array architectures in RAIDframe, a framework for prototyping disk arrays. Code reuse was consistently above 90%. The additional layers of abstraction did not affect the response time and throughput characteristics of RAIDframe; however, RAIDframe consumes 60% more CPU cycles than a hand-crafted non-redundant implementation.

RAIDframe employs roll-away error recovery, a novel scheme for mechanizing the execution of disk array algorithms without requiring that all actions be undoable. A barrier is inserted into each algorithm: failures prior to the barrier result in rollback, relying upon undo information. Once the barrier is crossed, the algorithm rolls forward to completion, and undo records are unnecessary. Experiments revealed this approach to have identical performance to that of non-logging schemes.

Acknowledgments

I am living a charmed life. I am continually blessed with an abundance of opportunity and the support necessary to capitalize upon it. It is my privilege in this brief but sincere statement to acknowledge my family and friends who have helped me, directly and indirectly, in the pursuit of the work described in this dissertation. While this tribute is woefully inadequate and decidedly incomplete, it is my hope that at the very least it serves as a notice of my awareness and appreciation of their concern and efforts. The honor associated with this degree is as much theirs as it is mine.

First and foremost, I am proud to be a part of a great family. Since my birth, my parents, Bill and Mariam, have worked without interruption to provide for my needs and teach me how to lead a rich and productive life. They continually remind me of the pride that they find in me; however, that feeling is surely equaled, if not surpassed, by the pride I have in being their son. My brother Joe, who is three years younger than myself, is more like a big brother to me. I often find myself setting personal goals based upon his precious achievements and counseling. Aside from my immediate family, I am also very close to my grandmothers, cousins, aunts, and uncles. They have all given selflessly to my betterment.

I have also been blessed with a large contingent of friends who have helped me to grow as a person and as a professional. For almost twenty years, I have been able to include Mike Scheaffer in my circle of good friends. Even though we are half a continent apart, Mike is able to see through any facade that I may present and help me through life's little bumps. While at Carnegie Mellon, I have been able to rely upon the friendship of Chris and Nicole Newburn, who made me an extended member of their family. They taught me more about myself than I would have thought possible. When I returned from Pittsburgh after only three years, it was clear that I had grown as much outside my degree program as I had within it. Recently, I have had the sincere pleasure of getting to know Marla Martinous, someone who is very special and provides the sunshine that illuminates my days.

I have had many teachers throughout my life, not all of whom are part of the academic community. Charles Button, my scoutmaster, has taught me a great deal, probably more than either of us realize. Trueman Smith, my high school math instructor, is the first instructor that challenged me on a personal level to excel. Curtis Martin, who I never had as instructor, was a mentor my first year of high school. Curtis left before I was able to enroll in his science courses, but he nevertheless opened the door to creative thinking and introduced me to my first computer: an Ohio Scientific with a cassette drive and 8K of memory (fully loaded!). Robert Higgins, my band instructor, taught me that success does not come easy, but is worth the effort. While studying at the University of Kansas, Gary Minden, a junior faculty member, made the time to take me under his wing and teach me the design skills that have made my career successful. Harry Talley, a senior faculty member who has recently retired, also took an interest me and has offered guidance throughout my career in engineering.

It goes without saying that Garth Gibson has been my greatest instructor. He has pushed me to points that I was certain I could never reach. What I didn't expect is that he would teach me so much about things that I considered to be outside of engineering. I continue to discover things that he has taught me and will probably never be able to fully assess the gifts that he has bestowed upon me. I am still amazed at the patience he displayed as I worked my way through the program.

My doctoral work is also a direct result of the companionship and collaboration of the people of the Parallel Data Lab (PDL). Specifically, Khalil Amiri, Chris Demetriou, Mark Holland, Patty Mackiewicz, Paul Mazaitis, Hugo Patterson, LeAnn Neal Reilly, Erik Riedel, Daniel Stodolsky, Rachad Youssef, and Jim Zelenka have all contributed directly to this work. Patty has been a wellspring of support, providing a ear when it was needed. Despite her busy schedule, she always made the time to take a personal interest in everyone, making Garth's research group a family. The enthusiasm and fresh viewpoints of Khalil and Erik, who both arrived late in my tenure as a graduate student, helped me make it through the final leg of my research. Mark, Hugo, LeAnn and Dan provided enlightened discussions and served as mentors. Mark was also the driving force behind the creation of RAIDframe. Dan, in addition to treating me to gourmet cooking, was also an author of RAIDframe. LeAnn helped with my writing skills. Rachad, who added the simulator to RAIDframe, used his untiring enthusiasm to keep everyone in the lab smiling, and for that I am thankful. Chris, Paul, and Jim provided invaluable support in the lab. Additionally, Chris wrote the striping driver used in Chapter 4 and Jim wrote and debugged large fractions of RAIDframe. Mandana Vaziri, a student who was not a part of the PDL, introduced me to model checking and furthered my understanding of formal verification. Her pleasant demeanor and thought-provoking conversations were always a joy.

At the Electrical and Computer Engineering graduate office, Lynn Philibin and Elaine Lawrence have provided moral and logistical support as I navigated the doctoral program. Jeannette Wing, a member of my thesis committee who hails from the School of Computer Science, has invested a great deal of time in my education, beyond what I would consider to be typical. Dan Siewiorek has provided keen insight at crucial points in my degree program. Jim Gray of Microsoft Research, a very busy and talented man, made the time to advise me over great distances. His genuine interest in my growth is exceeded only by his wisdom.

Finally, I would like to thank the people of Symbios Logic, who paid for my education. I hope to repay that debt many times over in the coming years. Specifically, I recognize Sharon Boyd, Joe Edens, Marty Francis, Ed Marchand, Denise Rajala, Stan Skelton, and Kim Walker. These people have been like family to me, always looking out for my best interest. Sharon is my mother away from home, providing words of encouragement always making sure that I don't work too hard. Similarly, Marty is more like a father than a supervisor. Marty hired me in 1986 and has been my supervisor for almost my entire tenure at Symbios Logic. He is the best. I've never worked for Stan, and I don't recall how we became acquainted, but he has become my guardian angel. If it were not for his patient counseling, I would have burned out on engineering long ago. Joe took the risk, as Director of Engineering, to send a brash young engineer off to pursue a Ph.D. with no assurance of what would become of that investment. Despite the disparity of our rank, I was always able to walk into his office and let him know just what I was thinking, even if that meant complete disagreement. Denise and Kim served as administrators of my Ph.D. program, making sure that tuition was paid but, more importantly, helped me keep my sanity throughout this entire process by listening and providing words of encouragement. Finally, I thank Ed Marchand, who has patiently forgiven missed schedules and excused low productivity the last seven months as I completed my degree.

Table of Contents

List of Tables

Appendix B: Modifying Graphs for Roll-Away Recovery

Appendix C: Data

Index

List of Figures

Chapter 4: RAIDframe: Putting Theory Into Practice

Chapter 5: Roll-Away Error Recovery

Chapter 6: Conclusions and Recommendations

References

Appendix A: Flow Graphs for Popular Array Architectures

Appendix B: Modifying Graphs for Roll-Away Recovery

Appendix C: Data

Index

Chapter 1: Motivation, Problem Statement, and Thesis

The importance of storage systems, historically sidelined as peripheral devices which supported the processing unit of the computer, has dramatically increased as computer installations have become data-centric, rather than processor-centric. Customers are no longer purchasing a single computer and building the storage system around it; they are instead designing the computer around the central database.

The importance of data to customers has necessitated a continual search for improvements in both the performance and the dependability of storage systems. Redundant disk arrays are designed to offer improved performance and, at the same time, increased dependability. This year, Disk/Trend estimates that world-wide shipments of redundant disk arrays will be $12.3 billion, reaching $18.6 billion by 1999 [Disk96a]. This outpaces the growth of commodity disk drives, estimated to by $29.7 billion this year, reaching $45.9 billion in 1999 [Disk96b].

Redundant disk arrays are manufactured by corporations such as Data General, Digital Equipment, EMC, Hewlett-Packard, IBM, Storage Technology, and Symbios Logic (formerly NCR). Despite the fact that storage systems based upon magnetic disk technology have been in production for forty years, commodity production of storage systems that employ redundancy to survive disk faults has only recently occurred. Redundant disk arrays, commonly referred to using the RAID taxonomy introduced by researchers at the University of California's Berkeley campus in 1987, began to increase in popularity when commodity pricing of the small-form-factor disk drives used in personal computers made redundant disk arrays more affordable, dependable, and higher performing than the single large expensive disks (SLEDs) paradigm [Patterson88].

Since the introduction of the original RAID paper in 1988, research in redundant disk arrays has flourished in an attempt to provide a broad spectrum of solutions for a variety of price/performance/dependability trade-offs. Our interest in the work described in this dissertation began with the casual observation that, for reasons of complexity, disk array vendors were limiting the scope of their product offerings to only the most basic redundant disk array architectures. After examining the properties of current redundant disk array implementations, we came to the conclusion that vendors were employing design practices that had been used to implement nonredundant disk systems, but were unsuited for the problem domain of fault-tolerant storage systems.

For example, servicing a user's request to write data to a redundant disk array typically requires the execution of a partially-ordered series of actions such as: the allocation of resources such as locks and buffers, the computation of new redundancy information, the writing of new user data and newly-computed redundancy information to disk, and the release of resources. If an error is detected during the execution of these actions, the array is required to complete the user's request, assuming that the fault which caused the error is specified to be tolerated by the array's fault model.

The process used to recover from the error and complete the request begins with an assessment of the state changes made by actions completed at the time the error was detected, and the damage caused by the error. Because some of these actions, such as the writing of new data and the computation of new redundancy information, may occur in parallel, the state space which must be explored may be significant in size. The process of identifying the state space is often manual, tedious, and prone to mistakes. Furthermore, the state space is architecture specific—as new algorithms are introduced, so are new execution states.

The second step in this process is to complete the user's request. Because errors may corrupt (or make unavailable) information which is necessary to complete the operation, the algorithm being used to service the request may need to be altered. The specific alterations to the algorithm are a function of the error and the current execution state—as the execution state space increases, so does the number of alternate algorithms that must be created by the programmer.

This process of mapping current execution state to a carefully prepared model of the entire execution state space and then altering execution to move from the point of error detection to request completion is known as *forward error recovery* [Lee9c, Stone89]. From informal conversations with practitioners, I discovered that 60-70% of the code found in implementations based upon forward error recovery may be devoted to error recovery. Friedman reports this number to be as high as 90% [Friedman96].

Because forward error recovery schemes are architecture specific, extending existing implementations to support new array architecture can be difficult. This is unfortunate because, as I will demonstrate in Chapter 2, a wide variety of disk array algorithms can be composed from a relatively small set of actions, such as disk read and XOR. Intuitively, it stands to reason that a basic library of these actions should be able to support a multitude of architectures.

Finally, verifying the correctness of a design which is built upon case-by-case analysis is difficult. Not only must the error-free execution of the algorithms be verified, but the identification and case-by-case treatment of all distinct error scenarios must be verified as well.

Database systems have long employed transactions, independent units of work which guarantee atomic (all-or-nothing) failure semantics without regard for the context in which the transactions are executing. By logging the state changes that are made by each transaction, errors which cause a transaction to abort (fail prior to completion) can be recovered automatically by returning the system to the state which existed prior to the execution of the transaction which failed. This all-or-nothing semantic eliminates the need for manually identifying and processing errors, instead allowing programmers to simply design transactions which begin in a state which is free from error—the underlying system assumes the responsibility for error recovery. This system also guarantees that the execution of each transaction is isolated from that of other transactions, a property commonly known as *isolation* or *serializability.*

Redundant disk array operations can be treated as transactions, meaning that the execution of these operations can be mechanized in a general fashion, independent of array architecture and the algorithms that are used to perform array operations. In this dissertation, I argue that forward error recovery schemes, used with arguable success in non-redundant disk systems, are unsuited for fault-tolerant disk systems. Employing a novel programming abstraction, which graphically represents disk array operations as directed acyclic graphs, and specializing error recovery technology found in transaction processing systems, it is my thesis that by modeling redundant disk array operations as transactions:

> *redundant disk array software can be constructed in a fashion that reduces the need for hand analysis of errors and concurrency, increases the fraction of reusable code by reducing architecture-specific error recovery, achieves performance comparable to hand-crafted implementations, and does not consume significant resources.*

To this end, I have written this dissertation which describes our experiments and approach to redundant disk array software implementation. Also included in this dissertation is a description of *RAIDframe*, a software package for implementing and evaluating redundant disk arrays developed by researchers at Carnegie Mellon's Parallel Data Laboratory [Courtright96c, PDLHTTP, RAIDframeHTTP]. RAIDframe employs the programming abstraction and error recovery technology described in this dissertation and was used extensively to demonstrate the claims made throughout the dissertation.

I hope that the work described in this dissertation will lead to an avoidance of software faults in production systems; unfortunately, I have no empirical data to support this belief. Rather, I argue that the simplicity of the approach, its demonstration in RAIDframe, and the ability to formally verify systems using these techniques as correct, are enough evidence to merit further investigation.

Before proceeding with a description of our approach to implementing redundant disk array software, the dissertation commences with a study of background material. Chapter 2 introduces the fundamental terminology used to describe fault-tolerant systems as well as disk drive and disk array technology. The fault tolerance terminology I use is consistent with that commonly found in the field [Gray93, Siewiorek92, and Lee90c]. Chapter 2 also reviews disk and disk array technology, including a description of the fault model used in our experiments. This model is based upon a general array controller architecture found in commercial products, and the accepted notion that disk arrays fail non-atomically when power failures and crashes are encountered.

Chapter 3 introduces a graphical programming abstraction based upon directed acyclic graphs in which the nodes represent actions such as disk read and XOR and the arcs represent data and control dependencies. The nodes are designed to be atomic and, by making them undoable, a mechanism for executing these graphs, which includes recovery from node failures, is described. Because the mechanism is general (independent of graph structure), the need for hand-crafting code to clean up after errors encountered during the execution of array operations is eliminated.

This graphical programming abstraction, and a modular partitioning of functions such as mapping and algorithm (graph) selection was the basis for *RAIDframe*, a prototyping framework I describe in Chapter 4 that permits researchers to quickly implement disk array architectures and evaluate them in real computing environments. This chapter presents an anecdotal history of our efforts as we developed eight disk array architectures in RAIDframe, in which code reuse was consistently above 90%. Also included in this chapter is an examination of RAIDframe's efficiency, and a validation of its response-time and throughput performance against expectation. For a nonredundant disk array, RAIDframe was found to provide results consistent with a non-redundant hand-crafted striping driver, but required 60% more CPU cycles. For redundant disk arrays, RAIDframe produced results consistent with simple models that predict throughput as a function of the amount of disk work in the system [Patterson88].

The study of error recovery is continued in Chapter 5, which begins by examining the cost of the naive execution mechanism described in Chapter 3. That mechanism maintained enough undo information to permit rollback from the failure of any node in the graph. Using RAIDframe, I show that the penalty for creating records for undoing disk writes, which requires a pre-read of the disk sectors to be overwritten, results in a 33-50% reduction in the small-write throughput of the eight architectures that we studied. This degradation motivates the development of a *roll-away error recovery*, an adaptation of the two-phase commit protocol found in many transaction monitors. Roll-away error recovery eliminates the need to perform expensive undo logging for actions such as a disk write while maintaining the simplicity of a mechanized execution. Each array operation is divided into two phases, separated by a barrier which requires completion of the first phase before execution of the second phase may commence. Undo logs are maintained for actions in the first phase of the operation—if an error is detected during the execution of the first phase, these logs are used to automatically roll the operation back to its beginning, failing it atomically. The operations are structured so that once execution of the second phase commences, the failure of any single action will not prohibit the remaining actions in the operation from being completed. This property guarantees that the an error detected during the execution of the second phase of the operation will not alter the operation's algorithm, enabling execution to simply roll forward to completion. Roll-away error recovery was implemented in RAIDframe and Chapter 5 demonstrates that its performance is identical to those systems which do not employ logging.

I conclude the dissertation with Chapter 6, which reviews the specific contributions and practicality of this work and presents a list of opportunities for future study. Notably, this list includes a proposal for extending roll-away error recovery, through the use of durable undo and redo logs, to ensure the atomic survival of power failure and software crashes.

Chapter 2: Fault-Tolerant Disk Storage

Disk storage systems offered forty years ago, such as IBM's Disk Drive RAMAC 350, were designed to tolerate only a minimal set of faults, relying instead upon external mechanisms to tolerate failures in the disk system. Over time, disk storage products have simplified the process of integrating them into systems by tolerating a wide variety of faults without the need for external intervention. Today, customers are able to choose between a variety of disk storage products, from commodity disk drives such as Seagate's Barracuda family which sell for less than 20¢ per MB, to single-fault tolerant disk arrays such as Compaq's server-based arrays at 25¢ per MB, subsystems such as Symbios Logic's MetaStor arrays which sell for 50¢ per MB, and EMC's Symmetrix arrays which employ large (1 GB) caches and sell for almost $2 per MB, to multiple-failure tolerating disk arrays such as Storage Technology's Iceberg which sells for $4 per MB. In fact, in 1995, 158 vendors collectively supplied 594 disk array products, ranging from software and board products for desktop systems to free-standing rack-mounted enclosures for mainframe systems [Disk96a].

It is my goal that, in addition to serving as compendium of background material for this dissertation, the material in this chapter will convince the reader that there is a great deal of commonality in redundant disk arrays which would suggest that there should be a great deal of commonality in redundant disk array implementations. Later, in Chapter 3, I demonstrate that exploiting this commonality is a problem and suggest a solution.

This chapter begins with a review of the terminology and metrics used throughout the dissertation. Section 2.3 describes well-known procedures for implementing dependable systems, which can be applied to arbitrary systems, including disk arrays. Disk arrays are then introduced by first describing disk drives, the fundamental building block of disk arrays, in Section 2.4. Redundant disk arrays, which are capable of tolerating a variety of failures, are then described in Section 2.5. This section includes a review of a variety of architectures which provide cost, performance, and dependability optimizations, their fault models, and the algorithms they employ for accessing data.

2.1 Terminology

Real systems are not perfect. Regardless of the care taken during design, construction, operation, and maintenance, defects will be introduced that result in exceptional behavior. In many applications, such as spacecraft navigation systems, the dependability of computing equipment is critical—even the most minute error can commit a crew to travel where no man has gone before! Computing equipment used in these and other applications must therefore be designed to cope with defects.

Fault-tolerant computing, the science of creating dependable systems from imperfect components, is well studied and the brief introduction provided here is by no means complete. Many excellent texts, such as *Reliable Computer Systems: Design and Evaluation* by Siewiorek and Swarz [Siewiorek92], *Transaction Processing: Concepts and Techniques* by Gray and Reuter [Gray93], and *Fault Tolerance: Principles and Practice* by Lee and Anderson [Lee90c], offer a much broader and deeper introduction to this field than is presented here. The purpose of this section is to introduce the terminology necessary to understand the design goals and failure mechanisms of redundant disk arrays.

2.1.1 Faults, Errors, and Failure

All computing devices, whether constructed from hardware, software, or some combination, have a specified operating behavior. That is, all devices are expected to behave in a predictable manner. When the behavior of a device is inconsistent with expectation, an *error* is said to exist. Erroneous behavior is a direct consequence of the presence of a *fault*, a defect in the device. The time between the introduction of a fault and detection of the error it manifests is referred to as *error latency.*

Faults can be either *man-made* or *physical*. Man-made faults may be introduced throughout the life of a device and are further subdivided into categories such as: *design*, *manufacturing*, *installation*, and *maintenance faults*. Physical faults are generally the result of environmental factors such as temperature, vibration, or chemical processes such as corrosion.

Faults are also categorized as a function of their duration. *Permanent faults*, also known as *hard faults*, are a result of defects which are always present and may only be removed by an explicit repair operation. *Transient faults*, also known as *soft faults*, are temporal and appear to repair themselves over time and disappear. Transient faults are often a result of an environmental anomaly. A recurring transient fault is referred to as an *intermittent fault*. Intermittent faults are the result of a combination of a permanent defect and an infrequent input pattern or environmental condition.

To better understand the relationship between faults, errors, and failure, consider an accounting system used by a bank to maintain savings accounts. The system may have a design fault (man-made fault) in the algorithm that computes interest, creating a latent error. At the end of the month, when interest is to be paid, the error becomes effective at the time when interest is computed incorrectly. The error is detected when the bank's customers all pay off their loans early. Because the design fault is always present, it is classified as a permanent (man-made) fault.

2.1.2 Fault Models and Semantics

Formulating an expectation of the behavior of a system is the first step in characterizing the dependability of a system. This "expectation" is documented in what is known as a *fault model* which specifies: the types of faults which are thought likely to occur, the damage that they cause, their effects upon the behavior of the system, and the frequency of their occurrence.

Each module in a system can often fail in many ways. However, the effects of the errors manifested by these faults may often be similar, or even identical. Instead of coping with each of these faults in a distinct manner, fault models often group sets of faults which produce similar or identical errors into what I call a *fault domain*. For example, there are many distinct faults that can lead to the loss of power within a disk system (e.g., loss of line power, power supply failure, operator error). Because the effect of each of these faults is to deprive the system of power, these faults can be treated in a like fashion.

The relative timing of faults in systems is an important part of the fault model. For example, a redundant disk array may be able to survive either a loss of line power or the loss of a battery without failure; however, if the two faults should be present simultaneously, failure (loss of data) will result. Such a model is commonly referred to as *single fault tolerant*, meaning that the system will survive only the failure of a single fault domain at any instant. Of course, it is possible to create *N-fault tolerant* systems which tolerate the simultaneous occurrence of at most (*N*) faults.

Finally, the fault model must describe the effect of faults upon the behavior of the system. Users of a system request work, and these requests are performed as a sequence of one or more *operations*. The operations are executed with an expected behavior, or *semantic,* and the fault model specifies the effects, if any, of predictable faults upon this behavior.

2.2 Metrics

Ultimately, two metrics are used to measure the dependability of computing equipment: *reliability* and *availability* [Laprie82]. Reliability is the probability that a device will operate without failing for a period of time, *t*, and is computed as:

$$R(t) = e^{-\int_0^t h(x)dx} \quad \textbf{(EQ 2-1)}$$

where *h(x)* is the *failure-rate* or *hazard function* of the system which specifies the instantaneous failure rate of the device. If a system is known to have a constant failure rate, the failure-rate function becomes constant and is often specified as reciprocal of the *mean-time-to-failure* (*MTTF)*, the expected time interval between some instant in time and the failure of the system.

When a system fails, an explicit repair operation is required to restore service. The expected time required to complete a repair operation, *mean-time-to-repair* (*MTTR*) is measured from error detection to completion of the repair. Availability is a prediction of the fraction of time that a device will be able to provide service. Assuming constant failure and repair rates, expected availability is defined as:

$$Availability \equiv \frac{MTTF}{MTTF + MTTR} \quad \textbf{(EQ 2-2)}$$

The denominator of this equation, *MTTF + MTTR*, represents the total time between failures and is sometimes represented as the *mean-time-between-failure (MTBF)*.

Finally, the performance of disk systems, is generally characterized by the metrics: *response time throughput*, and *capacity*. Response time is total time required to service a request, measured as the elapsed time from the arrival of a request, read or write, to completion. Throughput measures the rate at which operations are completed and is typically reported as IO/s. Throughput is sometimes used to indicate *bandwidth*, the rate at which data is moved in MB/s. In this dissertation, I will use the IO/s metric when discussing throughput. Capacity is simply the amount of storage, typically measured in gigabytes, of the disk system that is available for user data.

2.3 Improving System Dependability

Creating systems which are less likely to fail (i.e., have a higher MTTF) can be accomplished by either reducing the likelihood that faults will be introduced or by creating procedures for hiding their effects [Randell78]. The first approach, referred to as *fault intolerance* by Avizienis [Avizienis76] and commonly known as *fault avoidance*, is powerful but ultimately limited in its ability to improve dependability because faults can not be entirely eliminated. Typical approaches to fault avoidance employ quality assurance practices such as: using fewer, more reliable components; using only established and well-understood design, manufacturing, and maintenance practices; thorough validation; and restricting environmental conditions. STRIFE (Stress Life) testing is a common fault-detection procedure used by manufacturers. This testing generally exposes a product to environmental and operational extremes until a failure is detected. The root cause of the failure is analyzed, the fault is eliminated (when possible) and testing resumes. By pushing the product to failure, even though the failures may occur outside normal operating parameters, the weak points in the design and implementation are discovered and eliminated.

If the failure rate or time required to repair a failed system is too high, availability and reliability will drop below acceptable levels. When this occurs, fault avoidance techniques must be supplemented with procedures for hiding the effects of faults. These *fault-tolerant systems* are able to survive faults without manifesting their effects to the outside world. This process is typically accomplished through the use of redundancy [von Neumann56].

2.3.1 Detection, Diagnosis, and Isolation

The first steps taken in achieving fault tolerance are: *detection*, *diagnosis*, and *isolation*. Fault detection discovers the presence of a fault by detecting the errors it created. A good example of this is parity encoding, a popular method used by information systems to detect single faults in binary codewords [Hamming50]. Illustrated in Figure 2-1, parity is a single-fault-detection code because detection of all single bit errors is guaranteed.

Because fault detection schemes such as parity do not necessarily determine the location of the fault, methods for diagnosing the location of faults are necessary. Many proven techniques exist for diagnosing permanent faults. Faults in information are generally located by using error-correcting codes such as Hamming codes which, unlike parity, are able to detect, locate, and repair corrupted data [Hamming50, Arazi88]. However, as explained in Figure 2-1, if the location of errors in data are known, the data becomes an *erasure channel* and n-bit error detecting codes can be used to correct n-bit errors.

0 0 1 1 0 1 0 0	1	$\xrightarrow{\text{fault}}$	1 0 1 1 0 1 0 0	1
information	parity		information	parity
$\sum$bits = 0			$\sum$bits $\neq$ 0 (error!)	

Figure 2-1 Parity codes detect single errors

Parity codes detect single errors by introducing an extra "parity" bit that forces the binary sum of all bits, information plus parity, to be either even or odd. In this example, the codeword uses even parity, meaning the binary sum of all bits is expected to be zero. Over time, a fault occurs which causes the left-most bit to be transformed from a zero to a one. The error is detected by noting that the sum of the bits is no longer even. Note that the location of the error (the left-most bit in this example) can not be determined from the non-zero summation alone. However, if the location of the failed bit was also known, then that bit could be toggled to correct the error. In general, given the location of a set of detectable errors, an error detection scheme can be used to correct errors [Hamming50].

Duplicating components is a common method of diagnosing hardware faults. For example, Triple-Module Redundancy (TMR) requires that three modules (possibly identical) compare results [Kuehn69]. A voting module is used to compare the responses of the three modules. If a discrepancy is found, the faulty module is assumed to be the minority voter. Similarly, software systems employ *N-version programming* to detect faults associated with software design and construction [Chen78, Elmendorf72]. In these systems, the outputs of a number of independently developed program modules (*N*) are compared for inconsistencies. Again, a voting module is used to detect errors and locate failed modules.

An alternative to diagnosing faults by voting is to use a single component and monitor its output. This is the basis for a software technique called *recovery blocks* which detect software faults through the use of an explicit acceptance test [Anderson85, Horning74]. First proposed by Horning, Lauer, Melliar-Smith, and Randell, recovery blocks do not concurrently execute multiple software modules but instead monitor the output of a single primary module. If the acceptance test detects an error, a secondary module is called upon to replace the primary module and the previously-failed procedure is retried.

It is possible that malicious sources can introduce faults which are intended to avoid detection. These faults are classified by Lamport, Shostak, and Pease as *byzantine* because, like a Byzantine general looking for spies among his commanders, a system's fault diagnosis system has difficulty discerning their presence [Lamport82].

Once a fault has been detected and located, its effects can be contained by isolating the faulty module from the system. To minimize the damage caused by a fault, fault models commonly assume that modules are able to detect, diagnose, and isolate faults in an expedient manner. This behavior, referred to as *failfast* or *failstop*, requires modules to stop themselves as soon as an error is detected, thereby preventing the spread of the fault. For example, instead of a disk drive returning erroneous data which would be passed on to applications for processing, it returns a message indicating that the presence of an error prevents the request from being completed. The time required for detection is assumed to be small, hence the term "failfast" [Gray93].

Failfast modules are self-checking, meaning that diagnosing faults in systems composed of failfast modules is relatively straightforward. This is the basic premise of redundant disk arrays which rely upon error-correcting codes and other mechanisms internal to a disk drive to report any faults which may be present within the disk [Patterson88]. Furthermore, failfast modules simplify the task of isolating faults within a system, reducing the likelihood of their effects spreading to otherwise error-free components.

2.3.2 Recovery, Reconfiguration, and Repair

Once the presence of a fault has been detected and its location is known, the processes of *error recovery*, *reconfiguration*, and *repair* may commence. Error recovery removes the manifestations of the fault, restoring the system to a consistent state which is free from error. With the fault contained and the system in a consistent state, components may be reconfigured to restore service; however, until the failed component has been repaired, service may be degraded. Additionally, the dependability of the system may decline because the fault may have removed a component from service. To restore the fault tolerance of the system, the failed module must be repaired.

Time-based redundancy allows transient faults, which appear to repair themselves, to be survived by simply retrying the operation which failed [Gray93]. Intermittent faults however, are more difficult to overcome—retry may work, but in some instances the recurrence rate of the fault may be too high, driving the dependability of the system down to an unacceptable level. Therefore, a system's ability to tolerate intermittent faults is often a function of the robustness of the system's fault-diagnosis mechanisms—if the fault can not be diagnosed, the system is doomed to fail repeatedly.

The failed module, once removed from service, must be repaired. Because availability is a function of the time to repair a fault (EQ 2-2), many systems provide on-line repair services which are capable of repairing a failed module without taking the system off-line. For example, redundant disk arrays often include spare drives for use in replacing failed drives [Gibson93, Symbios95a]. When a drive fails, the array is reconfigured to use a spare and the data stored on the failed drive is reconstructed onto the spare. The array is available to service requests throughout the entire repair[1] process although, as Holland demonstrates, the array may offer lower performance during this interval [Holland94]. At a later time, the failed drive is physically removed from the array and another spare drive is inserted in its place.

2.3.3 A Closer Look at Error Recovery

Methods for removing errors from a system fall into two general classes: *forward error recovery* and *backward error recovery*. Forward error recovery methods remove the effects of an error by moving the system to a new, corrected state whereas backward error recovery returns the system to a previous state [Lee90c, Laprie82].

Perhaps the most popular method of forward correction is simply retrying an operation. This method can be used to correct errors due to the failure of *idempotent* operations which are the result of a transient fault. Idempotent operations have the property that state changes are not a function of the number of times the operation is executed; that is, repeated execution of an idempotent operation causes no additional state changes beyond those associated with its original execution.

Retry is unable to remove hard errors because they will only be repeated, and retrying operations that are not idempotent introduces undesired state changes. When retry is not applicable, forward error recovery schemes apply a distinct corrective measure given the current situation. For example, if the trajectory of a spacecraft is discovered to be in error, the appropriate action would be to fire the rockets in a manner that would guarantee that the rocket will safely reach its destination. To determine the corrective rocket firing, the current trajectory, position, intended destination, and other parameters must be known.

Systems that employ forward error recovery may construct a dedicated recovery scheme for each possible error scenario (error type and context) and therefore require an intimate understanding of the system. As the complexity of the system increases, the number of recovery schemes which the programmer must create grows with the number of error scenarios. Additionally, the ability to predict all possible error scenarios diminishes. The implications of this are two-fold: design faults are likely to be introduced and proper validation becomes increasingly difficult. Errors overlooked during design are easily overlooked during validation. Also, the number of combinations that must be verified may be large—this problem is compounded in systems which concurrently execute large numbers of independent operations.

Instead of trying to move forward to a new system state, backward error recovery schemes return the system to a previous state, referred to as a *recovery point*, that is assumed to be free from error [Randell78, Stone89]. This is generally accomplished by periodically storing *recovery data*, information that describes the state of the system, dur-

1. Holland divides what I call the “repair” process into two distinct phases: “repair” and “reconstruction.” Holland studied the time to recompute lost information after a drive was replaced and uses the term “repair” to measure the time to replace the failed drive and “reconstruction” as the time to initialize it’s contents.

ing normal processing. When an error is detected, the system is returned to the recovery point by reinstating the recovery data. The effects of the error, as well as all work completed since the recovery point, are undone. With the system restored to an error-free state, the operation that failed can be retried, either by using the same procedure or, given the presence of a fault in the system, by using a new procedure which does not rely upon the failed module.

Backward error recovery requires that the actions which compose an operation must be undoable. Unlike forward error recovery, backward error recovery techniques do not rely upon an intimate understanding of all possible error types and the context in which they occur; instead, backward error recovery blindly returns the system to a state prior to the detection of an error, irrespective of error type or context.

Because errors can be treated in a general manner, backward error recovery is amenable to mechanization. Two backward error recovery mechanisms found in use today are *checkpointing* and *audit trails*. Checkpointing systems establish a recovery point, known as a *checkpoint*, by saving a subset of the system state, known as *checkpoint data* [Chandy72, Siewiorek92]. When an error is detected, the system returns to the checkpoint, in a process called *rollback*, by restoring the checkpoint data. By returning to the checkpoint, all work completed in the system since the checkpoint was first established is lost and must later be redone.

An important optimization in backward error recovery systems is reducing the *unit of recovery*, or the scope of the rollback operation. The unit of recovery determines the amount of work that will be undone as a result of rollback. The unit of recovery can be reduced from a global rollback (restoring the entire system to a previous state) to a per-operation rollback (restoring the state changes made by the operation which failed to previous values) by *logging* the individual state changes of each operation. Logging, also known in the literature as *journalling* or *audit trails*, is widely used in database systems [Bjork75, Verhofstad78, Gray81].

2.3.4 Transactions

A special class of operations, *transactions*, are executed in a manner which guarantees the semantic properties of: *atomicity*, *consistency*, *isolation*, and *durability*. Collectively referred to as "ACID" by Haërder and Reuter, these are the defining properties of transactions [Haërder83, Gray93].

Atomicity requires that each transaction either completes successfully or leaves the system unchanged. Atomic transactions eliminate the need for programmers to interpret and correct incomplete state changes and therefore greatly simplify the process of coping with errors [Lomet77, Lynch94]. Consistency implies that each transaction in the system is only allowed to introduce valid state changes to the system. For example, a transaction would not be permitted to withdraw $100 from an account with a balance of $50.

Transactions are frequently used in systems that are characterized by high concurrency and it is important to ensure that independent transactions do not interfere with one another. Isolation ensures that transactions executing concurrently have no knowledge of one another. Because concurrently executing transactions that run in isolation appear to an external observer to execute in a serial fashion, the semantic property of isolation is sometimes referred to as *serializability.* The property of isolation is important in applications such as redundant disk arrays in which many transactions concurrently modify shared information. Durability ensures that when a transaction commits (completes successfully) the changes that it made to system state will survive subsequent faults such as loss of power and system crash.

Transactions, are an important programming paradigm that have been widely embraced by developers of complex fault-tolerant systems, such as those used in database applications [Bernstein87, Gray93, Lynch94]. In addition to guaranteeing ACID semantics, systems based upon transactions provide recovery on a per-transaction basis. Recovery is typically accomplished by recording information in a durable log that is used by a *recovery manager* to either remove the effects of transactions which failed prior to commit, or to complete the state transformations of transactions which have committed but were interrupted prior to completion.

A transaction is said to *commit* when it can guarantee success. If a transaction encounters an error prior to commit, it must undo all visible state changes. Therefore, the *undo rule* requires the recording of enough information to undo all visible state changes made by the transaction prior to its commit point. These changes are recorded in an *undo log*, which is generally required to be durable to ensure survival of system crashes. If an action can not be undone[1], the transaction must be designed so that it does not execute prior to commit. Similarly, if a transaction commits prior to completing all state changes visible outside the local scope of the transaction, the *redo rule* requires that these state changes be recorded in a durable *redo log* prior to commit.

An important commit protocol is the *two-phase commit*, which is used to coordinate the atomic commit of a transaction across multiple participants [Bernstein87]. A central coordinator asks each participant if they are able to commit. If one or more participants vote "no," the transaction aborts and each participant is asked to roll back by undoing their effects. If, however, all participants vote "yes," the transaction commits and each participant rolls forward to completion.

Gray et al describe an instance of the undo/redo approach used in the recovery manager of the System R database manager [Gray81]. Called the DO-UNDO-REDO protocol, this approach provides a transactional programming abstraction through the use of four distinct programs: DO, UNDO, REDO, and COMMIT. Illustrated in Figure 2-2, the DO program performs an *action* which composes a transaction. Executing the DO program results in execution of the specified action, thereby changing the state of the system. Prior

1. An action that can not be undone is called a *real action* by Gray and Reuter [Gray93].

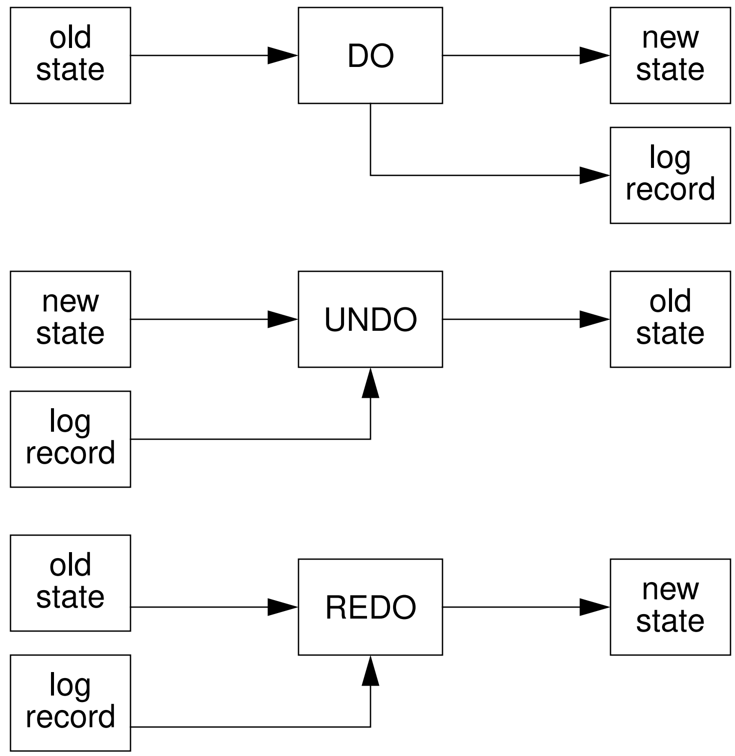


Figure 2-2 The DO-UNDO-REDO protocol

Taken from [Gray81], the transformations of the DO, UNDO, *and* REDO *programs are illustrated above. The* DO *program applies an action that moves the system to a new state and creates a log entry. The* UNDO *program uses the log entry to undo the effect of the* DO *action. In the event of a crash, the* REDO *program is used to restore work previously-completed since the last system checkpoint.*

to committing its changes, the DO program places a record in a log which contains enough information to later undo or redo the effects of the action.

If the transaction fails, any previously-completed actions composing the transaction must be undone. This is accomplished with the UNDO program which reads the undo log in LIFO order, applying the log records and removing the state changes of the failed transaction. If a system crash occurs, the system may be restarted by first restoring the most-recent checkpoint and then using the REDO program to redo transactions which committed after the checkpoint was taken.

2.3.5 Discussion

This section has quickly reviewed the basics of well-known mechanisms for tolerating failures. Dependable systems can reduce the likelihood of fault occurrence through the use of established design practices. However, the occurrence of faults can not be avoided entirely and therefore systems with higher dependability demands must be designed to tolerate faults.

Faults manifest themselves as errors which must be dealt with. I described two fundamental approaches to error recovery, forward and backward. Forward error recovery approaches are generally ad hoc because they rely upon a case-by-case treatment of errors. Furthermore, this case-by-case treatment prevents forward error recovery from being mechanized. Forward error recovery is necessary when dealing with actions which are not undoable.

Backward error recovery approaches, particularly transactions, offer general mechanisms which better manage complexity. By guaranteeing ACID operation, programmers are freed from the burden of interpreting and correcting the state changes made by partially-completed operations. The price of this simplification is the overhead associated with storing information which enables the system to undo the effects of transactions which fail. This information is stored during normal processing and will therefore introduce some performance degradation as well as resource consumption.

2.4 Disk Drives

Magnetic disk drives are the dominant form of secondary storage used in computer systems. Also known as "hard" or "rigid" disk drives, they can be found in applications from hand-held devices to mainframes. In 1995 alone, an estimated 89.6 million drives were shipped worldwide [Disk96b]. With an annual growth rate predicted to be 17.5%, shipments in 1998 are expected to exceed 149 million drives.

Disks are packaged and sold both individually and as collections. The technology, failure mechanisms, and fault model of commodity disk drives are the focus of this section. A discussion of fault-tolerant disk arrays is deferred to Section 2.5.

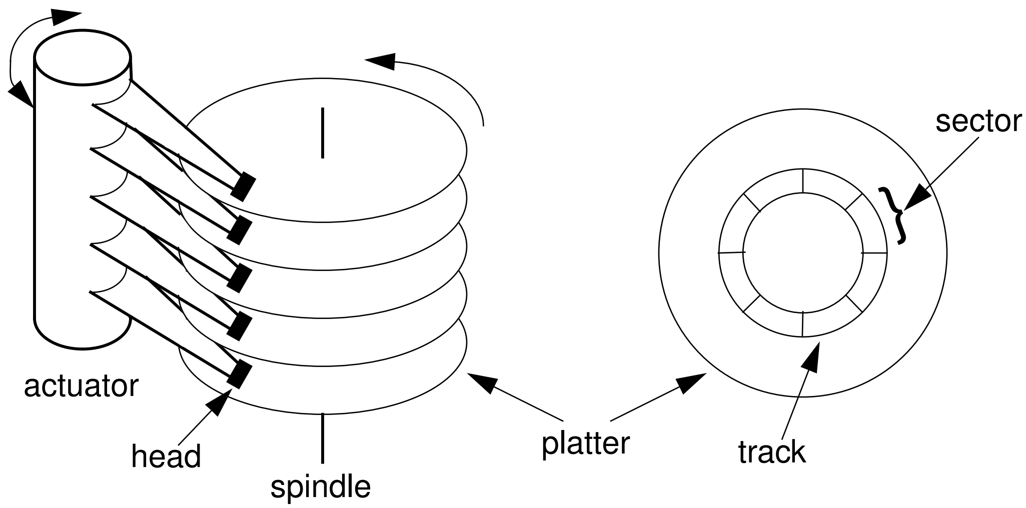


Figure 2-3 Accessing a sector

A disk drive consists of a set of platters, each possessing a dedicated read/write head. The platters are rotated by a common spindle and the heads are mounted on a rotary voice-coil mechanism called an "actuator." Data is recorded on each platter in concentric rings which are subdivided into sectors.

A sector is accessed by first selecting the head which is assigned to the platter containing the sector. The actuator then positions the head over the correct track in a process referred to as "seeking." Once the desired track has been reached, the head waits for the rotation of the platter to place the desired sector directly underneath the head.

2.4.1 Disk Technology

General information on disk technology can be found in the texts *Digital Storage Technology Handbook* by Digital Equipment [Digital89] and *An Introduction to Direct Access Storage Devices* Sierra [Sierra90]. Recent papers by Wood and Hodges [Wood93], and Grochowski and Hoyt [Grochowski96a] explain the driving forces behind current disk trends. A recent paper by Ruemmler and Wilkes provides an excellent discussion of disk performance modeling [Ruemmler94].

Disk drives use magnetic recording techniques to provide nonvolatile storage. Data is stored on rotating *platters*, usually constructed from aluminum and coated with an iron-oxide compound. Current commodity drives are available with 1.8", 2.5", 3.5", or 5.25" diameter platters. The coating is referred to as the *media*. The platters, illustrated in Figure 2-3, rotate on a spindle at a fixed rate of revolution, typically 5,400 or 7,200 rpm. A set of magnetic read/write heads is positioned using a voice-coil mechanism called an *actuator*. There is one head per media surface and all heads move in unison. The rotation

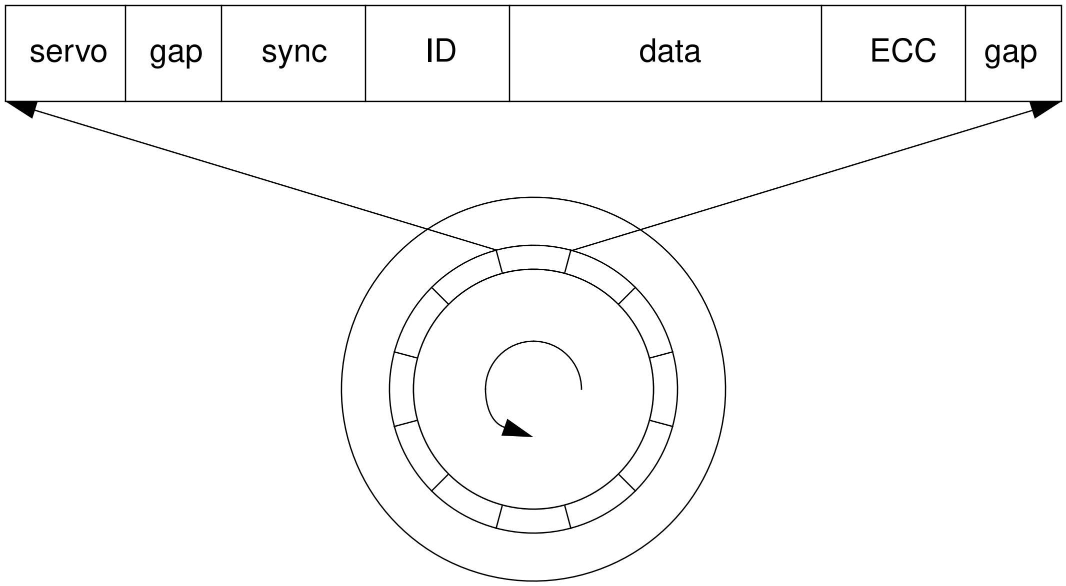


Figure 2-4 Typical sector formatting

A "sector" includes the minimal amount of data a drive can transfer as well as the information used by the drive to locate the data and detect and correct any errors during readback. The sector format begins with an embedded servo field which is used for centering the head on the track. A "gap" field is an unrecorded area and used to isolate the "servo" and "sync" fields. The "sync" field contains a pattern which synchronizes the drive's electronics to the information contained in the subsequent fields. Each sector is uniquely identified by information stored in the "ID" field. User data, typically 512 bytes, is stored in the "data" field followed by error detection and correction codes. The sector is terminated with an inter-sector "gap" which is used to isolate sectors.

of the platters creates an air bearing which, given current technology, separates the head from the media by a distance of less than 2 micro-inches [Grochowski96b].

Data is recorded on the media in concentric circles called *tracks* that are further subdivided into *sectors*. A sector is the minimum unit of access offered by the disk. The *areal density* of information which the head/media tribology can support is approaching 1 gigabit per square inch and is increasing at 60% per year. As Figure 2-4 illustrates, each sector contains positioning information as well as check data for error detection and correction during readback. An explicit format operation is used when the drive is installed to frame the sectors on the disk.

Sectors are accessed by first enabling the head for the platter surface which contains the sector. This enabling, called a *head switch*, typically requires 1 ms. With the proper head enabled, the actuator *seeks* to the track which contains the sector. Seek times vary, as a function of the diameter of the platter and workload. Today, the average seek time for specified by most manufacturers for a 3.5” disk is just below 10 ms [Disk96b].

With the head positioned over the proper track, then head reads the track as the platter spins, waiting for the desired sector to move into position under the head. This time, generally known as *rotational latency*, is on average one-half of the time required for platter to complete a full rotation. The average rotational latency for a 7,200 rpm disk is 4.2 ms. At this point, the desired operation, the read or write of the sector, occurs.

2.4.2 Fault Model

Because error correcting codes (ECC) used in disk drives are very successful at detecting errors and because verifying a write cannot be performed without a full rotation of the platter (8.4 ms for a 7,200 rpm drive), writes are not verified and subsequent reads discover bad writes. The primary goal of the check codes contained in the ECC field is the detection of all errors in the data—error correction is a desired but secondary priority. Great care is taken in the design of the check codes to ensure detection of all likely error types. For example, Quantum’s Atlas family of drives uses a 198-bit Reed-Solomon code which can detect a single burst error up to 73 bits in length and can correct up to 10 bytes in a sector [Quantum95].

In addition to ECC codes, disk drives use retry and other techniques to recover from read errors. For instance, some IBM drives which encounter a media error in a read operation invoke a 50-step recovery process to isolate the failure mechanism and recover the data without loss [IBM95]. Ultimately, disks, such as those in Seagate’s Barracuda family, specify a recovered error rate of less than 10 errors in every 10^{11} bits transferred [Seagate95].

When ECC and other recovery measures fail, the disk will mark the sector as “bad” and fail the read operation. For Barracuda drives, this happens for less than 1 sector in every 10^{14} bits transferred. Contemporary disk drives maintain a pool of spare sectors, and the disk has the capability of automatically performing reconfiguration, replacing the failed sector with a spare from the pool. This leaves the process of repair, (reconstructing the lost information) to an external client.

Most important of all, the likelihood that a disk drive will return incorrect data is negligible. For example, Barracuda drives specify a failure rate in error detection of less than 1 sector in every 10^{21} bits transferred. Ruemmler and Wilkes recently traced a number of disks which supported UNIX file systems [Ruemmler93]. The highest read rate they observed for a single disk was just under 5.2 million sectors read in a two month period. Using a Barracuda disk drive and assuming 512 byte sectors, the Barracuda’s error detection function would have an expected MTTF of over 7.8 billion years! Methods for tolerating the failure of a drive to return correct data exist in the form of end-to-end detection mechanisms. Throughout the remainder of the dissertation, I restrict my analysis to disks which report all errors.

2.4.2.1 Fault Model Used in This Work

I treat single-sector operations as atomic: the sector is either written in its entirety or not altered; subsequent reads either complete successfully or fail. Furthermore, disk operations, regardless of size, are idempotent, meaning that a failed write operation can be retried without ill effects.

Multi-sector disk operations are not atomic. A disk drive does not guarantee that once begun, a multi-sector operation will complete. Furthermore, individual disks are generally permitted to reorder a multi-sector request to reduce rotational latency [Digital89, ANSI91]. Therefore, any subset of a multi-sector write operation may fail.

Faults in head positioning and recording mechanisms are avoided through automated internal calibrations which occur as frequently as every 10 minutes. These calibrations compensate for changes in temperature which modify mechanical and electronic operating parameters. Other periodic maintenance include validating the integrity of the drive's software by recomputing a ROM checksum and sanity checks of the electronics [Seagate95].

Over time, disk drives will eventually fail in a catastrophic manner from which they can not recover. In his dissertation, Gibson studied exponential, Weibull, and Gamma models of disk failure distributions and concluded that exponential distributions are sufficient when modeling mature products [Gibson92]. Assuming an exponential distribution, the failure rate becomes constant and reliability is calculated as:

$$R(t) = e^{\frac{-t}{MTTF}} \tag{EQ 2-3}$$

where MTTF is specified by disk vendors as an estimate of the expected amount of time that the device will operate from the time it leaves the factory, assuming operation in a controlled environment [Stone90].

It is worth noting that this model, specifying MTTF to indicate a drive's reliability, has come under criticism due to recent quality problems associated with the younger disk drive products of several major disk drive vendors. The International Disk Drive Equipment and Materials Association (IDEMA) has created a subcommittee, composed of disk vendors and customers, to investigate better methods of specifying disk reliability [IDEMA96].

Disk drives generally have a warranty or expected operating lifetime of two to five years. To achieve this, disk drives such as the Seagate Barracuda drives have an MTTF of 800,000 hours (91 years). If these drives are to be used for a lifetime of 2.5 years, their expected reliability is 97.3%.

$$R(21,915\ \text{hours}) = e^{\frac{-21,915}{800,000}} = 97.3\% \qquad \textbf{(EQ 2-4)}$$

Clearly, Seagate does not have 91 years of data on this product, so how can their MTTF rating assumed to be credible? The answer is twofold: first, disk vendors base MTTF predictions on an analysis of the reliability of common disk drive components used in previous products. Second, testing is accelerated by subjecting the disk drive to extreme operational conditions (power, temperature, and activity) which identify the weak points of the design, enabling vendors to understand the mechanisms which are likely to cause the majority of drive failures.

A catastrophic disk failure implies that the entire contents of a disk drive are permanently lost. Catastrophic disk failures are often the result of a head crash or the failure of disk electronics. Recovering from a catastrophic disk failure requires replacing the disk and reconstructing its contents. The catastrophic failure of one disk does not affect the remaining disks in the system; however, this does not necessarily imply that the faults that lead to catastrophic disk failures are independent. For instance, *stiction*, the attractive force between a head which is parked (resting on a non-rotating platter), can be so large that the drive's spindle motor is unable to rotate the platters (the drive fails to "spin up"). This is a manufacturing defect and can therefore affect many drives from the same production run. If power is lost to a collection of disks with this fault, it is possible that several disks may fail due to the same fault when power is restored.

2.4.3 Discussion

Disk drives are commodity devices that provide nonvolatile storage. Lampson and Sturgis defined what has come to be the classic model of disk failures: write operations rarely fail and sectors which were written successfully may decay over time [Lampson79]. Operations which read a sector are expected to tolerate transient failures, but will fail when a permanent fault is detected. In addition to these sector-level failures, the drive may experience a catastrophic failure (e.g. head crash or the failure of the internal controller) which makes all sectors inaccessible.

The data that I presented in this section supports this model. I treat disk operations as consistent, serializable, and durable. Single-sector operations are treated as atomic but multi-sector operations are not. Finally, disk errors are self-identifying, meaning that disks can be treated as erasure channels. This important property will be used in the following section to simplify the redundancy necessary to tolerate disk failures.

Other fault models do exist. One example is the Mime disk architecture which maintains shadow copies of data to permit a checkpoint-based recovery [Chao92]. When an error is detected during a multi-sector write, Mime uses the checkpoint data to restore the surviving disk sectors to their original state, failing the operation atomically. At the time of this writing, Mime has not moved beyond simulation studies, and contemporary disk systems continue to offer non-atomic failure semantics for multi-sector operations.

In addition to disk drives, storage systems are composed of modules such as power supplies, fans, and cables. I defer a discussion of the failure of these components to Section 2.5.3 which follows the introduction of disk array controllers. This analysis is drawn largely from prior work [Chen94, Gibson92, Schulze89].

2.5 Disk Arrays

Disk performance improvements have not matched those in processors, creating what Pugh refers to as an “access gap” [Pugh71]. This widening disparity in performance is a consequence of the mechanical constraints faced only by disk drives. Disk head positioning mechanisms may reach accelerations as high as 50 g and the resulting forces acting on the head can only be decreased by reducing the mass of the disk arm. Similarly, rotational speedups are held in check by problems with heat and platter rigidity.

In 1990, 5.25” disk drives were the dominant form factor. These drives exhibited average seek times of 12 ms and rotational latencies of 5.6 ms (5,400 rpm) [Disk90]. Today, the average seek time for high-performance 3.5” drives has dipped below 10 ms and the fastest drives offer seek times below 7 ms. By increasing the rotational rate of the platters to 7,200 rpm, rotational latency has dropped to 4.2 ms [Disk96b]. Together, this implies an improvement in disk head positioning of 36% in the last five years.

By comparison, in this same period microprocessors have increased in performance from a SPECint rating of 25 to 325, an increase of 1,200% [Patterson96]! Microprocessors have directly benefited from continued advances in VLSI technology which not only increase clock rates, but also increase the number of transistor a device can support. This, in conjunction with improved design tools, has enabled the implementation of architectural advances which permit the concurrent processing of multiple instructions.

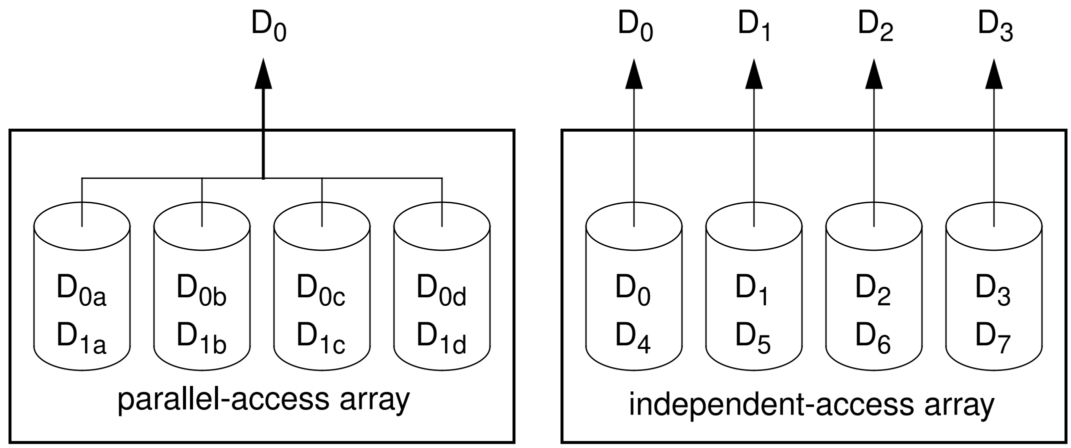


Figure 2-5 Data layouts which enable concurrency in disk storage

In this illustration, D_n *represents a unit of user data. In the parallel-access array, each data unit is distributed across all drives, effectively reducing the time spent transferring data to/from the media by a factor equal to the number of drives. Similarly, the independent-access array is designed to reduce head positioning time by allowing each drive in the array to concurrently service an independent request. This is possible because, unlike the parallel-access array, each data unit is contained entirely on a single drive [RAB95].*

2.5.1 Striping for Performance

Instead of relying upon disk drive technology improvements, disk storage system architects have employed concurrency to achieve higher throughput and decreased response time. By organizing commodity disks into *arrays*, architects are able to increase the overall performance of the disk system by placing data across the drives in a manner that either enables independent accesses to perform positioning operations concurrently, or enables large accesses to transfer data from several drives concurrently [Kim86, Salem86]. By striping data such that a unit of access spans all drives, the array offers an effective bandwidth equal to the transfer rate of a single disk multiplied by the number of disks in the array. Because all disks transfer data in parallel, this disk array architecture is known as a *parallel-access array* [RAB95] and is illustrated in Figure 2-5.

Parallel-access arrays are a common method of increasing disk performance in applications which are dominated by large transfer sizes. A good example of a high-bandwidth system is the Los Alamos High-Performance Data System (HPDS) which is designed to provide high-speed transfer rates to network-attached storage. The system moves approximately 120 gigabytes of data per day and supports data traffic rates up to 60 MB/s [Collins93].

Conversely, many applications are characterized by large numbers of small accesses. Database applications, such as those used in banking applications, are an excellent example. Updating a customer's account involves reading a small record, modifying it, and then writing the result back to disk. Contemporary database systems offer transaction rates which demand up to 10,000 disk operations per second [Gray93]. With an average access time of 11 ms, the throughput of today's disks is less than 100 I/Os per second. However, by distributing the records of the database uniformly across an array of disks, the workload can be evenly distributed, allowing all actuators in the array to be positioned independently. Such an array is called a *independent-access array* or a *stripe set* [RAB95].

2.5.2 Redundant Disk Arrays

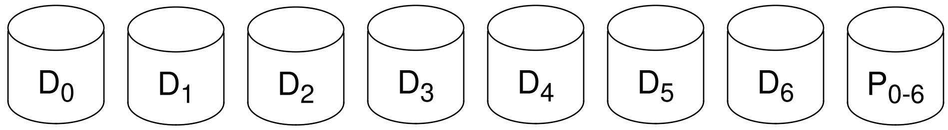
Increasing the number of disks and controllers in an array increases the effective capacity and performance of the array. Unfortunately, this has the simultaneous effect of reducing dependability [Gibson93]. Disk arrays that are designed to tolerate disk faults without loss of data or interruption of service are increasingly common. This subsection introduces the most common redundant disk array architectures in production today, the Berkeley RAID taxonomy, and presents a number of array architectures which are popular in today's research literature but have yet to be offered as products.

2.5.2.1 Encoding

Disk failures can be tolerated by creating a *codeword*, an encoding of user data and check data, and distributing the codeword across an array of disks such that each disk in the array contains at most one symbol (one bit) of the codeword. When a disk fails, the result is equivalent to the loss of (at most) one symbol in the codeword. Because disks are considered to be erasure channels, simple encodings such as single-copy or parity can be used to reconstruct the symbol which was lost.

The two types of data encoding most common to redundant disk arrays are duplicate copies and parity. Copy-based encoding is the most popular of all data encodings used in redundant disk arrays. Array architectures which employ copy-based encodings are referred to as *disk shadowing* or *mirroring* arrays [Bitton88, Gray90b]. Basic mirroring systems maintain two copies of user data. By placing each of the two copies on an independent disk, the failure of one disk can be tolerated by using the copy stored on the surviving disk.

Disk arrays that are based on mirroring are easily understood and can be implemented without specialized hardware. A significant disadvantage of mirrored arrays is that 50% of their total storage capacity is lost to redundant information (the mirror copy). To overcome this problem, disk arrays are also constructed using codewords based upon par-



$$P_{0\text{-}6} = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6$$

Figure 2-6 Codeword in a parity-protected disk array

As previously described in Figure 2-1 on page 12, parity can be used to correct a single error if its location is known. In this illustration, seven disks contain user data and one disk is devoted to parity. Thus, only 12.5% of the array's capacity is lost to redundancy, a significant improvement over mirroring systems which always sacrifice 50%.

---

ity encodings [Gibson92, Lawlor81, Park86, Patterson88]. Parity is computed simply as the XOR of all data symbols in the codeword:

$$P_{0-n} = D_0 \oplus D_1 \oplus D_2 \oplus \ldots \oplus D_n \quad \textbf{(EQ 2-5)}$$

As illustrated in Figure 2-6, parity-protected arrays reduce the capacity overhead lost to redundancy from 50% to:

$$Overhead_{parity} = \frac{1}{N_{disks}} \quad \textbf{(EQ 2-6)}$$

In order to increase significantly reduce the 50% capacity overhead lost to a mirror copy, parity-protected disk arrays generally need to stripe across four or five drives, resulting in an overhead of 25% to 20%.

2.5.2.2 Algorithms for Accessing Information

Reading information from a fault-free array that uses either a copy or parity-based encoding is accomplished by simply reading the data directly from disk. In the case of mirroring, because two copies of the data are stored in the array, the read may be directed

Figure 2-7 Writing and reading data in a mirrored disk array

A copy of a user data block (D) is stored on each disk in the array. Writes update both copies and reads may be directed to either copy. In the event that one of the disks fails, read and write requests are simply directed to the surviving copy.

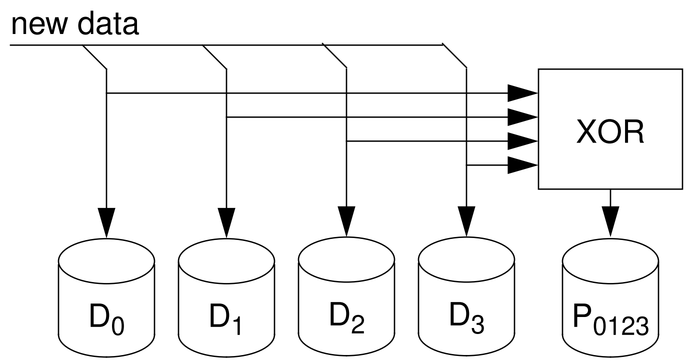


Figure 2-8 The large-write algorithm

The large-write algorithm is used to overwrite an entire codeword in a fault-free parity-protected disk array. New parity is computed from the new data to be written. New values of data and parity overwrite previous values.

to one of two disks. Similarly, as Figure 2-7 illustrates, writing data to a mirrored array requires updating both copies.

Writing data to a parity-protected disk array is not as straightforward. In parity-based arrays, the size of a codeword is constrained only by the number of disks in the array and the symbols are not duplicate copies of one another. Because some accesses may not overwrite all symbols in the codeword, a variety of algorithms are necessary to minimize the performance degradation due to parity maintenance.

For instance, if an access spans an entire codeword in a parity-protected disk array, the *large-write algorithm*, illustrated in Figure 2-8, is used. Parity is computed directly from the data to be written to the array and the entire codeword is overwritten. This algorithm is efficient in that the minimal amount of disk work, one write per symbol (including parity) is performed.

If an access overwrites a single symbol in the codeword, parity can be updated by using the *small-write algorithm* which updates parity via a read-modify-write process. Consider writing the symbol D_{0new} to a codeword which is in the initial (old) state:

$$P_{old} = D_{0old} \oplus D_{1old} \oplus D_{2old} \oplus \ldots \oplus D_{nold} \qquad \textbf{(EQ 2-7)}$$

When the write is complete, parity should be altered, such that:

$$P_{new} = D_{0new} \oplus D_{1old} \oplus D_{2old} \oplus \ldots \oplus D_{nold} \qquad \textbf{(EQ 2-8)}$$

Reading all of the data symbols in a wide (*n* is large) array is not efficient when we are only trying to update a single symbol (D_0). Therefore, using the following two properties of XOR:

$$D_{0old} \oplus D_{0old} = 0 \qquad \textbf{(EQ 2-9)}$$

$$D_{1old} \oplus 0 = D_{1old} \qquad \textbf{(EQ 2-10)}$$

we can compute parity from the data and parity values that we are about to change:

$$P_{new} = P_{old} \oplus D_{0old} \oplus D_{0new} \qquad \textbf{(EQ 2-11)}$$

Intuitively, this simplification can be understood by thinking of the term $(D_{0old} \oplus D_{0new})$ as representing the change made to D_{0old} which is then applied to P_{old}.

Figure 2-9 provides an illustration of an operation using the small-write algorithm. Pre-reading each symbol (data and parity) before their overwrite means that the small-write algorithm is twice as expensive in terms of the amount of disk work performed ver-

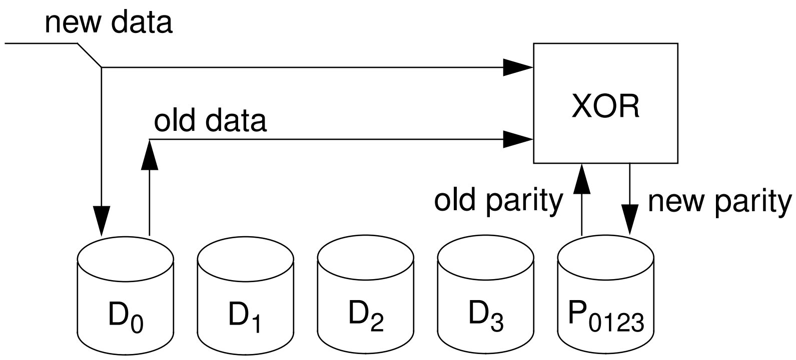


Figure 2-9 The small-write algorithm

To minimize the amount of disk work required to maintain parity when overwriting one symbol of a fault-free codeword, the small-write algorithm performs a read-modify-write of the parity symbol, resulting in a total four disk accesses. In this example, data is to be written to D_0. *This requires pre-reading the old value of* D_0 *and parity (*P_{0123}*), computing new parity as the XOR of old* D_0, *new* D_0, *and old* P_{0123}, *and writing new* D_0 *and new* P_{0123}.

sus symbols written when compared to the large-write algorithm. This disparity is often referred to as the *small-write problem*.

Unlike the large-write algorithm, which can only be used to write an entire codeword, the small-write algorithm can be used to write an arbitrary number of symbols in a fault-free codeword. However, because of its expense in terms of disk work, a third algorithm, the reconstruct write, has been devised. Illustrated in Figure 2-10, the *reconstruct-write algorithm* is similar to the large-write algorithm with the difference being that not all data symbols are overwritten. Those that are not overwritten are read, enabling parity to be directly computed from all data symbols in the new codeword. The reconstruct-write algorithm requires one disk access for each symbol in the codeword, regardless of the number of symbols being written. Therefore, if half or more (but not all) of a codeword is to be written, the reconstruct-write algorithm is generally used because it requires less disk work than the small-write algorithm.

Writing data to a codeword in which a fault has removed a symbol isn’t too tricky. If the disk containing the parity symbol has failed, the write is performed by simply writing the new data—the update of parity is simply ignored. Similarly, if the entire codeword is being overwritten, the large-write algorithm is used but the write of the disk which has failed is eliminated.

If a disk containing a data symbol that is to be overwritten has failed, and the entire codeword is not being written, then the *degraded-write algorithm*, illustrated in Figure 2-11, is used. This algorithm is similar to the reconstruct-write algorithm: the data symbols

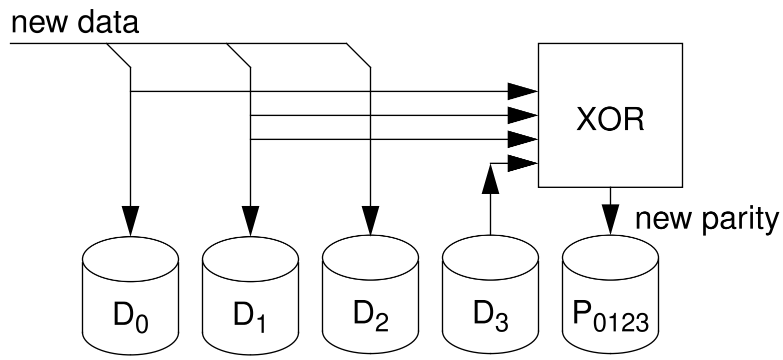


Figure 2-10 The reconstruct-write algorithm

The reconstruct-write algorithm is similar to the large-write algorithm in that new parity is computed from all of the data symbols in the codeword. The data symbols which are not being overwritten (D_3 *in this example) are read from disk.*

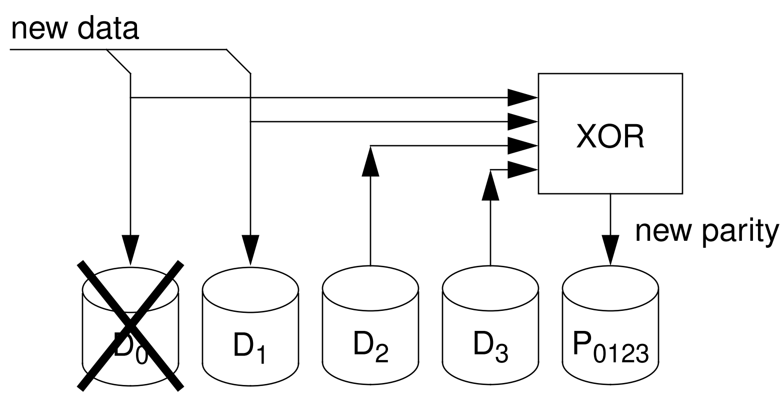


Figure 2-11 The degraded-write algorithm

If a disk containing a data symbol can not be overwritten, the simplest and least-expensive method of updating parity is to use read the data symbols not being overwritten and then compute parity from the entire set of data symbols. In this example, D_0 *and* D_1 *are to be written an array in which the disk containing* D_0 *has failed. The remaining data symbols,* D_2 *and* D_3*, are read from disk and new parity* P_{0123} *is computed as the XOR of all data symbols. The new values of* P_{0123} *and* D_1 *is are then written to disk.*

not being overwritten are read from disk and new parity is then computed from all of the data symbols in the new codeword.

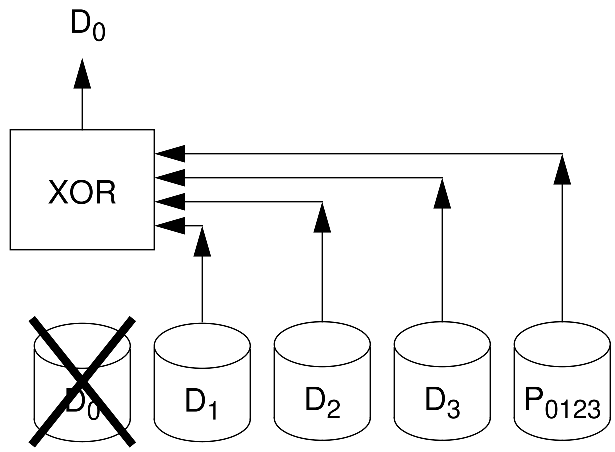


Figure 2-12 The degraded-read algorithm

In this example, a request is made to read D_0*, a symbol which is stored on a disk that has failed. Using the XOR function, the* D_0 *is recomputed from the surviving symbols in the codeword which are read from disk.*

Reading a symbol which was stored on a disk that has since failed is accomplished by reconstructing it from the surviving data and parity symbols. This algorithm, commonly known as the *degraded-read algorithm*, is illustrated in Figure 2-12.

2.5.2.3 The Berkeley RAID Taxonomy

In 1988, researchers at the University of California, Berkeley observed that 5.25" disk drives which were used in personal computers had become commodity devices. Their low cost/actuator made them an attractive building block for arrays; however, their MTTF was predicted to be only 30,000 hours, far below the MTTF of 100,000 hours found in the 14" IBM 3380 disk drives of the day. Realizing that disk performance was becoming increasingly important, Patterson, Gibson, and Katz introduced a taxonomy for redundant disk arrays based upon data layout and encoding [Patterson88, Gibson92]. The taxonomy defined five levels of Redundant Arrays of Inexpensive[1] Disks (RAID). RAID level 0 was later introduced by industry to denote a nonredundant disk array.

The Berkeley RAID levels are limited to single fault-tolerant array architectures. RAID level 1 is used to denote mirrored disk arrays. RAID level 2 is reserved for arrays which employ Hamming codes. The remaining RAID levels, 3 through 5, employ parity to protect data from the failure of a single disk. Parallel-access arrays are categorized as RAID level 3 and independent-access arrays are categorized as either RAID level 4 or RAID level 5. As illustrated in Figure 2-13, RAID levels 4 and 5 are distinguished by the

1. The RAID Advisory Board defines "RAID" as Redundant Arrays of Independent Disks.

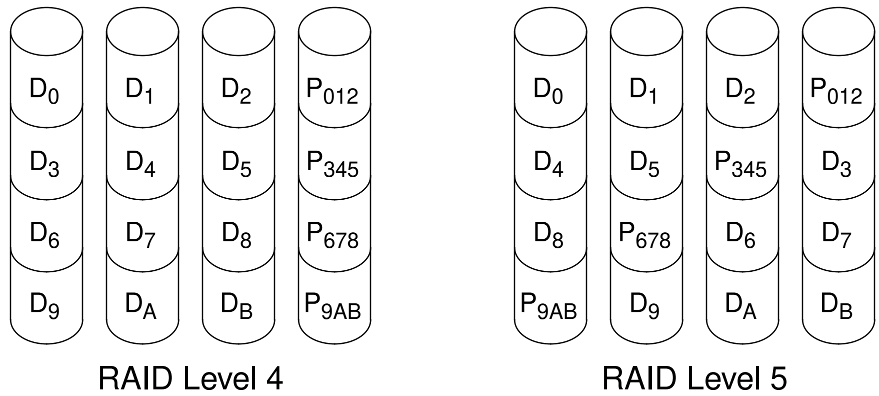


Figure 2-13 Parity placement in RAID levels 4 and 5

This figure compares the data layout and redundancy organizations for RAID levels 4 and 5 for an array of four disks. Data units represent the unit of data access supported by the array. Parity units represent redundancy information generated from the bit-wise exclusive-or (parity) of a collection of data units. The redundancy group formed by a parity unit and the data units it protects is commonly known as a parity group.

placement of parity: RAID level 4 arrays place all parity on a single "parity" disk while RAID level 5 arrays evenly distribute parity across the array.

Any write to a RAID level 4 disk array will involve the disk containing parity, making it a bottleneck in small-write intensive workloads which require a read-modify-write of the parity disk for each user I/O. In small-write-intensive workloads, the throughput of a RAID level 4 disk array will be equal to one-half of the throughput of the parity drive. RAID level 5 arrays, on the other hand, evenly distribute the parity and data workload across the array, and achieve an aggregate throughput equal to one-fourth of the total throughput of the combined disks in the array [Lee90b].

2.5.3 Fault Model

The internal workings of disk arrays (e.g. mapping, encoding, and algorithm selection) are abstracted from users by an interface that makes the array appear as a single disk with higher performance, capacity, and dependability. It is natural to assume that read and write operations should have the same semantics exhibited by disk drives, previously described in Section 2.4.2. This subsection briefly describes the failure mechanisms commonly found in disk array products, the likelihood of their occurrence, and their effect upon operation as perceived by the user. Detailed studies of disk array reliability are widely available in research literature [Chen94, Ganger94, Gibson92, Gibson93, Ng94, Patterson88, Schulze89, Savage96].

Recall that disk drives are assumed to be failstop devices. The same can be said for disk arrays, if they are either constructed from failstop devices, or employ sufficient redundancy to enable the failure of a non-failstop device to be detected in an expedient fashion. In addition to disk drives, disk arrays require power, cooling, cabling. Additionally, the array requires a control mechanism which is responsible for mapping, encoding, and algorithm selection and execution is sometimes implemented in specialized hardware. This *disk array controller* may be implemented entirely in either software or in some combination of software and dedicated hardware.

Representative MTTF values of each of these components are summarized in Table 2-1. With the exception of array control software, the data in this table was taken from a commercially available disk array manufactured by Symbios Logic [Symbios95a] and is consistent with the guidelines outlined by the United States Department of Defense [DOD81]. The reliability of array control software was estimated based on a limited survey of unpublished field return data and conversations with practitioners.

The power system can be treated as a failstop device because of the conditioning normally present in supplies which isolates potentially-damaging line voltage surges from disk array equipment. Array cooling is provided by assemblies which contain two fans. Control electronics are included to adjust fan speed and detect and report fan failure.

The backpanel provides all electrical communication in the subsystem. Backpanel faults can result in the loss of power, cooling, and the loss (or corruption) of communication between the controller and the disk drives and are a single point of failure. Backpanels have no self-checking mechanisms and therefore can not be treated as failstop devices. Instead, devices communicating across backpanels (or cables) must employ some form of

Table 2-1 Disk array component reliability

Component	MTTF (hours)
$MTTF_{back}$ (backpanel failure)	4,566,004
$MTTF_{cbl\text{-}pwr}$ (power cable failure)	10,000,000
$MTTF_{cbl\text{-}scsi}$ (SCSI cable failure)	1,718,200
$MTTF_{cool\text{-}asy}$ (cooling assembly failure)	60,000
$MTTF_{ctrl\text{-}hw}$ (array controller hardware failure)	81,000
$MTTF_{ctrl\text{-}sw}$ (array controller software failure)	40,000
$MTTF_{disk\text{-}cat}$ (catastrophic disk failure)	800,000
$MTTF_{pwr\text{-}supply}$ (power supply failure)	65,000

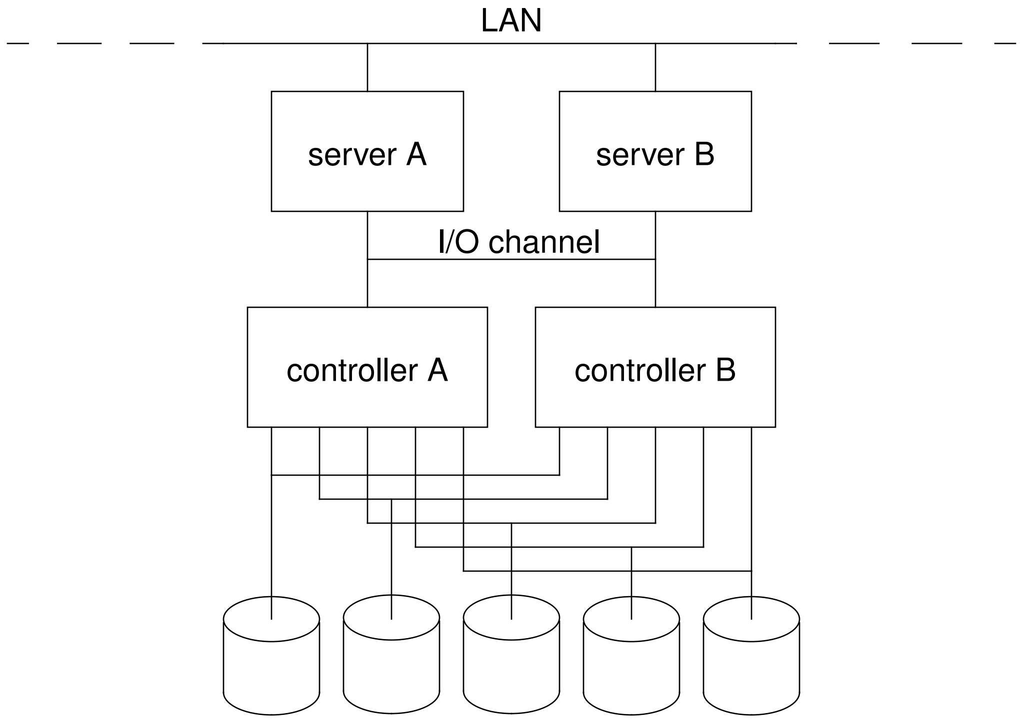


Figure 2-14 Disk array with redundant controllers

This figure illustrates a dual-controller disk array which provides storage to a dual-server cluster. If either controller fails, both servers are assured access to storage via the surviving controller.

end-to-end error detection mechanism, such as those used in SCSI and Fibre Channel [ANSI91, ANSI94] which can be used to identify and isolate backpanel faults.

Hardware-assisted array controllers off-load tasks such as I/O management and parity computation from a host CPU and may also isolate some data transfer operations from the primary system bus. Similar to disk drive controllers, hardware-assisted disk array controllers periodically perform a variety of sanity tests to provide failstop service [Symbios95b]. For instance, it is unlikely that a “brain-dead” array controller will be capable of correctly forming valid messages which impart bad information. When a controller does fail, all local volatile state is lost.

Many array products contain redundant controllers which give the array the ability to survive a controller fault without loss of service. For example, Figure 2-14 illustrates a dual-controller array. The failure of the array control mechanism should not result in a loss of the data from previously-completed write operations. In fact, I assume that the array controller should survive simultaneous failure of the array control mechanism and a disk. When a controller does fail, all work in progress in that controller is interrupted and all volatile state is lost. Generally, nonvolatile state that is required for controller failover

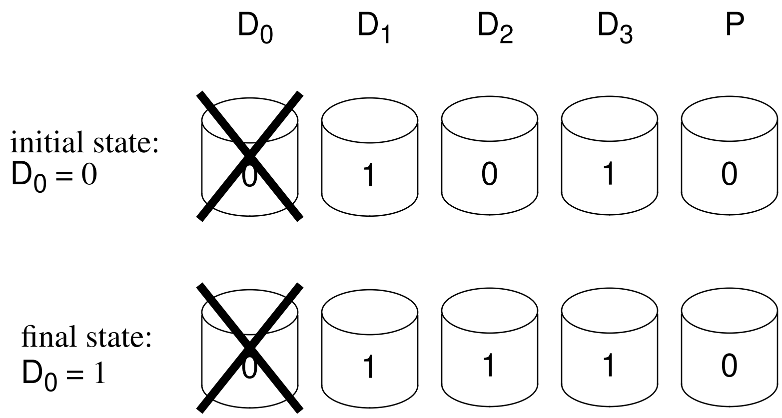


Figure 2-15 The write hole

In this example, the small-write algorithm, previously described in Figure 2-9, is being used to write data to disk D_2 in a parity-protected disk array in which disk D_0 has previously failed. Before the operation begins, the value of the symbol stored on disk D_0 is "0" and can be computed as the XOR of the remaining data and parity disks.

The write operation is interrupted at a point in which new information has been written to D_2 but the parity disk has not been updated. When the controller is restarted, the value of D_0 is computed as "1." This problem is called the "write hole." Unless measures are taken to complete (or remove the effects of) the write operation which was interrupted, data corruption of unrelated data (D_0) has occurred.

---

must be stored in a shared area which is accessible by the surviving controller(s). This could be either a reserved area of a disk or mirrored memory regions which reside on the array controllers.

Because disk arrays do not assume atomic failure of in-flight operations with respect to faults which result in the failure of the array control mechanism, redundant disk array implementors only need to worry about the side effects that can lead to the corruption of codewords. That is, if a user write fails due to a crash, the user can make no expectation (new data, old data, unknown data) about the region being written. However, the user should expect that the remaining data in the array is unaffected. This latter expectation can be difficult to maintain in the event of simultaneous power and disk failures.

Consider as an example the small-write operation illustrated in Figure 2-15, which writes data to an array in which a disk has failed. The operation begins by computing new parity and writing new data. Sometime between the writes of new data and new parity, the array controller crashes (power or software fault). This leaves the codeword in an inconsistent state and the original value of the failed disk can no longer be reconstructed, a problem commonly known as the *write hole* [RAB96]. Avoiding the "write hole" requires that array control mechanisms record sufficient information in non-volatile memory so that interrupted write operations can be completed after the crash.

Predicting the failure rate of array control software is more of an art than predicting hardware reliability. This is because software reliability is a function of many subjective measurements (schedule pressure, experience, code history, development environment, test strategy, etc.). Unfortunately, the literature has almost entirely ignored the reliability of array control software [Patterson88, Schulze89, Ganger94, Gibson93, Ng94, Savage96]. In a recent paper, Chen et al modeled the effects of failures which interrupt the array control mechanism [Chen94]. In this study, they define a crash as: "any event such as a power failure, operator error, hardware breakdown, or software crash that can interrupt an I/O operation on a disk array." A value of $MTTF_{crash}$ of 17,523 hours (1 month) was assumed; however, this value did specifically include array control software. In fact, in reliability calculations for implementations with hardware-assisted array control, system crashes were entirely ignored.

I found the task of determining the defect rates of array control software from field data to be a difficult one. First, many software faults are intermittent, appearing infrequently and when detected, are difficult to reproduce and isolate. Second, when a customer experiences a failure, the first thing that a vendor is likely to do is request that the customer upgrade to the most recent software—this usually eliminates the failure without isolating the fault. Finally, when a controller is returned to the factory for failure analysis (FA), a likely first step that a technician will take is to upgrade the software to its latest revision (no point in debugging software that's known to have bugs). Unfortunately, FA technicians are able to repeat only 15% of failures after upgrading controller software. Collectively, these problems suggest an under-reporting of software faults—failures that are not repeatable by a manufacturer are not categorized as software faults.

Informal conversations with development organizations and the minimal field-return data I have seen suggest a software $MTTF_{ctrl\text{-}sw}$ of 40,000 hours, which implies that software is the single weakest single component of a redundant disk array. I can not provide empirical data to confirm this; however, this finding is consistent with the general findings in the field of fault tolerant systems which report software faults as the leading contributor to failures in "fault-tolerant" systems [Gray90a].

2.5.4 Beyond the RAID Taxonomy

The Berkeley RAID taxonomy was immediately adopted as a de facto standard and an industry consortium, the RAID Advisory Board, was created to standardize the application of the RAID taxonomy to products [RAB95]. The demand for RAID systems exploded, exceeding $9.7 billion in 1995, and estimated to exceed $18.6 billion by 1999 [Disk96a]. This demand is driven by a broad spectrum of capacity, dependability, cost, and performance requirements. Researchers have generated a variety of architectures in an attempt to cover this spectrum. A continued growth in new, specialized array architectures is undermining the completeness of the RAID taxonomy. Summarizing all array architectures proposed in the last five years is well beyond the scope of this discussion. The purpose of this subsection is to demonstrate that by simply changing data placement, data encoding, and the algorithms used to access data, a wide variety of architectures is easily developed.

2.5.4.1 Improving Dependability

To begin, consider that as the number of disks (or any component) used in single-fault tolerant disk arrays increases, reliability will suffer. Burkhard and Menon suggest that by the year 2000, user capacity demands will require a large enough number of disks in the array that the dependability of single fault-tolerant disk arrays will be inadequate [Burkhard93]. A number of architectures have been designed to allow arrays to survive the simultaneous failure of two disk drives without loss of data. Most notable are two-dimensional parity [Gibson92] and EVENODD [Blaum95] which employ parity encodings, and RAID level 6 [ATC90, STC94] which employs a Reed-Solomon encoding.

In single-fault tolerant schemes such as RAID level 5, each bit of user data is a symbol in only a single codeword. In *two-dimensional parity* schemes [Gibson89], each block of data is a member of two independent codewords. As illustrated in Figure 2-16, the codewords are arranged orthogonally so that any two codewords have at most one common symbol. If two failures occur in a codeword, the missing data can be constructed from the orthogonal codewords.

Intuitively, accessing information in an array protected by two-dimensional parity occurs in much the same was as for a RAID level 5 disk array. Writes affect two codewords, requiring additional parity computation and disk accesses. Also, additional algorithms are necessary in order to provide operation in the face of two disk failures. In Chapter 3, I introduce a novel programming abstraction for disk array operations and in Appendix A I describe fifteen algorithms that can be used to access information stored in disk arrays protected by two-dimensional parity.

Two-dimensional parity increases the amount of storage capacity lost to redundancy and the additional disk work required to maintain the second parity disk reduces the throughput of the array. Minimal redundancy overhead occurs if the number of data columns is equal to the number of data rows. In this case, the amount of capacity lost to parity information is: $2\sqrt{N_{DataDisks}}$. If the data disks in a two-dimensional array are not organized in a square array, the fraction of capacity lost to parity information will increase.

Blaum, Brady, Bruck, and Menon introduced *EVENODD*, a parity-based redundancy scheme similar to two-dimensional parity, but with a different mapping of information that guarantees a minimal capacity overhead regardless of the array organization (the

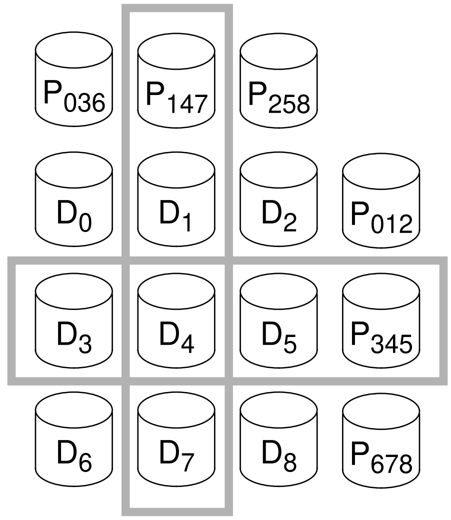


Figure 2-16 Two-dimensional parity

Two-dimensional parity protects data from any two disk faults by placing each block of data in two independent codewords. For example, D_4 *is protected by* P_{147} *and* P_{345}.

array does not need to be physically square). Similar to two-dimensional parity, each data symbol is a member of two distinct parity-based codewords. However, the codewords are distributed in a fashion that results in a constant (minimal) redundancy overhead of $2\sqrt{N_{DataDisks}}$. The name "EVENODD" is derived from the fact that while one set of codewords is always based upon even parity, the parity of the remaining codewords is allowed to dynamically change from even to odd.

By employing two check symbols, a parity and a non-binary code, a codeword can be constructed that tolerates two symbol (disk) failures. This approach, known as *RAID level 6*, is used in Storage Technology's Iceberg product line [STC94]. The non-binary code, a Reed-Solomon derivative, is typically computed using either large lookup tables or an iterative process involving linear feedback shift registers—a relatively complex operation which requires specialized hardware.

2.5.4.2 Improving Performance

Counter to Burkard and Menon's prediction, Savage and Wilkes propose *AFRAID*—A Frequently Redundant Array of Independent Disks, an array architecture that trades dependability for performance [Savage96]. Attempting to compensate for the small-write problem found in RAID level 5 arrays, parity stripes in AFRAID are allowed to become inconsistent for brief periods of time. When writing data to the array, if the parity drive is busy, they defer its update to a later point in time. They predict that a 23% reduction in the array's mean time to data loss can increase performance by as much as 97%. AFRAID changes the fault model of single-fault tolerant disk arrays because codewords which do not have up-to-date parity are susceptible to a disk failure.

Instead of sacrificing dependability to achieve higher performance, it is possible to increase performance through more traditional means, such as caching. Recognizing that disk traffic is bursty [Ousterhout85, Ruemmler92], a write-back cache can be used to defer updates until the array is idle [Golding95, Menon93a, Symbios95a]. By making the cache nonvolatile, the semantic that completed write operations are durable is preserved. Additionally, deferring write operations allows small sequential operations to be coalesced into larger, more efficient disk operations [Menon93a, Rosenblum92].

Alternatively, a variety of architectures have been proposed for trading capacity for performance. These include deferring updates in a disk log [Bhide92, Stodolsky94] and various schemes for modifying the logical to physical mapping of data that allow more efficient array operations to be utilized [Menon93b, Mogi94, Solworth91].

Stodolsky, Holland, Courtright, and Gibson proposed *parity logging*, an approach to avoiding the small write problem by using disks more efficiently. Parity logging defers updates to parity in RAID level 5 arrays by storing them in a FIFO log that is maintained partially in controller memory and partially on disk. Using the rule of thumb that full track accesses are ten times more efficient than sector accesses, parity logging collects large amounts of parity updates and then applies them en mass at track rates.

Figure 2-17 illustrates two algorithms for writing data to a fault-free array. The first algorithm replaces the small-write algorithm used in RAID level 5 disk arrays. Instead of immediately performing the read-modify-write update of parity for each write operation, an update record, reflecting the changes made to user data, is appended to the FIFO log. Similarly, a large-write operation appends a parity overwrite record to the log. When the log becomes full, it is emptied by reading its contents and that of the parity disk at track rates, applying the records, and then writing the parity, again at track rates. To preserve consistency, the log is processed in the same FIFO order that it was written.

A power failure that results in the loss of the portion of the parity log that is stored in controller memory can be recovered from by simply reconstructing parity, assuming that there are no disk failures. If the array is required to survive simultaneous disk and power failures, then the parity log must be durable.

Instead of deferring work, Menon, Roche, and Kasson propose *floating data and parity* which allows the physical location of disk blocks to be remapped [Menon93b]. Spare sectors are allocated in each disk cylinder—to reduce rotational latency, mappings of data and parity are swapped to these spare locations as needed. Unlike the architectures discussed to this point, the location of data and parity can not be statically determined. To survive power failures, the data and parity mapping tables must be stored in nonvolatile memory. Loss of mapping information results in the loss of all data in the array.

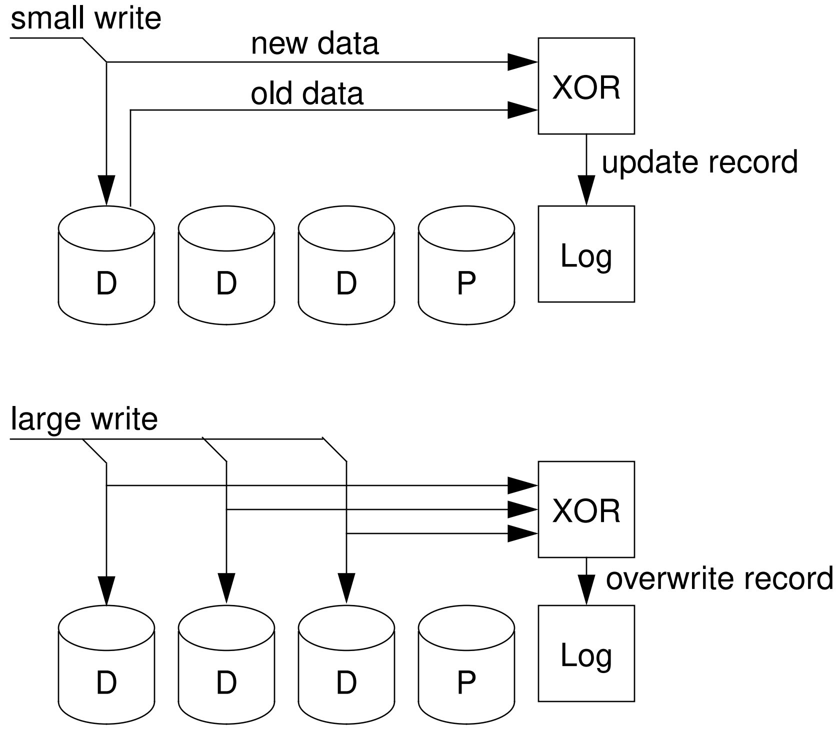


Figure 2-17 Fault-free write operations in a parity logging disk array

These two algorithms are used to write data to a parity-logging disk array. The uppermost algorithm is a variation of the small-write algorithm, previously described in Figure 2-9, and the lower algorithm is a variant of the large-write algorithm of Figure 2-8. In both cases, the fundamental difference is that parity changes are stored in an append-only log rather than being written to disk. In the case of a small write, a record containing changes which must be applied to parity is placed in the log. In the case of a large write, in which a new value of parity has been computed, an overwrite record is placed in the log.

Mogi and Masaru propose a logical remapping called *virtual striping* that is essentially a marriage of the floating data and parity architecture, and the write-back caching techniques previously discussed which coalesce small-write operations into more-efficient large-write operations [Mogi94]. To survive power failures, the data and parity mapping tables, as well as the write-back cache, must be stored in nonvolatile memory. To survive controller failures, this information must be mirrored elsewhere in the array.

Still other directions exist. Seagate has introduced hardware support for RAID operations into their SCSI and Fibre Channel disk drives by providing XOR and third-party (disk-to-disk) transfer capabilities that collectively reduce the amount of data traffic between a central array controller and the disks in some RAID level 5 operations [Seagate94]. These improvements, which effectively distribute portions of the array control mechanism to the disk drives, lead to a fundamental break with the architectures described to this point: array control mechanisms are not distributed. Inherent properties of the SCSI and Fibre Channel standards that weaken the disk array's fault model have limited the adoption of this technology. For example, a reset of the SCSI bus requires all devices, including disk drives, to erase all buffers [ANSI91]. Because bus resets are unpredictable, this may result in the destruction of state information necessary to complete an operation.

Finally, Cao, Lim, Venkataraman, and Wilkes propose another distributed controller architecture, *TickerTAIP*, designed specifically for RAID level 5 applications [Cao94]. User requests are received by *originator nodes*, centralized control mechanisms that select the appropriate algorithm to be used and then dispatch disk and parity (XOR) work to *worker nodes*. Each worker node manages a subset of the disks in the array. Similar to the Seagate model, worker nodes are relied upon to transfer information directly between themselves, bypassing the originator node. TickerTAIP did not address the aforementioned problems with SCSI.

The failure of a worker node leaves the disks it manages inaccessible. Therefore, to survive the failure of a worker node, codewords must be arranged so that each symbol is stored on a disk managed by an independent worker node. To survive the failure of an originator node, which manages the progress of the worker nodes involved in servicing each user request, its state information is mirrored on other originator nodes. By ensuring that enough information is duplicated across nodes, TickerTAIP is able to atomically survive the failure of any node or disk. Power failures are not survived atomically, and the array therefore suffers from the "write hole" problem that was described in Figure 2-15.

Because there are multiple originator nodes, TickerTAIP does not guarantee isolation (serializability) of user requests. TickerTAIP does permit users to specify an explicit ordering of requests; however, because users have no notion of the alignment of their requests to codeword boundaries, this does not avoid the problem of maintaining the consistency of parity in codewords that are simultaneously updated by independent user requests.

2.5.5 Discussion

Disk drive performance is limited by mechanical devices and its rate of performance increase does not match that of processors. This section began by introducing the concept of striping user data across an array of disks to improve performance for a variety of application workloads. Relying upon the fact that disks can be treated as erasure channels, I described two well-known methods of tolerating disk failures through the use of redundant data. The first, normally called mirroring, relied upon two copies of data. The second method, based upon parity, reduced the amount of disk capacity required for redundancy, but required more algorithms for accessing information. I then described RAID, an informal taxonomy of redundant disk arrays, which is based upon the striping and encoding of user data.

I defined a fault model for disk arrays that requires that array operations are serializable, and durable. I described various techniques for implementing RAID systems which adhere to this model and assumed that the components used to construct disk arrays were either inherently failstop, or easily made to behave as failstop devices. I concluded the section by reviewing a variety of disk array architectures that have recently been introduced to optimize for performance, dependability, and cost.

Throughout this section, I have provided examples of the algorithms used to access information stored in a disk array. An important observation is that despite the fact that the number of algorithms necessary to properly implement all of the architectures I discussed may be large, they were composed from a relatively small set of actions, such as disk read, disk write and XOR. Intuitively, it would seem that given a working disk array, altering its architecture should simply require changes to mapping and algorithm selection as well as creating new algorithms from an existing library of actions. Occasionally, as new devices (write-back cache, parity log, etc.) and encodings (Reed-Solomon) are introduced, programmers must additionally create the actions that operate upon them.

Another important point, which was only briefly examined here, is the potential difficulty of guaranteeing that disk arrays maintain the desired operating semantics. The example I used was the “write hole,” in which the failure of a write operation results in an unexpected loss of data. An interesting problem that I will later study at length is the method(s) for correctly sequencing algorithms to ensure appropriate recoverability which leads to correct operational behavior.

2.6 Conclusions

This chapter was written for three reasons. First, it was intended to educate the reader on the fundamentals of general fault-tolerant systems and, to this end, the terminology, metrics, and procedures necessary to improve the dependability of an arbitrary system were described. Second, the chapter reviewed fault-tolerant disk systems, describing a variety of disk array organizations that are able to tolerate a variety of faults, including the loss of a disk, without loss of availability. Third, and most importantly, this chapter provided insight into a variety of array architectures, describing their expected behavior and their commonality with other array architectures.

This third and final point is the springboard into the remainder of the dissertation. In Section 2.5.4, I demonstrated that by making simple changes to mapping and encoding, significant changes to the dependability, performance, and cost of a disk array are made. As these changes are introduced, new algorithms are required for accessing information in the array. Observing that these algorithms are constructed from a small set of actions that access the storage devices (disk, cache) and compute check information (parity, Reed-Solomon), it is reasonable to conclude that it is possible to construct disk arrays in a manner which exploits this commonality, allowing programmers to only worry about the new mapping, encoding, algorithm specification, and algorithm selection.

In Chapter 3, I investigate the validity of this hypothesis by examining methods for executing array operations and recovering from the errors encountered during this execution. I demonstrate that not all of the techniques described in Section 2.3 are well suited for use in redundant disk arrays, burdening the programmer with far more complexity than is necessary.

Chapter 3: Mechanizing the Execution of Array Operations

Chapter 2 demonstrated that array architectures are distinguished by mappings, encodings, and the algorithms that are used to access information stored in the array. These algorithms are composed from actions, such as disk read, which are common to many array architectures. This leads to the layered model of disk array software illustrated in Figure 3-1 and the casual observation that the well-known practice of modular design should lead to greater software reuse.

This chapter concentrates upon the problems of specifying and executing array algorithms. The chapter begins with a study of the goals that simplify the design and programming of disk array software. Section 3.2 examines the actions which are the building blocks of array algorithms and defines a consistent interface for all actions. Section 3.3 introduces a novel method of specifying array algorithms as directed graphs in which the actions are represented by the nodes of the graph and the dependencies between the actions by its arcs.

The remainder of this chapter concentrates upon the execution of these graphs. I dismiss ad hoc techniques of execution which rely upon forward error recovery as unreasonable, because they do not effectively exploit the commonality between array architectures—they require that new methods of recovery from errors must be defined and implemented for each algorithm and array architecture. Instead, I embrace methods for execution that are rooted in the design of dependable systems which employ transactions [Bernstein87, Gray93, Lynch94]. Because transactions guarantee programmers atomic behavior in the face of errors, a general execution mechanism which executes array algorithms without regard for their function or the architecture that they support can be created. Furthermore, the approach lends itself to the application of known techniques for correctness verification and deadlock detection. This execution mechanism, described in detail in Section 3.5, employs classic undo/no-redo recovery principles to guarantee atomic operation and a fine-grain (single operation) unit of recovery in a single-controller architecture. Chapter 4 describes an implementation based upon this execution mechanism. Later, in Chapter 5, this mechanism is revised to accommodate a structured method for reducing the amount of undo logging, without sacrificing these benefits. Multiple controller architectures, and the redo logging necessary to support controller failover, are not implemented.

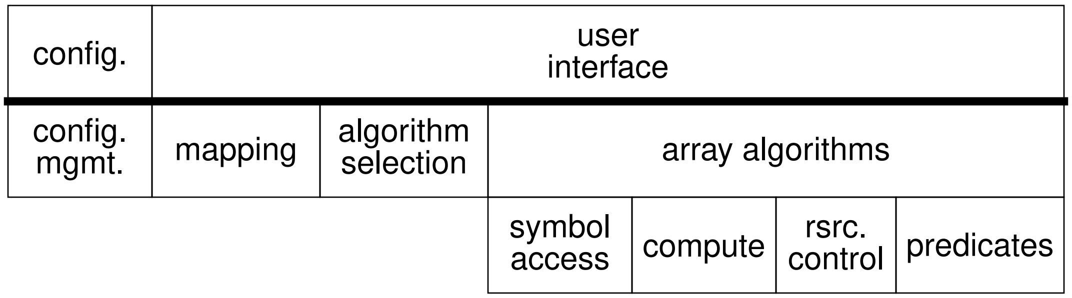


Figure 3-1 A layered software architecture

This diagram illustrates a basic partitioning of array software. The internal workings of the array (boxes below the bold line) are hidden from the user who is presented with a simple read/write abstraction, similar to the one presented by contemporary disk drives. The array software is represented by two layers: the upper layer represents the configuration management, data mapping and algorithms which distinguish array architectures. The lower layer represents the actions, common to array architectures, which are used to compose new algorithms.

This layered approach is common to software RAID systems such as those provided by NT which uses a layered driver architecture [Custer93]. Disk mirroring is implemented as a layer above the low-level disk drivers, isolating the array algorithms from the physical disk interface.

3.1 Goals of an Ideal Approach

Chapter 2 concluded with the observation that disk array architectures are differentiated by the mappings used to locate data and check information, the devices used to store this information, the algorithms used to access data, and the criteria for selecting an algorithm. Furthermore, after examining a variety of architectures optimized for either performance, reliability, or capacity, it became evident that the number of actions necessary to construct array algorithms for these architectures was quite small. Intuitively, it would seem that because almost all array architectures are implemented from a common set of actions such as accessing symbols (e.g. disk read and write), computing check information (e.g. XOR, Reed-Solomon), and manipulating resources (e.g. allocate), it would be possible to create an infrastructure for implementing array architectures which:

- limits the amount of code changes required to extend existing code to support a new array architecture

- simplifies error recovery by creating a process which guarantees that codewords are updated atomically, reducing the need for hand analysis

- does not introduce overhead which results in either significant resource consumption or performance degradation

- enables verification, early in the development cycle, of the ability of array algorithms to correctly tolerate faults

The remainder of this dissertation is devoted to the pursuit of these four ideals. Chapter 2 demonstrated the need to extend array architectures to explore distinct cost, performance, and dependability solutions. Minimizing the amount of code changes required to perform these extensions has a significant impact on production costs and the bandwidth of the development organization. Amortizing the cost of writing code for use in redundant array controllers across an entire development group (designers, coders, and testers), a typical programmer can develop 9,300 lines of code (LOC) per year at a cost of $18.50 per LOC [Potochnik96]. It is not uncommon for a fully-featured array controller which supports multiple RAID levels to have 250,000 lines of code. Rewriting just 30% (a conservative estimate) of these lines would require over eight man-years at a cost of $1,387,500.

Simply reducing the cost of extending code to support new array architectures is an incomplete goal. As described in Section 2.5.3, programmers must ensure that write operations which fail will contain their damage to the data symbols being written. When a write operation fails in the middle of execution, the programmer must therefore ensure that the integrity of the codeword is maintained. Failure to do this results in the problem commonly known as the *write hole*, in which data in the codeword, which was not a part of the write operation, is permanently lost. This problem was discussed in detail in Figure 2-15. The burden of maintaining codeword integrity can be greatly simplified if a system can be devised to ensure atomic codeword updates. This would eliminate the task of predicting and processing all incomplete codeword updates.

The cost of this system must be held in check. Performance is generally important and the resources used to construct arrays are often precious. Software arrays, which provide little more than tolerance of disk failures, are generally not permitted to increase the cost over today's 20¢ per MB cost of commodity disk drives. The cost of array subsystems that tolerate controller, fan, cabling, and power failures has fallen below $1 per MB and is expected to reach 11¢ per MB by the end of the decade for some applications [IDC95]. Nonvolatile memory devices, which are capable of surviving loss of power without corrupting data, currently cost as much as $20 for an 32KB part. Exotic solutions which employ large amounts of expensive resources such as nonvolatile memory may not be practical. Therefore, a general solution must be sensitive to both performance as well as resource consumption.

Finally, the process of verifying code as correct, regardless of the number of lines involved, is an important function. As with any type of development process, the earlier design defects are detected, the cheaper they are to repair, both in terms of cost and time. Therefore, an ideal approach to developing array software will be amenable to correctness verification during the early phases of implementation, rather than deferring all verification to the lab-testing of prototypes.

3.2 Isolating Action-Specific Recovery

The obvious and best-known method for minimizing the amount of code changes required to extend software is to create modular code which isolates functions that are known to change orthogonally [Parnas72]. In Figure 3-1, I described the boundaries between the modules which are changed to produce new disk array architectures. In this section, I focus upon the actions which are used to compose array operations. Specifically, I define a general interface which isolates recoverable errors detected during the execution of these actions from the layer responsible for executing array algorithms, hiding the internal details of the actions from the array architect. By requiring that all action-specific recovery be performed by the actions themselves, an infrastructure which allows a variety of array architectures to be implemented without regard for the manner in which actions are implemented becomes possible.

3.2.1 Creating Pass/Fail Actions

Irrespective of the type of an action, it is possible (and necessary) to define a set of rules by which all actions must abide. By knowing that actions behave in a common manner, the programmer can generalize the infrastructure used to execute array operations. The first such rule is designed to isolate action-specific recovery from the process of recovering from failed array algorithms, enabling programmers to create array algorithms from a library of actions, without regard for the internal details of the actions. This is accomplished by abstracting actions with a wrapper that is responsible for recovering from all recoverable errors encountered during the execution of an action. With such a wrapper, these actions can be viewed by the array architect as pass/fail building blocks in which "pass" implies successful completion and "fail" implies that the action can not be completed and the failed components have been removed from service [Courtright94, Courtright96a]. Actions are assumed to exhaust all known methods of recovering from errors and therefore actions that fail are not retried.

Recall from Section 2.3 that the process of tolerating faults requires six steps: detection, diagnosis, isolation, recovery, reconfiguration, and repair. For all errors, actions are required to detect, diagnose, and isolate all faults because these steps require information which is local to the device. Additionally, for all recoverable errors, actions are required to perform recovery, reconfiguration, and repair.

Because actions which fail remove devices from service, and because invariants may exist across multiple devices to provide fault tolerance, actions must maintain the independence of faults within a system. For instance, if an action is defined to update two symbols in a codeword and one of the symbols fails, the action should not arbitrarily fail the second symbol.

Actions may also (and often do) operate upon symbols in multiple codewords, but must continue to preserve the independence of faults—if a symbol in one codeword is lost, the action should not arbitrarily fail the symbols in the other codewords it is operating upon. For example, if the failure of a single sector is detected during a multi-sector disk read, the action performing the read should only mark the failed sector as “bad.”

One of the four ideals described in Section 3.1 was the elimination of coping with incomplete codeword updates by creating a process for executing array algorithms atomically. If array algorithms are to atomically modify codewords, the actions from which they are constructed must be known to execute atomically. This follows from the same notion used in atomic commit protocols in transaction systems: transactions can be made to operate atomically if the actions from which they are composed are themselves atomic [Bernstein87, Lynch94].

To summarize, there are four rules for creating actions:

1. Actions are responsible for detection, diagnosis, and isolation of all faults encountered during their execution.

2. Actions are responsible for recovery, reconfiguration, and repair of all tolerable faults detected during their execution.

3. Actions preserve the independence of faults within an array.

4. Actions are atomic.

3.2.2 Actions Commonly Used in Redundant Disk Array Algorithms

Four fundamental types of actions are necessary to implement the disk array algorithms described in Chapter 2: symbol access, resource manipulation, computation, and predicates. Table 3-1 presents examples of specific actions for each of these four types. This table is not meant to be a comprehensive list of all actions required to implement known array algorithms. Furthermore, because the study of array architectures is ongoing, it is likely that new actions will be developed in the future. For instance, new actions are necessary when new encodings (e.g. the Reed-Solomon encoding used in RAID level 6) are employed to protect data. Similarly, new actions are required if a new device type (e.g. the append-only log used in parity logging) is added to store symbols. Because the rules presented in Section 3.2.1 apply equally to all instances and types of actions, the fact that this table is incomplete is unimportant.

The remainder of this section describes the four basic types of actions. Each of these types is distinguished by data dependencies and state transformations. The data dependencies represent a dependence upon input parameters that are necessary for the action to begin execution. The state invariants represent the transformation the actions make to the system. Later, in discussions of the execution of array algorithms composed from these types of actions, this information will be used to establish constraints for the construction of array algorithms as well as to reason about the correctness of the error recovery procedures in mechanisms that execute these graphs.

Table 3-1 Actions common to most disk array algorithms

Type	Name	Function
symbol access	Rd	copy data from disk to buffer
symbol access	Wr	copy data from buffer to disk
symbol access	LogUpd	append a “parity update” record
symbol access	LogOvr	append a “parity overwrite” record
rsrc. manipulation	MemA	acquire a buffer
rsrc. manipulation	MemD	release a buffer
rsrc. manipulation	Lock	acquire a lock
rsrc. manipulation	Unlock	release a lock
computation	XOR	EVENODD decode (XOR variant)
computation	EO	EVENODD encode (XOR variant)
computation	$\overline{\text{EO}}$	EVENODD decode (XOR variant)
computation	Q	Reed-Solomon encode
computation	$\overline{\text{Q}}$	Reed-Solomon decode
predicate	Probe	if hit, return shared lock and pointer

3.2.2.1 Symbol Access

Actions are necessary to load and store symbols between memory and devices such as disk drives, append-only logs, and caches. Actions that read symbols are assumed to copy them from the device to an uninitialized buffer. Conversely, actions that write symbols copy them from an initialized buffer to a storage device. More formally, these actions can be defined in terms of the state invariants which exist prior to, during, and after their execution. For example, prior to the execution of a read action the following invariants must be true:

- The region of the storage device (e.g., an offset/length pair) to be accessed must be valid.
- A buffer must be supplied whose length is greater than or equal to the length of the extent to be read.
- The contents of the extent to be read are presumed to contain information from the previous write action to that extent.
- The contents of the buffer are assumed to be unknown.

Therefore, read actions have a dependence upon a valid address and buffer. Any algorithm that uses read actions must ensure that these dependencies have been satisfied prior to executing the read action.

During the execution of the read action, the contents of the storage device are unchanged and the contents of the buffer may be in either the original (uninitialized) state, a new (same as the extent of the storage device being read) state, or some arbitrary combination of these two. Read actions that fail will leave the buffer in one of these three states but without an indication of which one—therefore, it is only safe to assume that read actions that fail will leave the buffer in an uninitialized state. Once a read action completes, the extent being read is left unchanged and the contents of the buffer are identical to the extent being read from the storage device. This behavior is consistent with traditional disk semantics and the atomicity requirement of Section 3.2.1.

Write actions have similar dependencies and invariants. Prior to execution, write actions are dependent upon a valid device address and an initialized buffer. The amount of data in the buffer that is to be written must be of equal length to the extent to be written on the storage device. Once execution begins, the write action begins to copy information from the buffer to the storage device. The order that the information is copied is arbitrary; however, the information is copied in units of a predetermined length, such as a disk sector. If a write action fails, the buffer it is copying data from is left unchanged. Each unit of the storage device being modified is left in one of three states: unchanged, identical to the corresponding unit of the buffer, or inaccessible. Once a unit becomes inaccessible, it can not be returned to service without an explicit repair operation. Because storage devices are assumed to offer atomic operation on a predefined unit of access, write actions to these devices can be made atomic—if an action fails, it is conceivable that recovery code could be created to transition all of the previously-written units to their original states.

If the write action completes successfully, each unit of the storage device being written is identical to the corresponding unit in the buffer and the buffer is unchanged from its original value.

3.2.2.2 Resource Manipulation

Resource managers, whether they control locks, buffers, or some other resource, provide two basic actions: allocation and deallocation. Actions that perform some form of allocation must either return the requested resource or fail. Immediate return from an action is not necessary, so it is permissible to wait for a resource to become available.

As with actions that access symbols, actions that manipulate resources can be characterized by their dependencies and state invariants. Prior to execution of an action that allocates a resource, a valid resource (one that is known to exist in the system) must be known and the resource in question must be in either an “acquired” or “available” state. During execution, the allocation action waits for the resource to enter an “available” state. When the action completes, the resource is marked as “acquired” with the owner being the process that invoked the action.

Actions that perform a release begin with a valid resource that has been acquired by the process which invoked the release action. During execution, the action atomically modifies the state of the resource from “acquired” to “available.” At the conclusion of the release action, the state of the resource is marked as “available.”

3.2.2.3 Computation

Actions that perform computation require one or more buffers which contain the information to be operated upon, a buffer in which to place the result in (may be the same as one of the input buffers), and optionally, a parameter that specifies the type of computation to be performed. The initial states of the buffers are a function of the type of computation to be performed—generally, the buffers that contain the information to be operated upon are assumed to be initialized and the output buffer is assumed to be uninitialized. If the computation completes successfully, the output buffer will be initialized to its intended value and the input buffers may be in either their original or some predetermined state. For example, the computation may be designed to overwrite the result into one of the input buffers.

Similar to actions that write symbols, actions that perform computations are permitted to generate results in an arbitrary order, but are assumed to atomically perform that computation on units of information of a predetermined size. Therefore, computations that fail will leave each unit of the result buffer in either its original or final state.

3.2.2.4 Predicates

Unlike the actions previously described which modify the value of information (buffer contents, storage device contents, or resource states), predicate actions produce a result that is used to determine the flow of execution in the algorithm of which they are a part of. To do this, predicate actions begin with a set of information that is used to make a decision, the type of which is specific to each type of predicate action (e.g., two integers for use in an equivalence test). The predicate also requires a register, assumed to be uninitialized, to record the result of its decision. When the predicate completes execution, the inputs are left unchanged and the result register is in one of two or more predetermined states. Predicate actions are required to execute atomically.

3.3 Representing Array Operations as Flow Graphs

Creating storage operations from a library of functions is a technique which has been in use 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 of its introduction, 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. Using a linear sequence of commands, channel programs isolated these details from users by providing an abstract interface which was closer to that found in today's SCSI drives [ANSI91].

Similar methods for abstracting the details of disk array 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 the management of simultaneous actions occurring across the array, TickerTAIP uses a centralized table in which each entry contains a list of actions for a worker to execute. Once an array operation is initiated, each worker is responsible for sequencing its own activities.

3.3.1 Flow Graphs

Instead of using a table to represent an array operation, I propose the use of *flow graphs* which provide a system with enough information to correctly sequence instructions without requiring an understanding of their collective effect [Aho88, Courtright94]. As Figure 3-2 illustrates, flow graphs are traditionally used to illustrate program control flow between *basic blocks*, sequences of statements which are: single entry, single exit,

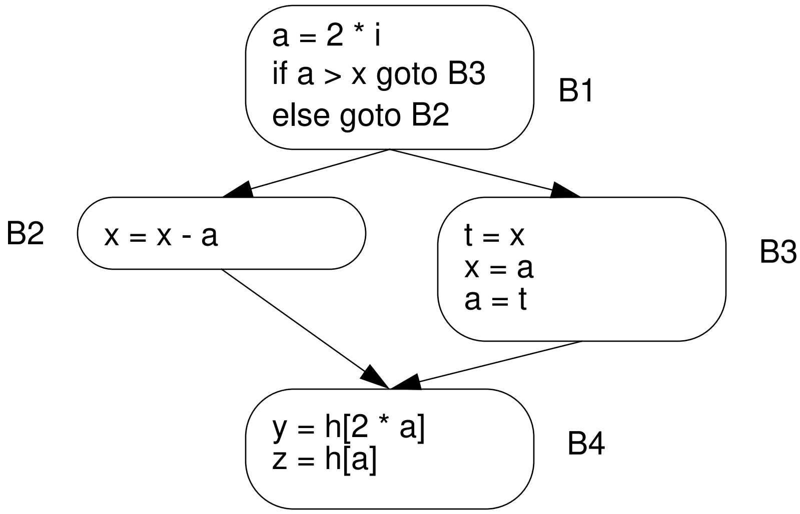


Figure 3-2 Flow graphs model program control flow

*In this illustration, arcs connect the basic blocks (*B1-B4*) of a program which follow each other in some execution sequence. In this example, when* B1 *completes, either* B2 *or* B3 *will be executed.*

---

and unconditional in their execution. The nodes of a flow graph are the basic blocks of a program and the arcs represent the flow of control through the program. In this example, control flows conditionally from block B1 to block B2. Block B4 is executed when either B2 or B3 completes.

When using flow graphs to model RAID operations, the actions described in Table 3-1 are represented as distinct nodes of a graph. Figure 3-3 illustrates a small-write algorithm represented as a flow graph. Because each action is represented by a single node, the properties of a node (e.g. atomic failure) are inherited from the defining properties of the actions.

Notice that the nodes in the graph of Figure 3-3 do not convey the context (e.g. "read old parity") of each action. This is because the context is known only by the designer of the graph. Section 3.5.4 capitalizes upon this independence of context to permit the construction of a general execution mechanism which is independent of array architecture.

The execution of actions within an array operation is constrained by the presence of dependencies (control and data) which are represented by the directed arcs that connect the nodes of the flow graph. An arc is drawn from a parent node to a child node if execution of 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.

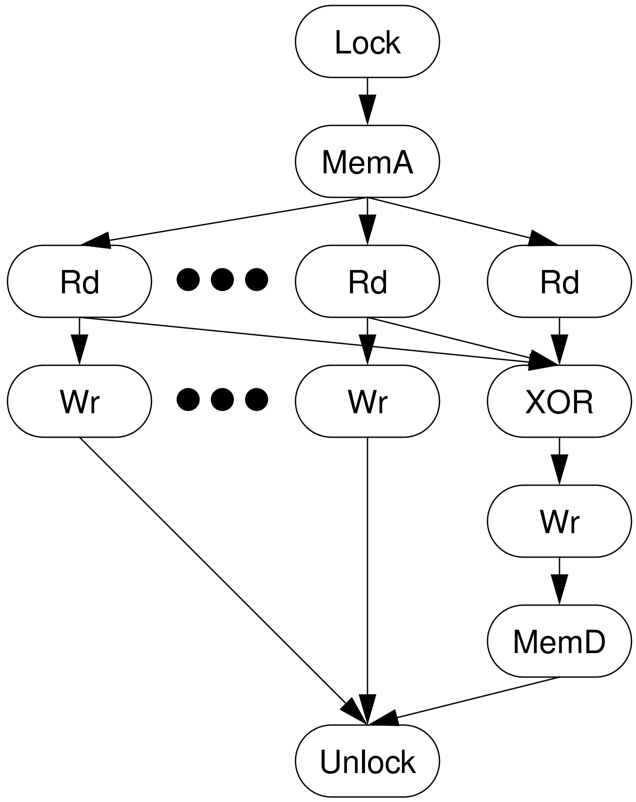


Figure 3-3 RAID level 4/5 small-write graph

This illustration presents the small-write operation, first described in Figure 2-9 and now represented as a flow graph. 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 over-writing 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. The* Lock *and* Unlock *nodes ensure that the operation runs in isolation.*

A *control dependence* exists between two actions when enabling the execution of the child is conditional upon the completion of the parent. For example, the arc between the Lock and MemA nodes in Figure 3-3 represents a control dependence from the locking hierarchy which requires that parity locks be acquired prior to the acquisition of buffers. A *data dependence* exists between actions which share data in some way. A *true data dependence*, also called a *read-after-write* (RAW) dependence, exists when an action produces (writes) a value consumed (read) by another. The arcs from the Rd nodes to the XOR node are all true data dependencies. Conversely, an *anti data dependence*, also called a *write-after-read* (WAR) dependence, exists when an action overwrites a value previously used (read) by an independent action. The arcs from the Rd to the Wr actions are examples of anti dependencies. Finally, *output dependencies*, also called *write-after-write* (WAW) dependencies, occur between actions that overwrite the same object. The presence of the output dependence guarantees a predictable ordering of the overwrite actions.

Unlike table-based representations, the visual information supplied in this representation provides an immediate understanding of the internal sequencing of actions which compose an operation. Appendix A presents the flow graphs for algorithms required to support RAID levels 0, 1, 3, 4, 5, and 6 as well as parity declustering, chained declustering, interleaved declustering, two-dimensional parity, and EVENODD architectures. Included in this discussion is a description of each graph's structure as well as when the graph should be used.

3.3.2 Predicate Nodes

Normally, all nodes in a graph are executed as soon as their parents complete. However, it is possible to create a graph in which some nodes are never executed. This is accomplished through the use of predicate nodes. A *predicate node* has two or more children and, after completion, selectively enables one or more of the children for execution.

The only additional structural constraint required to insert a predicate node into a graph is that any node which is a child of a predicate node may have no other parents than the predicate node. Also, the arcs which connect the predicate node to its children are labeled to indicate which branch will be taken given the result of the predicate.

3.3.3 Simplifying Constraints

To simplify execution, I require that the graphs be acyclic. I believe this is a reasonable requirement because I am aware of no array algorithms which require loops. Eliminating cycles does not eliminate predicate nodes and conditional execution. In the event that the array controller receives a request which is too large to process as a single operation, the request can be decomposed and implemented as a collection of smaller operations.

Additionally, I require that all graphs are *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, I require the presence of 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 to create one. Adding an extra NOP (no operation) node to create a single source or sink has no effect upon the array algorithm represented by the graph.

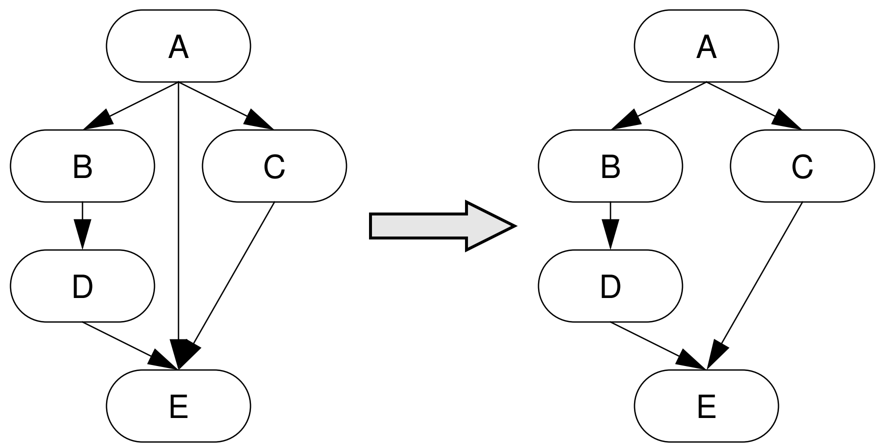


Figure 3-4 Eliminating redundant arcs

Arcs represent the presence of dependencies (control or data) and are used to constrain execution. If the arcs connect control-equivalent sequences of nodes, the arcs are treated equally, implying that redundant arcs can be eliminated. In this example, arc A-E *is eliminated because of chains* A-B-D-E *and* A-C-E*. However, if node A was a predicate that enabled either* A-B-D-E*,* A-C-E*, or* A-E*, then this optimization would not be possible.*

3.3.4 Graph Optimization

With the exception of arcs which connect a predicate node to its children, the type of dependence represented by each arc is unimportant to the execution of the graph. Therefore, arcs which have the same source and sink nodes are considered to be redundant and can be represented as a single arc. Furthermore, as Figure 3-4 illustrates, any control-equivalent arc whose source and sink nodes are identical to those of a sequence of connected arcs may also be removed.

Finally, function-preserving transformations used by compilers to eliminate *dead code* and *common subexpressions* can be employed to optimize a flow graph [Aho88]. Common subexpressions, such as redundant store actions, may appear if multiple graphs have been merged to form a single graph. Eliminating common subexpressions can produce dead code, a chain of nodes which is not connected to the sink node of the graph. Figure 3-5 demonstrates the removal of a common subexpression and resulting dead code. Dead code is eliminated by starting at the last node in the branch that has no children and walking toward the source node, removing all nodes which have no true data dependencies to their children.

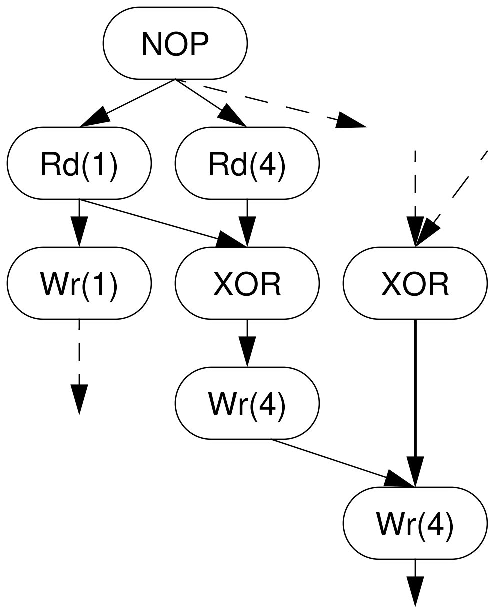


Figure 3-5(a): Original Flow Graph

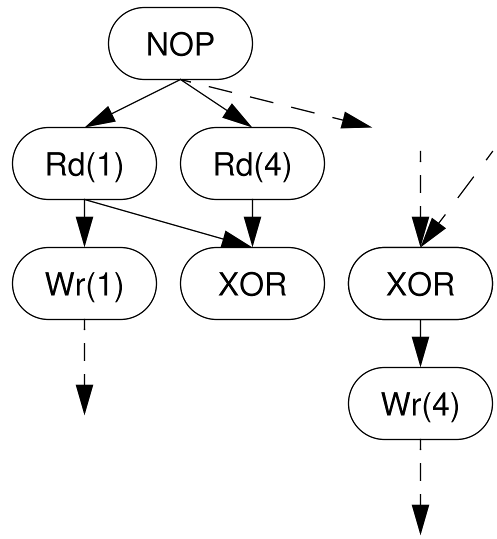


Figure 3-5(b): Common Subexpression Removed

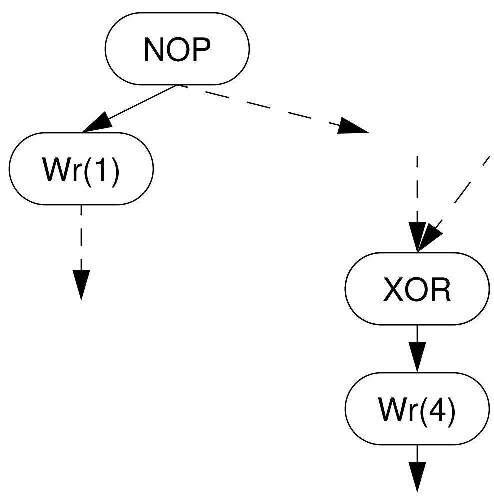


Figure 3-5(c): Dead Code Eliminated

Figure 3-5 Function-preserving transformations

This example demonstrates the removal of a common subexpression (redundant stores) and the subsequent elimination of the resulting dead code. Figure 3-5(a) shows a code fragment which contains Wr(4). *For reasons of space and simplicity, the dashed arcs represent dependencies to nodes not represented in this illustration. In Figure 3-5(b), the duplicate* Wr(4) *action has been eliminated, creating dead code in the path* NOP-RD(4)-XOR *which has no children. In Figure 3-5(c), the dead code has been eliminated.*

3.3.5 Automating Correctness Verification

Flow graphs naturally model array operations in a form which is amenable to automated correctness verification. For example, Vaziri, Lynch, and Wing have used *model checking* to validate the correctness of some of the graphs presented in Appendix A [Vaziri96]. This technique models an algorithm as a finite state machine and then exercises all possible orderings of the machine, verifying that the program invariants are satisfied at each step [Clarke82, Clarke94]. Correctness verification of flow graphs therefore requires the definition of the system invariants (e.g. even-parity codewords), the state changes imposed by each action in an operation, and the rules which govern graph execution.

During the course of their study, error in the design of a RAID level 6 small-write algorithm (taken from [Gibson95]) was uncovered. This error (a write hole) would have led to data corruption in specific execution sequences.

3.3.6 Discussion

This section described the rules for modeling array algorithms as flow graphs given a set of actions and their interdependencies. Included in this description were the mandatory structural constraints as well as some suggestions for simplifying the structure of graphs and reducing the amount of work within a graph. Additionally, the design of flow graphs can be exhaustively tested during the design phase, which should lead to reduced development costs.

Given a small library of actions which access symbols, manipulate resources, compute check information, and implement basic predicates, flow graphs can be used to construct arbitrary array algorithms. As a demonstration, Appendix A presented twenty two distinct flow graphs which are used in twelve disk array architectures.

In the remainder of this chapter, I describe the process of executing graphs. This process includes recovery from node (action) failures. In Section 3.4, I dismiss execution based upon forward error recovery because the constraints necessary to design graphs that are known to be recoverable are not obvious and a significant amount of architecture-specific error-recovery code is required. In Section 3.5, I describe a mechanism for executing graphs which guarantees that each graph is executed atomically and imposes no structural constraints upon the graphs. This guarantee is made by requiring that the effects of every action may be undone at any point prior to the completion of the algorithm. Later, in Chapter 5, I relax this requirement by inserting a barrier into each graph and only require that actions executed prior to the barrier be undoable.

allocate memory
read old parity
~~read old data~~
XOR: new data, old data, parity
write new data
write new parity
deallocate resources

error induces algorithm change →

mark disk as "failed"
discard old parity
allocate additional memory
read remaining data symbols
XOR: new data, remaining data
write new parity
deallocate resources

Figure 3-6 Forward error recovery

In this example, the small-write algorithm is scheduled to perform a write operation. The operation begins by allocating a buffer and scheduling reads of old data and old parity. When the read of old data fails, the read-modify-write of parity can not be performed because it relies upon the old value of data. Therefore, the array controller allocates additional memory, reads the remaining data symbols, and computes new parity as the XOR of the data symbols.

3.4 Execution Based Upon Forward Error Recovery is Unreasonable

Based upon conversations with representatives of six vendors who in 1995 were collectively responsible for over one-half of the world-wide disk array revenue [Disk96a], it is my suspicion that the majority of the code used to implement redundant disk array control mechanisms is currently based upon some form of forward error recovery. These same sources inform me that 50-60% of the software in disk array control mechanisms is devoted to error recovery. In a recent article in *Computer*, Friedman reports this fraction to be as high as 90% [Friedman96]. The unexpectedly-large size of this fraction is a direct consequence of the fact that any system which employs forward error recovery is required to provide a unique recovery procedure for each distinct error scenario.

In this section, I demonstrate that the burden of this approach upon the programmer, who is required to have an intimate understanding of an array architecture in order to guarantee correct operation, is significant. For instance, array operations may need to be constrained in non-obvious ways to avoid the *write hole*, a failure which results in the unexpected loss of user data. This leads to the secondary problem of validating array software as correct—exhaustive testing is necessary to ensure that each error scenario has been identified and a correct recovery procedure has been implemented.

When an error is encountered during the execution of an operation in a disk array based upon forward error recovery, the array control mechanism will attempt to complete the operation by altering the algorithm currently being executed. As an example, Figure 3-6 illustrates a write operation which is initially implemented using the small-write algorithm. When the operation begins, all disks in the array are presumed to be without fault. As the operation attempts to read the old values of data and parity, the disk containing old data is discovered to be bad, resulting in the failure of the read of old data and the subsequent inability to compute new parity. Parity is computed by altering the algorithm: the remaining data symbols in the codeword are read and new parity is computed as the XOR of all data symbols in the codeword.

Clearly, the corrective procedure necessary to recover from an error is a function of the type of error and the context in which it occurred. For instance, in the example of Figure 3-6, if the read of old parity (instead of old data) had failed, the algorithm would be changed to simply write the new data to disk and ignore parity.

3.4.1 Correct Design is Not Obvious

Providing recovery from all error scenarios is not as trivial as the example of Figure 3-6. In some instances, it is possible to construct algorithms which generate valid codewords but leave the array in a state from which it can not recover if execution is interrupted at certain points in time. That is, array algorithms may need to be constrained in non-obvious ways to avoid the write hole.

As an example, consider an operation using the reconstruct-write algorithm. In this example, illustrated in Figure 3-7, user data D_0, D_1, and D_2 are to be written to an array in which the disks are initially presumed to be free from faults. New parity is computed by reading D_3 and XOR'ing its contents with the new values of D_0, D_1, and D_2. This algorithm clearly must complete the read of D_3 and compute new parity before a new value of P can be written to disk. What is not obvious is that the read of D_3 must be completed before the writes of D_0, D_1, and D_2 may commence.

3.4.2 Exhaustive Testing is Required

Figure 3-7 provides a classic example of the subtle way in which operations which appear to be implemented correctly can lead to data corruption. In fact, in our early implementations of array software, we initially overlooked this very example. In addition to its being non-obvious, it was difficult to detect during testing, even when the test was designed specifically to exercise this algorithm. The unconstrained algorithm behaved correctly in fault-free as well as many degraded tests. Only after repeated testing was the timing just right so that the scenario described in Figure 3-7 was reached. Despite the fact that the error was rare, it resulted in the unexpected loss of data.

Verifying code constructed in this fashion requires exhaustive testing—knowing that the actions which compose an array algorithm are implemented correctly is not enough to

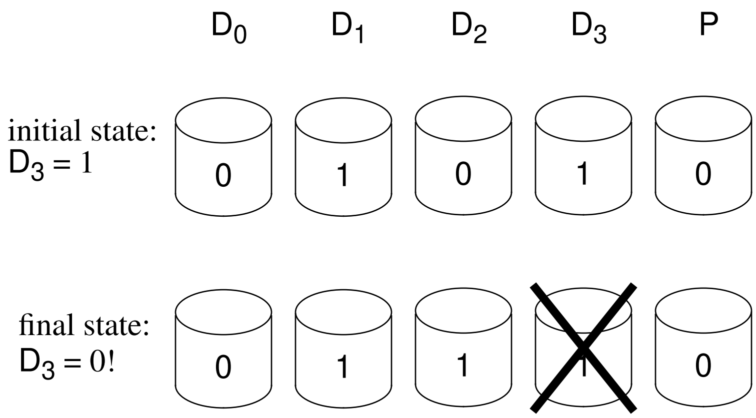


Figure 3-7 Constraining execution to ensure forward recovery

In this example, new data (0, 1, 1) is to be written to disks containing the symbols D_0, D_1, and D_2. The symbol D_3, whose value is currently "1," should not be affected by this operation. Because all of the disks in the array are presumed to be without fault and over one-half of the symbols in the codeword are to be modified, the reconstruct-write algorithm of Figure 2-10 is selected to carry-out the write operation. In this example, D_0, D_1, and D_2 have been written to disk and the read of D_3 has failed due to a catastrophic disk failure, meaning that P can not be updated. The value of D_3 has been corrupted because the current value of P does not reflect the changes in D_0, D_1, and D_2. Because the old values of D_0, D_1, and D_2 are not known, the codeword can not be restored to its original state and the corruption is therefore permanent.

ensure that the algorithm itself is implemented correctly. Relying upon finding bugs late in the development cycle, rather than at design time, is expensive. Boehm, in his book *Software Engineering Economics*, estimates the cost of finding a bug during the test phases of a project to be four to ten times the cost of finding it during the design phase [Boehm81].

3.4.3 Recovery Code is Architecture-Specific

In the beginning of this section, I presented estimates which indicated that the majority of the array control software is devoted to error recovery procedures. This is likely due to the fact that implementations based upon forward error recovery require a case-by-case treatment of each error. Unfortunately, extending an existing implementation by adding new array algorithms will require the addition of new error recovery code because existing error recovery code can not be reused.

3.5 Simplifying Execution Through Mechanization

Once action-specific error recovery is removed from the structure of the graph, it is possible to define and implement a general execution mechanism which automates recovery from errors due to failed actions [Courtright94, Courtright96b]. This mechanism, together with a library of actions, will allow rapid implementation of array operations.

This section introduces such a mechanism which employs an undo/no-redo recovery scheme, similar to the approach used in the System R recovery manager [Gray81]. In this approach, if an action fails at any time during the execution of a graph, the execution mechanism will automatically undo the effects of all previously completed actions. The information necessary to perform the undo is stored in a log as the graph is executed and deleted from the log when the graph completes execution.

To guarantee correct operation, the approach described in this section assumes that all actions are both atomic and undoable. These requirements are later relaxed in Chapter 5, allowing elimination of much of the performance and storage overhead required to achieve undoable actions. Finally, to tolerate crashes or other faults which interrupt execution, the undo functions must be idempotent. Crash recovery is described in Section 3.6.2.

3.5.1 Undoing Completed Actions

Removing the effects of failed graphs requires the ability to undo previously completed actions. This is accomplished using the undo log, previously described in Section 2.3.3. By guaranteeing that enough information exists to undo all completed actions in an graph, the system is capable of providing recovery without knowledge of the context in which the actions were used.

The information necessary to perform this recovery is recorded in the undo log as either *logical* (e.g. "computed new parity") or *physical* (e.g. "wrote the following information to disk: $011001..._2$") values. Generally, logical logging consumes less space and may even require less work than creating and recording physical log entries.

The actions described in this section, the actions used to undo them, and the data stored in the undo log, are summarized in Table 3-2. The remaining subsections briefly explain how common array actions can be undone.

3.5.1.1 Symbol Access

Assuming non-destructive read actions, undoing a read action is trivial: do nothing. This simple procedure is effective because the buffer that symbols are copied into by read actions is presumed to be uninitialized and there is no initial value to be restored.

Undoing a disk write is equally straightforward, but more expensive in terms of performance. A disk write is undone by restoring the sectors which were overwritten with their original data. Therefore, a copy of this data must be placed in the undo log. Unfortunately, to generate the copy, the original data must first be read from disk. This minimally requires an additional full disk rotation (8.4 ms on a 7200 rpm drive). Because disk utilization is a precious resource in the array, it is important to use the disks in an efficient manner. Therefore, it is desirable to eliminate the requirement that disk writes are undoable, something I examine in Chapter 5.

Parity logging algorithms require two actions: *append parity update* and *append parity overwrite*. Because the log is defined to be append only, undoing an append requires appending a record which undoes the previously appended record at a later time when the log is processed in FIFO order.

Undoing the parity update can be undone by simply appending another copy of the same record. This second record will cancel the first record in the same manner described in EQ 3-2 (transition logging). Undoing a parity overwrite is a bit trickier. The presence of a parity overwrite record in the parity log causes all previous parity information, whether it be stored on disk or in the parity log, to be ignored. Undoing a parity overwrite record could be performed by appending an additional parity overwrite record which contains the

Table 3-2 Methods for undoing actions

Action	Undone By	Log Data
Rd	Copy	none
Wr	Wr	previous disk contents
MemA	MemD	buffer pointer and size
MemD	MemA	buffer size
Lock	Unlock	lock name
Unlock	Lock	lock name
XOR	XOR	none
Q	$\overline{\text{Q}}$	none
$\overline{\text{Q}}$	Q	none
LogUpd	LogUpd	buffer contents or buffer pointer, parity address
LogOvr	LogInv	parity address

original value of parity. Because this previous value of parity is not known without scanning the entire parity log, creating an undo record is prohibitively expensive.

Alternatively, a third append action, *append parity invalidate*, could be defined to invalidate the previous overwrite record in the log. To guarantee that the correct record is undone, an operation must lock the parity log, allowing no other operations to append records, until it completes.

3.5.1.2 Computation

Similar to actions which read symbols from a device, the undo of actions which can compute new symbols is trivial if the result of the computation is written to an uninitialized buffer. If, however, the result overwrites one of the parameters provided to the action performing the computation, the original contents of that buffer must be restored. This can be accomplished by logging either physical or logical information, depending upon the type of action to be undone. For example, the XOR function is self-inverting, meaning that XOR actions undo themselves. If an action XOR'd the contents of three buffers and stored the result by overwriting the contents of one of the three:

$$B_{2new} = B_{2original} \oplus B_1 \oplus B_0 \qquad \textbf{(EQ 3-1)}$$

the action could be undone either by restoring the original physical data (copy original data into B_2) or by recomputing the original data:

$$B_{2original} = B_{2new} \oplus B_1 \oplus B_0 \qquad \textbf{(EQ 3-2)}$$

This type of logging, using a self-inverting action to undo itself, is referred to by Gray as *transition logging* and can complicate crash recovery procedures which assume that actions are idempotent [Gray93].

3.5.1.3 Resource Manipulation

Allocation of resources is easily undone by simply deallocating them. Because resource allocation actions are generally followed by deallocation actions in the same graph, the information required to perform the deallocation is available to be entered into the undo log without additional work.

O_1: $R_{L1} \rightarrow R_{L2} \rightarrow R_{L3} \rightarrow R_{L4} \rightarrow$ failure

Recovery Manager: $A_{L4} \rightarrow A_{L3} \rightarrow A_{L2} \rightarrow A_{L1}$

O_2: $A_{L1} \rightarrow A_{L2} \rightarrow A_{L3} \rightarrow A_{L4}$

time

Figure 3-8 Deadlock resulting from out-of-order allocation during recovery

Operation O_1 *releases the locks* L_1, L_2, L_3, *and* L_4 *and later fails, prior to completion. This failure causes the recovery manger to work backward through the undo log, undoing the effects of each action in LIFO fashion. In this example, the "release" actions are undone using "allocate" actions. At the same time, an independent operation,* O_2, *attempts to allocate locks* L_1 *through* L_4. *Because the recovery manager has acquired the locks out-of-order, deadlock results.*

Similarly, undoing a deallocation requires the (re)allocation of the resource. However, when using allocation to undo resource deallocation actions, great care must be taken to ensure that the locking hierarchy is not violated. Out-of-order resource allocation can result in deadlock, as illustrated in the example of Figure 3-8.

Therefore, a general recovery scheme which undoes the effects of a sequence of actions must guarantee that the allocation of resources strictly obeys the locking hierarchy. One very simple method of avoiding deadlock during rollback is to employ the *DAG locking protocol*. This protocol requires the sequential release of resources in the reverse order in which they were acquired, ensuring the preservation of the locking hierarchy during rollback [Gray93].

Another very simple solution is the *two-phase locking* protocol that requires that once a graph releases a lock, it cannot acquire any other locks. Applying this to our immediate problem, in which the undo process can begin at any point in the graph, requires that no resources are released until the operation is complete.

3.5.1.4 Predicates

The by product of a predicate action is information which determines the flow of execution. Predicates do not directly affect symbol values or buffers and therefore leave no visible state to be undone.

3.5.2 Node States and Transitions

Each node in a graph has three fields, summarized in Table 3-3: *do action*, *undo action* and *state*. Similar to the DO and UNDO programs described in Figure 2-2 on page 17, 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 3-3 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 3-4. The allowable transitions between these states are illustrated in Figure 3-9. When a graph is initially submitted for execution, all nodes are in the wait state, meaning that all of the actions represented by the graph are in their pre-execution states as defined in Section 3.2.2. A node enters the skip state if all of its parents are in the skip state or if its parent is a predicate node that has determined that the branch which contains the node will not be executed. Once entered, a node will never leave the skip state. The actions corresponding to nodes in the skip state are in their pre-execution states.

Nodes in the fired state represent actions that have begun, but have not completed, execution of their corresponding actions. A node enters the fired state 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 node fails, the graph must fail atomically. This requires that the effects of previously-completed nodes be undone. When a node is to be undone, it first enters the recovery state which indicates that the node's *undo action* may begin execution. Only nodes that have successfully completed execution enter the recovery state. Once the *undo action* completes, the node enters the undone state, which signifies that the state changes made by the execution of the (*do*) action associated with that node have been undone. This process of failing a graph atomically is described in further detail in Section 3.5.4.

3.5.3 Sequencing a Graph

Execution of a graph 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 dissertation. Assuming that the graph does not contain any predicate nodes and that all nodes complete successfully, this process contin-

Table 3-4 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
recovery	execution of undo action in progress
undone	previously executed node has since been undone

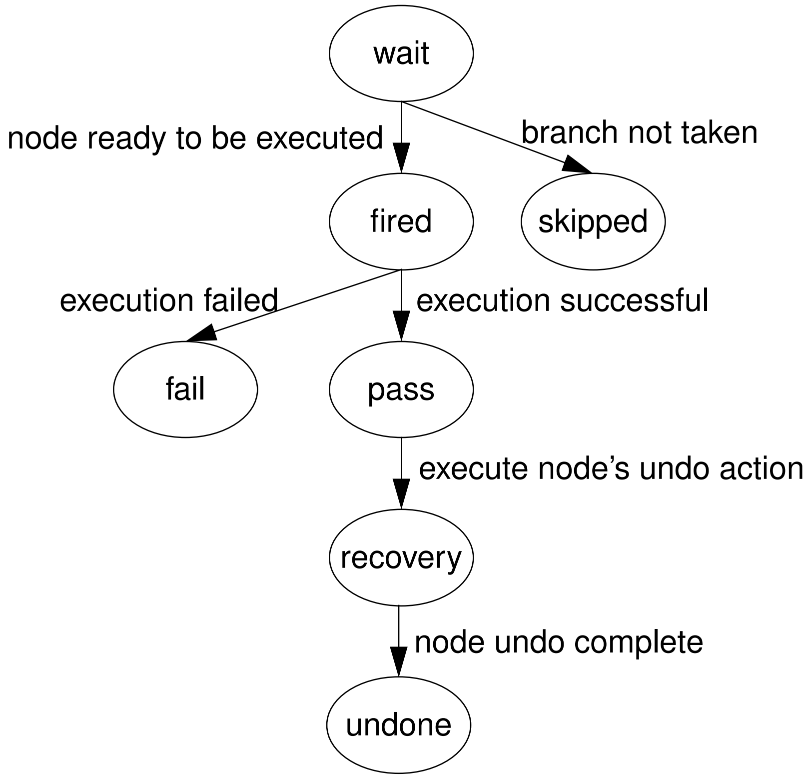


Figure 3-9 Node state transitions

All nodes in a graph begin in the wait *state. When a graph successfully completes execution, all nodes are in either the* pass *or* skip *states. The* recovery *and* undone *states, described later in Section 3.5.4, are reached only if the operation fails.*

---

ues until the sink node enters the pass state. At this point, all nodes are in the pass state, the execution of the graph is complete, and the operation is declared to be successful. Additionally, any information recorded in the undo log for this graph may be deleted.

3.5.3.1 Sequencing Graphs with Predicate Nodes

If a graph includes a predicate node, some nodes may not be executed. Again, forward execution begins with the source node and concludes with the sink node. Assuming again that no errors are detected, the graph completes and undo information may be discarded. Nodes which were skipped will be in the skip state; all remaining nodes will be in the pass state.

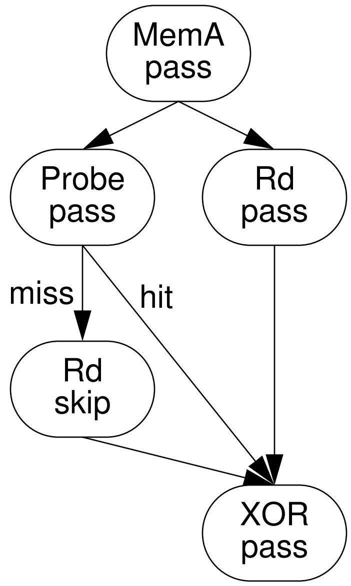


Figure 3-10 Sequencing a graph which contains a predicate

The Probe *node is a predicate, conditionally enabling execution of the* Rd *node. In this example, the* Probe *node returned a "hit" and the* Rd *node was not executed. The illustration presents the status of each node after the graph has completed forward execution.*

The sequencing of a graph which contains a predicate node is illustrated in Figure 3-10. In this example, a predicate node (Probe) is used to determine if a data block resides in a read cache. The scenario illustrated in Figure 3-10 assumes that the block was found in the cache; therefore, the Rd node, which loads the symbol from disk, is not executed. Unconditional execution resumes when the XOR node is reached. Because all of the XOR node's parents have completed (are in the either the skip or pass state) and at least one of its parents is in the skip state, the XOR node is executed rather than skipped.

3.5.4 Automating Error Recovery

When a node fails, it enters the fail state and forward execution of the graph is suspended, meaning that no more nodes are allowed to leave the wait state. Before error recovery may commence, all nodes that are in the fired state are allowed to complete execution. At this point, all nodes in the graph are in either the wait, pass, done, or fail states. Nodes in the fail state leave the modules that they modify in either an inaccessible or an unchanged state. Because of this, error recovery is limited to undoing the effects of the previously-completed nodes (i.e., those nodes that are in the pass state).

This process is easily performed by simply working backward through the graph, executing the *undo actions*. The graph is now executed from the point of failure back to the source node, in a process hereafter referred to as *backward execution*. This process is equivalent to undoing the effects of an aborted transaction by working backward through an undo log.

The rules for sequencing this process are straightforward. All nodes which are in the pass state will need to be undone. Any node in the pass state whose children are all in either the undone, fail, or skip states may enter the recovery state. When a node enters the recovery state, its *undo function* is executed and, when complete, the node enters the undone state. The process is complete when the source node has been undone; at this point, each node is in either the wait, skip, undone, or fail state and the effects of the graph have been completely removed. The array controller is now free to submit a new graph for execution.

As an example, assume that a user has requested that a single block be written to a RAID level 5 array. Because the array is in the fault-free state and the request is small, the array controller selected a small-write graph (similar to the one in Figure 3-3) and submitted it for execution. During execution, one of the Rd actions fails as illustrated in Figure 3-11. The execution engine, detecting that the node entered the fail state, suspends forward execution of the graph and begins backward execution. When backward execution completes, the effects of the graph will be completely undone. Because a fault is now present in the array, the array controller will retry the user's request, selecting a degraded-write graph which does not depend upon the failed disk.

3.5.4.1 Coping With Deadlock

Because error recovery is performed on a per-operation basis, operations can be individually aborted. This means that if deadlocked operations (e.g. because of contention for shared resources) can be detected, the deadlock can be eliminated by aborting one or more operations and then retrying them later. A common technique for detecting deadlock in a system is to use a *waits-for graph* which models the resource ownership and requests in a system in which concurrent processes compete for shared resources [Bernstein87]. As illustrated in Figure 3-12, the waits-for graph is a directed graph in which the nodes represent operations waiting on shared resources. Arcs are drawn for each request from the operation waiting on the resource to the operation which holds the resource. The presence of a cycle in a waits-for graph indicates that a deadlock condition exists.

In this example, three operations are competing for exclusive ownership of locks L_1, L_2, and L_3. Because the operations were allowed to allocate the locks in an arbitrary order, operations O_1 and O_2 have deadlocked. This situation could have easily been avoided if a *locking hierarchy* had been established which constrained the order in which locks were acquired. For instance, if the locking hierarchy required that operations acquire locks in

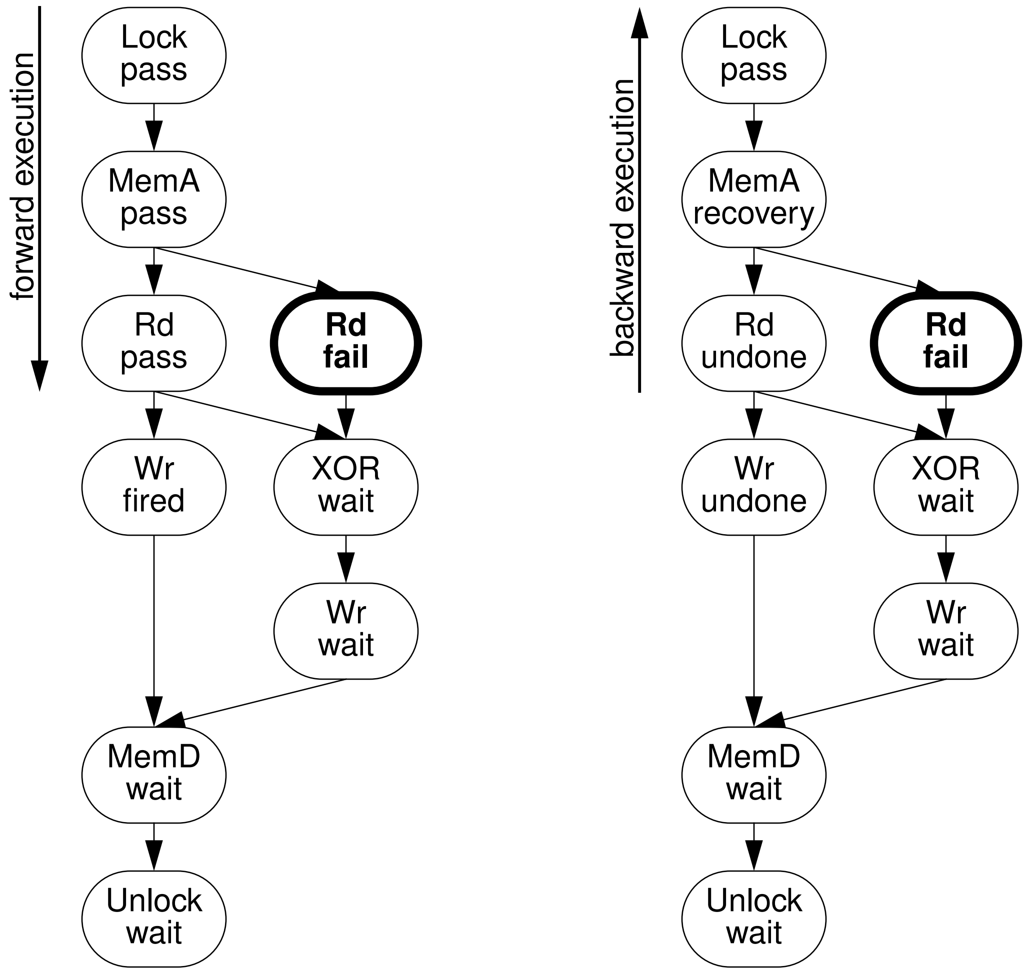


Figure 3-11 Error recovery from backward execution

The failure of the Rd *node, indicated in bold, causes forward execution to halt. Once the* Wr *node which was in the* fired *state completes, backward execution begins, undoing the previously completed actions by executing the corresponding undo functions from Table 3-2. In the illustration on the right, the* MemA *node is in the* recovery *state which implies that its undo function is currently being executed.*

numerical order (e.g. L_1, L_2, L_3), then deadlock between O_1 and O_2 would not be possible (both operations would compete for L_1 and the winner would be free to acquire L_2).

Once this deadlock condition has been detected, it can be resolved by aborting either O_1 or O_3. Aborting either one of these operations will cause its resources to be released, breaking the cycle in the waits-for graph. The other operation will then complete and the aborted operation can be retried.

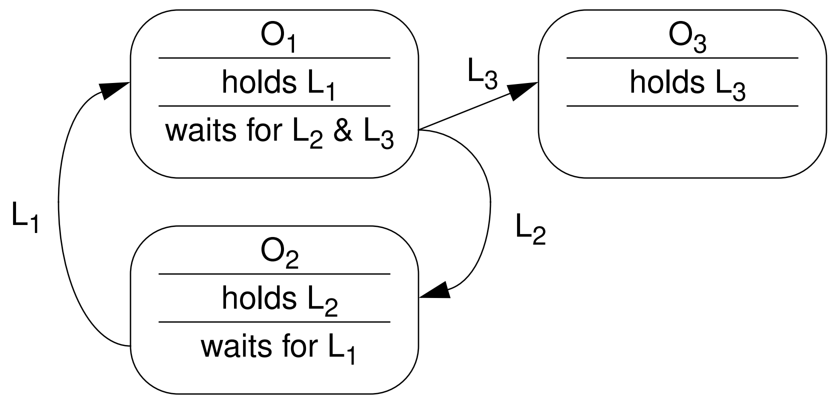


Figure 3-12 Detecting deadlock with a waits-for graph

In this waits-for graph, each node represents an operation and each arc indicates that an operation is waiting on a resource held by another operation. In this example, operations O_1 *and* O_2 *are deadlock because each is blocked, waiting on the other. This condition is indicated by the presence of the* O_1-O_2-O_1 *cycle.*

3.5.5 Distributing Graph Execution

It is possible to distribute the execution of a graph across multiple execution units. A graph may be pruned arbitrarily with a single node being the minimum unit which may be assigned to a processor. However, when the graph is pruned, the number of arcs which are cut will directly determine the amount of communication overhead required among the processors. First, as Figure 3-13 illustrates, a message will be required for each dependence between graph segments executing on distinct processors. Second, if each processor maintains a local undo log, each processor will need to know when a graph completes so that its undo information may be discarded.

Additional communication is required when an error occurs, requiring all processors to cease forward execution and commence backward execution. Because each subgraph can be recovered individually, no synchronization between processors is required to acquiesce the entire graph. Instead, each subgraph is individually acquiesced before it can begin backward execution.

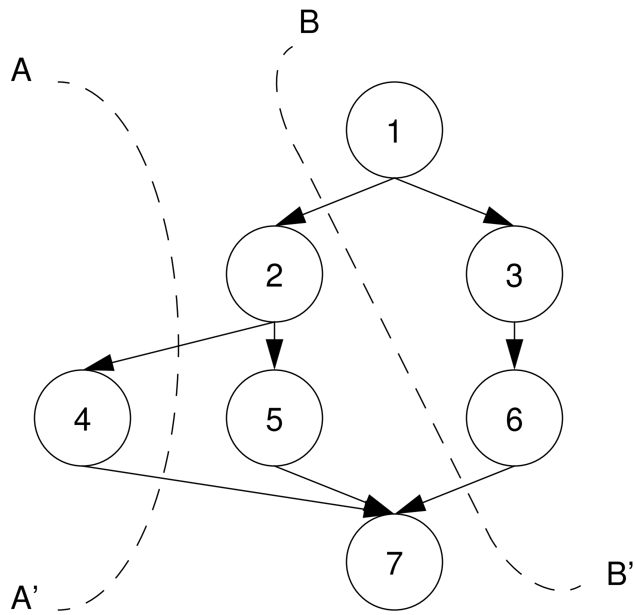


Figure 3-13 Pruning a graph for distributed execution

This graph has been pruned with cuts A-A’ *and* B-B’ *to allow execution on three separate processors. The processor responsible for executing node* 4 *will need to communicate three times with the processor executing nodes* 2, 5*, and* 7*: once to wait on a message that node* 2 *has completed execution, once to send a message that node* 4 *has completed execution, and once to wait on a message that node 7 has completed, indicating that the graph has completed and the local undo log may be discarded. Similar communication is required between the processor executing nodes* 1, 3*, and* 6 *and the processor executing nodes* 2, 5*, and* 7.

3.6 Fault Model

In the previous chapter, I discussed the physical fault models that characterize the devices commonly used to implement redundant disk arrays. These devices are assumed to offer failfast behavior. In this chapter, I described a programming abstraction for representing the algorithms that perform array operations. These algorithms are composed from device actions that are abstracted with a wrapper that detects, diagnoses, and isolates device faults. If possible, this wrapper also provides recovery, reconfiguration, and repair of the fault; otherwise, the fault is declared to be unrecoverable and the failed regions of the device are removed from service until an explicit repair operation is scheduled and completed. The wrappers are designed to guarantee atomicity of operation on each symbol. If a node that operates on multiple symbols (e.g., a multi-sector write to disk) encounters a failure in processing some of the symbols (e.g., a single sector fails), the unaffected symbols are either modified correctly or are left unchanged.

In order to guarantee that the fault model of the disk array is preserved, I now describe how these physical faults are modeled by the abstract programming representation and execution mechanism described in the preceding sections. I begin by examining node failures that correspond to the failure of some device. For the modeling and execution techniques described in this chapter to be generally applicable, it is imperative that they do not inherently weaken fault models that are defined for specific array architectures. Furthermore, it is necessary that the programmer understand the mapping of these physical faults into the logical abstractions presented here in order to implement a fault model.

I conclude this section with a discussion of failures that interrupt the execution of graphs but do not correspond to node failures. Specifically, I examine power, crash, and controller failures. A particular array implementation may survive any combination of these types of faults; therefore, it is important that the mechanism described here support all of these failure types.

3.6.1 Node Failures

Device failures are detected by nodes during the execution of a graph. Recoverable device failures are hidden by the nodes from the mechanism that executes the graphs, and their effects are therefore unnoticed outside the domain of the node. Unrecoverable device failures result in the failure of a node. Nodes that fail do so atomically, leaving all symbols being acted upon either unchanged, or marked as failed. Because the effects of all nodes are defined to be undoable, the effects of previously-completed nodes can be removed, permitting the graph to fail atomically. At this point, the system appears to the world as if a fault was detected on an idle system. There is no intermediate state to resolve, and the array is able to easily tolerate device failures as defined by the redundancy of the particular array architecture. In short, a device fault may result in the failure of a node. Node failures cause the atomic failure of a graph, with all observable states being either unchanged or marked as failed.

For example, if a write operation is initiated on a RAID level 6 array which during its execution encounters the failure of a disk sector, the write operation will suspend execution, permitting all in-flight nodes to complete but not scheduling new nodes for execution. When execution has acquiesced, the nodes which completed successfully are undone. The node that failed has marked the regions of the device that failed as inaccessible and has restored the original values to the remaining regions of that device. When the undo of all previously-executed nodes is complete, all visible state changes from the failed graph have been removed. Because the array is assumed to be initially in a consistent state, this process has returned the system to a state that is free from error. The array controller can now retry the write operation, possibly using a new algorithm that avoids the failed sector. Additionally, a repair operation can be scheduled to return the array to a fault-free state.

As long as the execution mechanism is known to be functioning and the nodes are known to operate correctly, device failures are easily handled in this general fashion, irrespective of type or device. In this dissertation, I do not address nodes or execution mechanisms that do not behave correctly, other than to assume that errors in their design will result in failstop behavior. I do address failures that result in the interruption of the execution of a graph.

3.6.2 Crash Recovery and Restart

If the execution of a graph is interrupted due to a controller failure, loss of power, or some other fault, a *crash* is said to have occurred and a process called *restart* is required to bring the system back to life and restore consistency. Because a crash is an asynchronous event, it is likely that a number of graphs will be incomplete when restart is initiated—furthermore, some of these graphs may have been executing in the forward state and some may be executing in the backward direction. Because redundant disk arrays are not obligated to fail atomically with respect to crashes (Chapter 2), the restart procedure is necessary only to prevent the appearance of a write hole. However, it is possible to make the array fail atomically when a crash occurs by making the entire undo log durable, guaranteeing that the restart procedure will be able to undo the effects of all operations which were executing at the time of the crash.

Assuming a single-fault-tolerant array (i.e. the array does not survive simultaneous disk and power failures), guaranteeing the semantics described Chapter 2 requires:

1. Each codeword in the array is in a consistent state, where "consistent" implies a legal codeword (e.g., even parity).

2. Operations interrupted by the crash will not affect the visible state of unrelated user regions.

3. Operations interrupted by the crash will reach completion (pass or fail).

The most simplistic restart procedure, common in systems which are not capable of surviving power failures without loss of all controller state, is to make each codeword in the array consistent by sweeping through the entire array and manually updating the check information if the codeword is found to be inconsistent. This process, commonly referred to as "scrubbing the array" can take several minutes for an array of even modest size. The process can be shortened if the assumption that the host will notify the array controller which operations were outstanding at the time of the crash, eliminating the need for the controller to examine all codewords in the array. Similarly, the array controller could devote a modest amount of nonvolatile memory (4KB) to recording this information locally [Symbios96].

Finally, it is possible that the restart procedure itself can be interrupted by a crash, implying that the restart procedure may be executed a number of times before being allowed to reach completion. Therefore, the restart procedure must be idempotent and the nodes of a graph must be guaranteed to have "exactly once" semantics [Gray93]. This may be accomplished by either making each action idempotent or testable, meaning that the system is able to discern whether the action has been executed or not.

3.6.3 Controller Failover

In a system with redundant controllers, if a controller fails due to a permanent fault, a process known as *controller failover* is used to transfer the work of the failed controller to a surviving controller. This process includes restarting the work which was in progress on the failed controller. To do this, the undo and redo logs of each controller must be visible to the surviving controllers, either by placing it in a central location (a disk) or by mirroring it in a second controller. Given the contents of the failed controller's logs, the surviving controller applies the standard restart procedure and then continues normal processing.

3.7 Summary

Instead of relying upon ad hoc methods for creating redundant disk array software, this chapter has introduced a structured approach which has the advantage that it is general (e.g. not specific to array architecture), does not require hand-crafting or analysis of errors, is amenable to automated correctness verification, and can be extended to provide atomic recovery from crashes. By isolating action-specific error recovery, array-level error recovery has been automated and the previous dependence upon a hand-analysis of array-specific error scenarios has been eliminated.

Array algorithms are represented as flow graphs, directed acyclic graphs whose nodes are the actions, such as a disk write, which perform work in the system, and whose arcs are the dependencies, control or data, between these actions. Section 3.2 presented the rules for constructing the actions, Section 3.3 presented the rules for constructing the graphs, and Appendix A demonstrated that from only a small set of actions, a complete library of flow graphs which implement the most popular redundant disk array architectures is easily created.

Executing redundant disk array operations represented as flow graphs is easily mechanized. If the nodes of a graph are atomic and undoable, a graph can be executed in a fashion similar to transactions, providing atomic failure of the graph and automated recovery from errors. Furthermore, because the execution mechanism is general, the task of verifying the implementation is limited to ensuring that the infrastructure (actions and execution mechanism) is correctly implemented and that the graphs are correctly designed—there are no case-by-case recovery procedures to validate. Finally, if the undo log is made durable, array operations can be made to fail atomically in the event of a crash.

Chapter 4 introduces RAIDframe, a framework for prototyping disk arrays which is based upon the paradigms (modular software, node/graph programming abstraction, mechanized execution) described in this chapter. Because creating undo information for actions such as a disk write may be expensive (a full disk rotation is required), Chapter 5 demonstrates that the requirement that all nodes are undoable may be relaxed. Using RAIDframe, the performance consequences of requiring all nodes to be undoable is studied and a novel method for eliminating the requirement that all actions, particularly the expensive ones, is introduced.

Chapter 4: RAIDframe: Putting Theory Into Practice

This chapter introduces RAIDframe, a software package for prototyping redundant disk arrays, developed by researchers at Carnegie Mellon's Parallel Data Laboratory in the mid-1990's [Courtright96a, Courtright96b]. One of the principal research thrusts of the Parallel Data Laboratory (PDL) is the exploration of new redundant disk array architectures [Holland94, Gibson95]. RAIDframe was developed to provide researchers with an easily-extended platform for implementing and testing new redundant disk array architectures. A thorough examination of RAIDframe can be found in: *RAIDframe: Motivation, Theory, and Implementation* [Courtright96c].

RAIDframe is based upon the approach introduced in Chapter 3: modular design, modeling operations as flow graphs, and automating error recovery. At the time of this writing, RAIDframe supports RAID levels 0, 1, 4, and 5 [Patterson88] as well as parity [Holland92] and chained [Hsiao90, Hsiao91] and interleaved declustering [Copeland89] array architectures. RAID level 6 [RAB96], EVENODD [Blaum95], and parity logging [Stodolsky94] are under study. I present RAIDframe as evidence that array software constructed using this approach is not only feasible, but desirable.

In this chapter, I introduce the internal structure of RAIDframe and the fundamental design decisions that guided its evolution and affect the nature of studies conducted in this dissertation. I then describe our experiences with creating and modifying array architectures, revealing that modular code and automated recovery from errors allowed new array architectures to be implemented with only minimal code changes. The chapter concludes with an examination of the efficiency of RAIDframe, comparing the results obtained through microbenchmark studies with those predicted by analytic models as well as direct comparisons to a hand-crafted striping (nonredundant) driver. Later, in Chapter 5, I use RAIDframe to evaluate the relative performance of various error recovery schemes.

4.1 Motivation

Historically, researchers studying redundant disk arrays have been forced to evaluate new architectures using either analytic or simulation methods [Blaum95, Cao94, Chen90, Lee91, Menon93b, Mogi94, Patterson88, Savage96, Stodolsky94]. Chapter 2 described a variety of these architectures, most of which have never been implemented beyond the first-order modeling required for event-driven simulation. In short, researchers were introducing new architectures and evaluating their performance, but were not demonstrating to implementors the feasibility of constructing working prototypes.

In an attempt to bridge the gulf between simulation models and concrete implementations, Ed Lee and others at the University of California's Berkeley campus developed an event-driven simulator which they called *raidSim* [Chen90, Lee90a]. By drawing code from the Sprite operating system [Ousterhout88], raidSim was designed to allow the code that implemented an array architecture in a simulation environment to be based upon code from a working system. This gave researchers the opportunity to analyze and refine an array architecture in a relatively-simple environment that was isolated from the complexities of a working system. Once the implementations was stabilized, the resulting code could be ported back into the real system. Since it's introduction, raidSim has been made publicly available [raidSimFTP] and was used extensively by Holland in his studies of arrays in which a disk has failed [Holland94].

We see raidSim as an important step toward the ability to rapidly prototype and evaluate new array architectures. While raidSim did reduce the labor required to move evaluation from the simulation to production environment, it did not fully capitalize upon the similarities between array architectures beyond data layout and encoding: raidSim was simply an event-driven simulator which provided an interface similar to that used by the Sprite operating system.

In 1993, Garth Gibson, Mark Holland, Daniel Stodolsky, and I realized that constructing a framework designed specifically to reduce the overall labor of defining, implementing, and evaluating array architectures could greatly benefit our work. Following the lead of raidSim, we isolated functions that controlled data layout and encoding. Furthermore, we developed a library of pass/fail actions as described in Chapter 3, a general method (flow graphs) of composing array operations from them, and an execution mechanism that automated recovery from errors, eliminating the need to develop architecture-specific error-recovery code. The framework we developed, which we called *RAIDframe*, allows architectures to be evaluated in each of three environments without the need for code changes: (1) an event-driven simulator, (2) a user process which communicates with real disks through their raw device interface, and (3) as a device driver installed in a UNIX kernel.

4.2 Architecture

The initial goal we established for the RAIDframe project was to simplify the task of implementing new redundant disk array architectures. Our basic approach was conventional: partition the architecture of RAIDframe into modules that are known to change orthogonally with array architecture. Because we were interested in implementing a variety of array architectures, we desired clear methods of reusing existing code to the fullest possible extent. This meant not only reusing actions such as "disk write" that are common to many array architectures, but also collecting and isolating as much architecture-specific code as possible. Therefore, we designed RAIDframe to execute array operations in a general manner in which the execution of graphs, including the recovery from node failures, is automated, irrespective of array architecture.

4.2.1 Design Decisions

As with any design, we made a series of decisions that affected the scope of the project, allowing a balance between several factors (in our case, efficiency and complexity) to be achieved. First, we simplified our disk fault model to recognize only catastrophic disk faults. We easily justified this because RAIDframe is intended to be strictly a prototyping framework and our foreseeable studies do not require the discernment of sector and catastrophic disk faults. However, should the need arise, it is possible to extend RAIDframe to distinguish the failure of an individual sector by incorporating additional maps to record the locations of the faults, so the fault model can be expanded at later time.

Second, we offered no services dedicated specifically to the survival of power, controller, or cooling faults. This limitation is due to the fact that RAIDframe is delivered as a software-only package which we hope will be eventually ported to a variety of platforms. Eliminating the requirement for nonvolatile memory was seen as a necessary measure to increase the likelihood of the future portability of RAIDframe. RAIDframe guarantees that after a crash, its internal structures (not disk state) are returned to their original power-on state. The damage observed by a user who experiences a crash is largely determined by the specific architecture being implemented—if, for example, dynamic mapping structures were being maintained in RAIDframe's volatile memory, then a crash would result in the loss of these maps and the inability to access previously-written data.

Third, RAIDframe will never attempt to construct a graph which accesses more than a single parity stripe. As Figure 4-1 illustrates, this restriction greatly simplifies the graph selection and creation routines by eliminating the need for scatter-gather actions which would otherwise be necessary to overcome the discontinuity of data at stripe boundaries. Requests that map to multiple parity stripes are broken into two subrequests which are executed concurrently. Each subrequest is executed independently and the entire operation is successful only if all subrequests complete successfully. In the event that one or more of

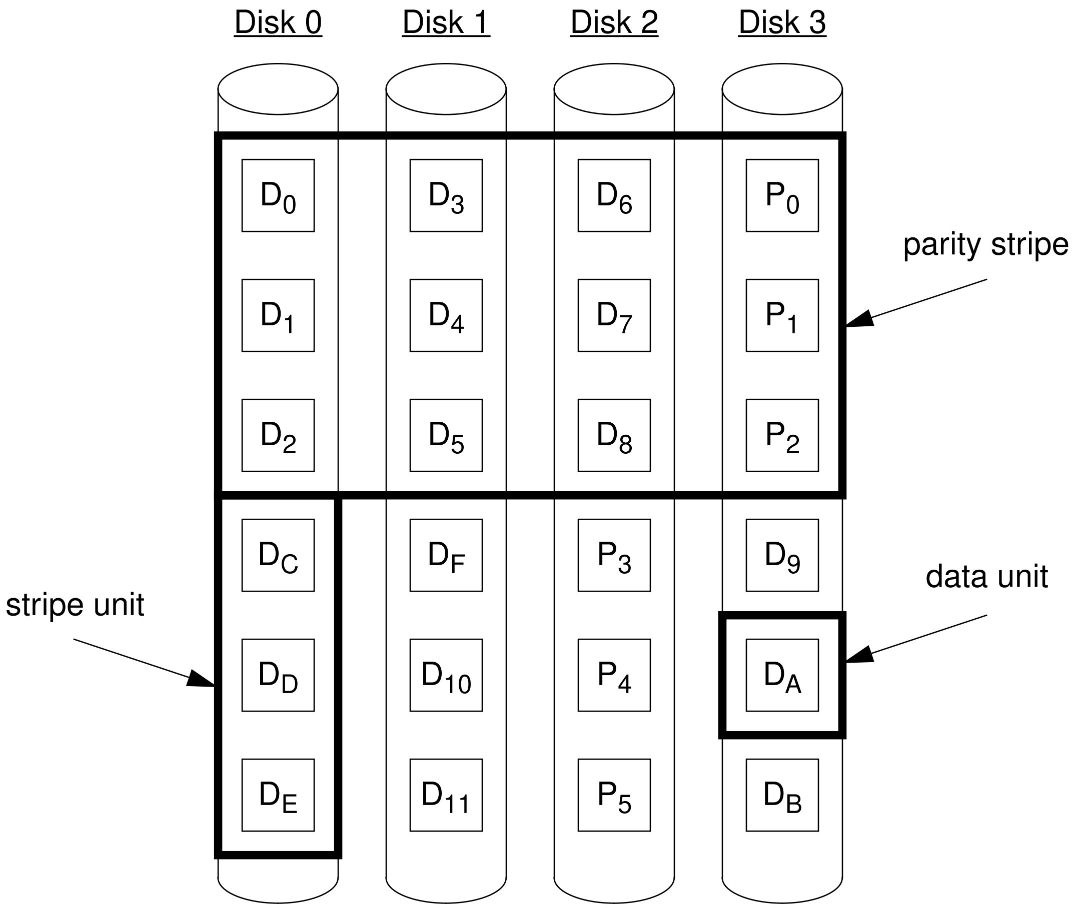


Figure 4-1 Processing parity stripes independently

A data unit is the minimum unit of access supported by the array. In this illustration, a four-disk RAID level 5 is presented in which the size of a strip unit is equal to three data units. A read request which accesses data units D_1 *through* D_8 *is contained entirely within a single parity stripe and can be implemented using three disk actions: read* D_1-D_2*, read* D_3-D_5*, and read* D_6-D_8*. Each read action independently transfers data into a buffer in a contiguous fashion, without the need for scatter/gather DMA. However, if the read request crossed a parity stripe boundary, for instance requesting* D_1 *through* D_{10}*, routing data from the disk actions to a buffer is not as straightforward—if a single disk action were used to read* D_1-D_E *from* Disk 0*, the data units* D_1-D_2 *and* D_C-D_E *would need to be routed to discontinuous regions of the buffer. To avoid the need for scatter/gather transfers, RAIDframe will break this request into two concurrently executed subrequests,* read D_1-D_8 *and read* D_C-D_{10}*. Requests to the same disk (e.g. read* D_1-D_2 *and read* D_C-D_E*) may be queued at the physical drive which is able to preserve performance by eliminating the need for a second pair of seek and rotate positioning operations.*

the subrequests are unable to complete, the request fails with the traditional disk semantic that some regions of the request were successfully written while others were not. Proposals for achieving atomic failure semantics of the entire request (instead of at each subrequest) are discussed in Chapter 6. Disk actions that are divided into multiple actions and queued at a drive may be later reassembled by the drive to preserve seek efficiency.

Fourth, an architecture implemented in RAIDframe should run with a minimal set of graphs. RAIDframe will attempt to construct a single graph for each parity stripe accessed in a request; however, we do not require that implementors provide graphs for all possible single-parity-stripe optimizations. If RAIDframe discovers that a single graph is not available to process an entire request, it will successively decompose the request into subrequests, first to a series of stripe unit, and then data unit, accesses. Therefore, an implementor is only required to install graphs for all possible single data unit accesses. Graphs that perform specialized algorithms given specific size and alignment combinations (e.g. the RAID level 5 reconstruct write of Figure 2-10) may be added seamlessly over time.

Finally, resource (buffers and locks) allocation and deallocation is performed outside of graph execution, limiting the scope of rollback during backward error recovery. RAIDframe provides simple mechanisms for locking address ranges and managing a shared buffer pool. Instead of releasing and then reacquiring locks on operations that have failed and are later retried, RAIDframe holds the locks until retried operations either complete successfully or fail in a manner in which retry is not possible (e.g. more than one fault in a single-fault-tolerant array). Buffers are allocated when graphs are created, allowing the fields of individual nodes to be statically initialized prior to graph execution. This greatly simplifies the process of debugging the graph-creation routines. When a graph completes, a generic routine is called to free the graph and all of the buffers that were attached to it.

We made other decisions which affected the design of RAIDframe, such as the manner in which we manage buffer pools, but I believe that the decisions presented here are the most significant in terms of differentiating RAIDframe from other implementations. The greatest deviations from array product offerings are the ones concerning the fault model, specifically RAIDframe’s lack of support for sector and crash failures. Because RAIDframe’s fundamental purpose is for prototyping, I do not believe that these omissions limit the scope of RAIDframe’s original goal, to prototype new disk array architectures. Furthermore, production disk arrays generally do not survive crashes atomically. Tolerating sector failures simply requires the addition of mapping information to track and repair the failed sectors. Surviving crashes requires making the undo log durable and other well-understood techniques for ensuring that the failure of a single controller can be tolerated [Chen94, Gray90b, Menon93a].

4.2.2 Libraries

RAIDframe's framework is partitioned into six primary modules that may be modified by implementors. These modules (mapping, actions, graph, graph selection, disk queueing, and disk geometry) contain the libraries from which arrays are implemented. The *mapping library* contains routines that transform the logical block address of the user into a set of physical disk locations. This process includes the identification of data and check (e.g. parity) units as well as any faults observable by the access. The *action library* contains the pass/fail actions, described in Section 3.2.1, out of which are created the nodes of graphs. The graph templates, which are used to implement array operations, are contained in the *graph library*. The *graph selection library* contains the architecture-specific routines that identify the appropriate graph template to use given the mapping information (location and size of access, location of faults, etc.). Examples of the graphs initially used in RAIDframe and the criteria for their selection are presented in Appendix A. These graphs were initially designed without any error recovery procedures and were later replaced by the graphs of Appendix B which rely upon the two-phase error recovery scheme described in Chapter 5.

Array performance has an intimate relationship with disk performance. RAIDframe allows researchers to exploit this relationship by providing disk queueing and geometry libraries. The *disk queueing library* allows the number of outstanding requests issued to a single disk drive to be specified as well as the method for queueing disk requests (e.g. FIFO, CVSCAN, SSTF, and SCAN [Geist87]) before dispatching them to the drive. When RAIDframe is used as a simulator, a variety of disk models are available in the *disk geometry library*. These models, taken from raidSim, allow researchers to study the sensitivity of the array to a wide-variety of disk parameters, including those pertaining to: sector layout, zone layout, and seek profiling,

4.2.3 Processing a User Request

Through the use of a programmable state machine, RAIDframe permits the basic sequence of events used to process user requests to be tailored to a specific array architecture. Specific optimizations such as the caching of user requests could be inserted on a per-architecture basis. However, we have found that the general sequence illustrated in Figure 4-2 works well for all of the architectures which we have implemented to date. For instance, some architectures may support read caching and therefore may include steps to probe a cache prior to commencing the process of selecting a graph and scheduling disk work. Regardless of architecture, all control sequences are constructed with the premise that requests are broken into graphs which are executed concurrently and fail atomically. If a graph fails, the (sub)request is retried and, assuming the state of the array has changed, a different graph is used. This process repeats until either all graphs complete successfully or a graph can not be selected.

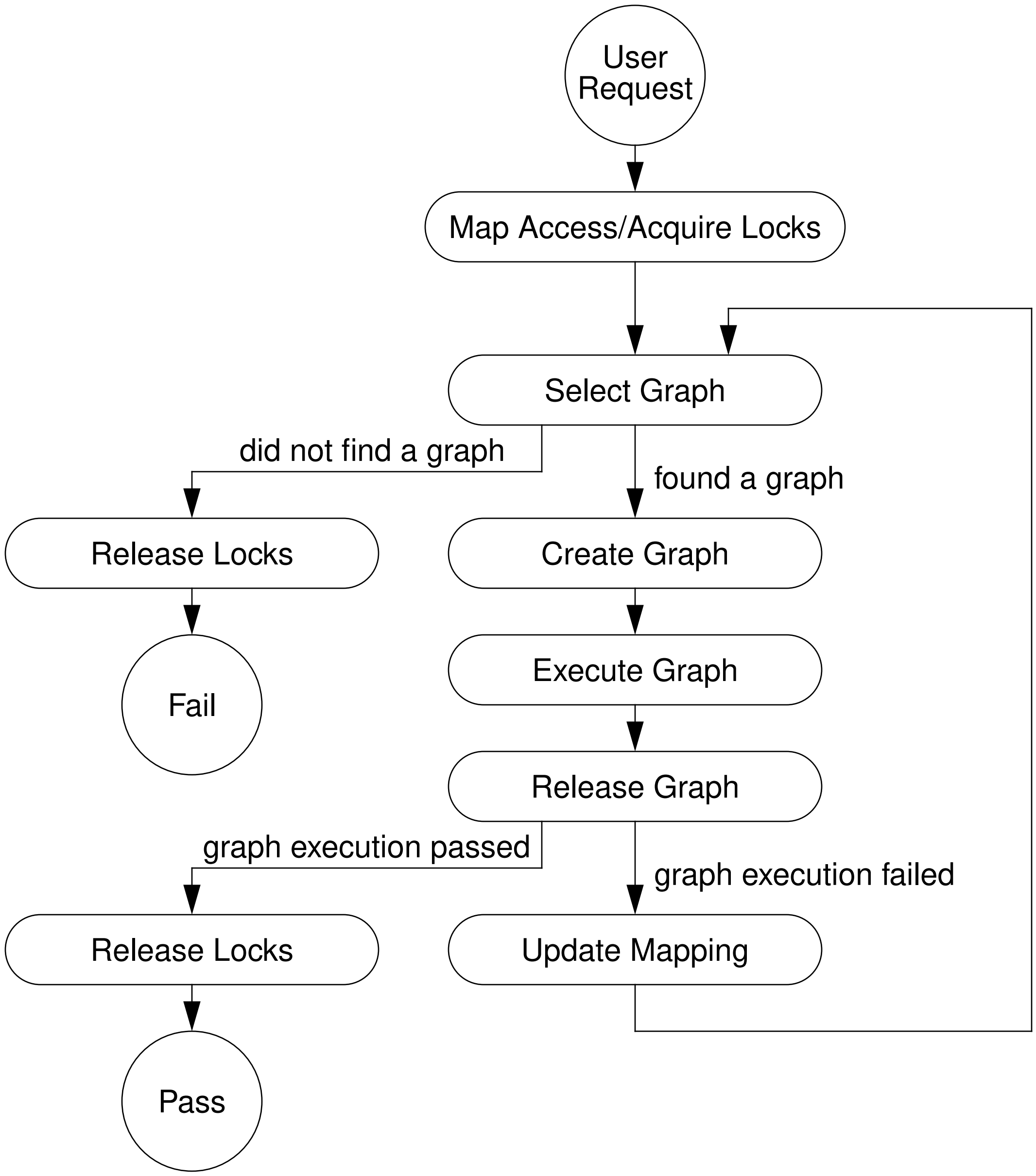


Figure 4-2 Mechanism for processing user requests

A user request only fails if a graph can not be selected. If a graph fails during execution, it is released and RAIDframe automatically retries the request, beginning with the graph selection process. Graph failures are the result of permanent faults which cause the configuration of the array to be altered. Therefore, because graph selection is a function of the presence of a fault, subsequent retries will select different graphs.

In Figure 4-2, requests are first mapped to the array and the necessary locks (e.g. write locks on parity ranges) are acquired. This step includes decomposing requests that span multiple parity stripes into subrequests, each of which access a single parity stripe. Next, using an architecture-specific graph-selection function, the appropriate type of graph is identified for each subrequest. Recall from the discussion of design decisions in Section 4.2.1 that if a single graph is not available for each parity stripe, each subrequest will be further subdivided. Graph selection will fail if the region being accessed contains more disk faults than can be tolerated, or if RAIDframe's graph library does not contain the minimal complement of graphs necessary to access all single data unit access patterns. If any part of graph selection fails, all locks are released and the request fails.

Assuming graph selection is successful, the selected graph-construction routines are used to create instances of the desired graphs. The process of creation includes allocating buffers that are required by the graph. Once graph creation has completed, the graphs are ready for execution. Graphs are submitted to an execution engine which guarantees that each graph will either complete successfully or fail atomically. When execution completes, the graphs and buffers attached to them are released. If all graphs complete successfully, the parity locks are released and the request is complete.

If the execution of a graph fails, using architecture-specific mapping routines, similar to those used in the original mapping function, the mapping information for the associated request is updated, allowing the presence of recently-detected faults to be incorporated. The process described above is then repeated. The graph selection process is re-entered, and a new (presumably different) graph is selected.

4.2.3.1 Locking

As Figure 4-2 illustrates, RAIDframe acquires locks prior to graph selection and execution and therefore the graphs used to represent these locks are not included as nodes in the graphs used by RAIDframe. The locking step of Figure 4-2 represents the acquisition of locks that ensure that each request is processed in isolation of other requests that may be in flight. Because the locks are held until processing of the request is completed, regardless of outcome, the rollback of a failed graph will not involve unlocking and the re-locking of state.

4.2.3.2 Error Recovery

As RAIDframe evolved, so did our understanding of how best to automate recovery from failed nodes. Initially, using the graphs of Appendix A, we experimented with array-specific techniques that mimicked backward error recovery [Courtright94]. We discovered that this approach was inadequate because it did not support arrays that tolerated the loss of more than one disk (e.g. RAID level 6). In the end, we developed a novel method of mechanized recovery, roll-away error recovery, that I later describe in Chapter 5.

In Chapter 5, I use RAIDframe to examine the relative performance of forward, backward, and roll-away error recovery schemes. To ensure an "apples-to-apples" comparison in the validation studies of Section 4.3.4, I assume forward error recovery and employ the graphs found in Appendix A without undo logging. However, all development work in RAIDframe is based upon roll-away error recovery and it is the graphs contained in Appendix B that are shipped as part of the RAIDframe package.

4.3 Evaluation

This section evaluates RAIDframe's ability to be extended, and the efficiency of array architectures implemented in RAIDframe, in terms of performance overhead. A discussion of the verification of RAIDframe's ability to tolerate faults is deferred to Section 5.5, which follows the discussion of the error recovery procedures used within RAIDframe.

To evaluate RAIDframe's ability to be extended, I examine the extension of a baseline RAID level 0 implementation to support seven additional architectures: parity declustering [Holland92], RAID levels 1, 4, 5 [Patterson88], RAID level 6 [ATC90, RAB96], chained declustering [Hsiao90, Hsiao91], and interleaved declustering [Copeland89, Teradata85]. The following analysis reveals that introducing these architectures required only modest code changes and that the changes were localized. While the majority of the architectures were implemented by members of the RAIDframe development team, the interleaved and chained declustering architectures were implemented by Khalil Amiri, a first-year graduate student who was not a member of the team.

I evaluate RAIDframe's performance by first examining its efficiency when compared to a hand-crafted striping (nonredundant) driver. RAIDframe returns the same response time versus throughput results as the striping driver, regardless of array size. However, RAIDframe's abstract programming interface demands a 60% CPU utilization premium over the hand-crafted driver. Finally, I evaluate RAIDframe's small-access performance for eight disk array architectures which shows a favorable comparison to the results predicted by simple throughput models. I perform this final evaluation for each of RAIDframe's three operating environments (simulator, user process, and device driver) and also evaluate their consistency.

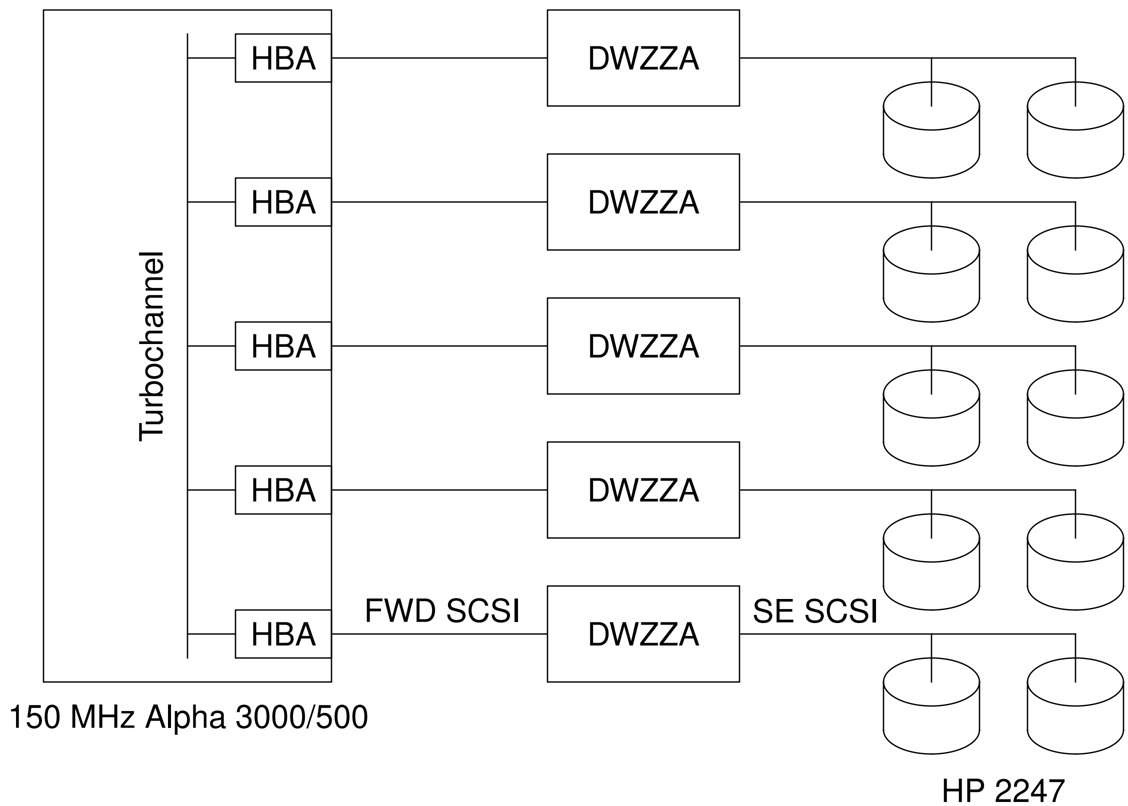


Figure 4-3 Setup used for collecting performance data

RAIDframe was installed on an Alpha workstation running Digital UNIX™ version 3.2. Because Digital UNIX does not support downloadable device drivers, a custom kernel which contained RAIDframe had to be created. Five KZTSA Turbochannel to Fast-Wide-Differential (FWD) SCSI Host Bus Adapters (HBA) were used to cable to the disk drives which were housed in StorageWorks cabinets. FWD SCSI was used out of the Alpha to accommodate the necessary cable lengths. The ten Hewlett-Packard 2247 disk drives were 8-bit single-ended (SE) SCSI 1 GB drives. Five FWD to SE SCSI converters (DWZZA), installed in the StorageWorks cabinets, were used to connect the single-ended drives to the fast-wide-differential cables.

4.3.1 Setup

RAIDframe was developed at Carnegie Mellon's Parallel Data Laboratory on DEC Alpha workstations running Digital UNIX™ version 3.2. All experiments in this dissertation were conducted on a DEC Alpha 3000/500 workstation with 128 MB memory connected to ten HP 2247 disk drives. This details of this setup are illustrated in Figure 4-3. All equipment used is commercially available and, with the exception of the kernel, was used without modification. The RAIDframe source code is available via anonymous ftp

[RAIDframeFTP]. Unfortunately, due to licensing restrictions, I am not able to provide a copy of the kernel code or the modules that we used to integrate RAIDframe into Digital UNIX. Fortunately, RAIDframe contains no kernel-specific dependencies which would alter its performance, so researchers wishing to duplicate these experiments should be able to do so by independently building their own kernels or running RAIDframe as a user process, entirely independent of the underlying kernel.

Experiments were conducted using configuration files similar to the one in Section C.3. All array configurations used a stripe unit size of 32KB (sixty-four 512 byte sectors per stripe unit) [Chen90]. Stripe size, the total storage associated with a stripe, varies with experiment. RAIDframe was permitted to dispatch up to five outstanding requests to each disk—beyond that, RAIDframe managed a separate queue for each disk using shortest seek time first (SSTF) scheduling. Parity declustering used a logical parity group of five disks distributed over the physical array of ten disks.

4.3.1.1 Workload Generation

The efficiency and accuracy studies presented in the remainder of this section were conducted using pseudo-random workloads. These workloads were executed using RAIDframe's trace-playback mechanism which applied identical, high-concurrency access sequences to each architecture executing in the simulation, user-level, and device-driver environments.

Workload files were created for 1, 2, 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50 threads. Each thread issues blocking (synchronous) 4 KB operations on random 4 KB aligned addresses with no intervening delay. Because accesses were aligned and smaller that the stripe size, the number of graphs executing at any instant was less than or equal to the number of threads. Read and write workloads were generated and executed separately. All experiments display the average of three experiment runs. The data presented in the performance figures, and their 95% confidence intervals, are presented in Appendix C. In all experiments, these data represent the average of three experiment runs, achieving a confidence interval which is typically 2% of the mean and does not exceed 7.8% of the mean.

4.3.2 Extensibility

To evaluate the cost of implementing new architectures in RAIDframe, I present an anecdotal history of the architectures that we have implemented, examining the size and locality of code changes necessary to extend RAIDframe. The initial coding effort to construct the basic RAIDframe platform resulted in the creation of 122 files and a total of 34,311 lines of code (LOC). RAIDframe is written in C and, in this study, I define LOC to include both source statements as well as comments. This initial framework, which was designed to eventually support RAID levels 4, 5, and parity declustering, included a working RAID level 0 implementation and provided the infrastructure necessary to conduct performance and correctness experiments (e.g. workload generation, tracing, data verification tools). Once this framework was established, the architectures listed in Table 4-1 were added in the order presented. This table measures “code reuse” as the ratio of unchanged code to the amount of total code required to support the new architecture.

Adding parity declustering to RAIDframe required the addition of 7 new files, totalling 2,021 LOC, which contained the nodes and graphs necessary to implement a parity-encoded array architecture. These files also included the architecture-specific mapping and graph selection routines particular to parity declustering. Additionally, 395 LOC of existing code, contained in 5 files, were modified. The infrastructure of RAIDframe was extended to perform reconstruction of a failed disk onto a spare. Debug functions which verify the correctness of parity, necessary for functional testing, were also added. Together, adding parity declustering affected 12 files (out of 189) and 2,416 LOC (out of 36,727) or about 6.5% of RAIDframe’s total LOC and file counts.

For the sake of comparison, consider the amount of work required to incorporate parity declustering into raidSim, a 92-file, 13,886-line simulator: 1 file was deleted, 11 files containing 1,204 lines of code were added and 46 files were modified, changing 1,167 lines and adding 2,742 lines. Collectively, to enable the research reported in one paper, the number of lines of code, 5,113, was equivalent to 36% of the size of the original simulator and affected over half of raidSim’s modules. This is about twice as much code, measured in LOC, as required to extend RAIDframe and the changes were distributed

Table 4-1 Cost of creating new architectures

Architecture	New Code	Modified Code	Code Reuse
RAID level 0	34,311 LOC 167 files	—	—
parity declustering	2,416 7 files	395 LOC 5 files	93.4%
RAID level 5	355 LOC 3 files	5 LOC 1 file	99.1%
RAID level 4	134 LOC 2 files	5 LOC 1 file	99.6%
RAID level 6	2644 LOC 7 files	88 LOC 4 files	93.2%
RAID level 1	373 LOC 2 files	35 LOC 2 files	99.0%
chained declustering	117 LOC 2 files	5 LOC 1 file	99.5%
interleaved declustering	119 LOC 2 files	5 LOC 1 file	99.5%

across five times as many files. Even though raidSim was not originally constructed with parity declustering in mind, it already supported RAID level 5, an architecture which differs only in the mapping of data and parity. It is also important to remember that raidSim is just a simulator—the changes to RAIDframe represent code which can operate in three environments, two of which move data to disk and verify the correctness of those transfers.

With parity declustering working, the process of implementing RAID levels 4 and 5 became trivial. These two architectures completely reused all nodes and graphs of parity declustering as well as its graph-selection function. In essence, these two architectures only required changes in the data and parity mapping functions. For RAID level 5, this resulted in the creation of 355 LOC contained in 3 new files and the modification of 5 LOC in 1 existing file. Similarly, for RAID level 4, this resulted in the creation of 134 LOC in 2 new files and the modification of 5 LOC in 1 existing file.

Adding RAID level 6 scaled RAIDframe support to multiple-failure-tolerating arrays by requiring the addition of nodes for encoding and decoding the Reed-Solomon encodings as well as the creation of graphs which maintained these codes. Beyond this, RAID level 6 required the standard additions of graph selection and mapping functions. Unlike the other architectures described here, RAID level 6 is not a complete implementation: the Reed-Solomon encoding/decoding actions are not fully debugged and the procedure which reconstructs the contents of a failed disk has not yet been modified. Ignoring reconstruction and assuming that the debug of the encoding/decoding nodes will not significantly change their current LOC count, RAID level 6 required the addition of 2,644 LOC contained in 7 new files and the modification 88 LOC contained in 4 files.

When I implemented RAID level 1, I maintained a detailed record of the time required to complete the implementation. The overall effort required 460 minutes, during which 370 lines of code were produced or modified. Broken down, the time until first compilation was attempted was 90 minutes. This time included the creation of new mapping and graph selection routines as well as a new graph used to write data to a fault-free array. Once begun, the first successful compilation occurred 15 minutes later. Read graphs first ran 55 minutes later, write graphs 105 minutes later. A run-time optimization of fault-free reads which selected the mirror-copy with the shortest queue required an additional 145 minutes. Finally, an additional 50 minutes were required to complete the implementation of degraded-mode operation.

The implementations of chained and interleaved declustering, created by Khalil Amiri, a member of the PDL who was not an author of RAIDframe, produced similar results. Khalil required approximately 240 minutes to implement each of these two architectures. In each case, approximately two-thirds of his time was spent coding and one-third debugging.

In the end, after implementing eight architectures and experimenting with parity logging, log-structured storage, and EVENODD, we have consistently reused over 90% of the existing code in RAIDframe. I contribute this level of reuse to the modular partitioning of RAIDframe and the elimination of architecture-specific error recovery from failed node failures. Code changes were confined and localized to: new actions, graphs, graph selection criteria, mapping, and reconstruction.

An equally important result is the elimination of architecture-specific testing. The architect only needs to ensure that the graphs are correctly designed and implemented—RAIDframe's infrastructure guarantees correct execution in the event of a node failure.

4.3.3 Efficiency

To evaluate RAIDframe's ability to efficiently process requests, we compared RAIDframe's implementation of RAID level 0 kernel device driver to a nonredundant striping driver which was independently developed for use in a file system project [Patterson95]. The first experiment established the performance of RAIDframe on a single disk. Figure 4-4 compares the single disk performance of RAIDframe's device driver to that of the hand-crafted striping driver. Both drivers show near-identical response times at a given throughput with a maximum throughput of approximately 80 IO/s.

Next, we examined the sensitivity of performance and CPU utilization to array size. With a constant workload of five threads per disk, each generating synchronous 4 KB requests, we measured response time versus throughput and CPU utilization for read and write workloads. Figure 4-5(a) illustrates that RAIDframe produces the same results as the hand-crafted driver. The general shape of this graph is not pleasing to the eye: as the number of disks increases, so does throughput; however, response time is not linear or even monotonic. This can be explained by several effects. First, as the system becomes more heavily loaded, the response time will increase due to the Alpha's limited ability to process interrupts. This explains the 300-500 IO/sec region of the curves. Second, as the number of disks is increased from one to two, so are the array capacity and workload. This means that in the experiment with one disk and five threads, the disk was processing one requests while four waited in it's queue. Because the workloads are random, the maximum queue depth per disk increases with the number of disks. In the two-disk experiment, it was possible for there to be up to nine requests queued at a single disk. This explains the sudden rise in response time (11%) and modest increase in throughput (17%). As the number of disks in the array increases, the likelihood of all accesses colliding on a single disk diminishes, and performance increases as one would expect (increasing throughput and flat response time). Again, regardless of shape, it is the strong correlation of RAIDframe to the hand-crafted driver which is important.

Figure 4-5(b) illustrates that, regardless of array size, RAIDframe consumes 60% more CPU cycles than the striping driver. This is due to the layered architecture of RAIDframe which simplifies programming through abstract operations and modular partitioning. I investigated this hypothesis by profiling the execution of the two drivers using *Atom*, a commercially-available tool sold by Digital Equipment. I found that RAIDframe's call tree contained sixty functions while the striping driver's contained only nine.

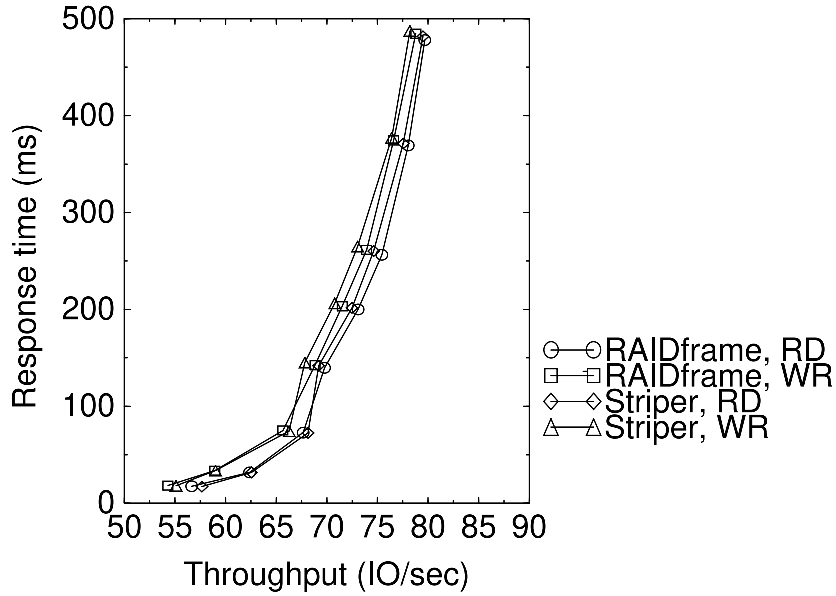


Figure 4-4 Single disk performance of striper and RAIDframe

Both implementations evaluated as device drivers executing in a Digital UNIX™ kernel. The eight datapoints for each curve were produced by a random workload of synchronous 4KB requests from 1, 2, 5, 10, 15, 20, 30, and 40 concurrently-requesting agents. SSTF disk queueing was used within the driver and up to 5 requests were allowed to be queued to the physical drive. This explains discontinuity in the graphs which occurs when the number of threads (10) exceeds the maximum number of simultaneous requests which may be dispatched to a drive (5). The reduction in performance suggests that increasing the queue depth would improve performance. Note that the x-axis does not begin with 0.

RAIDframe has four entry points: `open()`, `close()`, `read()`, and `write()`. The calls to `open()` and `close()` total less than 1% of execution time and I dismiss them from further study. With the exception of locking the extent to be written, execution profiles were not sensitive to the type of workload, read or write. This locking disparity was less than 1% and therefore no distinction is made between read and write workloads in the following presentation.

Table 4-2 presents a breakdown of RAIDframe's execution time based upon the functions presented in Figure 4-2. Keep in mind that idle time spent waiting on a disk drive to complete a request is not included and therefore, these percentages are of total CPU time and not total user (i.e. "wall clock) time. Without surprise, the big-ticket item here is the process of executing a graph. This process requires walking the graph and tracking the completion status of each node. An additional function, "state machine overhead," also appear in the table. This function represents the time spent initializing and processing the state machine of Figure 4-2.

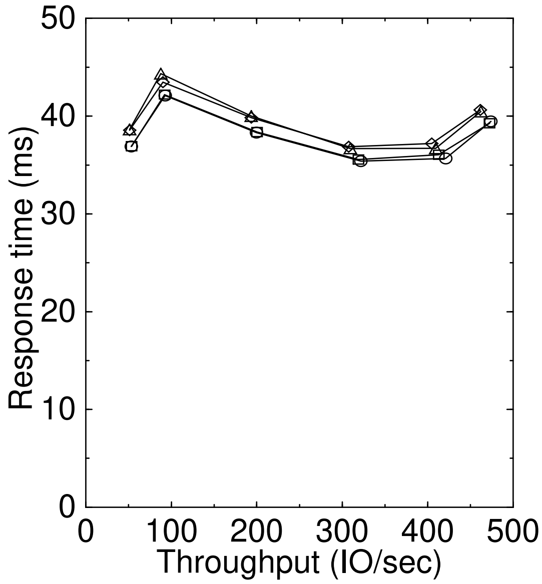


Figure 4-5(a): Constant workload/disk

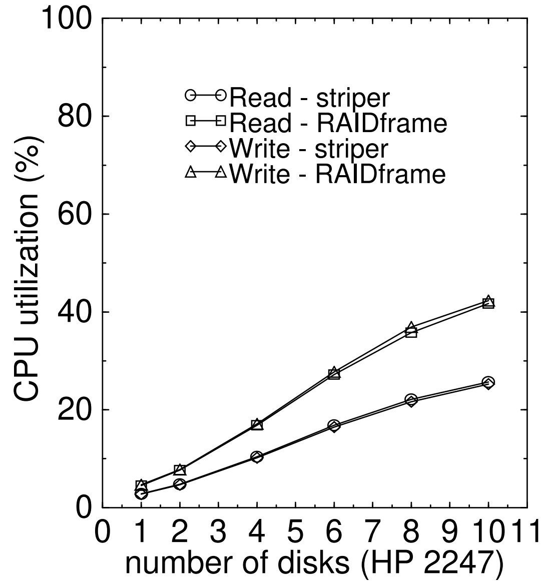


Figure 4-5(b): CPU consumption

Figure 4-5 Comparing RAIDframe to a hand-crafted implementation

This study compares the performance of RAIDframe's RAID level 0 implementation to that of a hand-crafted striping driver. Data was collected for arrays of 1, 2, 4, 6, 8, and 10 disks. The workload per disk was constant at five threads per disk, each thread generating synchronous, random 4 KB requests.

Table 4-2 RAIDframe execution profile

Function	%
mapping	7.93
lock acquisition	0.70
graph selection	7.37
graph creation	8.38
graph execution	46.78
graph release	7.86
lock release	4.22
state machine overhead	18.92

4.3.4 Verification

The expected performance of eight disk array architectures implemented in RAIDframe was verified by comparing the response time versus throughput characteristics of eight array architectures against the predictions of analytical models. Additionally, we compared the consistency of the results obtained from each of RAIDframe's three operating environments: simulator, user process, and device driver. In all experiments, we used ten disks and the 1, 2, 5, 10, 15, 20, 30, and 40 thread workloads, previously described in Section 4.3.1.1.

First, the ten-disk RAID level 0 read performance should improve by about a factor of ten over the results of Figure 4-4, which indicated a single-disk throughput of about 70 IO/sec at the "knee" of the curve and an eventual saturation of throughput of 79.7 IO/sec. Figure 4-6 illustrates the read performance of eight RAIDframe architectures for each of RAIDframe's operating environments. Concentrating, for the moment, on the kernel numbers of Figure 4-6(b), the ten-disk throughput of RAID level 0 reads in RAIDframe's device driver is 658 IO/sec.

With two exceptions, the remainder of the architectures, none of which require redundancy work, perform similarly. The throughput of RAID level 4 is slightly worse. This is because the array has only nine disks that are able to service read requests—one entire disk is dedicated to storing parity. RAID level 1 is slightly better because each block of user data is stored on two independent disks, allowing the read workload to be better balanced among the ten disks in the array. In our implementation of RAID level 1, read actions are dispatched to the copy with the shortest disk queue.

Finally, Figure 4-6 also illustrates the relative performance of RAIDframe's three environments. The user and device driver environments compare favorably with comparable throughput maximums and equivalent response times at given throughputs. However, the simulator predicts higher throughput maximums and lower response times at given throughputs. This is because the simulator does not account for the time required to execute code, data transfer bottlenecks, or limited interrupt-processing capabilities.

Predicting the small-write performance for an array of ten disks is not a straightforward task, but simple models can be used to estimate performance relative to that of a single disk [Patterson88]. As with reads, RAID level 0 write throughput should improve by a factor of ten over that of a single disk. Because RAID level 1, interleaved declustering, and chained declustering each requires two disk accesses (write the two mirror copies of user data) to service a user request, they will achieve a performance of five (10/2) times that of a single disk.

RAID level 4, with its parity contained entirely on a single disk, will achieve one-half the performance of a single disk. This is because each small-write operation performs four disk access—read old data, write new data, read old parity, write new parity—and

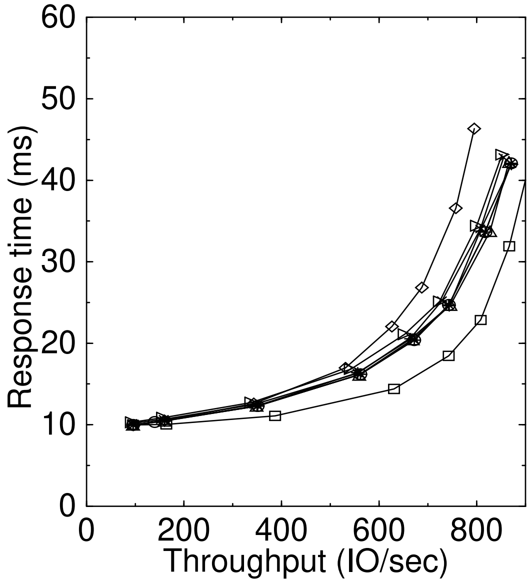


Figure 4-6(a): Simulator

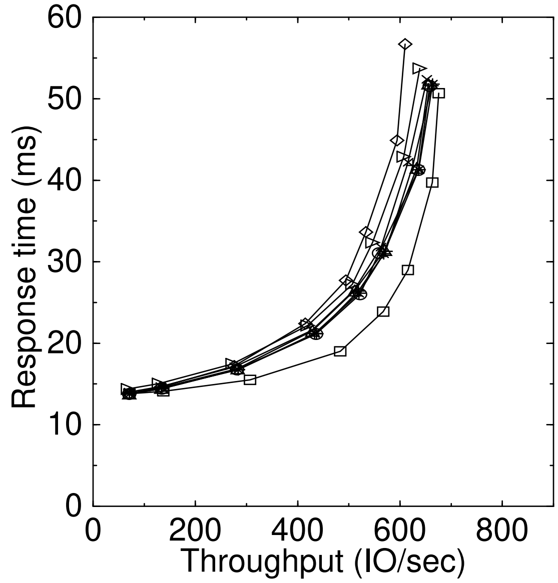


Figure 4-6(b): Kernel

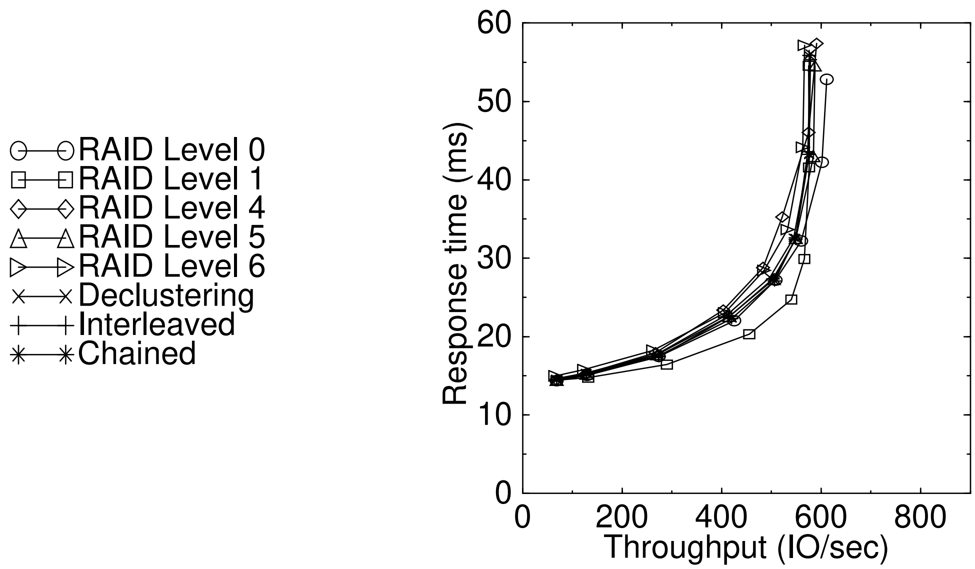


Figure 4-6(c): User

Figure 4-6 Small-read performance of RAIDframe's three environments

Performance was measured using 4 KB aligned workloads. Datapoints represent measurements for 1, 2, 5, 10, 15, 20, 30, and 40 thread workloads. Raw data, including 95% confidence intervals, appears in Appendix C.

performance is ultimately limited by the two disk access to the parity disk for each user request.

RAID level 5 and parity declustering distribute parity over the entire array. Their performance is therefore strictly a function of the number of disk accesses (four) required to complete a small user write to the array. Their expected performance is therefore 2.5 (10/4) times that of a single disk. Similarly, RAID level 6 with its additional two disk accesses which update the Reed-Solomon redundancy unit, should perform at (10/6) times that of a single disk.

Figure 4-7 illustrates the small-write performance we measured for these eight architectures in each of RAIDframe's three environments. The results, with the exception of interleaved and chained declustering, are consistent with our predictions. RAID level 1 reaches a throughput of 345 IO/sec, for an increase of just under 5x of the 78.8 IO/sec performance of a single disk. Like RAID level 1, interleaved and chained declustering both maintain a primary and a secondary mirror copy of each block of user data. However, unlike RAID level 1 which mirrors these two copies at identical disk offsets, these architectures place the copies on different halves of the disks. In write-intensive workloads, the two actuators in a RAID level 1 mirrored pair move in synchronization with an average seek time equal to the time required to seek across one-half of the disk. In interleaved and chained declustering, because the mirrored copies are stored at different disk offsets, the two actuators cover different seek ranges and the average is therefore greater than the minimal average of moving across one-half the total disk.

As expected, RAID level 4's write throughput reaches 44 IO/sec, or about one-half that of a single disk. Parity declustering and RAID level 5, each predicted to reach a write throughput of 2.5 times that of a single disk, achieved throughputs 176 IO/sec (2.23x) and 175 IO/sec (2.22x), respectively. RAID level 6 attained a maximum write throughput of 119 IO/sec (1.51x), just below its predicted value of 1.67x.

Similar to the Figure 4-6, Figure 4-7 shows that the relative write performance of RAIDframe's three environments is consistent between user and kernel environments, and that the simulator is optimistic. In the case of writes, RAIDframe is required to compute (e.g. xor) redundancy information. Because the simulator does not model execution time, the cost of these computations is absent from the final simulator performance, skewing the curves toward higher throughput.

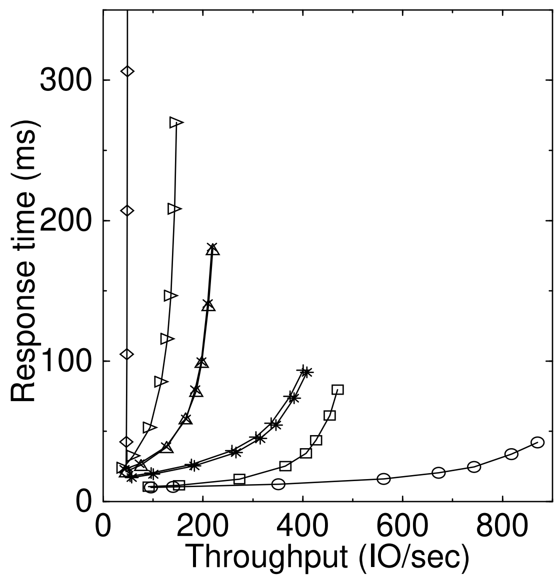


Figure 4-7(a): Simulator

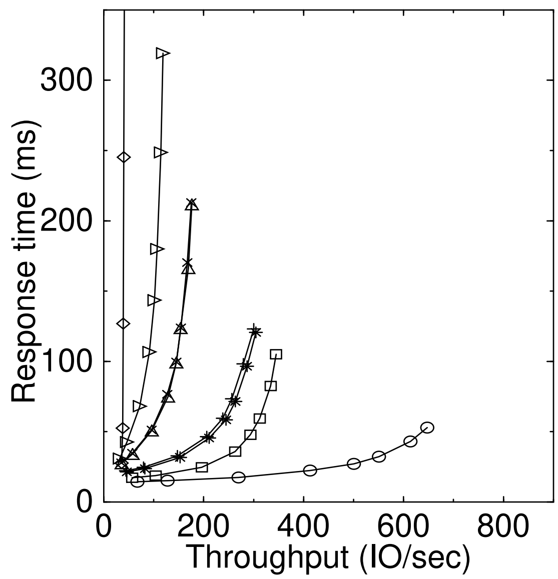


Figure 4-7(b): Kernel

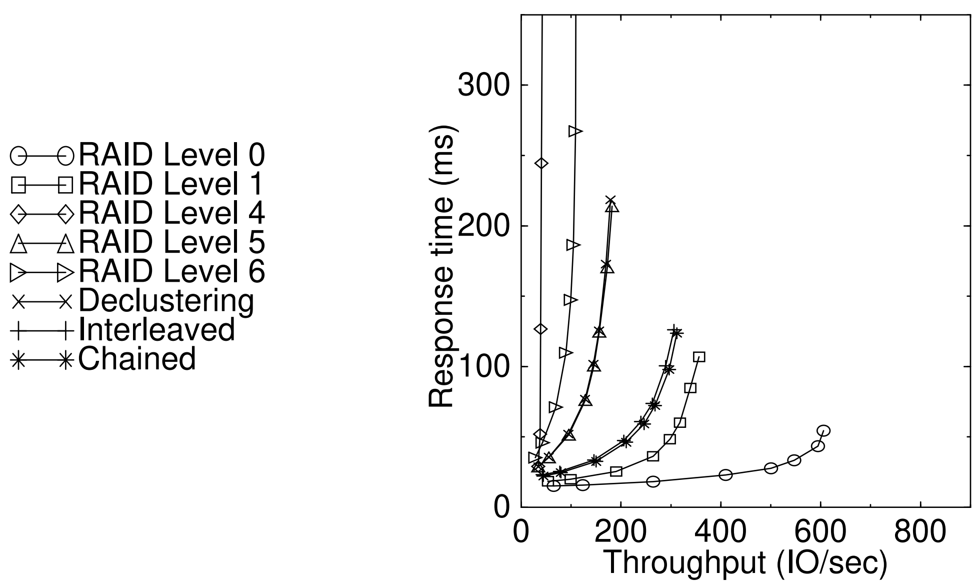


Figure 4-7(c): User

Figure 4-7 Small-write performance of RAIDframe's three environments

Performance was measured using 4 KB aligned workloads. Datapoints represent measurements for 1, 2, 5, 10, 15, 20, 30, and 40 thread workloads. Raw data, including 95% confidence intervals, appears in Appendix C.

4.4 Conclusions

RAIDframe allows array architectures to be implemented with only modest changes to existing code. Architectures implemented in RAIDframe can be evaluated using an event-driven simulator, or against real disks as either a user process or a device driver in a working UNIX kernel.

The cost of layering software in a manner which isolates an infrastructure designed to support the execution of array operations, irrespective of array architectures, is an increased consumption of CPU cycles. However, microprocessor performance has consistently increased at a rate of 35% per year [Patterson96], and I believe the additional cycles consumed by RAIDframe are easily absorbed by simply relying upon faster processors. I argue that trading a few extra CPU cycles consumed by RAIDframe is worth the reductions in complexity that are achieved.

The principal benefit of RAIDframe is its ability to experiment with working implementations of array architectures. By providing researchers with the ability to simplify the task of developing working prototypes, the performance of new array architectures can be evaluated in working systems which deliver workloads in real time. Furthermore, by demonstrating a working prototype, researchers are better prepared to convince implementors that the complexity of their proposals is manageable.

Chapter 5: Roll-Away Error Recovery

Recall, from Chapter 3, the goals of an ideal approach to implementing redundant disk array software: limited code changes to introduce new architectures, simplified error recovery through atomic operation, acceptable overhead, and verifiable correctness. Section 3.3 introduced a programming abstraction for composing redundant disk array operations from atomic actions. This approach simplified the design and execution of redundant disk array algorithms by isolating device-specific execution and error recovery (node failures) from array operation recovery (graph failures). Chapter 4 demonstrated that this programming abstraction can be used in practice. It showed that a modular partitioning of software and the elimination of architecture-specific error recovery results in an implementation which requires only modest code changes to implement new redundant disk array architectures.

This chapter begins by examining the performance consequences and resource consumption of the undo/no-redo error recovery scheme described in Section 3.5. Running the 4 KB random write benchmark used in Chapter 4, Section 5.1 demonstrates that the cost of pre-reading disk sectors before overwriting them degrades small-write performance by as much as 50%. Section 5.2 examines the necessity of full undo logging, concluding that it is possible to eliminate some undo log records, in a general manner, independent of architecture. Proceeding from this conclusion, Section 5.3 introduces *roll-away* error recovery, a two-phase method of error recovery which preserves the simplicity of a general error recovery mechanism but without the performance cost incurred by pre-reading disk sectors prior to overwrites. After examining the performance and correctness consequences of roll-away error recovery, a variety of techniques for manipulating graphs are described. The chapter concludes with a brief discussion of the possibility of extending roll-away error recovery to guarantee atomic crash semantics.

5.1 Full Undo Logging is Expensive

Section 4.3.4 examined the performance of random 4 KB write operations in eight array architectures without regard for error recovery. Using the undo/no-redo scheme described in Section 3.5, error recovery can be completely mechanized by requiring that

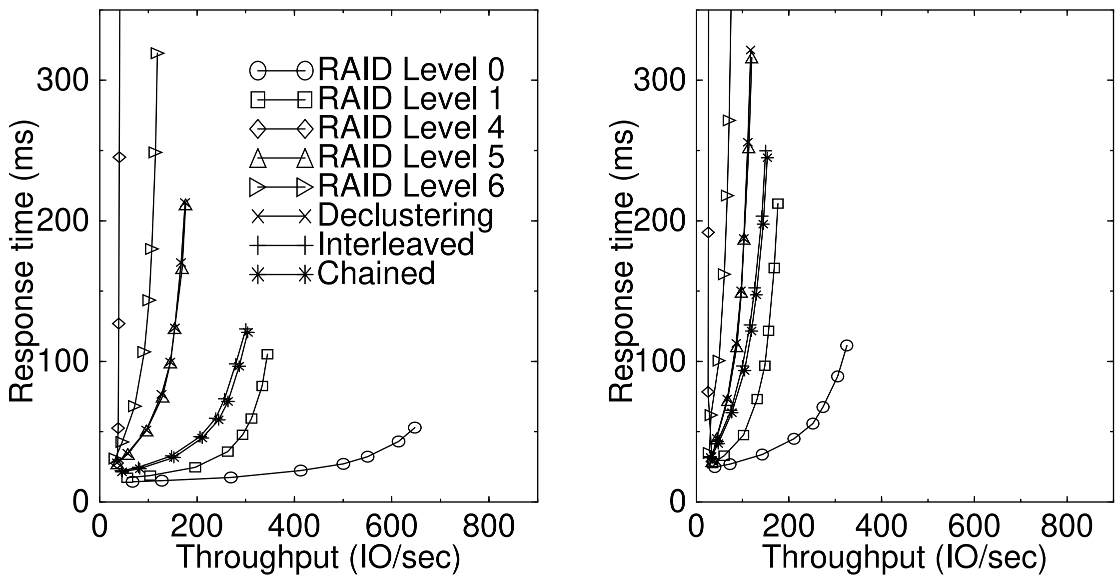


Figure 5-1(a): Forward

Figure 5-1(b): Undo/No-Redo

Figure 5-1 Relative performance of full undo logging

Figure 5-1(b) shows the results of the experiment of Figure 4-7 (random 4 KB writes) repeated with all nodes in the graphs generating undo records. To permit a direct visual comparison, Figure 5-1(a) presents the data from Figure 4-7(b). All data was for RAIDframe operating as a device driver.

---

the visible state changes made by each node can be undone. Before each node is executed, the information necessary to undo its effects are stored in an undo log. If the graph completes successfully (e.g. no node failures), its undo log entries are discarded. If a node fails during the execution of a graph, the underlying execution mechanism uses the undo log entries to fails the graph atomically.

To evaluate the cost, in terms of performance, of creating the undo log entries, I mimicked the maintenance of such a log, making each node generate an undo record entry as specified by Table 3-2 on page 64. Repeating the same workload of random 4 KB writes, the data displayed in Figure 5-1 demonstrates a degradation in small-write performance of 50% for operations in nonredundant and mirroring architectures and 33% for parity-based architectures. The principal source of this degradation is the cost of the creating the undo log records for the Wr nodes which require and extra disk access. The difference in the degree of degradation was a result of the fraction of nodes in the graph which performed Wr actions—the higher the fraction, the greater the degradation.

5.2 Reducing Undo Logging Requirements

Redundant disk array operations maintain codewords which are recorded on durable storage. These operations fall into two general classes: those which retrieve information contained in codewords and those which modify existing codewords. Reading information from a codeword may involve either simply returning the symbols as they appear (e.g. reading information from an array with no faults) or additionally performing a decoding computation to reconstruct missing information (e.g. reading data from an array in which a disk has failed).

Writing information to a redundant disk array will cause existing codewords to change. This change may be in the form of a direct overwrite (modify all symbols of the codeword) or modifications to selected symbols. In either case, this process can be generally described as the tasks of first creating new symbols followed by the task of writing of them to durable storage.

Why not simply rely upon some form of no-undo/redo recovery, which would entirely eliminate the need for undo logging? Such an approach would require that all changes to visible state made by a graph be stored in a *redo log* prior to being applied to the commit of the graph. If the execution of a graph was interrupted, the work which was in-flight at the time of the interruption could be restarted (redone) using the contents of the redo log. Unfortunately, this scheme is not well-suited for our application. We are primarily concerned with node failures, which imply that a faulty component has been detected in the system—simply retrying the same graph (algorithm) will fail because the same failed component will prevent subsequent retries from succeeding. This problem could be overcome by adding a *checkpoint*, a periodic snapshot of the system. If a graph failed, the checkpoint could be restored and the redo log could be played forward up to the instant before the failed graph was initiated. I also dismiss this implementation for the reason that the checkpoint would have to be a snapshot of the entire array. This is necessary because the array can not predict future updates and, in order to undo their effects (roll back by restoring the checkpoint), the checkpoint must contain copies of all data. Furthermore, the checkpoint must be constructed to survive the same faults as the fault model; that is, it must survive disk failures. This implies that we may be using a disk array to simplify the recovery of a disk array.

5.2.1 Limiting the Scope of Rollback

RAIDframe reduced the amount of undo logging required to guarantee error recovery by limiting the scope of the rollback operation. One way this was accomplished was by allocating reader/writer locks outside the creation and execution of flow graphs. When a graph failed, the locks were retained, eliminating the overhead associated with releasing and then reacquiring locks when operations were later retried using a different flow graph.

Another technique RAIDframe employed to reduce the scope of rollback, useful only in write operations, was the allocation and initialization of a buffer which contains the user data to be written to the array. By not discarding user write data when a graph failed, the undo of a write buffer allocation and initialization was eliminated.

5.2.2 Reclassifying Actions From Undoable to Real

The approach presented in Section 3.5 assumed that all actions were undoable and, when a node failed, error recovery consisted of walking backward through the entire graph from the failed node to the source node, undoing the effects of all previously-completed nodes. Recall from Table 3-2 on page 64 that creating the undo records for Wr actions, which required an additional disk access (read previous data) is expensive. If a general strategy for recovery could be created that eliminates the need for undoing these writes (making them real actions), the overall cost of ensuring recoverability should decrease.

Ensuring the recoverability of graphs which contain real actions requires that the real actions not be executed until the actions upon which they depend have reached a state which ensures that the graph will not roll back [Gray93]; that is, because real actions can not be undone, they should not be executed until enough work is completed to ensure that the graph will not need to roll back. Fortunately, because the write of symbols generally occurs at the end of a graph and, assuming that all new symbols are known, the symbols are written independently (the write of one symbol does not depend upon the write of another), it should be possible to eliminate the need to undo symbol writes, in effect making them real actions.

The following section describes a method for inserting a barrier in a graph that isolates the undoable nodes which precede the barrier from the real actions which follow it, Barriers are then inserted into the graphs of Appendix A and the rules for executing graphs with barriers are described.

5.3 Roll-Away Error Recovery

The previous section discussed the possibility of eliminating the undo of nodes by requiring that a graph be split by a barrier with undoable actions to the left and real actions to the right. Figure 5-2 illustrates such a graph in which a special node, Commit, has been inserted to represent the barrier, dividing the graph into two phases. Actions in the Phase-I subgraph either do not make durable state changes or make changes which are easily undone and therefore, undo logging is minimal. Actions in the Phase-II subgraph modify

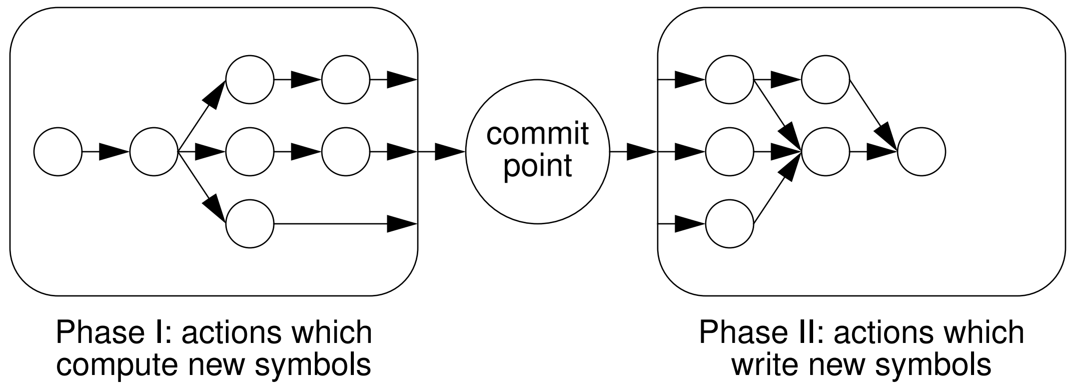


Figure 5-2 Dividing array operations into two phases

Array operations may be divided into two phases, the first which does not modify codeword symbols and the second which does. The two phases are isolated by a commit point which confines all undo logging to the actions in Phase-I and, if atomic crash semantics are required, redo logging to the actions in Phase-II.

---

symbols, but the structure of the graph guarantees that all symbol updates may occur independently, allowing all symbol updates to either complete successfully, or fail due to the presence of a permanent fault.

Recovering from errors in such graphs is performed using an approach that we call *roll-away error recovery*: node failures which occur prior to the barrier force the graph to roll back while node failures which occur after the barrier cause execution to roll forward. The name “roll away” was chosen because node failures force the execution to proceed away from the barrier. Transaction-processing literature commonly refers to this barrier as the *commit point*, because once the barrier is reached, the transaction is committed to move forward to completion rather than backing up. For the remainder of this dissertation, I will refer to the barrier as the “commit point.” The roll-away execution mechanism is described in detail in Section 5.3.4.

5.3.1 Properties of Phase-I Subgraphs

A Phase-I subgraph, and its nodes, have the same properties as those described in Section 3.3. In fact, all of the graphs of Appendix A are valid Phase-I subgraphs. Similarly, the execution rules described in Section 3.5 also apply: if a node fails, the engine walks backward through the graph, undoing the previously completed nodes.

With the graphs split into two phases, it is possible to add an additional constraint that will eliminate the deadlock problem described in Section 3.5.1.3. Recall that in order to avoid deadlock during rollback which results from allocating (undoing deallocation), locks, either a the DAG locking protocol or the two-phase lock protocol were necessary. The DAG locking protocol requires that resources be released in reverse-order from allocation and the two-phase protocol requires that once any lock is released, no more locks are acquired. With the graphs split into two phases, only one of which being undoable, it is very easy to apply the two-phase locking protocol by placing all allocation actions in the Phase-I subgraph and all deallocation actions in the Phase-II subgraph. Locks are only deallocated during rollback of a Phase-I subgraph or during the execution of a Phase-II subgraph. In either case, once deallocation begins, no new locks will be acquired.

5.3.2 Properties of Phase-II Subgraphs

Unlike Phase-I subgraphs, which rely upon undo/no-redo error recovery, Phase-II subgraphs rely upon no-undo/redo recovery. This change in error recovery protocols neccessitates changes in Phase-II structural constraints as well as execution rules. First, when a node fails during the execution of a Phase-II subgraph, execution will *roll-forward*, executing all nodes, regardless of pass/fail result, until execution of the sink node has completed. When the execution of the sink node has completed, all nodes will be in either the fail or pass state—a node in a Phase-II subgraph will never enter the recovery or undone states because nodes in Phase-II subgraphs are never undone.

Second, if a node fails in a Phase-II subgraph, its failure must not impede the execution of the remaining nodes in the subgraph. This can only happen if the node which fails generates a result which is used by its children. Formally, this is represented by a true data dependence in the graph. Therefore, a Phase-II subgraph can contain no true data dependencies.

Third, because execution of Phase-II subgraphs must be unconditional, predicate nodes are not allowed in the subgraph. Conditional execution is disallowed because the execution engine can not predict which branch to take should a predicate node fail, thereby preventing execution of the subgraph from completing.

Finally, as described in the preceding discussion of Phase-I subgraphs, Phase-II subgraphs may not contain actions which allocate resources and must contain all actions which deallocate resources.

5.3.3 Commit Node Determines Direction of Recovery

To keep the engine general, it is not designed to interpret the semantics of a graph as it is executed; the engine simply parses the graph, from source node to sink node. However, if an error occurs and roll-away error recovery is employed, the engine must understand which subgraph is being executed so that it may apply the appropriate recovery protocol: roll backward in the case of a Phase-I subgraph, roll forward in the case of a Phase-II subgraph.

To enable the engine to discern which subgraph it is currently executing, a a node whose do action is Commit, is inserted between the two subgraphs, completely isolating them from one another so that only one graph is being executed (has nodes in the fired state) at any moment in time. The engine executes the node containing the Commit action just like any other—the Commit action simply sends a message to the execution engine (e.g. sets a flag) which indicates that the Phase-I subgraph has successfully completed execution and that execution of the Phase-II subgraph has commenced. The commit node has the special property that it will never fail; therefore, it is only in either the wait, fired, or pass states.

5.3.3.1 Inserting a Commit Node Into a Read Graph

Read graphs are, by definition, strictly Phase-I subgraphs—they do not contain a subgraph which writes new symbols to storage. As a sanity check in our implementation, we inserted a commit node into all graphs, even if there were no Phase-II nodes. This permitted a general post-processing analysis of all completed graphs to ensure that a commit node was present and in the correct state. Therefore, the commit node is always positioned at the end of the read graph as the sink node. This placement is easily rationalized by realizing that it is not possible to successfully complete a read operation unless all actions complete successfully. If some of the actions fail, the operation is generally unable to return all of the requested data without scheduling additional work. Therefore, the graph is allowed to fail atomically and then a different graph is scheduled in its place.

As an example, Figure 5-3 presents the degraded read graph from Figure A-4 on page 146 which has been modified to support roll-away error recovery through the addition of a commit node.

5.3.3.2 Inserting a Commit Node Into a Write Graph

Phase-I graphs are known to be recoverable and it stands to reason that simply appending a commit node to the end of the write graph of Appendix A results in a graph which is recoverable. However, to reduce undo logging, it is desirable to propagate the commit node toward the head (source node) of the graph. Two conditions prevent propagation of the commit node: (1) the presence of predicate nodes and (2) the presence of true data dependencies. As Section 5.3.2 explained, predicate nodes are disallowed in Phase-II subgraphs because all nodes of a Phase-II subgraph are unconditionally executed. True data dependencies between the nodes in a Phase-II subgraph are disallowed because they violate the property that the failure of one node should not affect the execution of other nodes in the graph. If true data dependencies were permitted in Phase-II subgraphs, the failure of a node which produces a result (data) that is used by subsequent nodes would prevent the subsequent nodes from executing.

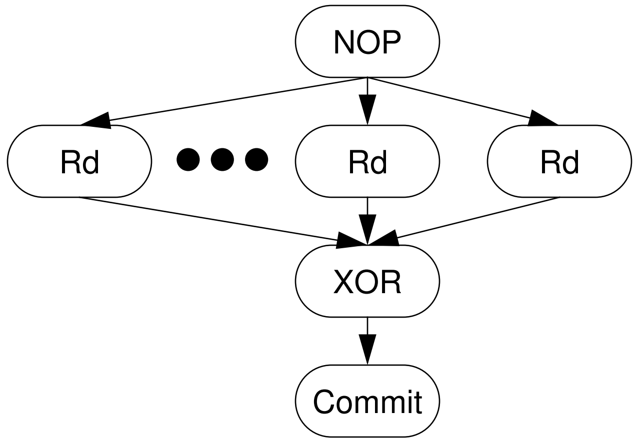


Figure 5-3 Degraded-read graph

A Commit node has been added as to the end of the graph. Remember: reaching the Commit node implies that the graph will complete successfully.

```
explore(node)
/* recursively explore toward the source node */
   for each parent of node
      if no data dependence between parent any child
         explore(parent)
      else
         place commit node between parent and node

main()
/* begin with sink node and work toward source node */
   explore(sink node)
```

Figure 5-4 Algorithm for inserting a commit node into a write graph

This algorithm inserts a commit node into a graph by beginning at the sink node and then recursively examining each branch, looking for data dependencies between two nodes. When a dependence is discovered, the commit node is inserted between these two nodes. As written, this algorithm does not look for predicate nodes. This check can easily be added to the routine `explore()`.

Figure 5-4 suggests an algorithm for inserting a commit node into a write graph. This algorithm looks for data dependencies, beginning with the sink node of the graph and working towards the source node. Each branch is explored until a node is discovered that has a data dependence to one or more of its children. The commit node is inserted between this node the node being examined. Note that this algorithm does not account for predicate nodes, which are not allowed in Phase-II subgraphs. The algorithm is easily extended to prevent predicate nodes from entering the Phase-II subgraph by looking for both data dependencies and predicate nodes as the branches are explored. If either is found, the commit node is inserted.

Appendix C contains the code for a program based upon this algorithm. This program creates a RAID level 5 small-write graph, which appends a commit node to the sink node of a graph, and then attempts to propagate the commit node toward the source node of the graph, in effect, moving nodes from the Phase-I subgraph to the Phase-II subgraph. This algorithm was used to transform the graphs of Appendix A into the graphs of Appendix B.

To demonstrate this algorithm, I illustrate the series of steps required to insert a commit node into the small-write graph originally presented as Figure 3-3 on page 55 and duplicated here as Figure 5-5(a). The process of inserting the commit node begins in Figure 5-5(b) in which the parents of the sink node (Unlock) are examined, beginning with the Wr node. The algorithm recursively walks up the graph until it reaches the Rd node which has a data dependence to one of its children (XOR). The Commit node is inserted between the Rd and Wr nodes. This process is repeated in Figure 5-6 with the second parent of the sink node, MemD, being examined. Again, the algorithm recursively walks up the graph until this time the XOR node is reached. Because the XOR node has a data dependence to its child (Wr), the Commit node is inserted between XOR and Wr. Because the sink node (Unlock) has no other parents, the process is complete.

Because the structure of the graph has changed, it is possible that some arcs in the graph may now have become redundant as defined in Section 3.3.4. Figure 5-7(a) illustrates the removal of the Rd-Commit, which is now redundant. Next, because the nodes in the Phase-II subgraph (descendants of the commit node) are not undoable actions, the strict serial ordering of resource deallocation actions, previously necessary to avoid deadlock during rollback, is no longer required. In this example, the MemD and Unlock nodes of Figure 5-7(b) are now permitted to execute concurrently by removing the control dependence (MemD-Unlock) which preserved this serial ordering. Finally, a NOP node is added to maintain the property that the graph has a single sink node.

5.3.4 Adjusting Graph Execution Rules

Node failures which occur prior to the execution of the commit node are handled by the execution engine in the same manner described in Section 3.5.4—the engine walks backward from the point of failure, executing the undo actions in the graph until the source node is reached and the graph has failed atomically. Using the small-write graph of Figure 5-7(b), the process of recovering from a node failure prior to the commit node is illustrated in Figure 5-8.

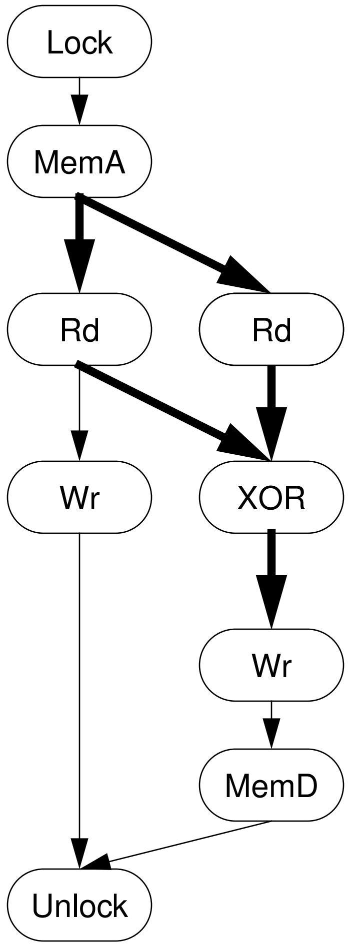


Figure 5-5(a): Initial graph

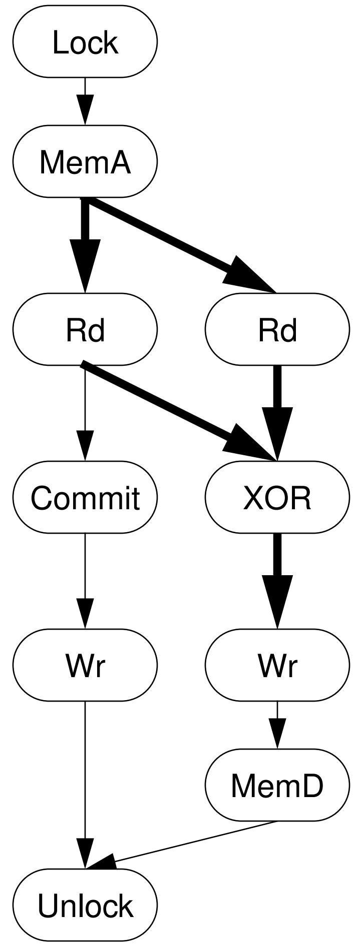


Figure 5-5(b): Step 1: explore left-most parents of Unlock node

Figure 5-5 Inserting a commit node into a RAID level 4/5 small-write graph

Figure 5-5(a) illustrates a small graph that we wish to add a commit node to. Beginning with the Unlock *node, the parents of each node are recursively examined until a data dependence (represented by the bold arrows) is encountered. At this point, the* Commit *node is inserted between the nodes that share the dependence. In Figure 5-5(b), the left-most branch has been walked until the left-most* Rd *node was reached. Because this node has a true data dependence to one of its children (*Rd-XOR*), the walk stops and the* Commit *node is inserted between the* Rd *and* Wr *nodes. This process continues in Figure 5-6.*

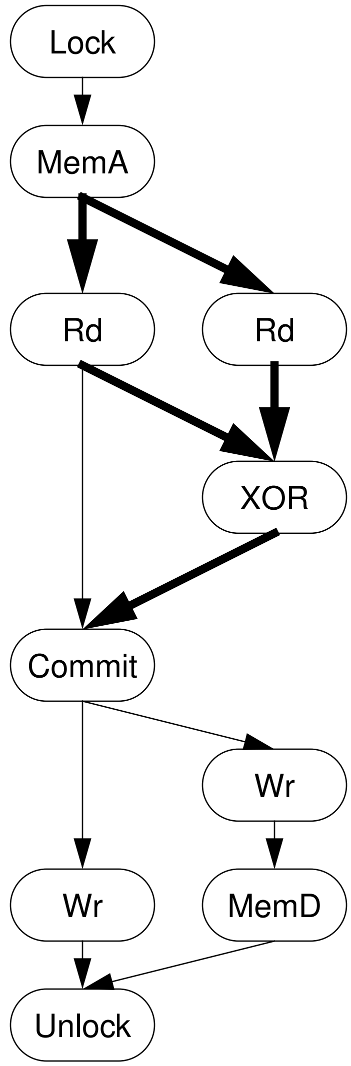


Figure 5-6 Inserting a commit node into a RAID level 4/5 small-write graph

Continuing from Figure 5-5(b), the right-most branch is walked until the XOR *node is reached. Because this node has a true data dependence between itself and its child (*Wr*), the walk stops and the* Commit *node is inserted between the* XOR *and* Wr *nodes. At this point, all branches leading to the sink node (*Unlock*) have been explored, and the insertion is complete. The resulting graph is shown above.*

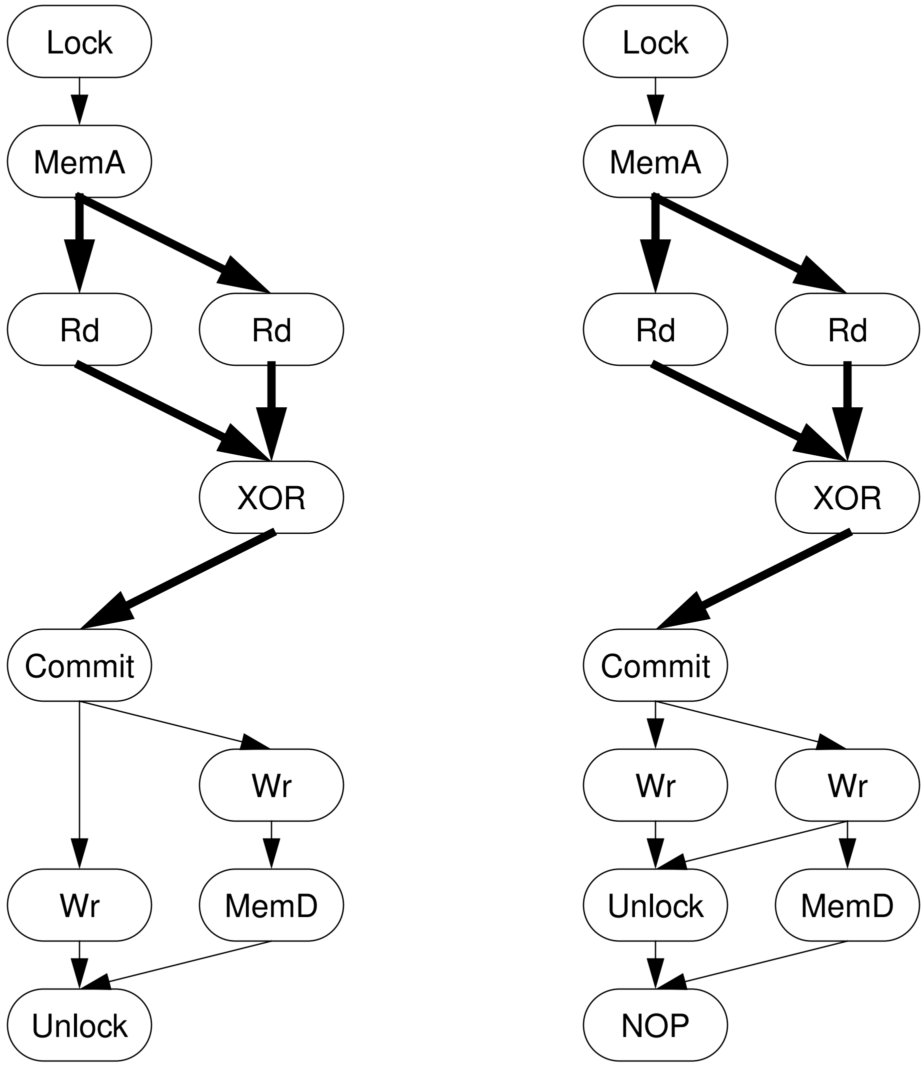


Figure 5-7(a): Step 4: eliminate a redundant arc

Figure 5-7(b): Step 5: eliminate an unnecessary control dependence

Figure 5-7 Graph optimization

After completing insertion of the commit node, the graph of Figure 5-6, redundant arcs are eliminated. Step 4 in this example eliminates the Rd-Commit *arc because it duplicates the* Rd-XOR-Commit *path. Finally, because resource deallocation actions in the Phase-II subgraph do not require the serial ordering necessary when they were members of a Phase-I subgraph, the* Unlock *and* MemD *nodes are allowed to occur in parallel. To guarantee a single sink node, a* NOP *node was added.*

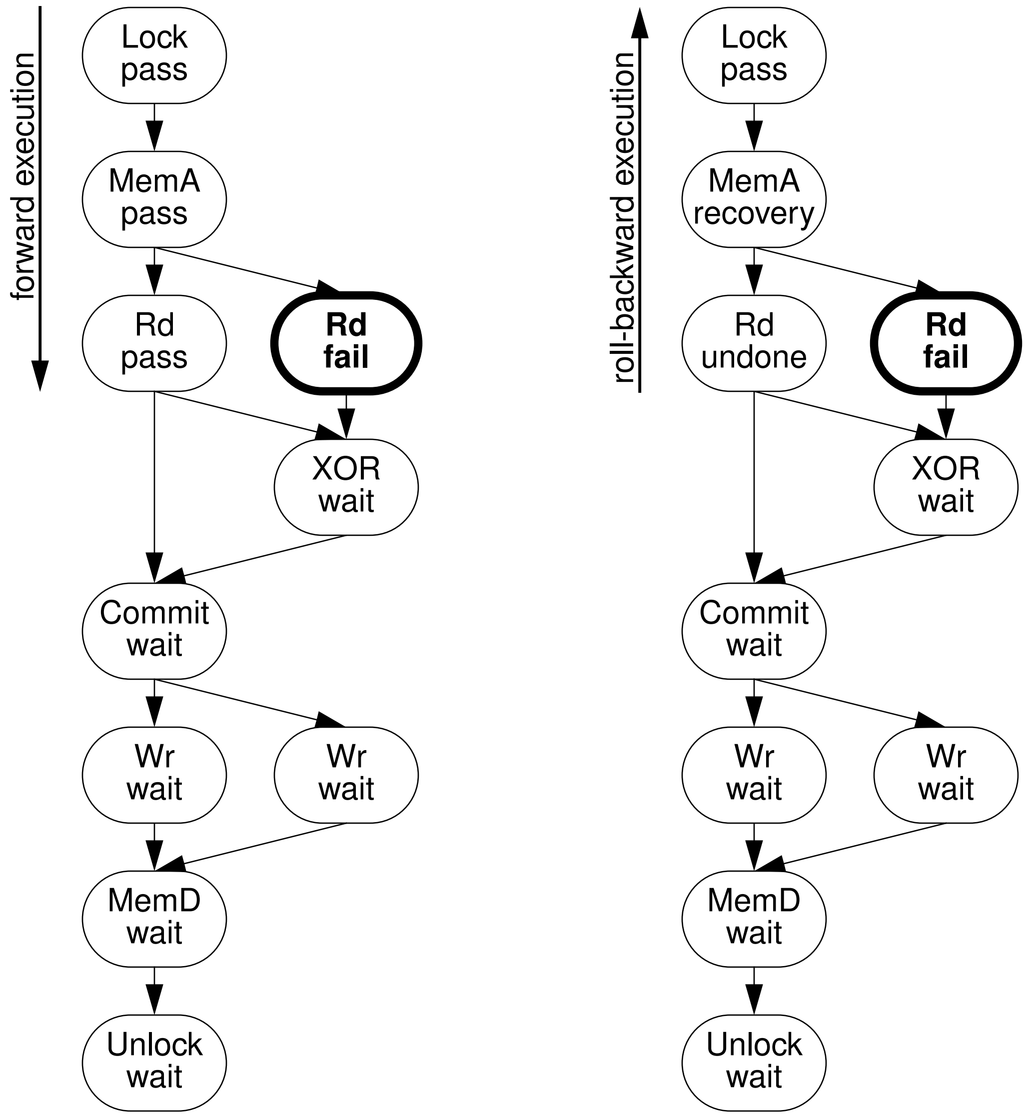


Figure 5-8 Recovering from errors prior to the commit point

The failure of the Rd *node, indicated in bold, causes forward execution to halt. Once the* Wr *node which was in the* fired *state completes, roll-backward execution begins, undoing the previously completed actions by executing the corresponding undo functions from each node. In the illustration on the right, the* MemA *node is in the* recovery *state which implies that its undo function is currently being executed. When roll-backward execution completes, the graph has failed atomically, all nodes in the Phase-I subgraph are in either the* undone *or* fail *state and all nodes in the Phase-I subgraph are in the* wait *state.*

Once the commit node is reached, execution of the graph will continue until the sink node is reached, regardless of whether a node subsequently fails or not. Because the undo information required to ensure the recoverability of the Phase-I subgraph is no longer necessary, the undo log records for this graph can be eliminated from the undo log when the commit node is reached.

Section 3.5.3 specified that during the process of forward execution, a node is ready for execution if all of its parents are in some combination the pass or skip states. Because forward execution of Phase-II subgraphs continues, regardless of node failures, this rule is changed to enable a node for execution when its parents are in some combination of the pass or fail states.

An example of a node failure which occurs after the commit point is reached is presented in Figure 5-9. In this example, execution continues forward, with all remaining nodes successfully completing. When the execution of a graph reaches the sink node, it is declared to be successful, regardless of node failures which may have occurred in Phase-II. This is possible because all symbols of the codeword, except those residing on failed devices, were written correctly. The completion of the graph and the failure of a device can be viewed as two independent events whose ordering is arbitrary.

5.3.5 Fault Model

The fault model presented in Section 3.6 still holds; however, the defining properties of nodes must be altered for nodes that appear in Phase-II subgraphs. Device faults continue to be observed as node failures. Node failures are survived in one of two ways: first, if the node fails prior to commit, the graph is rolled back as before, and the visible state changes made by the graph are undone. If a node fails after commit has been reached, execution continues forward to the end, and the system is left in a state in which all symbols have either been updated correctly or marked as "failed." The remainder of this section describes the changes to device actions (nodes) that are necessary to preserve the fault model.

Failures that interrupt the execution of a graph can still be tolerated through the use of durable undo and redo logs. As before, the undo log contains the information necessary to undo the effects of a failed graph. Any graph whose execution is interrupted prior to commit is undone at restart. However, this is not the case for nodes that follow commit. This is because these nodes are not undoable. Therefore, once commit is reached, enough information to redo each node that follows the commit is entered into a redo log. Because we know that the graph can reach completion once commit occurs, playing the redo log after restart will cause the graph to reach completion.

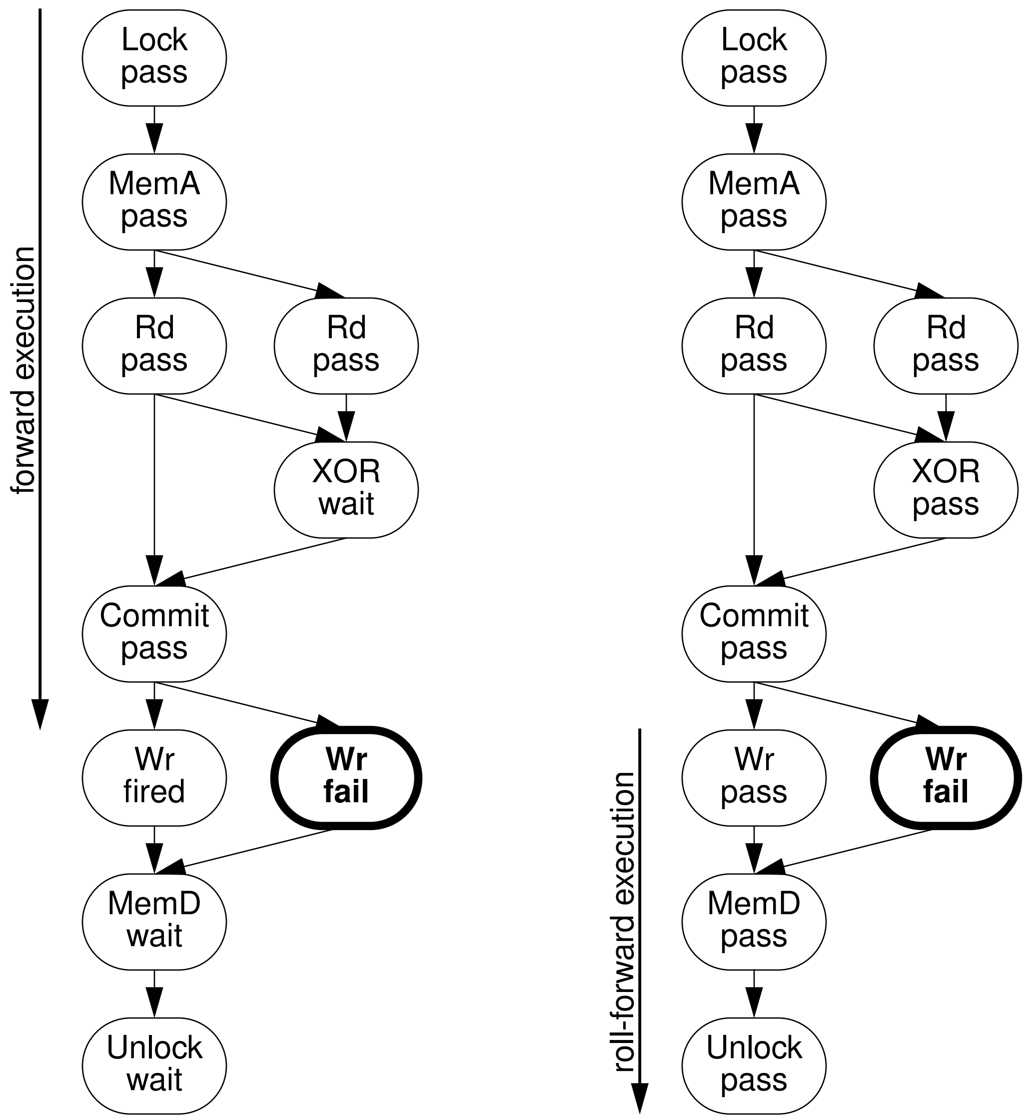


Figure 5-9 Recovering from errors following the commit point

The failure of the Wr *node, indicated in bold, causes forward execution to halt. Once the* Wr *node which was in the* fired *state completes, roll-forward execution begins, continuing until the sink node (*Unlock*) is executed and completed. When execution completes, all nodes in the Phase-I subgraph are in either the* pass *or* skip *state and all nodes in the Phase-II subgraph are in either the either the* pass *or* fail *state. At this point, the system appears as if the graph completed without error followed by a failure of a disk (which corresponds to the* Wr *failure).*

5.3.5.1 Adjusting Node Properties

Recall, from Section 3.2.1, that nodal actions were defined to atomically update all symbols that it operated upon—if the node failed, these symbols were left either unchanged or marked as failed. Additionally, actions were required to maintain the independence of the faults that they encounter—the failure of an action to properly operate (read, write, or compute) upon one symbol should not necessarily fail the other symbols. This atomic execution of actions greatly simplified the process of backward recovery; actions which failed left no state to clean up and actions which had previously completed were undone using the contents of the undo log.

Because roll-away error recovery does not rely solely upon backward recovery, the rules which specify how a node should fail must be reexamined. Specifically, instead of backing out successful changes to symbols, actions that fail during the execution of a Phase-II subgraph must complete all possible state changes. Therefore, unlike nodes in a Phase-I subgraph, nodes that fail in a Phase-II subgraph must leave all symbols either *changed*, or marked as failed. For example, if a Phase-II action that writes symbols to multiple sectors in a single disk encounters the failure of a single sector, it must complete the writes to surviving sectors. By doing this, the roll-forward recovery procedure of Phase-II subgraphs is preserved.

5.4 Performance Evaluation

Similar to the analysis in Section 5.1, which compared the response times of various architectures at given throughputs for the graphs of Appendix A with and without undo logging, I repeated the random 4 KB write benchmark of the eight array architectures, but this time using the graphs of Appendix B. Read operations were not retested because the graphs are identical to those used in Appendix A, with the exception that the sink node (NOP) was replaced by a commit node. Figure 5-10 presents the results of this experiment which are that roll-away error recovery graphs exhibited the same performance as the graphs which were executed with no undo logging. The actual data values, with 95% confidence intervals, are presented in Table C-8 on page 222. The average difference in response time for a given throughput was negligible (less than ±1%) and the confidence intervals were equally tight (less than ±4% of the mean).

This strong correlation was not surprising: the Phase-I subgraphs were composed entirely of Rd, Q, and XOR nodes, none of which required undo records. Therefore, the only potential impact on performance was the elimination of some concurrency within the graph, specifically at the point where the commit node was inserted.

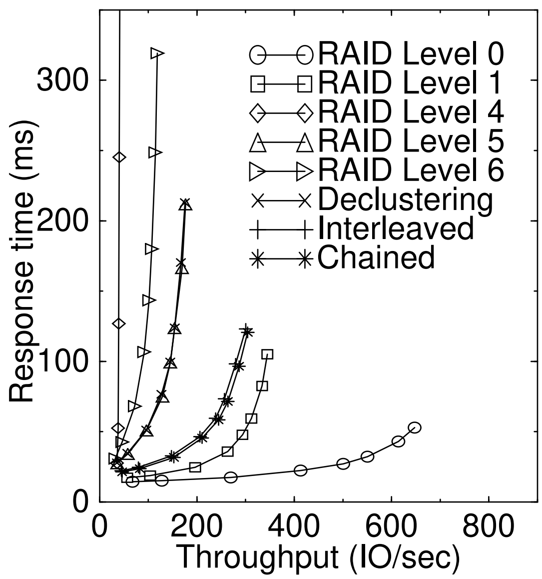


Figure 5-10(a): Forward

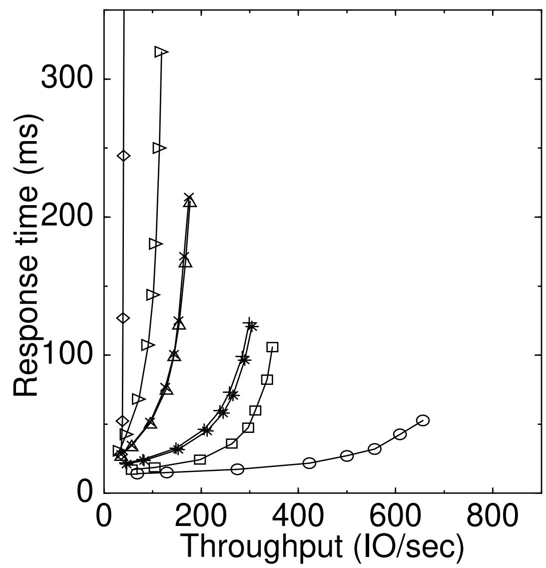


Figure 5-10(a): Roll-Away

Figure 5-10 Relative performance of roll-away recovery

Figure 5-10(b) shows the results of the experiment of Figure 4-7 (random 4 KB writes) repeated with all nodes in the graphs generating undo records. To permit a direct visual comparison, Figure 5-10(a) presents the data from Figure 4-7(b). All data was for RAIDframe operating as a device driver.

5.5 Correctness Testing

Roll-away error recovery is the sole method of recovery currently supported by RAIDframe [Courtright96c]. Extensive testing of this error recovery protocol was performed by injecting disk faults into an array, causing each graph type to either roll forward or roll backward, and then verifying that the array continued to operate in a consistent manner. The majority of our testing was performed with RAIDframe installed as a user process, running against real disks. Installing RAIDframe as a user process simplified the processes of fault injection and validation for two reasons: first, the simulator does not actually move data and therefore data corruption bugs are difficult to detect. Second, injecting faults and monitoring their effects in the kernel can be cumbersome. Because the same code is used when RAIDframe is installed as a simulator or device driver, the results obtained through user-level testing are expected to apply equally well to all three environments.

I employed two of RAIDframe's utilities, `LoopTest` and `ReconTest`, to perform the testing. Each of these tests begins by initializing the array's data and redundancy blocks and then proceeds by sending a series of random write requests to the array. Each write request is followed by a read request to confirm that the data was properly written to the array as well as a verification of the integrity of the codeword (i.e. that proper redundancy information was written). The number of write requests, as well as number of write requests which may execute concurrently, may be specified independently. The `ReconTest` has the additional property that at the end of the test, all data and redundancy information is read from the entire address space of the array, verifying not only that the locations which were to be overwritten contained the appropriate data, but also verified that no side effects were introduced (e.g. extents which were not overwritten remained unchanged). Finally, both utilities provided the ability to asynchronously fail a disk during the test and begin reconstruction without bringing the array off-line.

Initial testing was performed to verify that the graph correctly parsed the graphs. Testing began by modifying the engine code to first fail nodes in the Phase-I subgraph, and later, the Phase-II subgraph. This permitted validation that the two execution protocols, roll-backward and roll-forward, were operational and correctly parsing the graph. Using the small-write graph, I first failed Rd and XOR nodes (in different experiments) which forced rollback of the Phase-I subgraph. Next, I failed a Wr node to verify that the engine would continue to walk forward through a graph, given the failure of a Phase-II node. In addition to this testing, RAIDframe contains routines which, when enabled, analyze every graph that completes, performing sanity checks such as verifying that all nodes are in valid states (e.g. all pass/skip). These routines were enabled throughout the testing procedure.

Once this engine was known to be parsing the graphs correctly the disk failure mechanism provided by the utilities was used to inject asynchronous faults. Testing with 10,000 requests per thread, 30 threads, asynchronously failing a disk, and then reconstructing it, revealed no data corruption for random request sizes and alignments which exercised all of the graphs for RAID levels 1, 4, 5 and parity declustering[1] libraries. Testing was repeated until each graph was forced into both roll-back and roll-forward recovery scenarios.

Finally, limited testing was performed with RAIDframe installed as a device driver in a Digital UNIX™ 3.2c kernel. First, using the UNIX `dd` utility, random I/Os were sent to the RAIDframe device and a drive was physically removed while the array was servicing requests. Information describing the type of graphs which failed at the instant the drive was removed, and the action taken by the execution engine, and the final state of all nodes in the graph was displayed on the console for verification. Additionally, the utilities which automatically verify that all nodes of completed (pass or fail) graphs were in valid states

1. Development of interleaved and chained declustering, as well as RAID level 6, was not complete at the time of this testing. Because RAID level 0 is not fault tolerant, it could not be used in testing.

were enabled. Testing was limited to about a dozen attempts because Digital UNIX was not able to consistently tolerate the physical removal of SCSI devices, something it was not designed to cope with. On average, one half of the removals resulted in a condition known as a “SCSI bus freeze” in which all subsequent accesses to any device on the SCSI bus that contained the drive that was removed are disallowed. The only remedy for restoring access to these devices is to reboot the machine.

5.6 Summary

Roll-away error recovery, a novel two-phase error recovery scheme, automates the problem of recovering from failures of actions which compose redundant disk array operations. Roll-away error recovery has been shown to enjoy the performance advantages of forward error recovery and the simplicity of backward error recovery schemes. Building upon the graph-based representation defined in Chapter 3, roll-away error recovery fully-mechanizes the execution of array algorithms. Like backward error recovery, roll-away error recovery eliminates the need for architecture-specific error-recovery code, simplifying the processes of implementation, verification, and extension. However, unlike backward error recovery, roll-away error recovery is able to do this without a significant (30-50%) performance degradation; in fact, roll-away error recovery performance is identical to that of forward error recovery schemes which introduce no logging or other overhead during error-free processing.

This improvement in performance over strict backward error recovery schemes is a result of the realization that not all actions must be undoable; specifically, by removing the need to undo actions such as a disk write, significant overhead is eliminated from the logging process. Expanding the structural constraints of flow graphs, summarized here in Table 5-1, a commit node is inserted into each graph. Extending the execution invariants, summarized here in Table 5-2, flow graphs are executed in a manner that guarantees, in the absence of a crash, atomic operation. If atomic crash semantics are required, the undo log, already in use, must be made durable. Also, a durable redo log must be created. However, because the redo records are easily created from state which currently exists in the array controller, (disk accesses are not necessary), performance is not significantly degraded.

Roll-away error recovery has been implemented in RAIDframe and was demonstrated to be correct. A general method for inserting a commit node into a flow graph was developed and applied to each of the twenty one graphs of Appendix A. The resulting twenty one graphs were introduced as Appendix B.

Table 5-1 Structural constraints of graphs with commit nodes

Graph Segment	Constraint
Global	Single commit node in each graph.
	Phase-I and Phase-II Subgraphs isolated by commit node.
	Single sink node.
	Single source node.
	No cycles.
Phase-I	Single source node.
	Single sink node (commit node).
	All nodes are atomic and undoable.
Phase-II	Single source node (commit node).
	Single sink node.
	No predicate nodes.
	No true data dependencies.
	Nodes preserve the independence of symbol operations. If a node which operates upon multiple symbols detects a failure when operating upon a subset of those symbols, the remaining symbol operations are completed.

Table 5-2 Execution invariants of graphs with commit nodes

Execution Protocol	Invariants
Normal (error free)	Each nodes is initially in the wait state.
	Execution begins with source node.
	A node may be executed if all parents are in the pass or skip states.
	Execution terminates at the sink node. Each node is in either the pass or skip state.
Roll Backward	Commit node is in wait state.
	All nodes in the Phase-II subgraph in the wait state.
	Execution terminates at the source node. At completion, each node in the Phase-I subgraph is in the wait, undone, or fail state.
Roll Forward	Commit node is in the pass state.
	Each node in the Phase-I subgraph is in either the pass, skip, or wait state.
	Execution terminates at the sink node. At completion, each node in the Phase-II subgraph is in either the pass or fail state.

Chapter 6: Conclusions and Recommendations

In Chapter 1 of this dissertation, I stated my belief that the contemporary methods utilized in the implementation of redundant disk array software, which rely upon a case-by-case treatment of errors, are inadequate in that they unnecessarily complicate the processes of coding, verification, and extension. In addition to demonstrating the validity of this belief, this dissertation made four principal contributions which directly support my thesis:

1. A new programming abstraction to promote code reuse.

2. A reduction of architecture-specific error recovery code by isolating action-specific recovery from algorithm-specific recovery.

3. A mechanism for execution disk array algorithms that includes the recovery from errors detected during execution.

4. A significant reduction, in comparison to a naive mechanical scheme, in the logging penalties required to mechanize error recovery.

5. A programming abstraction which is amenable to automated correctness verification.

6.1 Validating the Problem

The work described in this dissertation was motivated by the fundamental belief that the ad hoc techniques currently employed in the implementation of redundant disk array software result in long development cycles due to the limited the ability of vendors to quickly implement basic (e.g. RAID level 5) array architectures, and, once implemented, validate their correctness.

Chapter 2 provided a compendium of background information, including a review of a variety of disk array architectures. The necessity of error recovery in redundant disk arrays, as well as several methods for coping with errors, was described. This review also demonstrated that a wide variety of disk array algorithms can be created from a relatively small set of actions, such as *disk read* and *XOR*, suggesting that it should be possible to quickly, and easily, extend a working disk array to support new algorithms and architectures.

Chapter 3 introduced a programming abstraction built upon the premise that by encapsulating the actions which compose disk array algorithms with a pass/fail interface, recovery from action-specific errors can be isolated from array-level recovery. A graphical method of representing the sequencing of these actions was then introduced in which the actions are represented nodes of the graph and the dependencies between the nodes (control or data) are represented by directed arcs. The merits of using forward recovery in the execution of these graphs was then examined and dismissed as unreasonable for several reasons. First, correctly designing a graph to ensure that it is recoverable is not obvious—subtle execution and failure timings which interrupt the execution of a graph can leave the system in a state from which recovery is impossible. Second, such a scheme requires the creation of a unique recovery procedure for each distinct error scenario. Because recovery is not generalized, the execution of these graphs is not easily mechanized. Third, these recovery procedures must be individually verified, something which is not easily done at design time. Finally, industry sources report that a significant fraction (over 50%) of the code that is written for systems based upon this approach is devoted to error recovery [Friedman96].

If the execution of a graph, including the recovery from failed nodes, could be generalized, the problem of designing and verifying algorithm-specific procedures would be eliminated. Given the programming abstraction developed in Chapter 3, which isolates the action-specific error recovery from the execution of array algorithms, mechanization should be possible, and therefore, the complexity of case-by-case error recovery procedures are unnecessary.

6.2 Eliminating Architecture-Specific Error Recovery Code

The final contribution of Chapter 3 was the introduction of a mechanized approach, borrowed from those used in transaction-processing systems, for executing disk array algorithms represented as graphs. The approach required that the actions contained in the nodes of a graph be atomic and undoable. As a graph executed, each node recorded enough information so that, if the graph failed, its effects could be undone. This resulted in the atomic failure of a graph.

Chapter 4 introduced *RAIDframe*, a prototyping framework built upon this graphical programming abstraction. Implementations of eight disk array architectures revealed that RAIDframe's modular construction permitted code reuse to be consistently above 90%. Additionally, all changes were localized, further simplifying the process of extending existing code. Analysis indicated that RAIDframe's performance was consistent with that of a nonredundant hand-crafted striping driver; however, RAIDframe's CPU consumption was 60% higher than that of the hand-crafted driver. Nevertheless, the response time versus throughput characteristics of the eight architectures implemented in RAIDframe performed as expected.

6.2.1 Reducing Logging Penalties

Using RAIDframe, Chapter 5 examined the cost, in terms of performance, of guaranteeing that all nodes in a graph are undoable. Because undoing disk writes is expensive, requiring the pre-read of disk sectors which are to be overwritten, the maximum small-write throughputs of the eight array architectures were degraded by 30-50%.

RAIDframe already limited the scope of the rollback by eliminating allocating resources (e.g. locks and buffers) and acquiring the data to be written to the array outside the graphs, eliminating the need for them to be undone. Chapter 5 introduced roll-away error recovery, a method for further reducing the need to log undo information. This was accomplished by eliminating the need to undo nodes, such as disk write, which occurred at the end of a graph. A commit point was inserted in each write graph, separating the actions that write information to disk from the actions that created the information which was to be written. If a node fails prior to commit, its effects are undone as before. If a node failed after commit, the graph rolls forward to completion. The rules for inserting the commit point, which enable this roll-forward approach, were described and the graphs (and nodes) which were used in the previous experiments were modified accordingly. The result was that the performance was that the logging overhead was eliminated.

6.2.2 Enabling Correctness Validation

Finally, because the programming abstraction introduced in Chapter 3 models array algorithms as state machines, techniques such as model checking can be used to verify that array algorithms are recoverable [Vaziri96]. This approach exercises all possible interleavings of a state machine to ensure that the correctness invariants are not violated. Chapter 5 provided a list of invariants which govern the creation and execution of these machines.

6.3 Practicality

The previous section summarized the principal contributions of this dissertation. This section summarizes the practicality of the approach outlined in this dissertation and put to use in RAIDframe:

- *design of a graph is straightforward*—Appendix A described the design of twenty-one flow graphs. A commit point was inserted into each of these graphs which were then presented in Appendix B.

- *many graphs can be generated from a small set of primitives*—The twenty-two graphs of Appendix B were created from nine actions: NOP, Rd, Wr, XOR, Commit, Q, Q, LogOvr, and LogUpd. Through the addition of an XOR-based encoding, an additional six graphs were possible by simply replacing the Q/Q nodes with E/E nodes.

- *roll-away error recovery does not weaken the semantics of the storage system*—The traditional semantics for disk storage are that: completed write operations are durable, write operations which fail leave the area being overwritten in a an unknown state, and write operations which fail do not affect areas not being overwritten. Roll-away recovery not only preserves these semantics but also, as the next section will describe, enables them to be strengthened to atomically survive power failures and system crashes.

- *deadlock avoidance is simplified*—by ensuring that all deallocation actions occur after the barrier, the problem of avoiding deadlock during rollback (reallocating previously-released resources) is eliminated.

6.4 Suggestions for Future Work

As explained in Chapter 1, this dissertation broadly examined problems in redundant disk array software. As a result, we discovered many problems which have yet to be examined in adequate detail. I present a list of these problems here in the hopes that other researchers in this field will find them worthy of examination.

- *graph compilation and optimization*—A cursory examination of techniques for optimizing flow graphs was presented in Section 3.3.4. I see three opportunities for interesting work in this area: first, graphs could pass through an optimizer which performs the function-preserving transformations described in this dissertation. Coupling this optimizer with a cache and appropriate deferment strategies, it seems likely that this optimizer could, at the very least, eliminate redundant load/stores of shared information (e.g. parity) and thereby improve performance.

Second, basic-block optimizations operate using only the structural constraints (dependencies) present within a graph—no understanding of the semantics of the graph are used to optimize the algorithms found in the graphs. For example, if a number of graphs which use the small-write algorithm to update independent blocks in a common parity stripe are merged, it is possible that the separate algorithms could be replaced by a single large-write operation. This type of knowledge is currently only available during the process of mapping user requests into flow graphs.

Third, instead of specifying array operations as flow graphs, it would be interesting to see if a an abstraction, similar to a programming language, could be developed. This may simplify the process of requesting storage access by providing an interface which is more general, deferring details of implementation to a compiler which understand the details of the current implementation and is able to transform the high-level request into a flow graph.

Fourth, because disk arrays are used to improve performance, it would be interesting to see work on a tool which could assist the process of designing and validating the ability of graphs to achieve performance goals. This could be used in conjunction with the compiler mentioned above, to produce graphs which don't simply guarantee the correct state transformations, but also guarantee properties such as maximum buffer consumption per IO.

- *model checking*—This is really work in progress [Vaziri96]. I repeat this item in this section because I hope work will continue in this area, and that collaboration with industry partners will develop in the near future. In following the work of Nancy Lynch, Mandana Vaziri and Jeannette Wing, I have seen interesting results: through the use of their models, they were able to predict write holes in the RAID level 6 small-write and RAID level 5 reconstruct-write graphs with two graphs, one of which we did not catch.

  One obstacle that I have noticed is that because model checking is not integrated into the array development framework (in our case, RAIDframe), the duplicate specifications of flow graphs necessary for each environment lead to holes in the verification process. For instance, separate, but hopefully identical, specifications were created for RAIDframe, model checking, and this dissertation. During the process of working on each of these three projects, errors, such as the inadvertent omission of an arc, were introduced which caused these specifications to become inconsistent. Despite the fact that the inconsistencies were often the result of typographical errors, their presence often impeded progress and required many hours of careful checking (by hand) to ensure that the specifications remained consistent. Therefore, it would be ideal if there was some way to represent the flow graphs in a manner which eliminated the redundant specifications. For example, the code which specifies a graph used in RAIDframe is the same code used to specify a graph in the model checking software, and, can be submitted to a printing routine which translates the code into a visual representation. This common specification could lead to an integration of model checking into a disk array prototyping framework, such as RAIDframe, in which once a graph were entered into the framework, it would be automatically validated prior to including it in the graph-creation library.

- *node and infrastructure design*—This dissertation largely ignored the design of nodes, focussing instead upon the design and execution of array algorithms. However, this does not mean that the design of nodes is trivial. In fact, after developing RAIDframe, the pacing item in our development of new array architectures became the creation of nodes, such as parity log append, and the infrastructure routines (parity log reintegration) for supporting them. I have no doubt that simplifying the implementation of such functions would greatly simplify the process of array prototyping.

- *nested execution*—The mechanism for executing graphs described in this dissertation assumed that the nodes from which graphs were constructed were both atomic and, in some cases, undoable. This mechanism guaranteed that graphs execute atomically in the face of node failures. Given the complexity of node design and infrastructure routines, I believe that there may be merit in decomposing these functions into a sequence of functions instead of designing a single massive function.

- *distributed execution*—This study presumed the existence of a centralized controller which has total knowledge of a flow graph's execution state. Section 3.5.5 proposed a method of distributing the execution of a flow graph across multiple nodes connected with a message-passing mechanism. Implementing and evaluating the generality of roll-away error recovery in a distributed environment would not only be useful for redundant disk array applications, but could also provide interesting insights in the development of distributed file systems.

- *caching array architectures*—Caching disk arrays are able to increase performance by deferring work until a time when it can be performed with greater efficiency. My study of redundant disk array software excluded caching architectures. Understanding the ramifications of recovery from errors encountered when completing deferred work are important to demonstrating the applicability of this approach to future array architectures which are increasingly likely to rely upon some form of deferment mechanism.

- *atomic disk semantics*—Most fault models presume that disk arrays operations which are interrupted by crashes do not fail atomically, leaving the area to be written in an unknown state (old data, new data, or unknown data). What is required is that operations which fail, regardless of fault, do not affect data which is not being overwritten. This semantic, carried forward from the traditional semantic of disk drives, continues to plague the implementors of dependable systems which require ACID behavior in a storage system—architects have been forced to implement procedures outside the storage system to create stable storage.

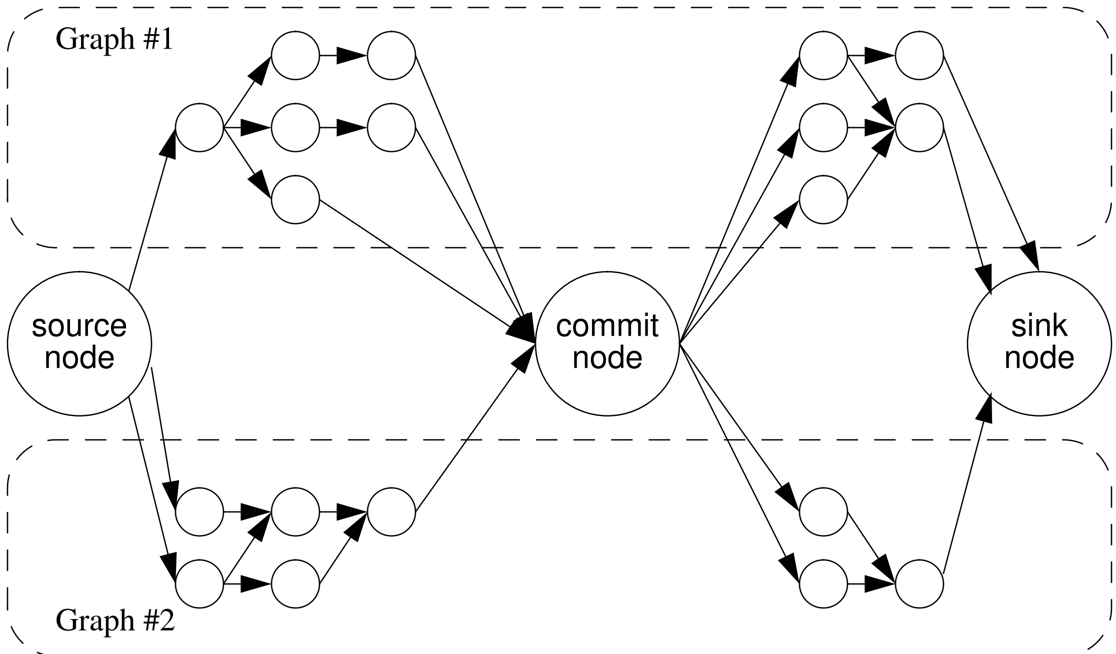


Figure 6-1 Synchronized commit coordinates recovery of multi-graph requests

In this example, two flow graphs have been merged to share a single commit node, effectively forcing both graphs to complete their Phase-I subgraph before either graph is permitted to crossing the commit point. This effectively creates global Phase-I and Phase-II subgraphs and enables global atomic recovery.

By using nonvolatile memory to hold array operating state, disk array vendors are able to produce controllers that atomically survive crashes. By adding a durable redo log that contains enough information to repeat the Phase-II nodes, and making the undo log durable, roll-away recovery permits the atomic survival of crashes (and pending recovery from Phase-I node failures). At restart, replaying the contents of the undo log would remove the effects of all graphs which were interrupted prior to the commit node. Similarly, replaying the contents of the redo log would complete the execution of graphs which were interrupted after the commit node. Extending this recoverability to multi-graph operations is possible by forcing both graphs to commit simultaneously. In the illustration of Figure 6-1, multiple flow graphs are joined with a a common commit node. After two or more flow graphs have been joined, the newly created graph can be converted to a proper flow graph by simply guaranteeing the existence of single sink and source nodes. By mirroring the undo and redo logs in multiple controllers, the permanent failure of a controller can be survived atomically.

References

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

[Anderson85] Anderson, T., Barret, P. A., Halliwell, D. N., and Moulding, M. R. “An evaluation of software fault tolerance in a practical system.” *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing (FTCS-15)*. Los Alamitos, CA: IEEE Computer Society Press, Ann Arbor, MI (June 19-21, 1985) 140-145.

[ANSI91] American National Standard for Information Systems (ANSI). *Small Computer System Interface - 2 (SCSI-2)*. document X3.131-1991. Global Engineering Documents, 2805 McGaw, Irvine, CA 92714. (October 17, 1991).

[ANSI94] American National Standard for Information Systems (ANSI). *Fibre Channel Arbitrated Loop (FC-AL)*. Global Engineering Documents, 2805 McGaw, Irvine, CA 92714. (June 17, 1994).

[Arazi88] Arazi, B. *A Commonsense Approach to the Theory of Error Correcting Codes*. Cambridge, MA: MIT Press (1988).

[ATC90] Array Technology Corporation. *Product Description: RAID+ Series Model RX*. Boulder, CO (1990).

[Avizienis76] Avizienis, A. “Fault-tolerant systems.” *IEEE Transactions on Computers* C-25(12). (December 1976), 1304-1312.

[Bellcore95] Bellcore. *Reliability Prediction Procedure for Electronic Equipment*. Technical Reference TR-TSY-332, Issue 5. (December 1, 1995).

[Bernstein87] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesly (1987).

[Bhide92] Bhide, A. and Dias, D. *RAID architectures for OLTP*. IBM Computer Science Research Report RC 1789. IBM Corp., Almaden, Calif. (1992).

[Bitton88] Bitton, D. and Gray, J. “Disk shadowing.” *Proceedings of the 14th Conference on Very Large Data Bases (VLDB-14)*. (September 1988) 331-338.

[Bjork75] Bjork, L. A. Jr. “Generalized audit trail requirements and concepts for data base applications.” *IBM Systems Journal* 14(3). (1975) 229-245.

[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 21st Annual Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA: IEEE Computer Society Press. Chicago. (April18-21, 1994) 245-254.

[Boehm81] Boehm, B. W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall (1981).

[Brown72] Brown, D. T., Eibsen, R. L., and Thorn, C. A. “Channel and direct access device architecture.” *IBM Systems Journal* 11(3). (1972) 186-199.

[Burkhard93] Burkhard, W. and Menon, J. “Disk array storage system reliability.” *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS-23)*. Toulouse, France. (June 1993).

[Cao94] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. “The TickerTAIP parallel RAID architecture.” *ACM Transactions on Computer Systems* 12(3). (August 1994) 236-269.

[Chandy72] Chandy, K. M. and Ramamoorthy, C. V. “Rolllback and recovery strategies for computer programs.” *IEEE Transactions on Computers* C-21(6). (June 1972) 546-556.

[Chao92] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. “Mime: a high performance parallel storage device with strong recovery guarantees.” Hewlett-Packard technical report HPL-CSP-92-9. (November 1992).

[Chen78] Chen, L. and Avizienis, A. “N-version programming: a fault-tolerance approach to reliability of software operation.” *Proceedings of the 8th Annual International Symposium on Fault-Tolerant Computing (FTCS-8)*. Los Alamitos, CA: IEEE Computer Society Press. Tolouse, France (June 1978) 3-9.

[Chen90] Chen, P., and Patterson, D. “Maximizing performance in a striped disk array.” *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA: IEEE Computer Society Press. Seattle, WA. (May 1990) 322-331.

[Chen94] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. “RAID: high-performance, reliable secondary storage.” *ACM Computing Surveys* 26(2). (June 1994) 143-185.

[Clarke82] Clarke, E. and Emerson, E. A. “Synthesis of synchronization skeletons for branching time temporal logic.” Proceedings 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) 52-71.

[Clarke94] Clarke, E., Grumberg, O., and Long, D. “Model checking.” *Proceedings of the International Summer School on Deductive Program Design.* Marktoberdorf, Germany. (July 26 - August 27, 1994).

[Collins93] Collins, W., Brewton, J., Cook, D., Jones, L., Kelly, K., Kluegel, L., Krantz, D., and Ramsey, C. “Los Alamos HPDS: high-speed data transfer.” *Proceedings of the 12th IEEE Symposium on Mass Storage Systems*. Los Alamitos, CA: IEEE Computer Society Press. Monterey, CA (April 26-29, 1993) 111-118.

[CRW96] “Suppliers undaunted by Hewlett-Packard....” *Computer Retail Week*. (February 5, 1996) 62.

[Copeland] Copeland, G. and Keller, T. “A comparison of high-availability media recovery techniques.” *Proceedings of the ACM Conference on Management of Data (SIGMOD).* (1989) 98-109.

[Courtright94] Courtright, W. V. II and Gibson, Garth A. “Backward error recovery in redundant disk arrays.” *Proceedings of the 20th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG94).* Computer Measurement Group, 414 Plaza Drive, Suite 209, Westmont, IL 60559. Orlando, FL (December 4-9, 1994) 63-74.

[Courtright96a] Courtright, W. V. II, Gibson, G., Holland, M., and Zelenka, J. “RAIDframe: rapid prototyping for disk arrays.” *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS ‘96)*. Philadelphia, PA (May 23-26, 1996) 268-269.

[Courtright96b] Courtright, W. V. II, Amiri, K., Gibson, G., Holland, M., and Zelenka, J. “A structured approach to redundant disk array software.” *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS ‘96)*. Urbana-Champaign, IL (September 4-6, 1996) 11-20.

[Courtright96c] Courtright, W. V. II, Holland, M., Gibson, G., and Reilly, L. N. *RAIDframe: A Rapid Prototyping Tool for RAID Systems*. Computer Science Technical Report CMU-CS-96-xxx, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213 (1996).

[Custer93] Custer, H. *Inside Windows NT.* Microsoft Press, One Microsoft Way, Redmond, WA 98052-6399. (1993).

[Digital89] Digital Equipment Corporation. *Digital Storage Technology Handbook*. (1989).

[Disk90] Disk/Trend, Inc. *1990 Disk/Trend Report: Rigid Disk Drives*. Mountain View, CA (October 1990).

[Disk96a] Disk/Trend, Inc. *1996 Disk/Trend Report: Disk Drive Arrays*. Mountain View, CA (November 1996).

[Disk96b] Disk/Trend, Inc. *1995 Disk/Trend Report: Rigid Disk Drives*. Mountain View, CA (May 1996).

[DOD81] U.S.A. Dept. of Defense. *Military Standard: Reliability Modeling and Prediction*. MIL-STD-756B. Washington 20301. (November 18, 1981).

[DOD86] U.S.A. Dept. of Defense. *Military Handbook: Reliability Prediction of Electronic Equipment*. MIL-HDBK-217E. Washington 20301. (October 27, 1986).

[Elmendorf72] Elmendorf, W. R. "Fault-tolerant programming." *Proceedings of the 2nd International Symposium on Fault-Tolerant Computing (FTCS-2)*. Los Alamitos, CA: IEEE Computer Society Press, Newton, MA (June 19-21, 1972) 79-83.

[Friedman96] Friedman, M. B. "RAID keeps going and going and...." *IEEE Spectrum* 33(4). (April 1996) 73-79.

[Fujitsu87] Fujitsu Corporation. *M2361A: Mini-Disk Drive Engineering Specifications, B03P-4825-001A*. (February 1987).

[Ganger94] Ganger, G. R., Worthington, B. L., Hou, R. Y., and Patt, Y. N. "Disk arrays: high-performance, high-reliability storage systems." *Computer* (March 1994) 30-36.

[Geist87] Geist, R. and Daniel, S. "A continuum of disk scheduling algorithms." *ACM Transactions on Computer Systems* 5(1) (February 1987) 77-92.

[Gibson89] Gibson, G. A., Hellerstein, L., Karp, R. M., Katz, R. H., and Patterson, D. A. "Coding techniquest for handling failures in large disk arrays." *Proceedings of the Third International Conference on Architectural Support for Programming Languages (ASPLOS III)*. Boston, MA. (April 1989) 123-132.

[Gibson92] Gibson, G. A. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. Cambridge, MA: MIT Press (1992).

[Gibson93] Gibson, G. A., Patterson, D. A. "Designing disk arrays for high data reliability." *Journal of Parallel and Distributed Computing* 17(1-2). (1993) 4-27.

[Gibson95a] Gibson, G. A., et. al. "The Scotch Parallel Storage Systems." *Digest of Papers: Fortieth IEEE Computer Society International Conference (COMPCON Spring '95)*. Los Alamitos, CA: IEEE Computer Society Press, San Francisco, CA. (March 5-9, 1995) 403-410.

[Gibson95b] Gibson, G. Courtright, W. V. II, Holland, M., and Zelenka, J. *RAIDframe: Rapid Prototyping for Disk Arrays*. Computer Science Technical Report CMU-CS-95-200, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213 (1995).

[Golding95] Golding, R., Bosch, P., Staelin, C., Sullivan, T., and Wilkes, J. "Idleness is not sloth." *Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*. USENIX Association, Berkeley, CA (1995) 201-212.

[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* 13(2). (June 1981) 223-242.

[Gray85] Gray, J. "Why do computers stop and what can be done about it?" Tandem Computers Technical Report 85.7 part number 87614. 19333 Vallco Parkway, Cupertino, CA 95014. (June 1985).

[Gray90a] Gray, J. "A census of Tandem system availability: 1985-1990." Tandem Computers Technical Report 90.1 part number 33579. 19333 Vallco Parkway, Cupertino, CA 95014. (January 1990).

[Gray90b] Gray, J. and Walker, M. "Parity striping of disk arrays: low-cost reliable storage with acceptable throughput." *Proceedings of the 16th Conference on Very Large Data Bases (VLDB-16)*. Brisbane, Australia (August 13-16, 1990) 148-159.

[Gray93] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann Publishers (1993).

[Grochowski96a] Grochowski, E. and Hoyt, R. F. "Future trends in hard disk drives." *IEEE Transactions on Magnetics* 32(3). (May 1996) 1850-1854.

[Grochowski96b] Grochowski, E. "40 Years of Innovation in Hard Disk Drive Technology." *Computer Technology Review*. (Fall 1996) 104-107.

[Haërder83] Haërder, T. and Reuter, A. "Principles of transaction-oriented database recovery." *Computing Surveys* 15(4). (December 1983) 287-317.

[Hamming50] Hamming, W. R. "Error detecting and error correcting codes." *Bell Systems Technical Journal* 29(2). (April 1950) 147-160.

[Holland92] Holland, M. and Gibson, G. "Parity declustering for continuous operation in redundant disk arrays." *Proceedings of the International Conference on Architectural Suport for Programming Languages and Operating Systems* (ASPLOS). Boston (October 1992) 23-25.

[Holland94] Holland, M. *On-Line Reconstruction in Redundant Disk Arrays*. Ph.D. dissertation. Computer Science Technical Report CMU-CS-94-164, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. (May 1994).

[Horning74] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B. “A program structure for error detection and recovery.” *Lecture Notes in Computer Science* Vol. 16. Wein, Austria: Springer-Verlag (1974) 171-187.

[Hsiao90] Hsiao, H. and DeWitt, D. “Chained declustering: a new availability strategy for multiprocessor database machines.” *Proceedings of the International Data Engineering Conference*. (1990).

[Hsiao91] Hsiao, H. and DeWitt, D. ““A performance study of three high-availablity data replication strategies.” *Proceedings of the International Conference on Parallel and Distributed Information Systems*. (1991) 18-28.

[IBM95] International Business Machines. *SCSI Logical Interface Specification: DFMS/ DFHS SCSI Models, All Capacities, 3.5 Inch Drives* (Rel. 3.0). publication #3303. (February 20, 1995).

[IDC95] International Data Corporation. *1995 Worldwide HDD and RAID Storage Subsystem Market Review and Forecast*. Vol. 1. Five Speen Street, Framingham, MA 01701. (October 1995) 42.

[IDEMA96] *MTBF Redefinition Subcommittee Meeting Minutes*. International Disk Drive Equipment and Materials Association (IDEMA), 710 Lakeway, Suite 140, Sunnyvale, CA 94806. `http://www.idema.org` (August 28, 1996).

[Kim86] Kim, M. Y. “Synchronized disk interleaving.” *IEEE Transactions on Computers* 35(11). (November 1986) 978-988.

[Kuehn69] Kuehn, R. E. “Computer redundancy: design, performance, and future.” *IEEE Transactions on Reliability* R-18(1). (February 1969) 3-11.

[Lamport82] Lamport, L., Shostak, S., and Pease, M. “The byzantine generals’ problems.” *ACM Transactions on Programming Languages and Systems* 4. (1982) 382-401.

[Lampson79] Lampson, B. W. and Sturgis, H. E. “Crash recovery in a distributed data storage system.” XEROX Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 (April 27, 1979).

[Laprie82] Laprie, J. C. and Costes, A. “Dependability: a unifying concept for reliable computing.” *Proceedings of the 12th International Symposium on Fault-Tolerant Computing (FTCS-12)*. Los Alamitos, CA: IEEE Computer Society Press. Santa Monica, CA (June 22-24, 1982) 18-21.

[Lawlor81] Lawlor, F. D. “Efficient mass storage parity recovery mechanism.” *IBM Technical Disclosure Bulletin* 24(2). (July 1981) 986-987.

[Lee90a] Lee, E. K. “Software and performance issures in the implementation of a RAID prototoype.” Technical report UCB/CSD 90/573, Computer Science Division (EECS), University of California, Berkeley, CA 94720. (May 17, 1990).

[Lee90b] Lee, E. K. and Katz, R. H. “Performance considerations of parity placement in disk arrays.” *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing (FTCS-12)*. Los Alamitos, CA: IEEE Computer Society Press. Santa Monica, CA (June 22-24, 1982) 190-199.

[Lee90c] Lee, P. A. and Anderson, T. A. *Fault Tolerance: Principles and Practice* (2nd ed.). Wein, Austria: Springer-Verlag (1990).

[Lee91] Lee, E. K. and Katz, R. H. “Performance Consequences of parity placement in disk arrays.” *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. Palo Alto, CA. (April 1991) 190-199.

[Lomet77] Lomet, D. B. “Process structuring, synchronization, and recovery using atomic actions.” *ACM SIGPLAN Notices* 12(3). (March 1977) 128-137.

[Lynch94] Lynch, N., Merrit, M., Weihl, W. and Fekete, A. *Atomic Transactions*. San Mateo, CA: Morgan Kaufmann Publishers (1994).

[Massiglia86] Massiglia, P. *Digital Large System Mass Storage Handbook*. Digital Equipment Corporation. (1986).]

[Menon93a] Menon, J. and Cortney, J. “The architecture of a fault-tolerant cached RAID controller.” *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA-20)*. Los Alamitos, CA: IEEE Computer Society Press. San Diego, CA (May 16-19, 1993) 76-86.

[Menon93b] Menon, J., Roche, J., and Kasson, J. “Floating parity and data disk arrays.” *Journal of Parallel and Distributed Computing*. (January 1993).

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

[Mogi94] Mogi, K. and Masaru, K. “Dynamic parity stripe reorganizations for RAID5 disk arrays.” *Proceedings of the Third Conference on Parallel and Distributed Information Systems*. Los Alamitos, CA: IEEE Computer Society Press, Austin, TX. (September 28-30, 1994) 17-26.

[Ng94] Ng, S. W. “Crosshatch disk array for improved reliability and performance.” *Proceedings of the 21st Annual Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA: IEEE Computer Society Press. Chicago. (April18-21, 1994) 255-264.

[Ousterhout85] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, j. A., Kupfer, M., and Thompson, J. G. “A trace-driven analysis of the Unix 4.2 BSD file system.” *Proceedings of the 10th Symposium on Operating Systems Principles*. Orcas Island, WA (1985) 15-24.

[Ousterhout88] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, Michael N., and Welch, B. B. “The sprite network operating system.” *IEEE Computer* 21(2). (February 1988) 23-35.

[Park86] Park, A. and Balasubramanian, K. “Providing fault tolerance in parallel secondary storage systems.” Technical Report CS-TR-057-86. Dept. of Computer Science, Princeton University, Princeton NJ (1986).

[Parnas72] Parnas, D. L. “On the criteria to be used in decomposing systems into modules.” *Communications of the ACM* 15(12). (December 1972) 1053-1058.

[Patterson88] Patterson, D. A., Gibson, G. A., and Katz, R. H. “A case for redundant arrays of inexpensive disks (RAID).” *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*. ACM Press, Chicago (June 1988) 109-116.

[Patterson95] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. “Informed prefetching and caching.” *Proceedings of the 15th Symposium on Operating Systems Principles*. (December 1995).

[Patterson96] Pattterson, D. A. and Hennessy, J. L. *Computer Architecture: A Quantitative Approach* (2nd ed.). San Francisco, CA: Morgan Kaufmann Publishers (1996).

[PDLHTTP] `http://www.cs.cmu.edu/Web/Groups/PDL`

[Polyzois93] Polyzois, C., Bhide, A., and Dias, D. “Disk mirroring with alternating deferred updates.” *Proceedings of the Conference on Very Large Data Bases*. (1993) 604-617.

[Potochnik96] Potochnik, J. Manager, Symbios Logic OEM Software Development. personal communication. (April 22, 1996).

[Pugh71] Pugh, E. W. “Storage hierarchies: gaps, cliffs, and trends.” *IEEE Transactions on Magnetics*. Vol. MAG-7. (December 1971) 810-814.

[Quantum95] Quantum Corp. *Atlas XP31070/XP32150/XP34300S Product Manual*. publication number 81-108333-01. (April 1995).

[RAB96] RAID Advisory Board. *The RAIDbook: A Source Book for Disk Array Technololgy* (5th ed.). ISBN 1-879936-90-9. St Peter, MN. (February, 1996).

[raidSimFTP] `ftp://ftp.cs.cmu.edu/project/pdl/raidSim`

[RAIDframeFTP] `ftp://ftp.cs.cmu.edu/project/pdl/RAIDframe`

[RAIDframeHTTP] `http://www.cs.cmu.edu/Web/Groups/PDL/RAIDframe`

[Randell78] Randell, B., Lee, P. A., and Treleaven, P. C. “Reliability Issues in Computing Systems Design.” *Computing Surveys* 10(2). (June 1978) 123-165.

[Rosenblum92] Rosenblum, m. and Ousterhout, J. K. “The design and implementation of a log-structured file system.” *ACM Transactions on Computer Systems*, 10(1). (February 1992) 26-52.

[Ruemmler93] Ruemmler, C. and Wilkes, J. “UNIX disk access patterns.” *USENIX Winter 1993 Technical Conference Proceedings*. San Diego, CA (January 25-29, 1993) 405-420.

[Ruemmler94] Rummler, C. and Wilkes, J. “An introduction to disk drive modeling.” *Computer* 25(3). (March 1994) 17-28.

[Salem86] Salem, K. and Garcia-Molina, H. “Disk striping.” *Proceedings of the 2nd International Conference on Data Engineering*. Los Alamitos, CA: IEEE Computer Society Press. (1986) 336-342.

[Savage96] Savage, S. and Wilkes, J. “AFRAID—a frequently redundant array of independent disks.” *Proceedings of the 1996 USENIX Technical Conference*. San Diego, CA. (January 22-26, 1996) 27-39.

[Schulze89] Schulze, M. E., Gibson, G. A., Katz, R. H., and Patterson, D. A. “How reliable is RAID?” *Proceedings of the 1989 Computer Society International Conference (COMPCON89)*. Los Alamitos, CA: IEEE Computer Society Press, San Francisco, CA, (spring 1989) 118-123.

[Seagate94] Seagate Technology. *RAID 5 Support on SCSI Disk Drives* (Rev. 1.51). 925 Disc Drive, Scotts Valley, CA 95066-4544. (November 3, 1994).

[Seagate95] Seagate Technology, Inc. *Barracuda 4 ST15150N/ND ST15150W/WD/WC/DC Disc Drive Product Manual* (Vol. 1). Publication Number: 83328880-B. 925 Disc Drive, Scotts Valley, CA 95066-4544. (March 1995).

[Seltzer90] Seltzer, M., Chen, P., and Ousterhout, J. “Disk scheduling revisited.” *Proceedings of the USENIX Winter Technical Conference*. Washington, DC (January 1990) 313-323.

[Sierra90] Sierra, H. M. *An Introduction to Direct Access Storage Devices*. Boston: Academic Press (1990).

[Siewiorek92] Siewiorek, D. P. and Swarz, R. S. *Reliable Computer Systems: Design and Evaluation* (2nd ed.). Bedford, MA: Digital Press (1992).

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

[STC94] Storage Technology Corporation. Iceberg 9200 Storage System: Introduction. STK Part Number 307406101. Corporate Technical Publications, 2270 S. 88th St., Lousiville, CO 80028. (1994).

[Stodolsky94] Stodolsky, D., Holland, M., Courtright, W. V. II, and Gibson, G. A. "Parity-logging disk arrays." *ACM Transactions on Computer Systems* 12(3). (August 1994) 206-235.

[Stone89] Stone, R. F. "Reliable computing systems - a review." Computer Science Technical Report YCS 110, University of York, England (1989).

[Symbios95a] Symbios Logic. *Functional Specification: 6210 10 Drive Subsystem* (Rev. C). FS 348-0029527. 3718 N. Rock Rd, Wichita, KS 67226 (December 1995).

[Symbios95b] Symbios Logic, *Hardware Functional Specification for the Symbios Logic Series 3 RAID Controller Model 3620* (Rev. B). FS 348-0029459. 3718 N. Rock Rd, Wichita, KS 67226 (December 1995) 48.

[Symbios96] Symbios Logic. *Software Integrator's Guide: Series 3 RAID Controllers*. 348-0026028 Rev. C. 3718 N. Rock Rd., Wichita, KS 67226 (March 1996).

[Teradata85] Teradata Corp. *DBC/1012 Data Base Computer System Manual*. (Rel. 13) C10-0001-01. (1985).

[TMC87] Thinking Machines Corp. *Connection Machine Model CM-2 Technical Summary.* Technical Report HA87-4. (April 1987).

[Vaziri96] Vaziri-Farahani, M. *Proving Correctness of a Controller Algorithm for the RAID Level 5 System*. Master's thesis. Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (August 1996).

[Verhofstad78] Verhofstad, J. S. M. "Recovery techniques for database systems." *Computing Surveys* 10(2). (June 1978) 167-195.

[von Neumann56] von Neumann, J. "Probabilistic logics and the synthesis of reliable organisms from unreliable components." In Shannon, C. E., and McCarthy, J. (Eds.) *Automata Studies*. Princeton, NJ: Princeton University Press (1956) 43-98.

[Wood93] Wood, C. and Hodges, P. "DASD Trends: cost, performance, and form factor." *Proceedings of the IEEE* 81(4). (April 1993) 573-585.

Appendix A: Flow Graphs for Popular Array Architectures

This appendix demonstrates the process of composing popular array operations from a set of atomic actions which are assumed to be undoable. Graphs, and the criteria for selecting them, are presented for twelve array architectures. The graphs contained in this appendix are the actual graphs used in the forward and backward error recovery studies described in Chapter 5. Later, in Appendix B, these graphs are modified to support roll-away error recovery.

Before proceeding, recall Section 4.2.3.1 that RAIDframe performs stripe locking and memory allocation outside of the execution graph. Locks (shared or exclusive) that protect the address range of the request are first allocated and then buffers are acquired. Locks are held until the request is completed, regardless of outcome. Buffers are acquired at during the creation of each graph and released at the completion of its execution. Because locks and buffers are acquired and released outside the scope of graph execution, the graphs presented here do not contain any resource allocation or deallocation actions.

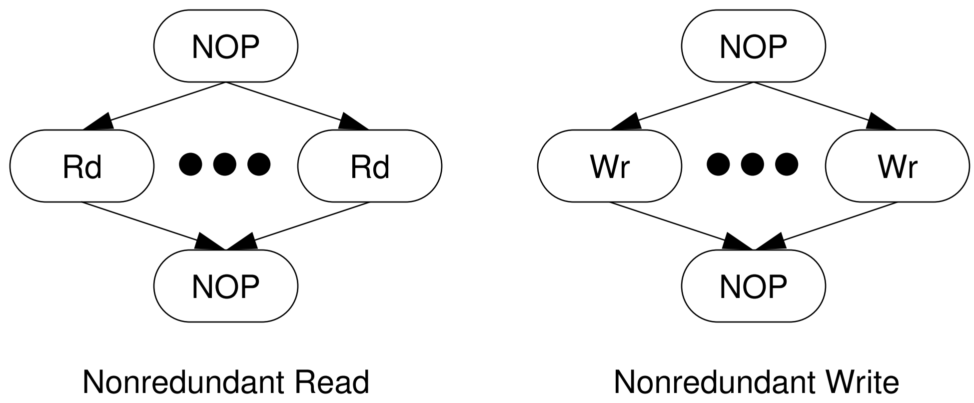


Figure A-1 Nonredundant graphs

A.1 RAID Level 0

RAID level 0 arrays do not encode data; therefore, the array is not fault-tolerant and because of this, only nonredundant operations are available for use. Figure A-1 illustrates the structure of nonredundant read and write operations, represented as flow graphs. The NOP actions guarantee that each graph has single source (head) and sink (tail) nodes. Each graph is capable of supporting one or more simultaneous disk actions, allowing the graph to scale with the size of the user request.

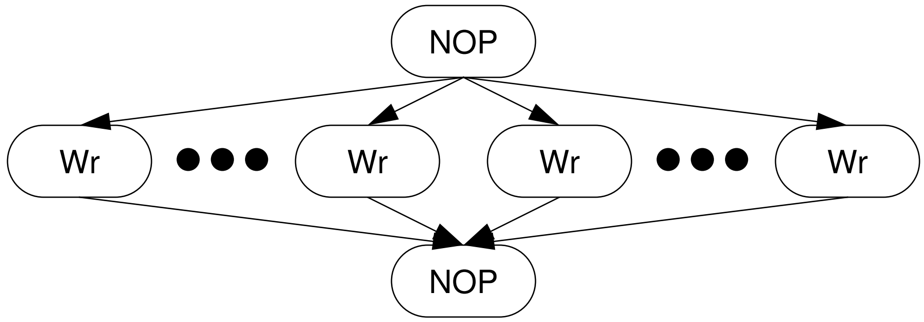


Figure A-2 Mirrored-write graph

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 actions on the left represent writes to a primary disk(s) and write actions on the right represent writes of data to secondary disk(s).

A.2 RAID Level 1, Interleaved Declustering, and Chained Declustering

RAID level 1 [Patterson88], chained declustering [Hsiao90, Hsiao91], and interleaved declustering [Copeland89, Teradata85] arrays are single-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 A-1 specifies which operations are used to service a request given the state of the disks.

In addition to the nonredundant graphs described in Figure A-1, RAID level 1 arrays require the 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 A-2, contains twice the number of write actions as a nonredundant write operation because a copy of each symbol is written to both a primary and a secondary disk.

Table A-1 RAID level 1 graph selection

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

A.3 RAID Level 3

RAID level 3 arrays are single fault-tolerant and use even-parity encoding to protect data from disk failures. The criteria for selecting graphs is summarized in Table A-2. Data is bit-striped across the array, guaranteeing that all accesses, regardless of size, will involve all disks in the array. Because of this, the operation used to write data to the array is known as a *large write*. The large-write operation, illustrated in Figure A-4, computes the parity of a codeword and then simultaneously writes all symbols, data and parity, to independent disks. The large-write operation (with one less disk write action) is also used when a data disk fails. If the disk which holds parity fails, a nonredundant write operation (Figure A-1) is used.

The nonredundant read operation (Figure A-1) is used to read data from a fault-free array or an array in which the disk containing parity has failed. If a data disk has been lost, a *degraded-read* operation reconstructs the missing data symbol by reading the entire codeword (surviving data and parity) and XOR'ing them as described in Section 2.5.2.2.

Table A-2 RAID level 3 graph selection

Request	Disk Faults	Graph
read	none	nonredundant read
read	data disk	degraded read
read	parity disk	nonredundant read
write	none	large write
write	data disk	large write
write	parity disk	nonredundant write

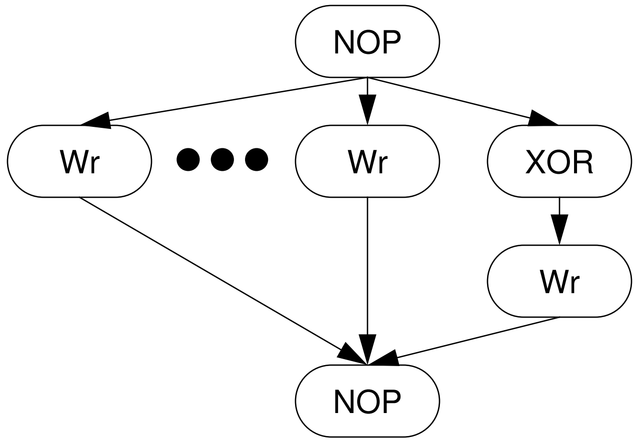


Figure A-3 Large-write graph

This operation overwrites an entire codeword in a parity-protected array. When the graph is submitted for execution, all new data is known. The XOR *node computes new parity which is then written to disk. The other* Wr *nodes write new data to disk.*

If a disk which contains data has failed, this graph is still used to write data to the array with the modification that the Wr *node which involves the failed disk is eliminated.*

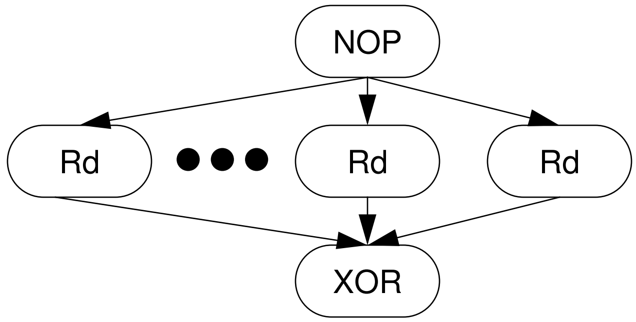


Figure A-4 Degraded-read graph

This operation is used to reconstruct missing data from a parity-protected array. Data is "reconstructed" by reading all surviving symbols of the codeword: data and parity. In this graph, the Rd *nodes on the left represent reads of surviving data and the* Rd *node on the right, the read of parity. Once all symbols (except the missing symbol) are known, the missing symbol is computed as the XOR of the surviving symbols.*

A.4 RAID Levels 4 and 5 and Parity Declustering

Similar to RAID level 3, RAID levels 4 and 5 [Patterson88] and parity declustering [Holland92] arrays tolerate disk faults through the use of a 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 A-5) which is used to write data to less than half of a codeword and the *reconstruct-write* operation (Figure A-6) which is used to write data to more than half, but less than a full codeword. Table A-3 provides a breakdown of graph selection for RAID level 4 and 5 arrays. Because these arrays differ only in mapping, the same table applies to both architectures.

The small-write operation, illustrated in Figure A-5, writes both data and parity to disk. Parity is computed as:

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

The cluster of read actions on the left side of the graph represent the read of old data and the single read action 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.

Table A-3 RAID levels 4 and 5 graph selection

Request	Disk Faults	Graph
read	none	nonredundant read
read	data disk	degraded read
read	parity disk	nonredundant read
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

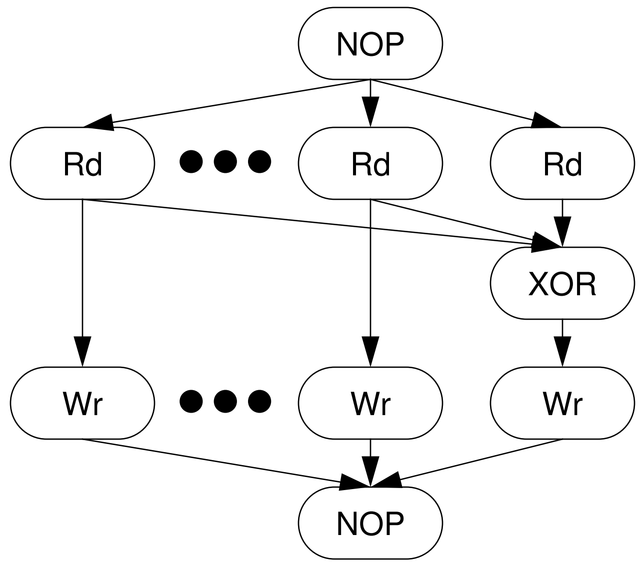


Figure A-5 Small-write graph

In the reconstruct-write operation, illustrated in Figure A-6, parity is computed from all symbols in the codeword. The Rd actions 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.

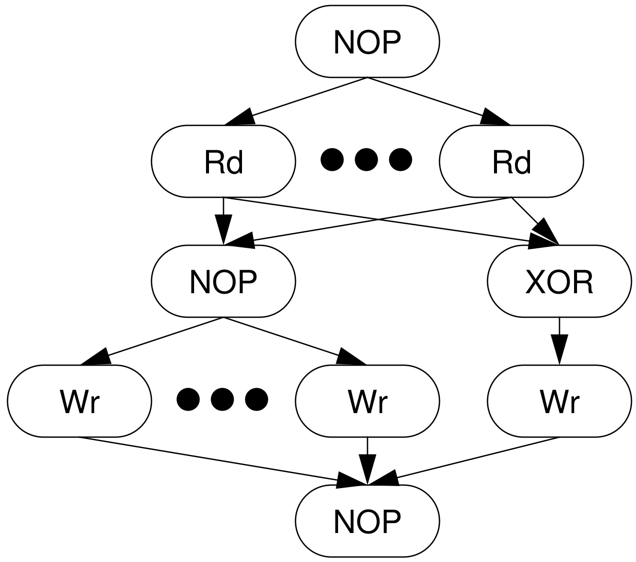


Figure A-6 Reconstruct-write graph

This graph, as drawn, was used in forward error recovery experiments. The Rd *actions read the data symbols which are not being overwritten. The left-most* Wr *actions overwrite data symbols and the* Wr *action on the right overwrites parity. As discussed in Section 3.4.1, the* NOP *node is necessary to avoid the write hole in implementations that employ forward error recovery.*

The NOP *node was removed for backward error recovery experiments which assumed that all nodes could be undone. With the* NOP *node removed, the data writes are allowed to begin immediately (i.e. were direct descendents of the source node.*

A.5 Parity Logging

Instead of committing parity information directly to disk, parity logging records changes to parity in an append-only log [Stodolsky94]. Later, when the log fills, the parity updates are applied en mass. If a disk fails in a parity logging array, I assume that the array is acquiesced and that the contents of the log are applied, converting the array to a RAID level 5 array; therefore, all degraded operations in parity logging arrays, as well as read operations, are identical to those used in RAID levels 4 and 5. Table A-4 summarizes the criteria used to select graphs in a parity logging array.

In general, the structure of write operations in parity-logging arrays is similar to that of operations for writing data in RAID level 4 and 5 arrays—the principal difference being that the disk actions which write parity are replaced by log actions. Figure A-7 illustrates a parity-logging small-write operation. In this graph, the read of old parity has been eliminated and the write of parity to disk has been replaced by a log action which places an "update" record in the parity log. An update record contains the exclusive-or of old and new data.

A reconstruct-write operation in parity logging is identical to a reconstruct-write operation in RAID levels 4 and 5 (Figure A-6) with the exception that the Wr action which overwrites parity on disk is replaced by a log action which appends an "overwrite" record to the parity log. The parity-logging reconstruct-write operation is illustrated in Figure A-8.

Similarly, the parity-logging large-write operation, illustrated in Figure A-9, is identical to the large-write operation of Figure A-4 with the exception that the write of parity to disk is replaced by a log action.

Table A-4 Parity logging graph selection

Request	**Disk Faults**	**Graph**
read	none	nonredundant read
read	any single disk	use RAID level 5 graphs
write < 50% of codeword	none	parity-log small write
write 50%, < 100%	none	parity-log reconstruct write
write = 100%	none	parity-log overwrite write
write	any single disk	use RAID level 5 graphs

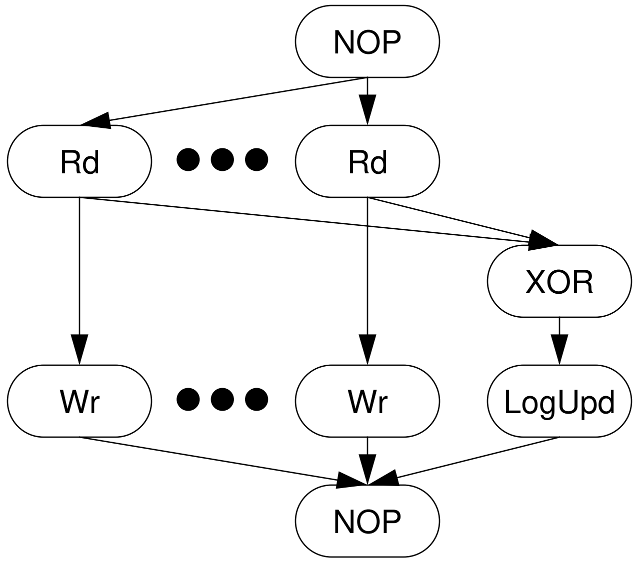


Figure A-7 Parity-logging small-write graph

Similar to a RAID level 5 small-write operation, the parity-logging small write computes an update to parity based upon the exclusive-or of old and new data. However, instead of updating parity directly, this operation records the update record in an append only log.

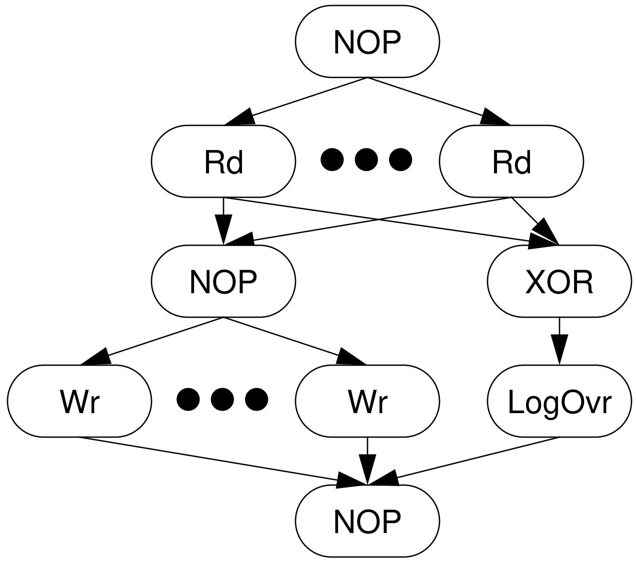


Figure A-8 Parity-logging reconstruct-write graph

This graph implements the same algorithm used in the RAID level 5 reconstruct-write operation except that the new parity which was previously overwritten on disk is now recorded in the parity log as an overwrite record.

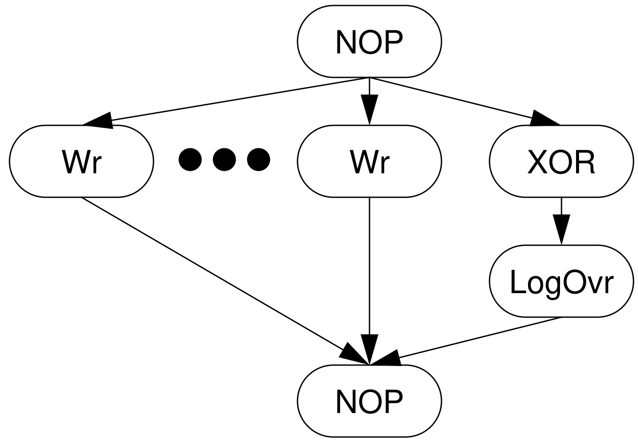


Figure A-9 Parity-logging large-write graph

This graph is identical to the RAID level 3 large-write operation with the exception that instead of overwriting parity which is stored on disk, an overwrite record is recorded in the parity log.

A.6 RAID Level 6 and EVENODD

In addition to parity, RAID level 6 [RAB96] and EVENODD [Blaum95] arrays employ a second check symbol to allow survival of two simultaneous disk failures. In RAID level 6, I refer to this second symbol as “Q” and in EVENODD I refer to it as “E.” The graphs and graph selection for each of these two architectures is identical, and I present only the graphs in terms of RAID level 6. To create the EVENODD graphs, simply replace the Q nodes with E nodes. The graphs used by these two architectures are summarized in Table A-5.

Table A-5 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
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

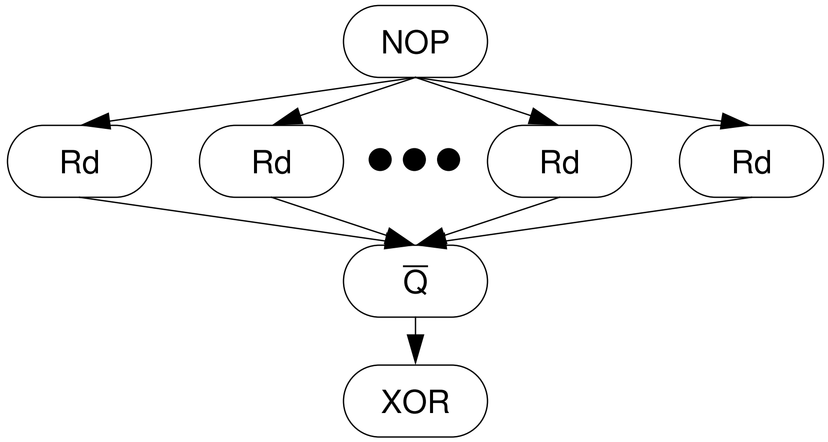


Figure A-10 PQ double-degraded read graph

This operation is used when two data units are missing from the codeword. The left-most Rd *action reads the old value of parity and the right-most action reads the old value of Q. The center Rd actions read all surviving data in the codeword. The* $\overline{\text{Q}}$ *action regenerates a single missing data symbol and the* XOR *node regenerates the other missing symbol.*

Read operations to a 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 A-10, is used. This operation is simply an extension of the degraded-read operation previously defined in Figure A-4, the only difference being the addition of an extra decoding step.

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 A-11.

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 A-12, writes new data symbols and computes new values of parity and “Q” using Equation A-1 on page 147. 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.

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 A-13, 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, par-

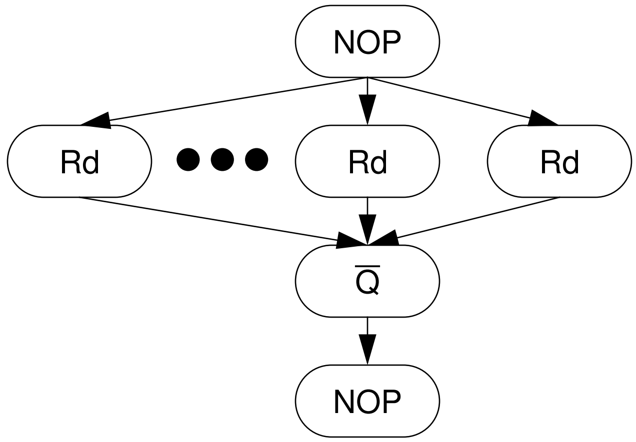


Figure A-11 PQ degraded-DP-read graph

Same as the degraded-read graph of Figure A-4 but uses $\overline{Q}$ *instead of* XOR.

---

ity, 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.

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 A-14, is used. This graph employs an algorithm similar to the one used in the PQ degraded write operation, but must reconstruct the failed data which is not overwritten.

Finally, writing data to the entire codeword is simply performed using the *PQ large-write* operation. Illustrated in Figure A-15, the operation overwrites every symbol in the codeword.

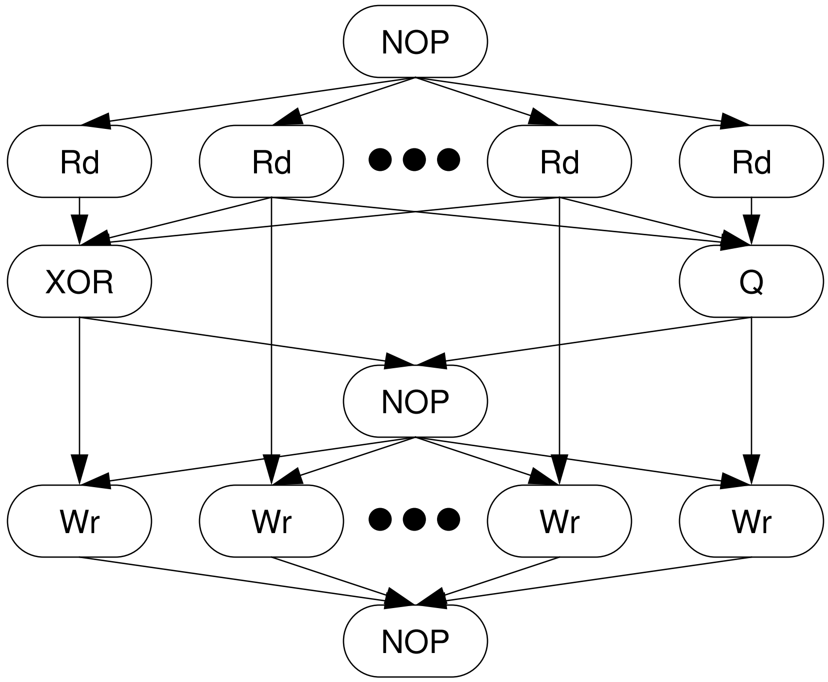


Figure A-12 PQ small-write graph

Similar to the small-write graph (Figure A-5) but with an extra chain added to update the "Q" disk. Additionally, a NOP *node was added to avoid a write hole in systems which employ forward error recovery. This node prevents graphs which fail from two faults from partially modifying a codeword, making recovery impossible. The* NOP *node is removed when backward error recovery is employed. The redundant arcs (e.g.* Rd-Wr*) appear are necessary when the NOP node is removed.*

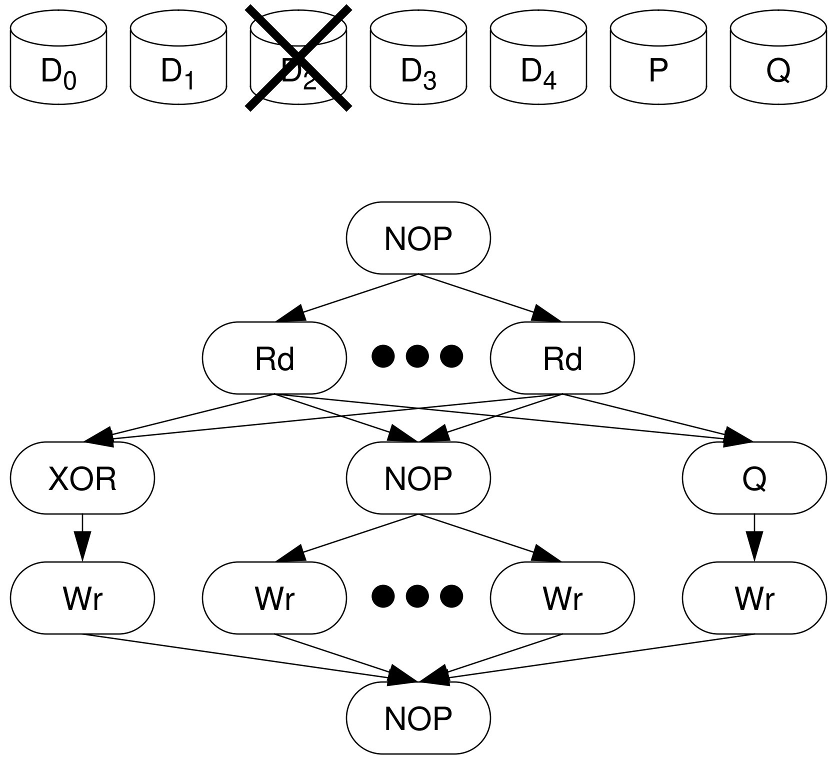


Figure A-13 PQ Reconstruct-write graph

Similar to the reconstruct-write graph (Figure A-6), 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 *actions 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* NOP *node prevents the write of new data (*D_1*) from executing until the entire codeword is stored in the controller. is necessary in forward error recovery implementations to ensure recovery. Implementations employing backward error recovery are allowed to remove the* NOP *node, allowing the write of new data to occur as soon as the graph begins execution.*

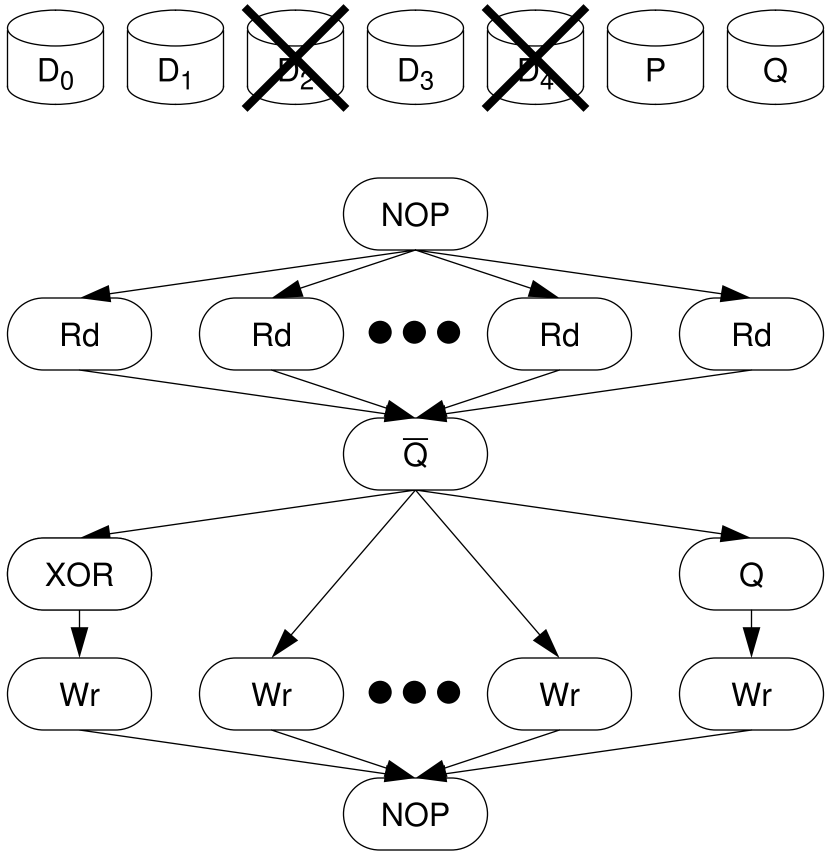


Figure A-14 PQ double-degraded write graph

Assume that D_1 *and* D_2 *are to be overwritten. Because* D_4 *is missing, the PQ reconstruct operation can not 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 *actions on the ends read old* P *and* Q*. The* Q *action reconstructs* D_4*. At this point, the entire codeword is known and the computation and writing of parity, Q and data can commence.*

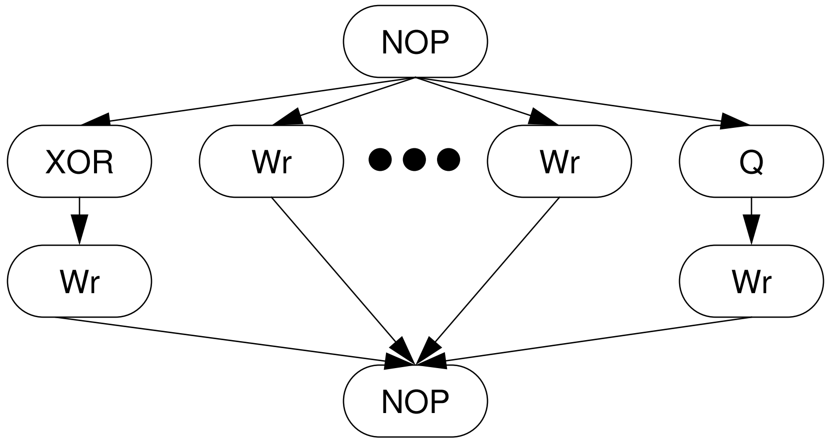


Figure A-15 PQ large-write graph

Similar to the large-write graph (Figure A-3) but with an extra chain added to update the "Q" disk.

A.7 Two-Dimensional Parity

Disk arrays which employ two-dimensional parity are capable of surviving two simultaneous disk failures [Gibson89]. Naturally, this implies that a greater set of operations is required in a two-dimensional parity implementation. Table A-6 summarizes the criteria for selecting an operation given the number and location of disk faults in the array. For simplicity, I have included the basic set of operations necessary to implement two-dimensional parity. It is possible to enrich this set of operations to with operations which increase performance by optimally manipulating parity given the access size (e.g. reconstruct-style update of horizontal parity and small-write update of vertical parity).

Fault-free two-dimensional parity array operations are the same as those used in RAID level 4 and 5 arrays. The same is true for read operations which involve a single failed data disk or one or two failed parity disks.

A new read operation, *2D double-degraded read*, is required when two data disks have failed. This case and the graph which implements the operation are illustrated in Figure A-16. The operation reconstructs missing information using the vertical codewords. This same operation is used in the case that a data and a horizontal parity disk are lost with the modification that only one data disk will need to be reconstructed.

Table A-6 Two-dimensional parity graph selection

Request	Disk Faults	Graph
read	none	nonredundant read
read	one data disk	degraded read
read	one parity disk	nonredundant read
read	two data disks	2D double-degraded read
read	data + vert. parity	degraded read
read	data + horiz. parity	2D double-degraded read
read	two parity disks	nonredundant read
write	none	2D small write
write	one data disk	2D degraded write
write	vertical parity	2D small write, omit V
write	horizontal parity	2D degraded-H write
write	data + vert. parity	2D degraded-DV write
write	data + horiz. parity	2D degraded-DH write
write	two data disks	2D degraded-DH write
write	two parity disks	nonredundant write

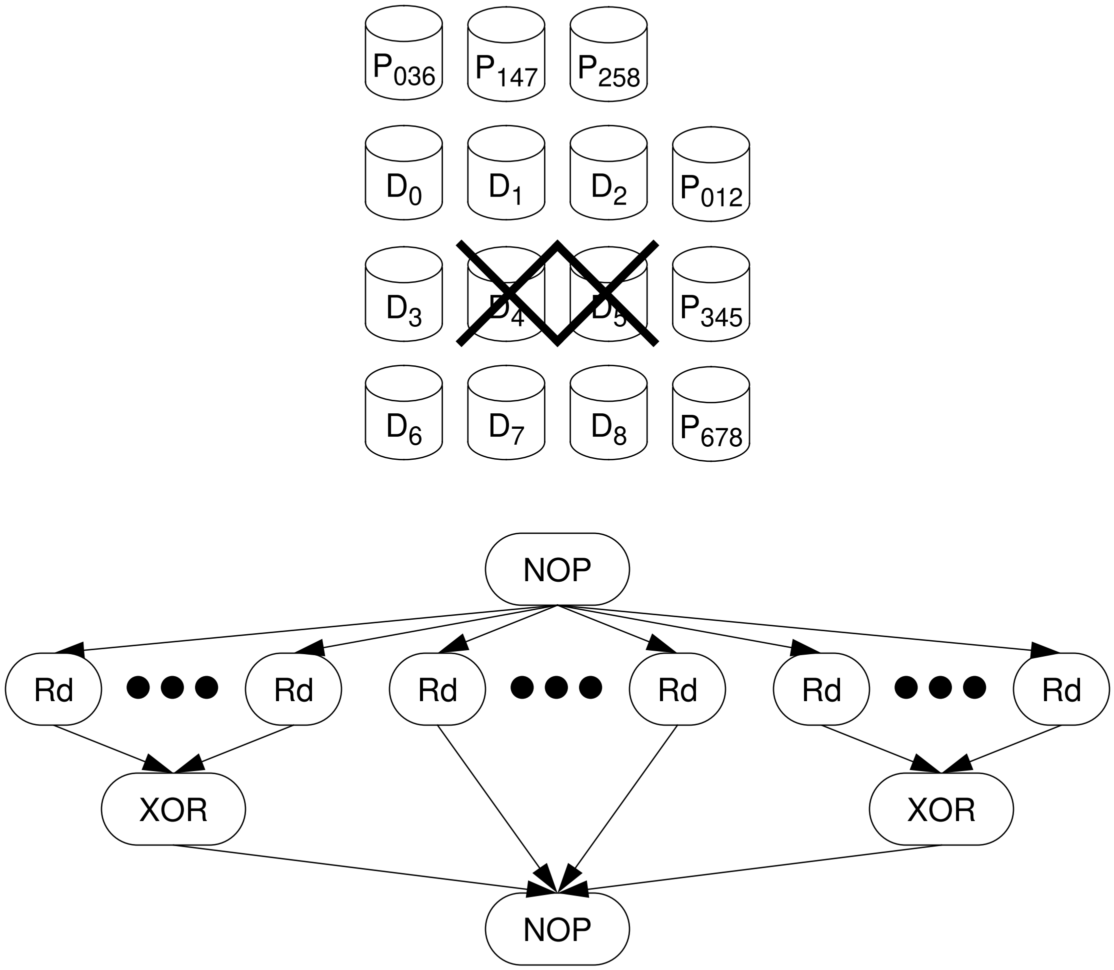


Figure A-16 2D double-degraded read graph

This operation is similar to the degraded-read operation used in RAID level 5 arrays, but extended to reconstruct missing data from two codewords. Assume that D_3-D_5 *are being read. The left-most* Rd-XOR *actions gather surviving information to reconstruct a unit of missing data (e.g.* D_4*). Similarly, the right-most* Rd-XOR *actions reconstruct the second missing unit of data (e.g.* D_5*). The center* Rd *actions retrieve data from non-failed disks (e.g.* D_3*).*

Because write operations must update parity for two codewords, they must perform additional work not found in the RAID level 5 graphs. The *2D small write* operation, illustrated in Figure A-17, uses the same principle (read-modify-write) as the RAID level 5 small-write operation (Figure A-5), but extends the approach to include vertical codewords. Additionally, this graph, as do all 2D write graphs, requires that all writes are enabled simultaneously to guarantee that all error scenarios are recoverable in implementation that employ forward error recovery. If backward error recovery is used, this constraint is removed because the graph may back up from any failure point.

Writing data to a codeword in which a single data disk has failed requires that the 2D small write operation be modified, replacing small-write style parity updates which involve the failed data to reconstruct-style updates. This operation, called a *2D degraded write*, is illustrated in Figure A-18.

If, in instead of a failed data disk, a disk containing vertical parity has failed, the 2D small-write operation is used, with the update of the failed vertical parity eliminated. If, the disk containing horizontal parity has failed, the update of horizontal parity is removed from the 2D small-write operation. This variant, called the *2D degraded-H write* operation, is illustrated in Figure A-19.

If both a data disk and the disk containing horizontal parity have failed, the write is performed using a *2D degraded-DH write* operation, illustrated in Figure A-20. This operation is essentially a 2D degraded write with the update of horizontal parity removed. Similarly, if a data disk and a disk containing vertical parity is removed, the 2D degraded write operation is modified and the update of vertical parity which protected the failed data disk is removed. This variant, called the *2D degraded-DV write* operation, is illustrated in Figure A-21. If the data and vertical parity failures don't overlap, a combination of the 2D degraded write and 2D degraded-V

Finally, I point out that many other variants of this basic set of graphs are possible. Hopefully, this appendix has provided you, the reader, enough insight to begin constructing these operations autonomously.

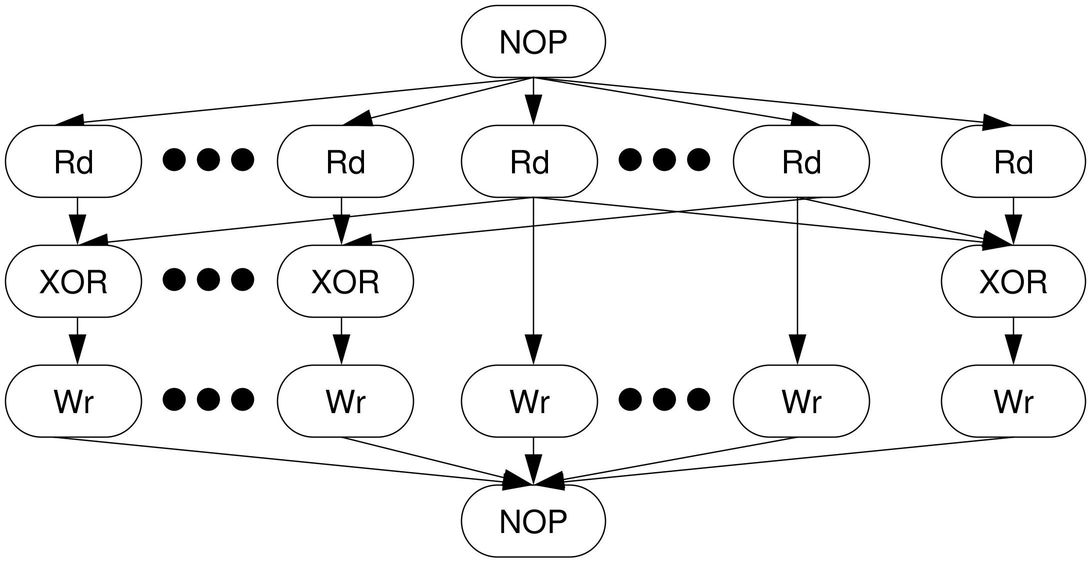


Figure A-17 2D small-write graph

This operation writes data to one or more blocks of a single horizontal codeword and employs the small-write method of updating associated parity. The right-most Rd-XOR-Wr *chain is used to update horizontal parity. The center* Rd-Wr *chains represent the reading of old data and the writing of new data. The left-most* Rd-XOR-Wr *chains represent the updates of vertical parity. Notice that these* XOR *actions depend only on one block of old data while the* XOR *used to compute horizontal parity depends on all "old data" blocks.*

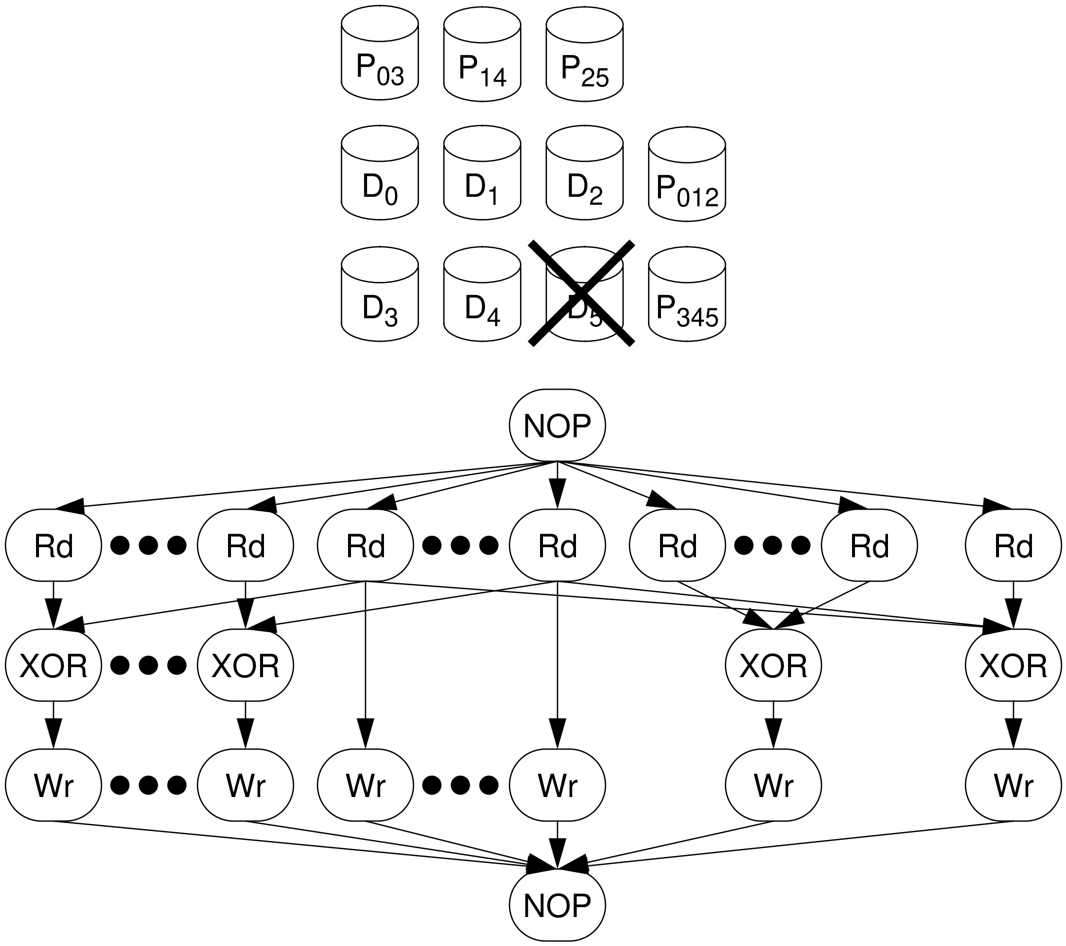


Figure A-18 2D degraded-write graph

To write data to D_4 *and* D_5*, a 2D degraded-data operation is used. This operation uses the small-write algorithm to update* P_{147} *and the reconstruct-write algorithm to update* P_{258} *and* P_{345}.

The Rd-XOR-Wr *chains on the left represent the read-modify-write of the vertical parity of non-failed data disks (e.g.* P_{14}*). The left-center* Rd *nodes gather old horizontal data (e.g.* D_3*). The left-center* Wr *nodes write new data (e.g.* D_4*). The right-center* Rd-XOR-Wr *chain reads old vertical data of the disk that failed (e.g.* D_2*) and updates vertical parity (e.g.* P_{25}*). The right-most* Rd-XOR-Wr *chain updates horizontal parity (e.g.* P_{345}*).*

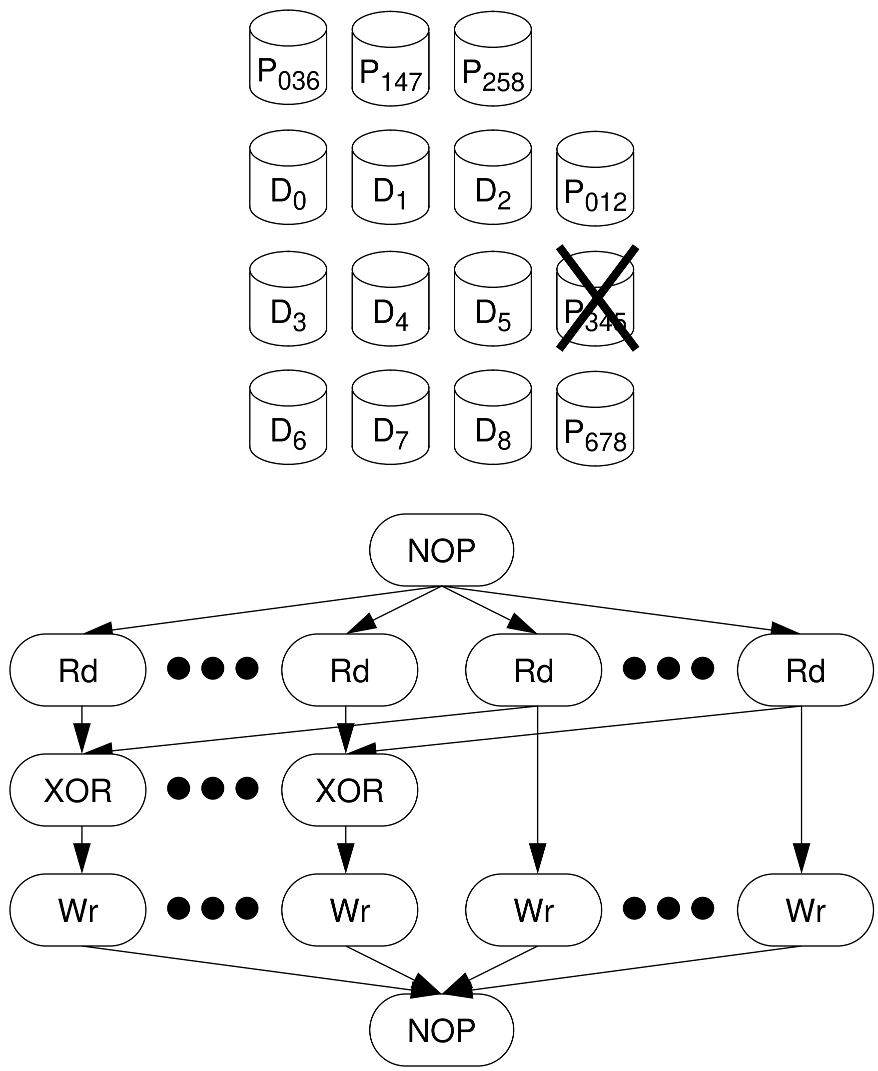


Figure A-19 2D degraded-H write graph

Similar to 2D small write but with the update of horizontal parity removed. Assuming that D_4 *and* D_5 *are to be written, the* Rd-XOR-Wr *chains on the left update vertical parity (e.g.* P_{147} *and* P_{258}*). The* Rd-Wr *chains on the right read and write data (e.g.* D_4 *and* D_5*).*

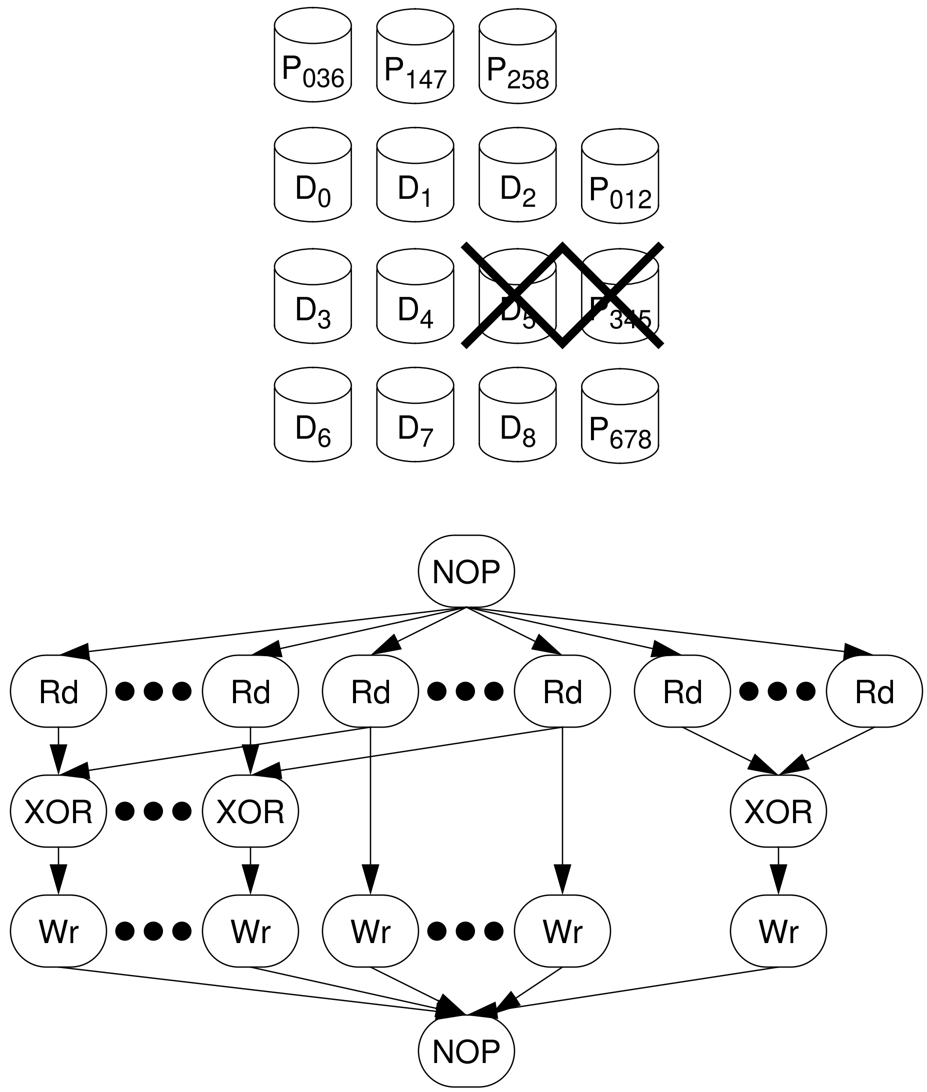


Figure A-20 2D degraded-DH write graph

Same as 2D degraded write but with update of horizontal parity removed.

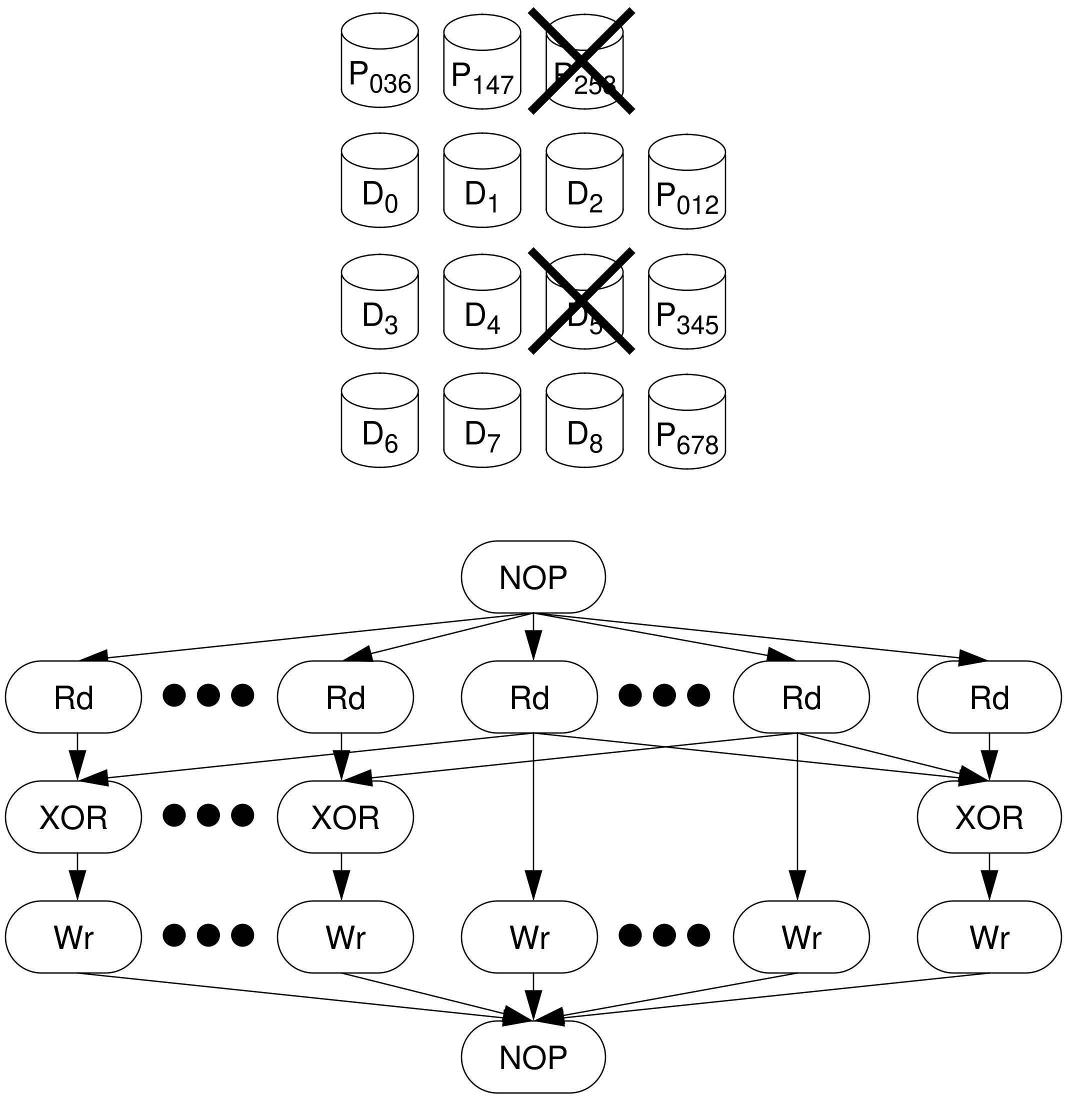


Figure A-21 2D degraded-DV write graph

Same as 2D degraded write but with update of vertical parity of failed data disk removed.

Appendix B: Modifying Graphs for Roll-Away Recovery

This appendix presents the flow graphs created in Appendix A but adapted for roll-away error recovery. This presentation assumes an understanding of the graph structure, which was explained in Appendix A.

The rules for placing a commit node in a graph were described in Section 5.3.3. 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.

These structure of these graphs is identical to those used in the roll-away error recovery experiments of Chapter 5. The graphs are presented in the same order that they were introduced in Appendix A.

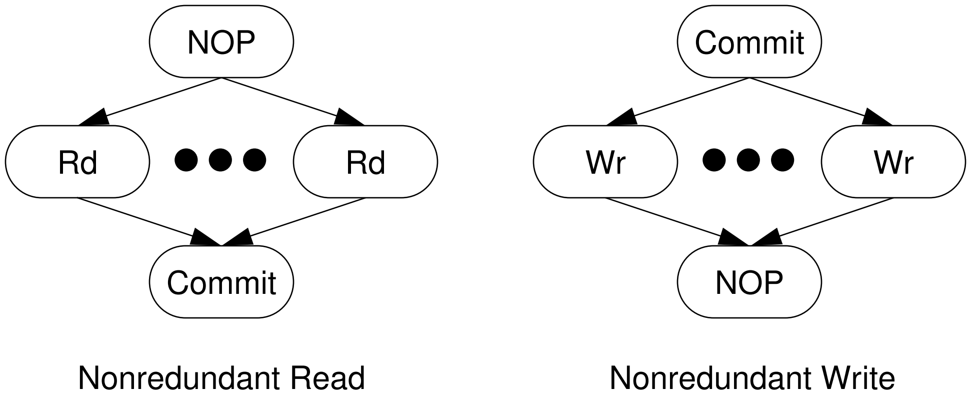


Figure B-1 Nonredundant graphs

The NOP *sink node of the nonredundant read graph is replaced by a* Commit *node. The* NOP *source node of the nonredundant write graph is replaced by a* Commit *node.*

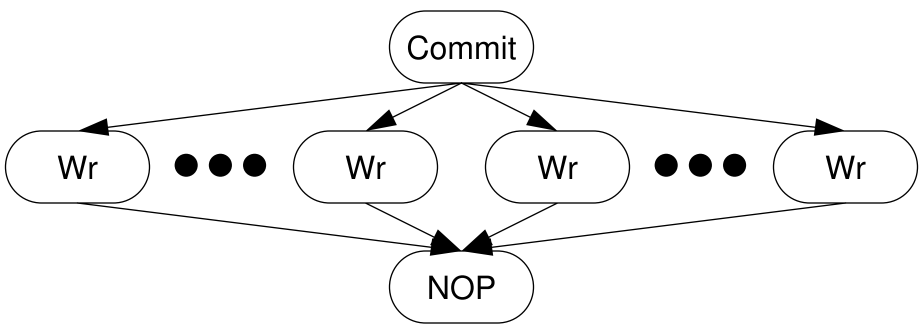


Figure B-2 Mirrored-write graph

The NOP *source node of the nonredundant write graph is replaced by a* Commit *node.*

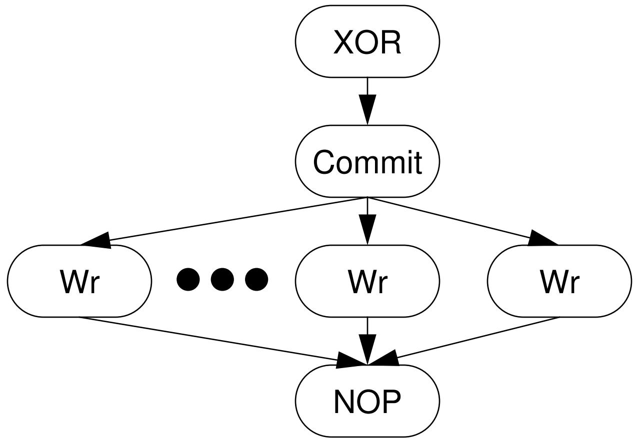


Figure B-3 Large-write graph

Instead of writing new data concurrently with the computation of new parity, a Commit *node is inserted to block the writes of new data until the* XOR *node has completed execution.*

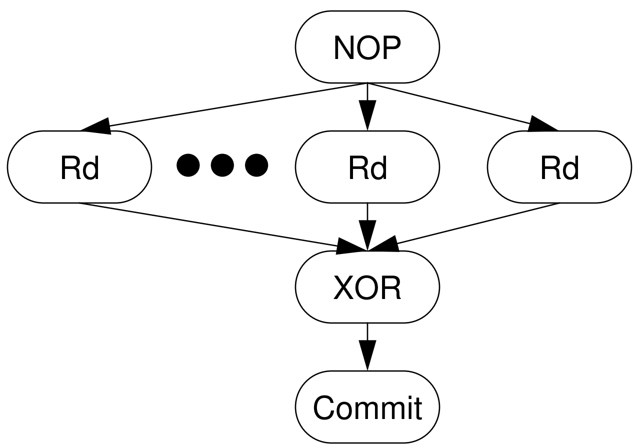


Figure B-4 Degraded-read graph

A Commit node has been added as to the end of the graph. Remember: reaching the Commit node implies that the graph will complete successfully.

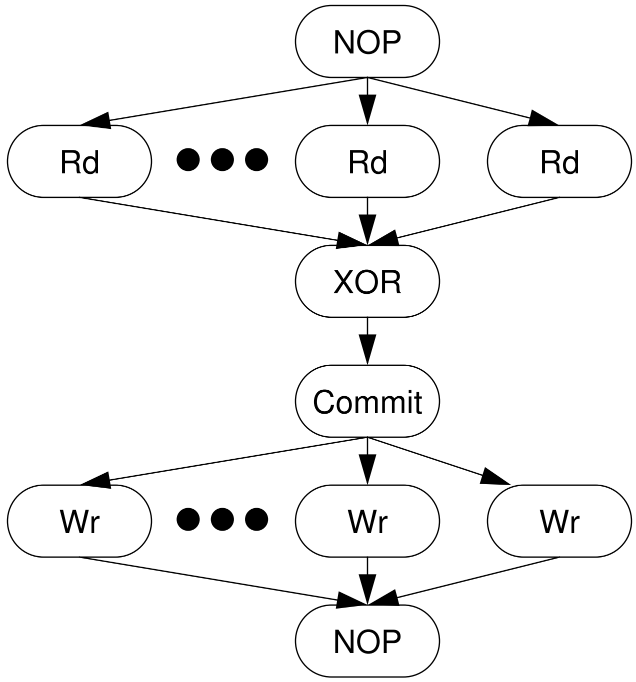


Figure B-5 Small-write graph

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.*

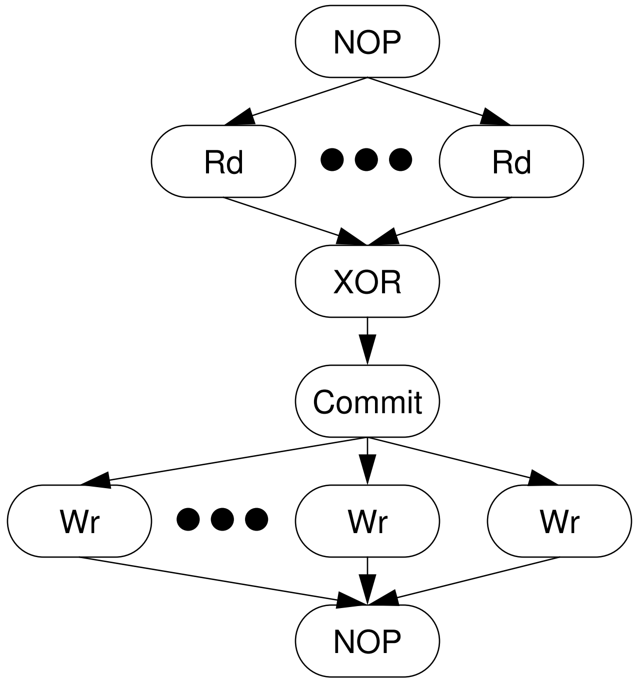


Figure B-6 Reconstruct-write graph

The middle NOP *was replaced with a* Commit *node which prevents the writes of new data from being executed until new parity has been computed.*

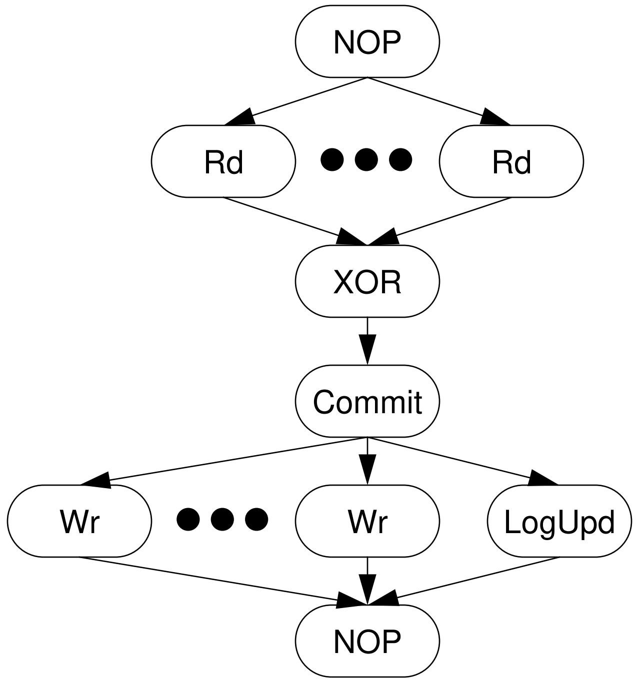


Figure B-7 Parity-logging small-write graph

A Commit *node is inserted to prevent writes of new data from proceeding until all reads of old data have completed and new parity has been computed.*

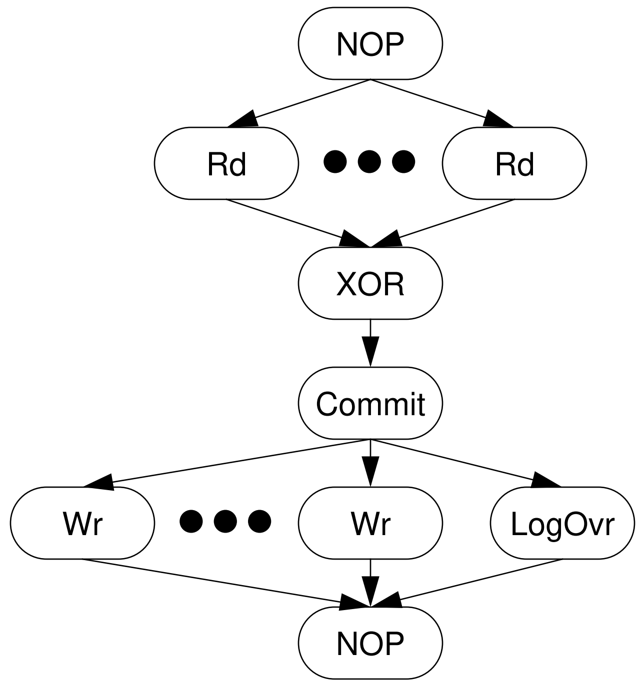


Figure B-8 Parity-logging reconstruct-write graph

The middle NOP *node, which prevented writes of new data from proceeding until all* Rd *nodes had completed has been replaced by a* Commit *node which prevents writes from proceeding until all* Rd *nodes have completed and the parity update record has been computed.*

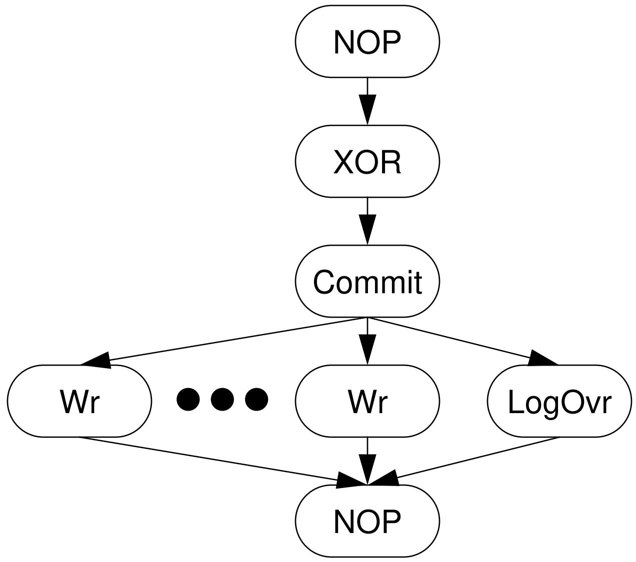


Figure B-9 Parity-logging large-write graph

Instead of writing new data concurrently with the computation of the parity overwrite record, a Commit *node is inserted to block the writes of new data until the* XOR *node has completed execution.*

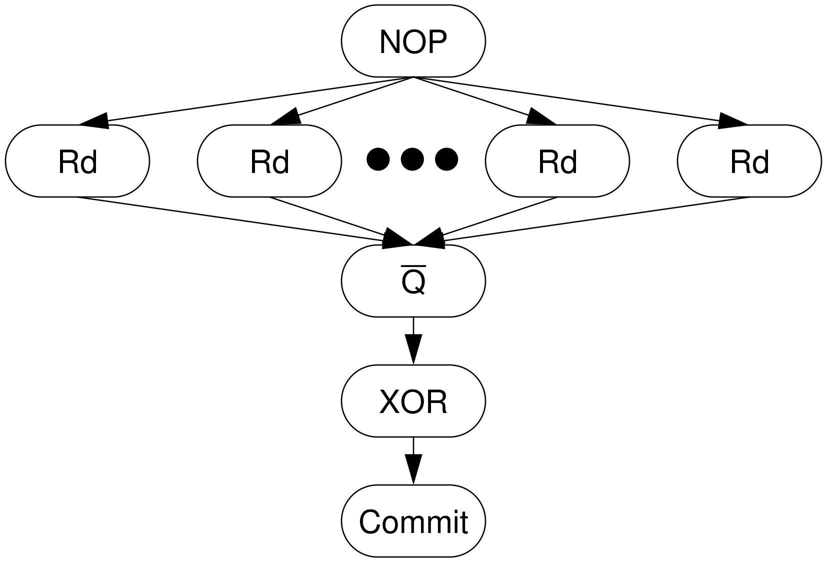


Figure B-10 PQ double-degraded read graph

A Commit node has been added as to the end of the graph. Remember: reaching the Commit node is reached implies that the graph will complete successfully.

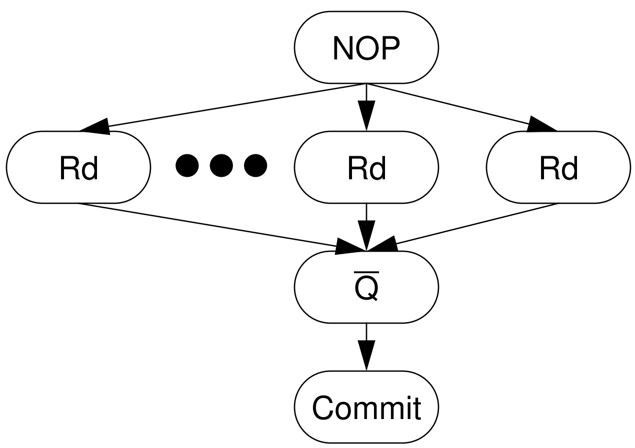


Figure B-11 PQ degraded-DP-read graph

The NOP *sink node was replaced by a* Commit *node.*

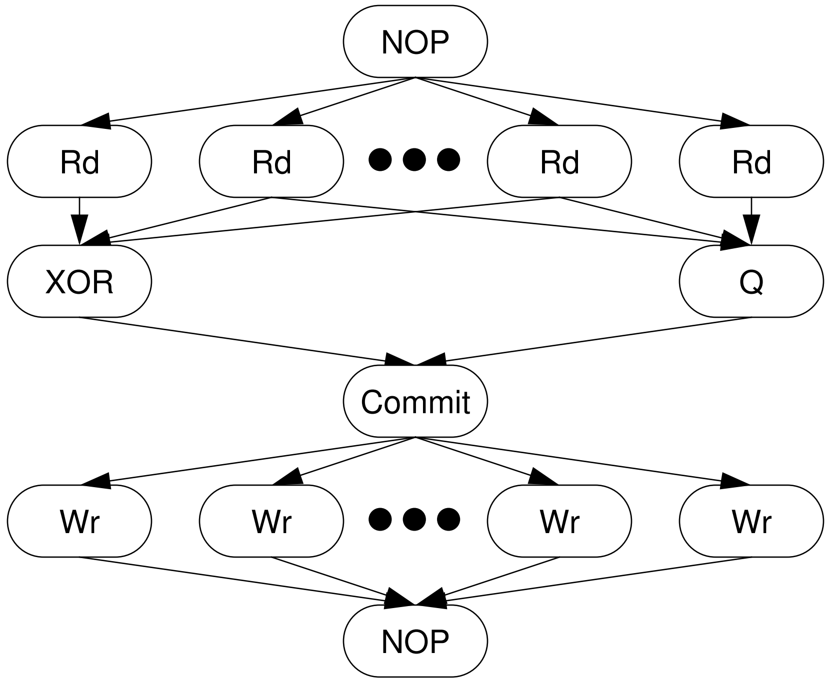


Figure B-12 PQ small-write graph

A Commit *node was added to block all writes from initiating until all new symbols (data, parity, and Q) have been computed.*

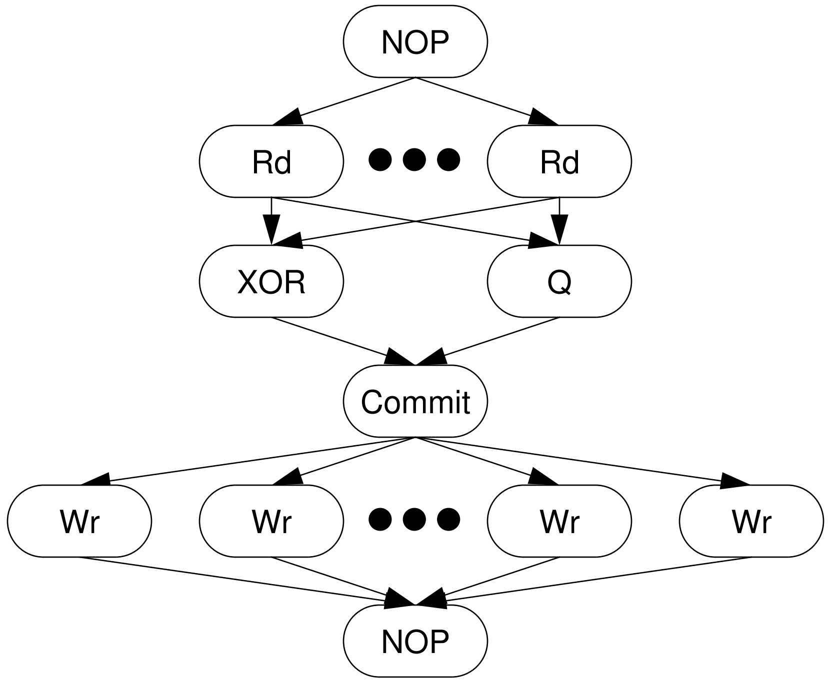


Figure B-13 PQ Reconstruct-write graph

Replaced central NOP *node with a Commit node which blocks all Wr nodes from executing until all new symbols have been computed.*

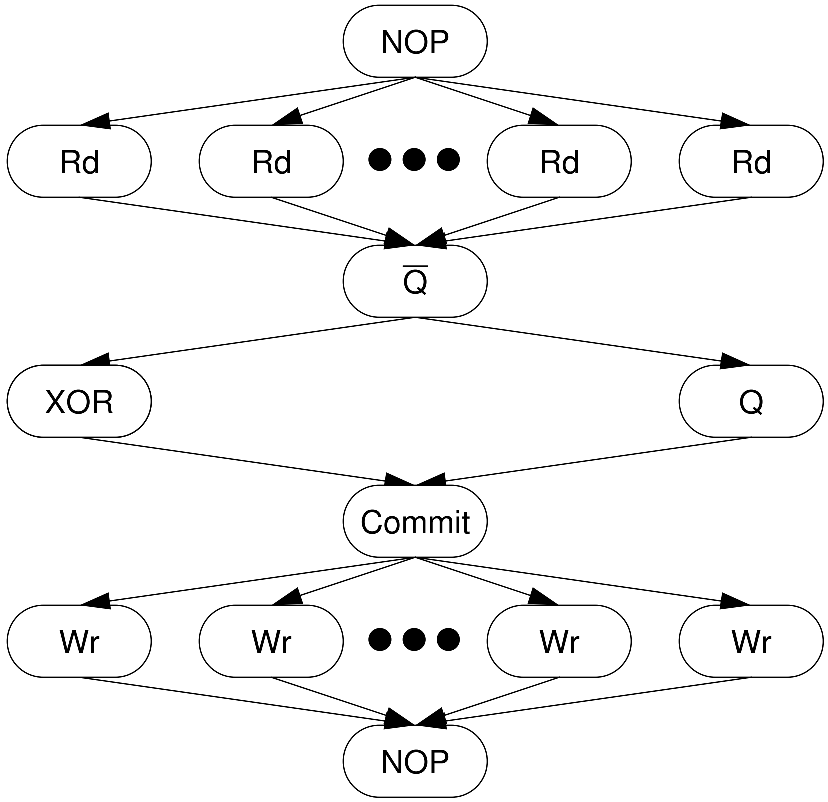


Figure B-14 PQ double-degraded write graph

A Commit *node was added to prevent* Wr *actions from executing before the* XOR *and* Q *nodes have completed.*

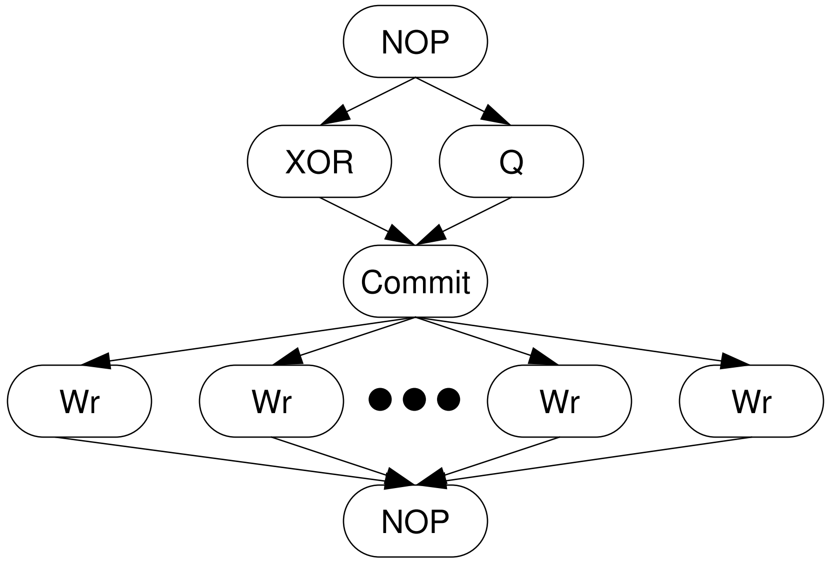


Figure B-15 PQ large-write graph

Instead of writing new data concurrently with the computation of the parity overwrite record, a Commit *node is inserted to block the writes of new data until the* XOR *and* Q *nodes have completed execution.*

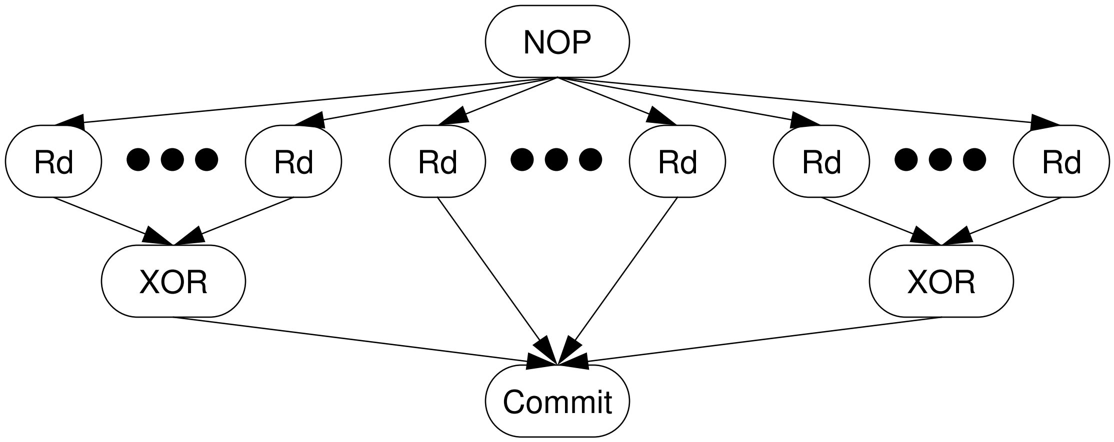


Figure B-16 2D double-degraded read graph

Replaced NOP *sink node with a commit node.*

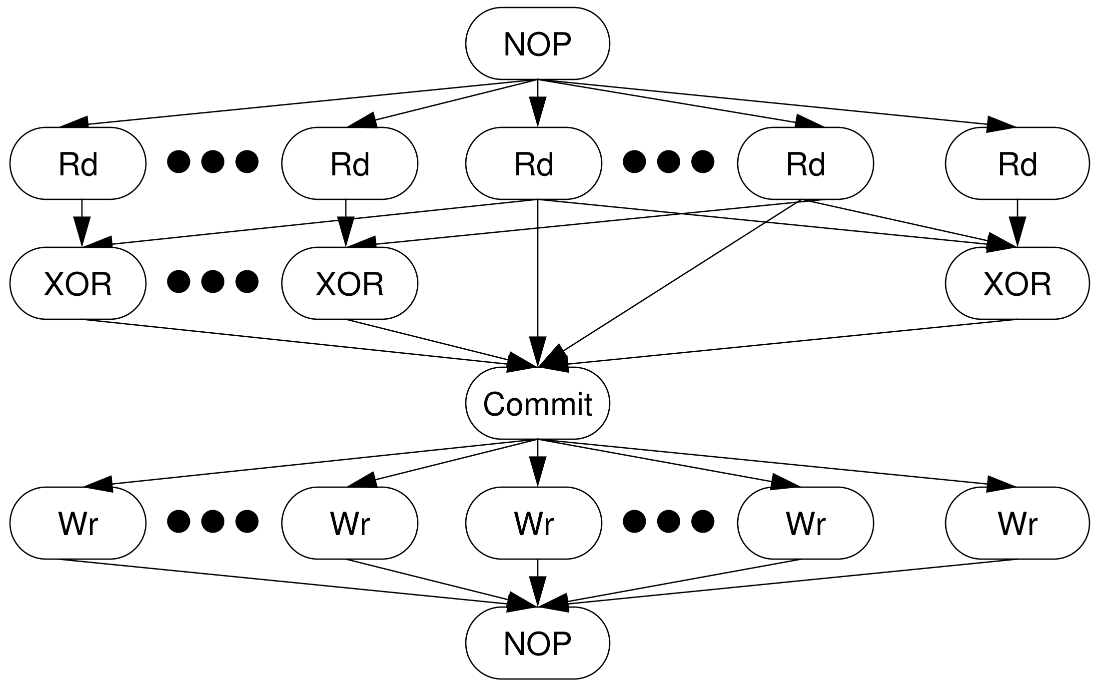


Figure B-17 2D small-write graph

A Commit *node was added, eliminating the need to undo the* Wr *actions.*

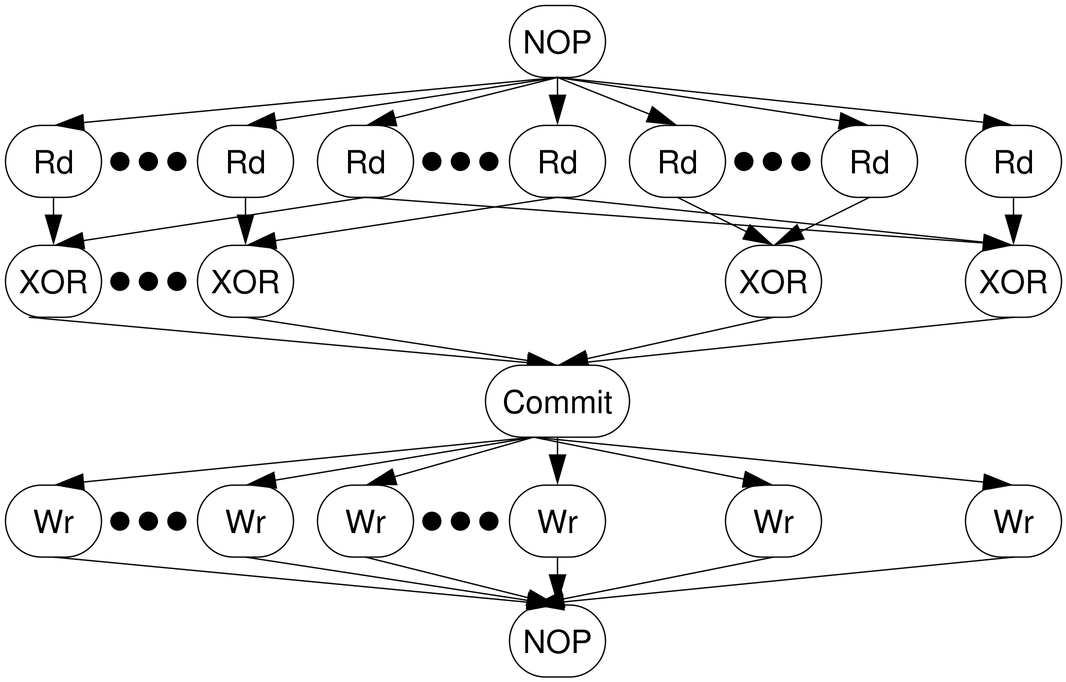


Figure B-18 2D degraded-write graph

A Commit *node was added, eliminating the need to undo the* Wr *actions.*

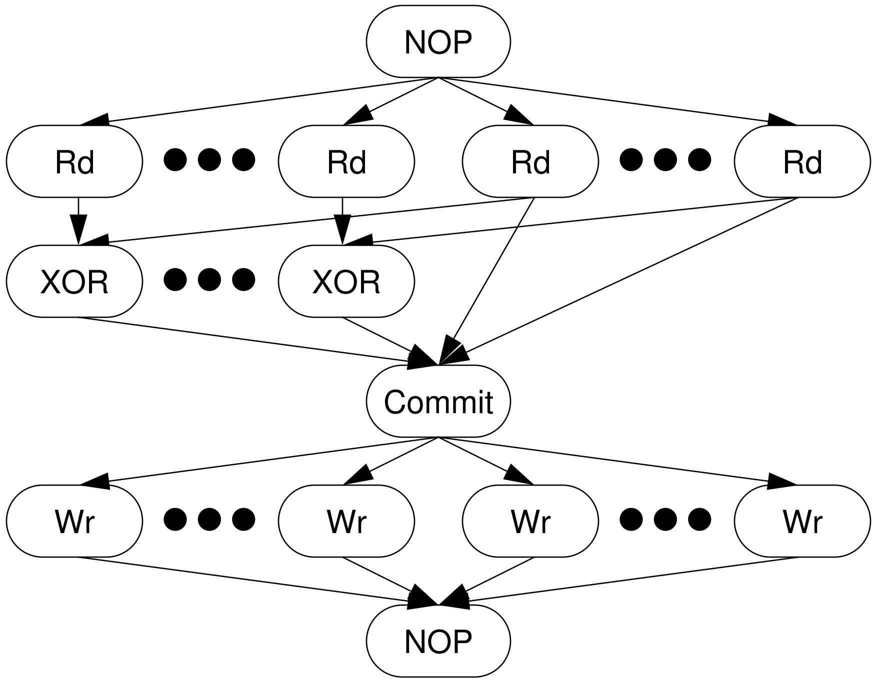


Figure B-19 2D degraded-H write graph

A Commit *node was added, eliminating the need to undo the* Wr *actions.*

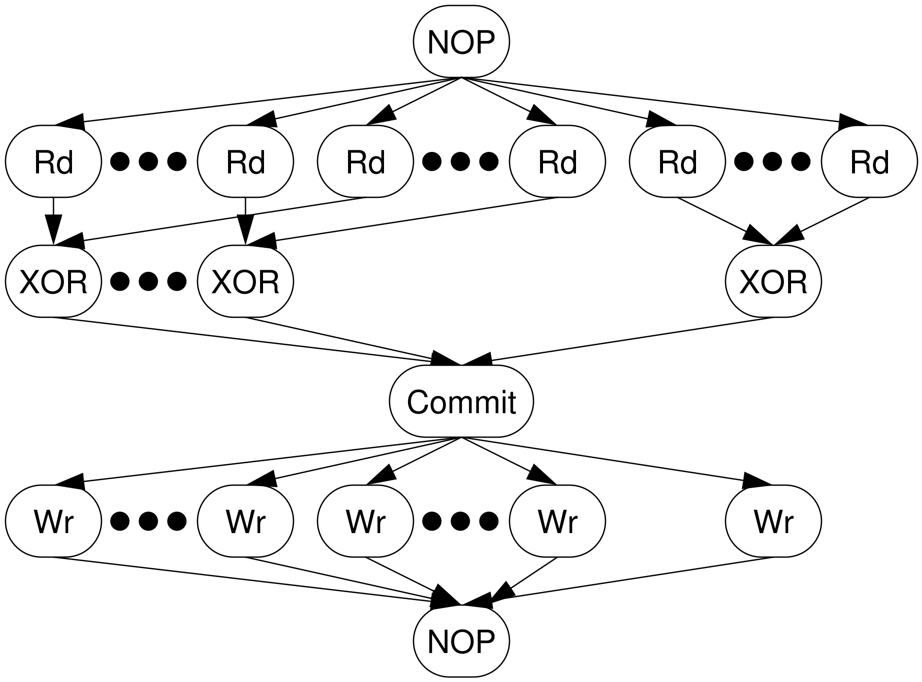


Figure B-20 2D degraded-DH write graph

A Commit *node was added, eliminating the need to undo the* Wr *actions.*

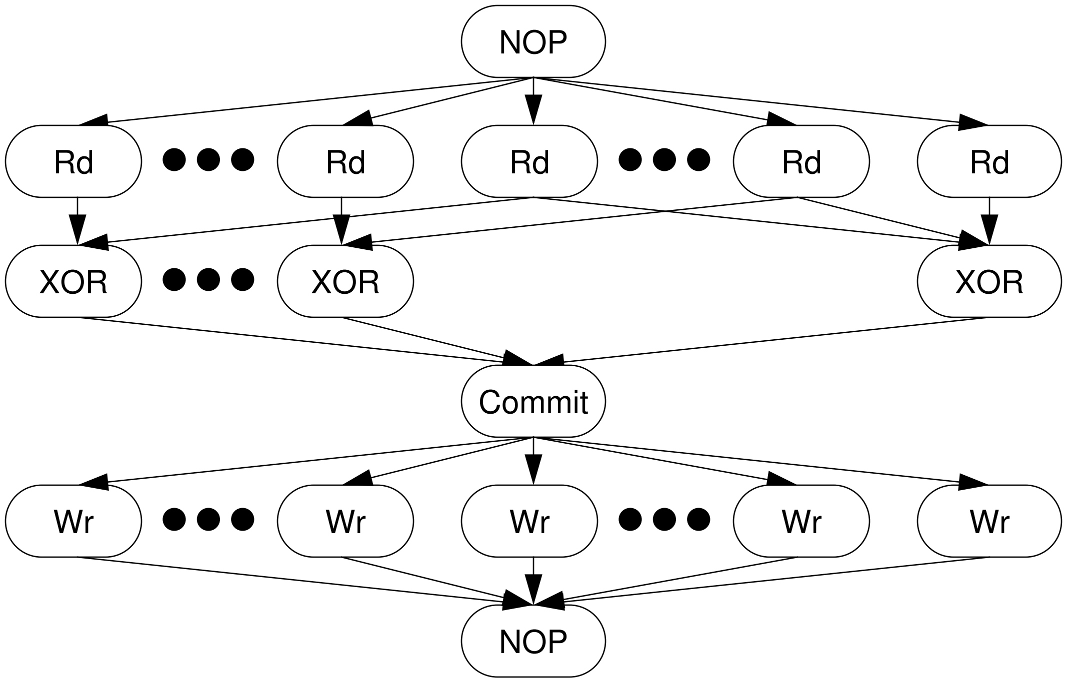


Figure B-21 2D degraded-DV write graph

A Commit *node was added, eliminating the need to undo the* Wr *actions.*

Appendix C: Data

This appendix contains the algorithm that was used to insert a commit node in the example of Section 5.3.3.2 (Section C.1), the raw data for each figure presented in the dissertation (Section C.2) as well as a sample configuration file (Section C.3). Table C-1 correlates the figure numbers with the tables which contain the raw data presented in the figure.

Table C-1 Cross-reference of performance figures and raw data

Performance Figure		**Performance Data**	
Figure Number	**Page Number**	**Table Number**	**Page Number**
4-5(a)	94	C-2	196
4-5(b)	94	C-3	196
4-4	93	C-4	197
4-6	96	C-5	198
4-7	98	C-6	206
5-1	102	C-7	214
5-10	117	C-8	222

C.1 Algorithm for Inserting a Commit Point Into a Write Graph

The following program creates a RAID level 5 small-write graph without a commit node, prints the graph, inserts a commit node, and then prints the new graph.

```
/*
 * code for inserting commit node into a graph
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>

#define MAX_PARENTS 8
#define MAX_CHILDREN 8
#define MAX_NODES 100



typedef struct node_s {
  char* name;
  int num_p; /* number of parents */
  int parent[MAX_PARENTS]; /* ptrs to parents */
  int num_c; /* number of children */
  int child[MAX_CHILDREN]; /* ptrs to children */
  int dat_dep_child; /* 1 if one or more children are data
dependent */
} node_t;

typedef struct dag_s {
  int length; /* number of nodes in the dag */
  int cmt_ptr; /* ptr to the commit node */
} dag_t;

node_t node[MAX_NODES];
dag_t dag;




static void build_dag()
{
  int i;
```

```
/* build a RAID level 5 small-write graph */

/* head of dag is always node 0
   tail of dag is always node (length - 1) */

dag.length = 9;
dag.cmt_ptr = dag.length;

node[0].name = "Lock";
node[0].num_p = 0;
node[0].num_c = 1;
node[0].child[0] = 1;
node[0].dat_dep_child = 0;

node[1].name = "MemA";
node[1].num_p = 1;
node[1].parent[0] = 0;
node[1].num_c = 2;
node[1].child[0] = 2;
node[1].child[1] = 3;
node[1].dat_dep_child = 1;

node[2].name = "Rd";
node[2].num_p = 1;
node[2].parent[0] = 1;
node[2].num_c = 2;
node[2].child[0] = 4;
node[2].child[1] = 5;
node[2].dat_dep_child = 1;

node[3].name = "Rd";
node[3].num_p = 1;
node[3].parent[0] = 1;
node[3].num_c = 1;
node[3].child[0] = 5;
node[3].dat_dep_child = 1;

node[4].name = "Wr";
node[4].num_p = 1;
node[4].parent[0] = 2;
node[4].num_c = 1;
node[4].child[0] = 8;
node[4].dat_dep_child = 0;

node[5].name = "XOR";
node[5].num_p = 2;
```

```
  node[5].parent[0] = 2;
  node[5].parent[1] = 3;
  node[5].num_c = 1;
  node[5].child[0] = 6;
  node[5].dat_dep_child = 1;

  node[6].name = "Wr";
  node[6].num_p = 1;
  node[6].parent[0] = 5;
  node[6].num_c = 1;
  node[6].child[0] = 7;
  node[6].dat_dep_child = 0;

  node[7].name = "MemD";
  node[7].num_p = 1;
  node[7].parent[0] = 6;
  node[7].num_c = 1;
  node[7].child[0] = 8;
  node[7].dat_dep_child = 0;

  node[8].name = "Unlock";
  node[8].num_p = 2;
  node[8].parent[0] = 4;
  node[8].parent[1] = 7;
  node[8].num_c = 0;
  node[8].dat_dep_child = 0;

  /* initialize commit node but do not connect to dag */
  node[9].name = "Commit";
  node[9].num_p = 0;
  node[9].parent[0] = 0;
  node[9].num_c = 0;
  node[9].dat_dep_child = 0;
}



static void print_node(int node_id)
{
  int i;

  printf("Node ID: %d\n",node_id);
  printf("     name: %s\n",node[node_id].name);
  if (node[node_id].num_p > 0) {
    printf("     parents:");
    for (i = 0; i < node[node_id].num_p; i++)
```

```
      printf(" %d", node[node_id].parent[i]);
    printf("\n");
  }
  else
    printf("     parents: none\n");

  if (node[node_id].num_c > 0) {
    printf("     children:");
    for (i = 0; i < node[node_id].num_c; i++)
      printf(" %d", node[node_id].child[i]);
    printf("\n");
  }
  else
    printf("     children: none\n");

  if (node[node_id].dat_dep_child)
    printf("     * node originally had data-dependent chil-
dren\n\n");
  else
    printf("\n");
}



static void print_dag()
{
  int i;

  printf("\n\nPrinting %d-node DAG\n\n",dag.length);
  for (i = 0; i < dag.length; i++)
    print_node(i);
}



static void insert_commit(int p, int c)
{
  int i, done;

  /* insert the comment node between the nodes p and c */
  /* p is the parent of c */

  /* commit node has an additional child, c */
  /* make sure c isn't already a child of the commit node */
  done = 0;
  for (i = 0; i < node[dag.cmt_ptr].num_c; i++)
```

```
    if (node[dag.cmt_ptr].parent[i] == c)
      done = 1;
  if (!done) {
    node[dag.cmt_ptr].child[node[dag.cmt_ptr].num_c] = c;
    node[dag.cmt_ptr].num_c++;
  }

  /* c's only parent is the commit node */
  node[c].num_p = 1;
  node[c].parent[0] = dag.cmt_ptr;

  /* p replaces child c with commit node */
  for (i = 0; i < node[p].num_c; i++)
    if (node[p].child[i] == c)
      node[p].child[i] = dag.cmt_ptr;

  /* p is added to the list of parents in the commit node */
  /* make sure p isn't already a parent of the commit node */
  done = 0;
  for (i = 0; i < node[dag.cmt_ptr].num_p; i++)
    if (node[dag.cmt_ptr].parent[i] == p)
      done = 1;
  if (!done) {
    node[dag.cmt_ptr].parent[node[dag.cmt_ptr].num_p] = p;
    node[dag.cmt_ptr].num_p++;
  }

}



static void explore_branch(int node_id)
{
  int i, p_id;

  /* recursively look at all of node_id's parents */
  /* stop searching if a data dependency is encountered */
  /* and then insert the commit node between nodes */

  for (i = 0; i < node[node_id].num_p; i++) {
    p_id =  node[node_id].parent[i];
    if (!node[p_id].dat_dep_child)
      explore_branch(p_id);
    else
      insert_commit(p_id, node_id);
  }
```

```
}



static void add_commit()
{
  int node_id;

  /* look for first sign of data dependencies */
  /* begin with sink node and work towards source node */

  explore_branch(dag.length - 1);
  dag.length++; /* commit node now a part of the dag */
}



int main(int argc, char **argv)
{
  build_dag();
  printf("Original DAG, without commit node:\n");
  print_dag();
  add_commit();
  printf("Final DAG, with commit node:\n");
  print_dag();
}
```

C.2 Raw Data

Table C-2 Comparing RAIDframe to a hand-crafted implementation

Number of Disks	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval			
	RAIDframe		Striping Driver	
	Read	Write	Read	Write
1	53.55 ± 0.48 36.93 ± 0.35	51.17 ± 0.12 38.65 ± 0.10	53.12 ± 1.05 36.90 ± 0.22	51.31 ± 0.62 38.55 ± 0.47
2	92.12 ± 0.74 42.02 ± 0.16	87.64 ± 0.62 44.34 ± 0.30	92.77 ± 1.20 42.10 ± 0.48	90.19 ± 1.38 43.47 ± 0.48
4	200.30 ± 2.28 38.36 ± 0.50	193.40 ± 2.19 40.00 ± 0.31	199.64 ± 0.64 38.31 ± 0.47	193.64 ± 3.12 39.85 ± 0.71
6	318.88 ± 2.24 35.55 ± 0.48	309.52 ± 6.68 36.69 ± 0.55	321.85 ± 4.46 35.39 ± 0.23	307.62 ± 9.02 36.86 ± 0.41
8	412.93 ± 4.38 36.05 ± 0.16	409.47 ± 4.78 36.70 ± 0.33	421.12 ± 4.20 35.68 ± 0.22	404.78 ± 1.13 37.20 ± 0.20
10	472.44 ± 7.51 39.27 ± 0.52	462.30 ± 3.14 40.48 ± 0.35	473.99 ± 9.13 39.45 ± 0.54	461.50 ± 1.53 40.62 ± 0.16

Table C-3 Comparing RAIDframe to a hand-crafted implementation

Number of Disks	CPU Utilization (%) ± 95% confidence interval			
	RAIDframe		Striping Driver	
	Read	Write	Read	Write
1	4.53 ± 0.13	4.67 ± 0.10	2.84 ± 0.05	2.75 ± 0.02
2	7.69 ± 0.24	7.78 ± 0.13	4.80 ± 0.11	4.67 ± 0.05
4	16.82 ± 0.12	17.12 ± 0.04	10.39 ± 0.07	10.17 ± 0.21
6	27.21 ± 0.27	27.73 ± 0.49	16.83 ± 0.28	16.45 ± 0.36
8	35.77 ± 0.37	36.90 ± 0.22	22.10 ± 0.28	21.63 ± 0.30
10	41.73 ± 0.97	42.30 ± 0.39	25.66 ± 0.41	25.26 ± 0.16

Table C-4 Single disk performance of striper and RAIDframe

Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval			
	RAIDframe		Striping Driver	
	Read	Write	Read	Write
1	56.6 ± 0.62 17.26 ± 0.10	54.31 ± 0.09 18.08 ± 0.03	57.67 ± 0.40 17.00 ± 0.09	55.10 ± 0.18 17.86 ± 0.07
2	62.38 ± 0.43 31.64 ± 0.19	58.94 ± 0.10 33.52 ± 0.06	62.55 ± 0.19 31.60 ± 0.08	59.01 ± 0.22 33.52 ± 0.14
5	67.66 ± 0.12 72.81 ± 0.22	65.65 ± 0.40 75.01 ± 0.73	68.14 ± 0.33 72.40 ± 0.29	66.28 ± 0.20 74.28 ± 0.30
10	69.78 ± 0.34 139.65 ± 0.47	68.83 ± 0.44 142.40 ± 0.77	69.19 ± 0.45 141.70 ± 0.80	67.84 ± 0.40 145.03 ± 1.44
15	73.08 ± 0.25 200.05 ± 0.55	71.52 ± 0.69 203.20 ± 1.67	72.48 ± 0.49 201.62 ± 1.36	70.79 ± 0.08 206.16 ± 0.80
20	75.43 ± 0.28 256.26 ± 2.90	73.93 ± 0.50 261.44 ± 2.57	74.63 ± 0.31 260.21 ± 0.81	73.04 ± 0.27 264.79 ± 0.51
30	78.05 ± 0.25 369.21 ± 2.74	76.60 ± 0.25 374.47 ± 1.35	77.55 ± 0.62 371.23 ± 2.47	76.41 ± 0.48 377.27 ± 2.16
40	79.68 ± 0.28 478.04 ± 6.87	78.80 ± 0.23 484.56 ± 1.01	79.50 ± 0.35 481.85 ± 2.55	78.20 ± 0.41 487.45 ± 2.94

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 0	1	95.65 ± 0.32	68.53 ± 0.81	70.60 ± 1.37
		9.69 ± 0.03	14.37 ± 0.17	13.78 ± 0.20
	2	139.94 ± 0.54	129.68 ± 0.15	134.41 ± 1.61
		10.34 ± 0.02	15.06 ± 0.02	14.43 ± 0.18
	5	351.18 ± 1.63	273.99 ± 2.30	281.41 ± 1.38
		12.29 ± 0.04	17.47 ± 0.15	16.80 ± 0.12
	10	561.78 ± 1.20	425.43 ± 3.83	435.68 ± 5.55
		16.18 ± 0.10	22.04 ± 0.26	21.15 ± 0.22
	15	671.88 ± 8.55	508.25 ± 4.36	521.78 ± 5.17
		20.37 ± 0.32	27.21 ± 0.22	26.00 ± 0.09
	20	742.96 ± 11.82	559.80 ± 6.83	558.87 ± 22.19
		24.71 ± 0.76	32.19 ± 0.42	31.06 ± 0.54
	30	817.81 ± 9.85	601.80 ± 1.80	635.85 ± 6.59
		33.64 ± 0.36	42.26 ± 0.32	41.27 ± 0.59
	40	870.37 ± 3.04	611.49 ± 4.82	657.91 ± 24.20
		42.07 ± 0.20	52.84 ± 0.86	51.57 ± 0.44

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 1	1	95.50 ± 0.54	68.29 ± 0.17	70.86 ± 0.58
		9.97 ± 0.06	14.42 ± 0.05	13.79 ± 0.11
	2	163.81 ± 0.91	131.99 ± 0.52	137.14 ± 1.26
		10.04 ± 0.05	14.76 ± 0.08	14.11 ± 0.06
	5	386.96 ± 3.86	290.07 ± 0.85	306.71 ± 2.11
		11.08 ± 0.09	16.44 ± 0.07	15.49 ± 0.07
	10	630.33 ± 1.14	455.63 ± 2.55	483.27 ± 7.33
		14.38 ± 0.06	20.31 ± 0.13	19.02 ± 0.14
	15	742.95 ± 6.48	540.03 ± 2.43	567.02 ± 9.75
		18.45 ± 0.19	24.72 ± 0.14	23.90 ± 0.24
	20	808.30 ± 3.93	566.24 ± 3.38	615.53 ± 4.79
		22.85 ± 0.09	29.91 ± 0.43	28.98 ± 0.14
	30	866.52 ± 9.41	575.72 ± 1.70	663.27 ± 11.11
		31.89 ± 0.33	41.59 ± 0.91	39.73 ± 0.51
	40	902.41 ± 8.16	575.11 ± 1.70	675.66 ± 7.06
		40.63 ± 0.11	54.56 ± 0.65	50.70 ± 0.29

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 4	1	95.99 ± 0.67	68.11 ± 0.38	70.66 ± 0.54
		9.92 ± 0.07	14.47 ± 0.09	13.82 ± 0.11
	2	159.95 ± 0.16	127.82 ± 0.27	132.72 ± 0.46
		10.53 ± 0.02	15.27 ± 0.03	14.61 ± 0.06
	5	343.05 ± 0.16	266.95 ± 1.70	276.01 ± 0.33
		12.60 ± 0.05	17.94 ± 0.13	17.22 ± 0.10
	10	531.25 ± 8.41	403.36 ± 2.36	414.83 ± 5.91
		16.99 ± 0.20	23.38 ± 0.14	22.42 ± 0.21
	15	626.28 ± 12.87	483.17 ± 1.90	494.29 ± 3.93
		22.04 ± 0.41	28.80 ± 0.15	27.72 ± 0.29
	20	687.60 ± 5.63	521.92 ± 4.38	533.38 ± 7.29
		26.83 ± 0.51	35.25 ± 0.48	33.65 ± 0.24
	30	757.43 ± 9.82	574.74 ± 4.02	594.30 ± 7.23
		36.57 ± 0.19	46.05 ± 0.79	44.90 ± 0.53
	40	794.92 ± 4.47	591.26 ± 4.75	610.14 ± 11.38
		46.35 ± 0.54	57.38 ± 0.94	56.71 ± 0.26

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 5	1	95.38 ± 0.54	68.18 ± 0.35	70.59 ± 0.21
		9.99 ± 0.06	14.45 ± 0.07	13.84 ± 0.03
	2	160.04 ± 1.12	128.24 ± 1.13	133.44 ± 0.46
		10.49 ± 0.08	15.22 ± 0.14	14.51 ± 0.02
	5	349.00 ± 6.98	270.41 ± 2.07	282.42 ± 4.04
		12.34 ± 0.19	17.69 ± 0.15	16.89 ± 0.19
	10	559.25 ± 0.47	415.55 ± 3.04	428.84 ± 11.72
		16.11 ± 0.04	22.59 ± 0.17	21.52 ± 0.12
	15	668.97 ± 6.47	504.88 ± 2.76	511.51 ± 11.37
		20.47 ± 0.06	27.29 ± 0.29	26.48 ± 0.29
	20	746.40 ± 4.02	549.31 ± 1.55	570.40 ± 16.52
		24.71 ± 0.30	32.54 ± 0.08	31.45 ± 0.35
	30	827.39 ± 11.28	584.73 ± 2.43	632.95 ± 13.58
		33.75 ± 0.44	42.98 ± 0.35	41.52 ± 1.15
	40	866.40 ± 11.03	587.51 ± 3.24	655.52 ± 5.13
		42.23 ± 0.20	54.54 ± 0.52	51.86 ± 0.50

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 6	1	92.76 ± 0.54 10.29 ± 0.06	65.77 ± 0.13 14.99 ± 0.03	67.93 ± 0.06 14.40 ± 0.01
	2	157.03 ± 0.65 10.86 ± 0.06	123.63 ± 1.02 15.80 ± 0.14	128.68 ± 1.08 15.06 ± 0.09
	5	337.84 ± 1.50 12.73 ± 0.12	262.25 ± 0.91 18.26 ± 0.05	272.77 ± 1.11 17.47 ± 0.10
	10	542.03 ± 3.60 16.83 ± 0.15	406.78 ± 3.50 23.05 ± 0.23	419.38 ± 10.10 22.19 ± 0.31
	15	652.23 ± 4.39 21.09 ± 0.28	485.14 ± 0.97 28.44 ± 0.06	506.79 ± 3.98 27.24 ± 0.23
	20	724.28 ± 9.49 25.12 ± 0.03	532.74 ± 2.97 33.65 ± 0.49	546.41 ± 6.06 32.35 ± 0.10
	30	800.19 ± 7.60 34.44 ± 0.40	562.90 ± 1.48 44.15 ± 0.35	607.25 ± 3.13 42.89 ± 0.27
	40	852.07 ± 5.83 43.15 ± 0.28	566.53 ± 0.97 57.16 ± 0.85	638.45 ± 22.79 53.74 ± 0.80

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
parity declustering	1	93.77 ± 0.45 10.17 ± 0.06	67.40 ± 0.32 14.63 ± 0.08	69.80 ± 0.13 14.00 ± 0.03
	2	158.58 ± 0.62 10.66 ± 0.04	127.12 ± 0.52 15.63 ± 0.06	132.39 ± 0.51 14.67 ± 0.07
	5	344.41 ± 3.40 12.48 ± 0.07	269.55 ± 0.69 17.75 ± 0.04	276.80 ± 3.29 17.10 ± 0.08
	10	556.15 ± 9.66 16.32 ± 0.21	412.13 ± 0.33 22.83 ± 0.05	427.86 ± 4.21 21.59 ± 0.23
	15	665.44 ± 4.28 20.72 ± 0.25	497.37 ± 7.64 27.58 ± 0.35	509.90 ± 4.00 26.46 ± 0.27
	20	728.87 ± 10.58 25.23 ± 0.16	544.82 ± 4.33 32.61 ± 0.25	562.73 ± 5.09 31.79 ± 0.35
	30	807.74 ± 11.51 34.02 ± 0.19	576.08 ± 1.88 43.17 ± 0.74	615.98 ± 17.31 42.20 ± 0.58
	40	855.15 ± 11.14 42.79 ± 0.79	578.48 ± 1.04 56.11 ± 0.44	652.66 ± 7.53 52.39 ± 0.20

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	**User**	**Kernel**
interleaved declustering	1	95.59 ± 0.79	68.26 ± 0.34	70.97 ± 0.17
		9.96 ± 0.08	14.43 ± 0.07	13.76 ± 0.04
	2	160.36 ± 0.04	128.87 ± 0.58	134.46 ± 0.48
		10.46 ± 0.05	15.13 ± 0.07	14.44 ± 0.05
	5	349.55 ± 0.58	274.26 ± 0.94	283.08 ± 2.23
		12.31 ± 0.04	17.43 ± 0.07	16.77 ± 0.11
	10	556.33 ± 7.60	415.49 ± 2.20	436.73 ± 6.19
		16.37 ± 0.23	22.57 ± 0.10	21.17 ± 0.25
	15	665.21 ± 3.38	503.34 ± 2.51	518.33 ± 5.65
		20.56 ± 0.19	27.36 ± 0.17	26.39 ± 0.15
	20	743.47 ± 5.52	548.57 ± 3.69	571.88 ± 13.38
		24.59 ± 0.08	32.41 ± 0.30	31.27 ± 0.65
	30	807.02 ± 2.59	573.39 ± 1.94	625.22 ± 11.38
		22.76 ± 0.45	43.58 ± 0.17	41.81 ± 0.75
	40	871.36 ± 6.54	577.79 ± 0.89	661.86 ± 11.70
		41.96 ± 0.09	55.33 ± 0.65	51.60 ± 0.88

Table C-5 Small-read performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
chained declustering	1	95.78 ± 0.39 9.94 ± 0.04	68.15 ± 0.41 14.46 ± 0.09	70.94 ± 0.31 13.77 ± 0.06
	2	160.19 ± 0.16 10.47 ± 0.02	127.91 ± 0.42 15.25 ± 0.05	133.76 ± 1.36 14.51 ± 0.13
	5	351.18 ± 1.63 12.29 ± 0.04	272.91 ± 0.16 17.52 ± 0.03	283.80 ± 4.39 16.75 ± 0.12
	10	561.78 ± 1.20 16.18 ± 0.10	419.80 ± 3.85 22.31 ± 0.21	432.97 ± 4.53 21.21 ± 0.18
	15	671.88 ± 8.55 20.37 ± 0.32	506.52 ± 6.02 27.09 ± 0.37	519.66 ± 4.88 26.13 ± 0.27
	20	742.96 ± 11.82 24.71 ± 0.46	548.44 ± 3.60 32.59 ± 0.44	568.20 ± 12.65 30.98 ± 0.11
	30	817.81 ± 9.85 33.64 ± 0.36	575.78 ± 1.48 43.26 ± 0.40	636.86 ± 6.26 41.30 ± 0.55
	40	870.37 ± 3.04 42.07 ± 0.20	575.74 ± 2.53 55.87 ± 0.30	663.80 ± 6.26 51.80 ± 1.50

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 0	1	95.65 ± 0.32 9.96 ± 0.03	65.25 ± 0.71 15.13 ± 0.16	67.61 ± 0.15 14.46 ± 0.03
	2	139.94 ± 0.54 10.34 ± 0.02	123.72 ± 1.76 15.80 ± 0.23	127.87 ± 0.95 15.20 ± 0.12
	5	351.18 ± 1.63 12.29 ± 0.04	264.25 ± 1.59 18.20 ± 0.11	269.77 ± 1.64 17.51 ± 0.02
	10	561.78 ± 1.20 16.18 ± 0.10	409.90 ± 9.22 22.97 ± 0.44	413.06 ± 0.71 22.34 ± 0.19
	15	671.88 ± 8.55 20.37 ± 0.32	500.64 ± 2.96 22.73 ± 0.23	500.72 ± 9.49 27.02 ± 0.13
	20	742.96 ± 11.82 24.71 ± 0.46	546.89 ± 5.59 33.32 ± 0.35	550.66 ± 6.39 32.15 ± 0.42
	30	817.81 ± 9.85 33.64 ± 0.36	594.74 ± 3.04 43.36 ± 0.32	614.04 ± 5.25 42.96 ± 0.57
	40	870.37 ± 3.04 42.07 ± 0.20	605.90 ± 0.91 54.39 ± 0.11	647.42 ± 11.65 52.99 ± 0.58

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 1	1	91.06 ± 1.13	53.62 ± 0.22	56.76 ± 0.38
		10.48 ± 0.14	18.43 ± 0.08	17.29 ± 0.11
	2	151.98 ± 0.58	98.88 ± 0.61	103.98 ± 0.62
		11.61 ± 0.07	19.82 ± 0.13	18.77 ± 0.10
	5	272.92 ± 1.97	190.31 ± 0.64	196.25 ± 2.14
		15.97 ± 0.25	25.43 ± 0.08	24.59 ± 0.14
	10	364.87 ± 2.71	264.36 ± 2.94	262.96 ± 0.90
		25.17 ± 0.25	36.28 ± 0.45	35.86 ± 0.39
	15	406.36 ± 5.76	297.85 ± 3.66	293.62 ± 6.67
		34.37 ± 0.44	48.27 ± 0.50	47.84 ± 0.53
	20	426.38 ± 6.69	317.98 ± 4.17	311.95 ± 4.95
		43.56 ± 0.24	60.07 ± 0.88	59.41 ± 0.91
	30	453.89 ± 3.76	338.90 ± 1.08	334.09 ± 5.61
		61.27 ± 0.48	84.80 ± 0.32	82.49 ± 1.23
	40	469.72 ± 4.03	356.54 ± 3.97	344.97 ± 1.47
		79.67 ± 0.63	106.88 ± 0.59	105.11 ± 0.72

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 4	1	45.93 ± 0.20	34.28 ± 0.59	35.70 ± 0.43
		21.27 ± 0.09	28.95 ± 0.51	27.67 ± 0.34
	2	46.51 ± 0.11	38.24 ± 0.44	37.80 ± 0.12
		42.49 ± 0.11	51.91 ± 0.60	52.49 ± 0.17
	5	47.35 ± 0.12	39.19 ± 0.54	39.09 ± 0.14
		104.93 ± 0.26	126.65 ± 1.73	126.84 ± 0.46
	10	48.01 ± 0.28	40.56 ± 0.32	40.20 ± 0.18
		207.12 ± 1.12	244.54 ± 1.74	245.32 ± 1.38
	15	48.64 ± 0.02	42.23 ± 0.07	41.48 ± 0.11
		306.45 ± 0.14	352.62 ± 0.55	354.54 ± 2.40
	20	48.90 ± 0.23	43.09 ± 0.17	42.37 ± 0.03
		405.80 ± 1.76	461.04 ± 2.17	459.08 ± 1.38
	30	49.11 ± 0.25	44.13 ± 0.09	43.33 ± 0.13
		604.41 ± 2.90	675.60 ± 1.32	666.99 ± 0.96
	40	49.76 ± 0.17	45.00 ± 0.09	43.98 ± 0.28
		793.36 ± 2.66	883.81 ± 1.88	861.25 ± 4.37

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 5	1	45.79 ± 0.01 21.34 ± 0.01	34.32 ± 0.40 28.91 ± 0.33	36.06 ± 0.34 27.39 ± 0.26
	2	75.78 ± 0.35 25.88 ± 0.12	55.68 ± 0.20 35.48 ± 0.14	57.84 ± 0.54 34.07 ± 0.30
	5	127.37 ± 0.62 38.74 ± 0.18	95.38 ± 0.87 51.55 ± 0.46	96.23 ± 1.39 50.75 ± 0.73
	10	165.00 ± 0.87 59.22 ± 0.37	128.74 ± 1.14 76.30 ± 0.74	96.23 ± 2.06 74.90 ± 0.73
	15	186.31 ± 2.07 78.79 ± 0.76	146.01 ± 1.05 100.68 ± 0.70	145.15 ± 1.49 99.13 ± 0.88
	20	197.51 ± 2.04 99.34 ± 1.07	156.60 ± 1.61 124.87 ± 1.58	153.93 ± 0.95 123.58 ± 0.58
	30	210.95 ± 2.31 139.82 ± 1.05	172.38 ± 1.68 170.52 ± 1.58	169.24 ± 0.95 166.31 ± 0.58
	40	219.55 ± 0.97 179.63 ± 0.67	182.06 ± 1.12 214.01 ± 2.07	176.31 ± 1.25 211.70 ± 0.29

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
RAID level 6	1	40.72 ± 0.06	28.13 ± 0.43	32.01 ± 0.15
		24.05 ± 0.03	35.33 ± 0.54	30.90 ± 0.14
	2	60.58 ± 0.23	43.05 ± 0.63	46.41 ± 0.32
		32.51 ± 0.18	45.94 ± 0.68	42.61 ± 0.34
	5	93.83 ± 0.57	69.27 ± 1.15	72.21 ± 0.46
		52.75 ± 0.32	71.09 ± 1.24	68.15 ± 0.45
	10	116.23 ± 0.99	89.52 ± 0.76	91.51 ± 1.16
		85.39 ± 0.75	109.81 ± 0.86	106.67 ± 1.57
	15	128.61 ± 0.81	99.57 ± 0.94	101.65 ± 0.62
		115.89 ± 0.72	147.44 ± 1.44	143.50 ± 0.43
	20	135.60 ± 1.71	104.76 ± 0.44	107.07 ± 0.98
		146.61 ± 1.81	186.42 ± 0.87	179.97 ± 1.87
	30	142.96 ± 0.65	108.90 ± 0.29	114.56 ± 0.60
		208.51 ± 0.90	267.21 ± 1.04	248.72 ± 1.33
	40	147.12 ± 1.75	109.74 ± 0.53	118.68 ± 0.81
		270.00 ± 2.69	354.15 ± 1.15	319.30 ± 1.99

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
parity declustering	1	42.59 ± 0.09 22.98 ± 0.05	34.49 ± 0.43 28.77 ± 0.37	35.45 ± 0.41 27.87 ± 0.32
	2	72.84 ± 0.61 26.95 ± 0.22	55.36 ± 0.75 35.71 ± 0.48	57.09 ± 0.22 34.58 ± 0.10
	5	126.07 ± 1.20 39.13 ± 0.38	94.80 ± 0.89 51.87 ± 0.49	95.87 ± 1.25 51.11 ± 0.69
	10	165.92 ± 1.71 58.76 ± 0.49	128.07 ± 0.79 76.72 ± 0.48	127.04 ± 1.24 76.55 ± 0.95
	15	184.33 ± 1.47 76.69 ± 0.54	144.78 ± 1.23 101.64 ± 0.83	145.28 ± 1.55 76.55 ± 0.54
	20	196.73 ± 0.60 99.65 ± 0.24	156.14 ± 1.95 125.33 ± 1.67	154.44 ± 2.20 123.80 ± 1.07
	30	209.32 ± 1.48 140.95 ± 1.10	170.02 ± 1.64 172.81 ± 1.61	167.04 ± 1.42 170.24 ± 1.23
	40	217.81 ± 0.81 181.06 ± 0.93	179.01 ± 0.64 218.58 ± 0.63	175.45 ± 0.69 212.93 ± 0.94

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
interleaved declustering	1	53.96 ± 0.37 18.03 ± 0.13	42.79 ± 0.48 23.16 ± 0.25	44.37 ± 0.20 22.21 ± 0.10
	2	96.34 ± 0.68 20.25 ± 0.14	77.48 ± 0.48 25.42 ± 0.16	79.57 ± 0.23 24.70 ± 0.07
	5	177.10 ± 0.77 26.33 ± 0.13	145.81 ± 1.25 33.55 ± 0.32	147.57 ± 1.93 32.74 ± 0.34
	10	258.00 ± 0.75 36.18 ± 0.24	205.63 ± 1.90 47.34 ± 0.43	206.44 ± 1.93 46.43 ± 0.31
	15	306.26 ± 4.27 45.78 ± 0.37	239.86 ± 1.67 60.77 ± 0.42	238.45 ± 1.46 59.58 ± 0.09
	20	336.65 ± 2.31 55.73 ± 0.27	262.81 ± 2.29 73.85 ± 0.85	256.55 ± 2.47 73.42 ± 0.53
	30	374.74 ± 1.61 74.93 ± 0.10	289.42 ± 1.93 100.46 ± 0.41	279.33 ± 2.26 98.27 ± 0.64
	40	400.72 ± 3.34 93.52 ± 1.06	306.28 ± 0.82 126.16 ± 0.33	300.10 ± 4.24 123.05 ± 1.21

Table C-6 Small-write performance of RAIDframe's three environments

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval		
		Simulator	User	Kernel
chained declustering	1	57.23 ± 0.02 16.97 ± 0.01	44.53 ± 0.46 22.23 ± 0.23	46.13 ± 0.30 21.35 ± 0.15
	2	100.51 ± 0.73 19.40 ± 0.14	79.73 ± 1.02 24.69 ± 0.32	82.59 ± 0.44 23.74 ± 0.11
	5	182.83 ± 0.94 25.41 ± 0.18	150.37 ± 1.25 32.48 ± 0.27	152.79 ± 1.14 31.75 ± 0.03
	10	265.96 ± 0.13 34.87 ± 0.20	210.62 ± 3.06 46.17 ± 0.68	210.75 ± 1.42 45.48 ± 0.33
	15	314.52 ± 3.15 44.85 ± 0.29	246.31 ± 1.75 59.09 ± 0.47	244.47 ± 3.33 58.44 ± 0.67
	20	346.18 ± 1.84 54.41 ± 0.07	268.33 ± 0.94 72.20 ± 0.24	263.48 ± 1.63 71.47 ± 0.31
	30	382.18 ± 2.01 73.49 ± 0.59	296.46 ± 4.01 97.94 ± 1.38	286.71 ± 1.62 96.60 ± 0.41
	40	407.90 ± 6.35 92.03 ± 0.94	312.01 ± 0.75 123.70 ± 0.40	304.51 ± 1.31 120.60 ± 0.81

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval Forward	Backward
RAID level 0	1	67.71 ± 0.15 14.46 ± 0.03	39.88 ± 0.51 24.74 ± 0.31
	2	127.87 ± 0.95 15.20 ± 0.12	73.06 ± 0.98 26.88 ± 0.30
	5	269.77 ± 1.64 17.51 ± 0.02	142.44 ± 1.54 33.77 ± 0.06
	10	413.06 ± 0.71 22.34 ± 0.19	210.88 ± 3.93 45.00 ± 0.90
	15	500.72 ± 9.49 27.02 ± 0.13	251.46 ± 4.26 55.85 ± 0.77
	20	550.66 ± 6.39 32.15 ± 0.42	273.33 ± 2.44 67.45 ± 0.03
	30	614.04 ± 5.25 42.96 ± 0.57	305.29 ± 7.68 89.38 ± 1.90
	40	647.42 ± 11.65 52.99 ± 0.58	324.90 ± 3.85 111.33 ± 0.96

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval — Forward	Backward
RAID level 1	1	56.76 ± 0.38 17.29 ± 0.11	35.65 ± 0.44 27.72 ± 0.34
	2	103.98 ± 0.62 18.77 ± 0.10	59.95 ± 0.62 32.89 ± 0.32
	5	196.25 ± 2.14 24.59 ± 0.14	102.28 ± 1.70 47.61 ± 0.81
	10	262.96 ± 0.90 35.86 ± 0.39	131.43 ± 2.38 73.24 ± 1.00
	15	293.62 ± 6.67 47.84 ± 0.53	148.22 ± 0.86 96.94 ± 0.81
	20	311.95 ± 4.95 59.41 ± 0.91	156.14 ± 1.80 121.85 ± 1.39
	30	334.09 ± 5.61 82.49 ± 1.23	168.25 ± 1.35 166.43 ± 1.49
	40	344.97 ± 1.47 105.11 ± 0.72	176.07 ± 3.25 212.28 ± 2.12

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Backward**
RAID level 4	1	35.70 ± 0.43 27.67 ± 0.34	34.33 ± 0.31 28.76 ± 0.26
	2	37.80 ± 0.12 52.49 ± 0.17	25.40 ± 0.05 78.33 ± 0.14
	5	39.09 ± 0.14 126.84 ± 0.46	25.94 ± 0.04 191.76 ± 0.38
	10	40.20 ± 0.18 245.32 ± 1.38	26.85 ± 0.02 367.94 ± 0.19
	15	41.48 ± 0.11 354.54 ± 2.40	27.65 ± 0.03 532.72 ± 2.16
	20	42.37 ± 0.03 459.08 ± 1.38	28.19 ± 0.10 692.19 ± 1.89
	30	43.33 ± 0.13 666.99 ± 0.96	29.05 ± 0.09 999.41 ± 5.53
	40	43.98 ± 0.28 861.25 ± 4.37	29.39 ± 0.17 1303.31 ± 7.88

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Backward**
RAID level 5	1	36.06 ± 0.34 27.39 ± 0.26	33.96 ± 29.07 29.07 ± 0.20
	2	57.84 ± 0.54 34.07 ± 0.30	43.86 ± 0.69 45.12 ± 0.75
	5	96.23 ± 1.39 50.75 ± 0.73	67.81 ± 0.40 72.43 ± 0.84
	10	129.03 ± 2.06 74.90 ± 0.73	87.46 ± 1.07 110.73 ± 1.49
	15	145.15 ± 1.49 99.13 ± 0.88	97.36 ± 0.73 149.54 ± 1.39
	20	153.93 ± 0.82 123.58 ± 1.08	103.13 ± 1.15 187.30 ± 2.44
	30	169.24 ± 0.95 166.31 ± 0.58	113.39 ± 1.72 252.30 ± 3.69
	40	176.31 ± 1.25 211.70 ± 0.29	119.84 ± 0.24 316.15 ± 0.78

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Backward
RAID level 6	1	32.01 ± 0.15	28.38 ± 0.16
		30.90 ± 0.14	34.87 ± 0.20
	2	46.41 ± 0.32	32.13 ± 0.33
		42.61 ± 0.34	61.72 ± 0.66
	5	72.21 ± 0.46	49.22 ± 0.27
		68.15 ± 0.45	100.54 ± 0.45
	10	91.51 ± 1.16	60.65 ± 0.49
		106.67 ± 1.57	161.98 ± 1.45
	15	101.65 ± 0.62	67.31 ± 0.74
		143.50 ± 0.93	217.96 ± 1.92
	20	107.07 ± 0.98	71.04 ± 0.60
		179.97 ± 1.87	271.96 ± 2.59
	30	114.56 ± 0.60	76.04 ± 0.77
		248.72 ± 1.33	377.80 ± 1.15
	40	118.68 ± 0.81	79.40 ± 0.71
		319.30 ± 1.99	479.17 ± 3.59

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Backward
parity declustering	1	35.45 ± 0.41 27.87 ± 0.32	33.88 ± 0.48 29.16 ± 0.42
	2	57.09 ± 0.22 34.58 ± 0.10	43.69 ± 0.39 45.28 ± 0.43
	5	95.87 ± 1.25 51.11 ± 0.69	66.85 ± 0.23 73.25 ± 0.33
	10	127.04 ± 1.24 76.55 ± 0.95	86.15 ± 0.48 112.86 ± 1.50
	15	145.28 ± 1.55 99.48 ± 0.54	96.64 ± 0.40 150.23 ± 0.20
	20	154.44 ± 2.20 123.80 ± 1.07	102.65 ± 0.90 187.41 ± 0.80
	30	167.04 ± 1.42 170.24 ± 1.23	111.32 ± 1.72 255.85 ± 4.62
	40	175.45 ± 0.69 212.93 ± 0.94	117.45 ± 0.72 321.50 ± 1.54

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Backward
interleaved declustering	1	44.37 ± 0.20 22.21 ± 0.10	30.63 ± 0.38 32.32 ± 0.40
	2	79.57 ± 0.23 24.70 ± 0.07	45.98 ± 0.43 43.04 ± 0.41
	5	147.57 ± 1.93 32.74 ± 0.34	75.09 ± 1.01 65.30 ± 1.12
	10	206.44 ± 1.93 46.43 ± 0.34	100.03 ± 1.36 96.77 ± 1.48
	15	238.45 ± 1.46 59.58 ± 0.09	115.90 ± 0.13 125.87 ± 0.60
	20	256.55 ± 2.47 73.42 ± 0.53	126.21 ± 0.25 152.40 ± 1.07
	30	279.33 ± 2.26 98.27 ± 0.64	141.86 ± 1.40 203.25 ± 2.13
	40	300.10 ± 4.24 123.05 ± 1.21	149.92 ± 0.31 249.95 ± 2.53

Table C-7 Relative performance of full undo logging

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Backward
chained declustering	1	46.13 ± 0.30 21.35 ± 0.15	31.38 ± 0.24 31.53 ± 0.24
	2	82.59 ± 0.44 23.74 ± 0.11	47.68 ± 0.33 41.49 ± 0.32
	5	152.79 ± 1.14 31.75 ± 0.03	76.94 ± 0.45 63.44 ± 0.61
	10	210.75 ± 1.42 45.48 ± 0.33	104.57 ± 0.64 93.48 ± 0.49
	15	244.47 ± 3.33 58.44 ± 0.67	119.46 ± 0.28 121.60 ± 0.08
	20	263.48 ± 1.63 71.47 ± 0.37	129.61 ± 0.44 147.45 ± 0.33
	30	286.71 ± 1.62 96.60 ± 0.42	144.53 ± 0.37 197.74 ± 1.38
	40	304.51 ± 1.31 120.60 ± 0.81	153.46 ± 0.50 244.89 ± 1.27

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Rollaway**
RAID level 0	1	67.71 ± 0.15 14.46 ± 0.03	68.56 ± 0.20 14.25 ± 0.03
	2	127.87 ± 0.95 15.20 ± 0.12	129.47 ± 1.18 14.97 ± 0.06
	5	269.77 ± 1.64 17.51 ± 0.02	274.56 ± 4.94 17.30 ± 0.28
	10	413.06 ± 0.71 22.34 ± 0.19	422.39 ± 2.16 21.74 ± 0.18
	15	500.72 ± 9.49 27.02 ± 0.13	499.77 ± 6.90 27.06 ± 0.16
	20	550.66 ± 6.39 32.15 ± 0.42	556.92 ± 6.08 31.96 ± 0.39
	30	614.04 ± 5.25 42.96 ± 0.57	609.11 ± 9.72 42.62 ± 0.14
	40	647.42 ± 11.65 52.99 ± 0.58	656.20 ± 15.47 52.63 ± 0.18

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Rollaway**
RAID level 1	1	56.76 ± 0.38 17.29 ± 0.11	57.26 ± 0.64 17.14 ± 0.19
	2	103.98 ± 0.62 18.77 ± 0.10	104.17 ± 0.59 18.75 ± 0.11
	5	196.25 ± 2.14 24.59 ± 0.14	197.35 ± 1.05 24.35 ± 0.27
	10	262.96 ± 0.90 35.86 ± 0.39	263.06 ± 5.63 35.95 ± 0.50
	15	293.62 ± 6.67 47.84 ± 0.53	296.52 ± 3.21 47.48 ± 0.72
	20	311.95 ± 4.95 59.41 ± 0.91	311.62 ± 3.58 59.93 ± 0.61
	30	334.09 ± 5.61 82.49 ± 1.23	335.51 ± 3.53 82.20 ± 0.86
	40	344.97 ± 1.47 105.11 ± 0.72	346.77 ± 2.35 105.90 ± 1.16

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Rollaway**
RAID level 4	1	35.70 ± 0.43 27.67 ± 0.34	35.05 ± 1.05 28.20 ± 0.85
	2	37.80 ± 0.12 52.49 ± 0.17	37.97 ± 0.16 52.24 ± 0.19
	5	39.09 ± 0.14 126.84 ± 0.46	39.07 ± 0.20 126.85 ± 0.54
	10	40.20 ± 0.18 245.32 ± 1.38	40.24 ± 0.18 244.45 ± 1.19
	15	41.48 ± 0.11 354.54 ± 2.40	41.51 ± 0.27 354.23 ± 3.96
	20	42.37 ± 0.03 459.08 ± 1.38	42.25 ± 0.15 458.54 ± 0.95
	30	43.33 ± 0.13 666.99 ± 0.96	43.42 ± 0.18 664.18 ± 3.93
	40	43.98 ± 0.28 861.25 ± 4.37	44.16 ± 0.23 865.27 ± 5.40

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Rollaway
RAID level 5	1	36.06 ± 0.34 27.39 ± 0.26	35.60 ± 0.21 27.76 ± 0.15
	2	57.84 ± 0.54 34.07 ± 0.30	57.15 ± 0.62 34.55 ± 0.40
	5	96.23 ± 1.39 50.75 ± 0.73	95.60 ± 0.75 51.04 ± 0.15
	10	129.03 ± 2.06 74.90 ± 0.73	128.64 ± 1.31 75.65 ± 0.60
	15	145.15 ± 1.49 99.13 ± 0.88	144.51 ± 1.11 100.00 ± 1.12
	20	153.93 ± 0.82 123.58 ± 1.08	154.76 ± 0.84 122.89 ± 0.68
	30	169.24 ± 0.95 166.31 ± 0.58	167.63 ± 0.86 167.88 ± 0.88
	40	176.31 ± 1.25 211.70 ± 0.29	177.06 ± 1.15 211.45 ± 2.33

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Rollaway
RAID level 6	1	32.01 ± 0.15	32.12 ± 0.44
		30.90 ± 0.14	30.79 ± 0.41
	2	46.41 ± 0.32	46.39 ± 0.30
		42.61 ± 0.34	42.59 ± 0.28
	5	72.21 ± 0.46	71.86 ± 0.53
		68.15 ± 0.45	68.20 ± 0.82
	10	91.51 ± 1.16	90.92 ± 1.12
		106.67 ± 1.57	107.36 ± 1.77
	15	101.65 ± 0.62	101.52 ± 1.09
		143.50 ± 0.93	143.73 ± 0.77
	20	107.07 ± 0.98	106.72 ± 1.19
		179.97 ± 1.87	180.63 ± 1.02
	30	114.56 ± 0.60	114.30 ± 1.39
		248.72 ± 1.33	250.08 ± 3.38
	40	118.68 ± 0.81	118.57 ± 0.97
		319.30 ± 1.99	319.74 ± 3.78

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Rollaway
parity declustering	1	35.45 ± 0.41 27.87 ± 0.32	35.20 ± 0.37 28.07 ± 0.29
	2	57.09 ± 0.22 34.58 ± 0.10	56.63 ± 0.46 34.88 ± 0.29
	5	95.87 ± 1.25 51.11 ± 0.69	94.98 ± 1.13 51.58 ± 0.50
	10	127.04 ± 1.24 76.55 ± 0.95	126.10 ± 2.00 76.89 ± 1.00
	15	145.28 ± 1.55 99.48 ± 0.54	144.05 ± 0.27 100.65 ± 0.31
	20	154.44 ± 2.20 123.80 ± 1.07	153.32 ± 0.83 125.00 ± 0.98
	30	167.04 ± 1.42 170.24 ± 1.23	164.59 ± 2.17 171.65 ± 1.61
	40	175.45 ± 0.69 212.93 ± 0.94	174.89 ± 1.04 214.52 ± 0.47

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	Rollaway
interleaved declustering	1	44.37 ± 0.20 22.21 ± 0.10	44.73 ± 0.24 22.03 ± 0.12
	2	79.57 ± 0.23 24.70 ± 0.07	80.44 ± 0.51 24.42 ± 0.15
	5	147.57 ± 1.93 32.74 ± 0.34	148.26 ± 1.16 32.60 ± 0.37
	10	206.44 ± 1.93 46.43 ± 0.34	206.50 ± 3.74 46.26 ± 0.37
	15	238.45 ± 1.46 59.58 ± 0.09	238.76 ± 0.92 59.78 ± 0.34
	20	256.55 ± 2.47 73.42 ± 0.53	258.51 ± 3.78 73.25 ± 0.68
	30	279.33 ± 2.26 98.27 ± 0.64	284.59 ± 2.89 99.03 ± 0.75
	40	300.10 ± 4.24 123.05 ± 1.21	299.08 ± 2.95 123.48 ± 0.37

Table C-8 Relative performance of roll-away recovery

Architecture	Number of Concurrent Requesting Processes	Throughput (IO/s) ± 95% confidence interval Response Time (ms) ± 95% confidence interval	
		Forward	**Rollaway**
chained declustering	1	46.13 ± 0.30 21.35 ± 0.15	46.61 ± 0.31 21.13 ± 0.14
	2	82.59 ± 0.44 23.74 ± 0.11	82.59 ± 0.77 23.67 ± 0.16
	5	152.79 ± 1.14 31.75 ± 0.03	152.64 ± 0.60 31.67 ± 0.18
	10	210.75 ± 1.42 45.48 ± 0.33	212.03 ± 1.64 45.14 ± 0.13
	15	244.47 ± 3.33 58.44 ± 0.67	245.16 ± 3.37 58.09 ± 0.70
	20	263.48 ± 1.63 71.47 ± 0.37	265.58 ± 1.25 70.89 ± 0.60
	30	286.71 ± 1.62 96.60 ± 0.42	289.40 ± 3.34 96.45 ± 1.64
	40	304.51 ± 1.31 120.60 ± 0.81	304.40 ± 1.27 120.80 ± 0.28

C.3 Sample Configuration File

```
START array
# params are: numRow numCol numSpare
1 10 0

START disks
# a list of device files corresponding to physical disks
/dev/rrz18c
/dev/rrz26c
/dev/rrz34c
/dev/rrz42c
/dev/rrz50c
/dev/rrz19c
/dev/rrz27c
/dev/rrz35c
/dev/rrz43c
/dev/rrz51c
/dev/rrz20c
/dev/rrz28c
/dev/rrz36c
/dev/rrz44c
/dev/rrz52c

START spare
# a list of device files corresponding to spare disks
# spare device goes here

START layout
# general layout params:
# sectPerSU SUsPerParityUnit SUsPerReconUnit parityConfig
64 1 1 5

START queue
# generic queue params: queue type, num concurrent reqs that
can be sent to a disk
sstf 5
# queue-specific configuration lines:
#    (none for FIFO)

START debug
accessTraceBufSize 100
maxTraceRunTimeSec 30
```

Index