Backward Error Recovery in Redundant Disk Arrays

William V. Courtright II and Garth A. Gibson

27 September 1994
CMU-CS-94-193

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

To appear: Proceedings of the 1994 Computer Measurement Group Conference (CMG94)

Abstract

Redundant disk arrays are single fault tolerant, incorporating a layer of error handling not found in non-redundant disk systems. Recovery from these errors is complex, due in part to the large number of erroneous states the system may reach. The established approach to error recovery in disk systems is to transition directly from an erroneous state to completion. This technique, known as forward error recovery, relies upon the context in which an error occurs to determine the steps required to reach completion, which implies forward error recovery is design specific. Forward error recovery requires the enumeration of all erroneous states the system may reach and the construction of a forward path from each erroneous state. We propose a method of error recovery which does not rely upon the enumeration of erroneous states or the context in which errors occur. When an error is encountered, we advocate mechanized recovery to an error-free state from which an operation may be retried. Using a form of backward error recovery, we are able to manage the complexity of error recovery in redundant disk arrays without sacrificing performance.

http://www.cs.cmu.edu:8001/afs/cs/project/pdl/WWW/HomePage.html

This research was partially supported by the National Science Foundation under grant number ECD-8907068 and an AT&T fellowship. 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 AT&T or the U.S. government.

Keywords: concurrency, error recovery, fault tolerance, RAID, redundant disk arrays

1. Introduction

Through the use of redundancy, disk arrays can be made single fault tolerant. Disk arrays which provide single fault tolerance, categorized by a taxonomy known as RAID (Redundant Arrays of Inexpensive Disks) in 1988 and summarized in Figure 1, have become a important class of storage devices [Patterson88]. As such, many companies are now engaged in the design of RAID systems. While many of these companies possess a tremendous amount of experience developing nonredundant disk systems, RAID systems introduce a new design challenge: isolating, at the level of the redundant disk array control, the effects of single faults, thereby transparently handling a large class of errors. Figure 2 shows the structure of a redundant disk array control system. This new error recovery task is made particularly difficult since an important feature of disk arrays is high concurrency, making the number of erroneous states which the system may reach uncountable.

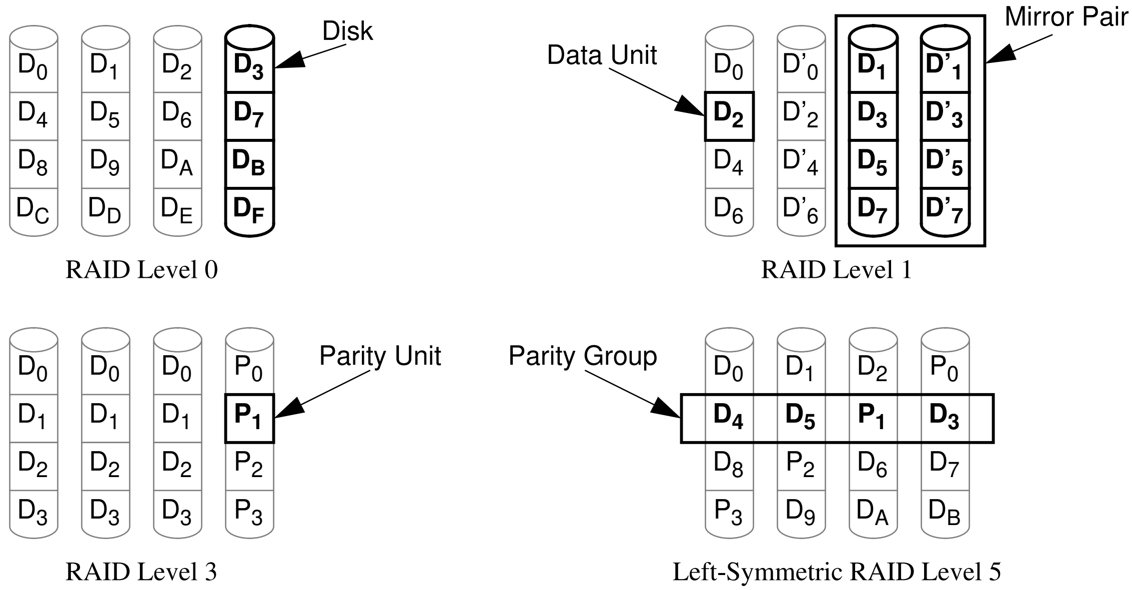

Figure 1 **RAID Levels 0, 1, 3, and 5.** This figure depicts the data layout and redundancy organizations for the most prevalent RAID levels using an array of four disks. *Data units* represent the unit of data access supported by the array. *Parity units* represent redundancy information generated from the bitwise 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*.

RAID level 0 offers no redundancy so it is not single fault tolerant. In this illustration, data is block interleaved meaning that the array is optimized for small transfer sizes with each request being serviced by a single drive, increasing throughput. Data may also be bit interleaved, optimizing performance for large transfer sizes by using every drive to transfer data in parallel, increasing bandwidth.

RAID level 1 offers the simplest form of redundancy, maintaining two copies of each data unit, each stored on a different disk. Also referred to as mirroring, this method of redundancy has the highest capacity overhead of the RAID architectures: 50% of disk space is consumed by redundant information. Because the redundant information is a simple copy and both copies reside on independent disks, read operations can be serviced by either copy, making balancing the read workload across the array easier.

RAID level 3 is bit interleaved to optimize bandwidth and relies on parity-based redundancy to reduce capacity overhead. Redundancy is maintained on a single drive, the parity drive. Parity is calculated as the bitwise exclusive-or of the data drives.

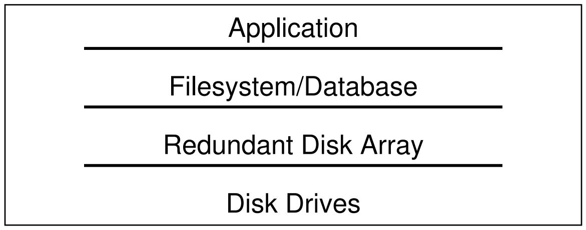

Figure 2 **Abstractions in the Storage Hierarchy.** Shown are four layers of abstraction found in a typical storage hierarchy and their relationship. Disk drives provide durable storage and implement read and write actions. Redundant disk array control systems increase the reliability and availability of disk storage and implement read and write operations as a composition of disk actions. Filesystem and database abstractions provide more natural interfaces to durable storage, implementing operations such as file open, delete file, add record, and read-modify-write. Finally, applications run as a part of the user process and provide an interface to the outside world, implementing tools such as: word processors, simulators, data visualization, and accounting systems.

Disk drives are not fault tolerant, meaning that any error encountered during a disk action results in loss of data. Redundant disk arrays make disk storage single fault tolerant, guaranteeing no loss of data and successful completion of operations when a single disk failure occurs. Errors due to the existence of multiple disk failures may result in loss of data in the array. Filesystems generally do not provide additional fault tolerance, but do attempt to minimize the effects of data loss presented to an application [Leffler90]. Databases, however, usually do provide higher levels of fault tolerance; in particular, operations are generally atomic, meaning that transactions which fail leave the applications's view of the system unchanged.

The traditional approach to recovering from errors in RAID systems has been to transition directly from an erroneous state to a completion state. This approach, known as forward error recovery, relies upon the enumeration of each erroneous state the system may reach. Since this is a large number, the task of guaranteeing correct error recovery while maintaining high concurrency can quickly become overwhelming.

Another approach, backward error recovery, eliminates the complexity associated with identifying all erroneous states and transitions from them. This is accomplished by saving the state of the system prior to each operation and then restoring this state if an operation fails. Once restoration is complete, the system is free from error and processing resumes. Unfortunately, systems employing backward error recovery can expect degraded performance due to the cost associated with saving the state of the system prior to each operation. In some cases, such as write operations, the amount of work required to save state information is comparable to the requested work, leading to a large degradation in performance.

We propose a method of error recovery based upon backward error recovery, but do not degrade performance by saving additional state information during normal processing. When an error is encountered, recovery is performed to a convenient error-free state, not necessarily the same state the system was in at the beginning of the operation. Once the system is in an error-free state, operation resumes and the operation which encountered the error is retried. This approach eliminates the complexity of transitioning from an erroneous state to a completion state. Furthermore, because we do not require recovery to a previously existing state, we do not incur performance degradation normally associated with backward error recovery.

The remainder of this paper demonstrates the problem of error recovery in redundant disk arrays, current solutions and their shortcomings, and concludes with our approach to this problem. Section 2. discusses the motivations for our interest in this problem. These include a description of the complexity of error recovery in redundant disk arrays, the shortcomings of the current approach to error recovery, and the benefits of pursuing a better approach. In Section 3., we examine other approaches to error recovery and their weaknesses. Section 4. presents our approach to error recovery, which manages complexity and enables exploration of the disk array design space without degrading performance overhead during normal processing. Finally, in Section 5., we conclude with a summary of the benefits of our approach and a brief review of work in progress to demonstrate this approach as valid.

2. Motivation

2.1. Motivations for Redundant Disk Arrays

Disk arrays are a well established method of using parallelism to reduce response time in disk storage systems [Kim86, Salem86, Katz89, Reddy89]. *Response time* is the total amount of time required to service a request made to a disk system and is composed of three components: *queueing time*, the time a request spends in a queue waiting to begin execution; *positioning time*, the time required to position the disk head to useful data; and *transfer time*, the time required to transfer data to or from the disk. Queueing time is reduced when multiple requests are serviced concurrently and transfer time is reduced by transferring data from disks in parallel.

Simple disk systems are not *fault tolerant*; a single fault can lead to data loss. The accepted failure model of such *nonredundant* disk systems requires only *error detection*, the recognition of the presence of an error [Lampson79]. This is acceptable since applications which require fault tolerance implement schemes to survive data loss at the application level of the system in the following way. After the detection of an error, the disk system notifies a client who is responsible for: *damage confinement* and *assessment*, the understanding and limitation of the effects of the detected error; *error recovery*, the removal of the effects of the detected error from the system; and *fault treatment*, isolating the fault which caused the detected error and removing it from the system.[1]

Disk systems, particularly when configured as arrays, may be composed of large numbers of disks. As the number of disks in the array increases, *reliability*, the probability that the disk array will function correctly, and *availability*, the probability that the disk array is able to service requests, may decrease to unacceptable levels since data loss occurs on the first failure [Gibson93]. This problem may be further aggravated because, as Patterson, Gibson, and Katz suggest, commodity disks may be employed in order to reduce the cost of storage in the array [Patterson88].

Because reliability and availability are critical characteristics of storage systems, disk arrays are usually designed to be single fault tolerant. This is accomplished by storing redundant data in the disk array [Gibson89, Gibson92]. Instead of propagating errors resulting from a disk failure to a client to handle, the redundant disk array now performs recovery from these errors, hiding their effects from clients and providing continuous service throughout the life of the fault.

2.2. Error Recovery is Complex

Redundant disk arrays are required to provide service from two distinct operating states: normal and degraded. The array exists in a *normal operating state* when no faults are present. Because redundant disk arrays are single fault tolerant, they are also expected to provide service in a *degraded operating state* which exists when a single fault, in our case a disk failure,[2] is present in the system. When two or more faults exist, the array may be in a *failed operating state* in which data is lost and service discontinued.

When an error is encountered, the system enters an *erroneous state*, meaning that the physical array state, “containing a failed disk”, is inconsistent with the state of the system as perceived by operations initiated prior to the error. Recovery from errors requires the array to be restored to a *consistent state*, a state free from error, and the successful completion of operation(s) in-flight at the time of the error.

---

1. The definitions presented here are consistent with those of the IEEE Technical Committee on Fault Tolerant Computing [Melliar-Smith77, Anderson82, Lee82].
2. Other faults, such as loss of power, mechanical failure of cabling, can be converted into independent single faults in “orthogonal” redundant disk arrays [Gibson93].

Work:
    read: old data, old parity
    exclusive-or: new data, old data,
                   old parity
    write: new data, new parity
Resources:
    memory for old data, old parity

Figure 3(a): Write Operation - Normal State

Work:
    read: surviving data
    exclusive-or: new data, surviving data
    write: new parity
Resources:
    memory for surviving data

Figure 3(b): Write Operation - Degraded State

Figure 3 **Work and Resources are a Function of Current Operating State**. This figure compares the work and resources required for a RAID level 5 small-write operations in the normal and degraded operating states. Figure 3(a) shows the normal case while Figure 3(b) illustrates the case when a data drive has failed. A RAID level 5 array uses parity-based redundancy to achieve single fault tolerance and requires that all write operations update the parity which protects the data being written. When the array is in the normal state, the update is computed as the exclusive-or of the data to be written, the new data, and the data being overwritten, old data. New parity is then generated as a result of the exclusive-or of the old parity and the update.

When the drive which contains the data to be written has failed, writes must be performed in a different manner. Write operations to a failed data drive are not able to write new data but are still able to write parity. The write is accomplished by reading surviving data, exclusive-or'ing this with the new data, and then writing the result as new parity. The new data can retrieved by exclusive-or'ing the surviving data and new parity.

The process of error recovery in redundant disk arrays is complex for several reasons: many operations may be executed concurrently, operations may be initiated and complete in different operating states, the array may reach a large number of erroneous states, and recovery of errors due to single faults must be transparent to client requestors.

First, disk arrays increase performance by executing many operations concurrently, requiring careful synchronization of shared resources. Errors require operations to alter their usage, and hence synchronization, of shared resources in order to complete. As concurrency increases, the complexity of synchronization escalates.

Second, operations in-flight when an error occurs are initiated in the normal operating state and should complete in the degraded operating state. As Figure 3 illustrates, the work and resources required to complete an operation in the normal state varies greatly from the degraded state. Because of this, recovery must dynamically change the operation's algorithm. It is this problem which makes error recovery in redundant disk arrays particularly difficult.

Third, the number of erroneous states which a redundant disk array operation can encounter, and be required to recover from, is large. This is because operations in redundant disk arrays perform more work than operations in simpler disk systems. Figure 4 demonstrates that the added disk work required to complete a small-write in a RAID level 5 disk array alone creates twenty potential erroneous states. Allowing multiple concurrent operations multiplies the number of erroneous states the array may encounter.

Finally, redundant disk arrays often guarantee *continuous service*, meaning that service is provided without interruption throughout the life of a fault. This requires that all error recovery and fault treatment be performed transparently while the array continues to accept and complete operations from client applications, filesystems, or databases.

1) RD_{OD} precedes WR_{ND}
2) RD_{OD} and RD_{OP} precede WR_{NP}

Figure 4(a): Disk-Work Ordering Constraints

1) RD_{OD} RD_{OP} WR_{ND} WR_{NP}
2) RD_{OD} RD_{OP} WR_{NP} WR_{ND}
3) RD_{OD} WR_{ND} RD_{OP} WR_{NP}
4) RD_{OP} RD_{OP} WR_{ND} WR_{NP}
5) RD_{OP} RD_{OP} WR_{NP} WR_{ND}

Figure 4(b): Valid Disk-Work Orderings

Figure 4 **Ordering of Disk Work for RAID Level 5 Small-Write Operation.** The four disk operations, reading old data (RD_{OD}), reading old parity (RD_{OP}), writing new data (WR_{ND}), and writing new parity (WR_{NP}), may execute concurrently as long as the constraints in Figure 4(a) are satisfied. This allows five possible orderings, shown in Figure 4(b). Because the array is single fault tolerant, any of the four disk operations is allowed to fail. When multiplied across the five possible orderings, this produces twenty erroneous states which the array must handle. This is in stark contrast to a single-actuator disk system which is only required to write new data to disk, fails in only one way, and does not execute multiple operations concurrently.

2.3. Forward Error Recovery is Inadequate

The traditional approach to error recovery in disk systems, *forward recovery*, attempts to remove an error by applying selective corrections to the erroneous state, simultaneously moving operations forward to completion and bringing the system to a consistent state [Anderson81]. Construction of these corrective actions requires detailed foreknowledge of the errors which may occur and the damage that they cause. This requires enumeration of all erroneous states the system may reach.

A significant drawback of this type of recovery is that the *context* of an error, information describing the specific type of activity which failed, is required in order to determine the appropriate set of corrective actions to take. This is because the effects of an error are a function of the context in which it was detected. For instance, in a RAID level 5 array, each disk contains an equal fraction of the data and redundancy information stored in the array. When a disk becomes inaccessible, some operations will not require its data (their parity and data are stored on other disks), some operations will experience loss of data, while others will experience loss of parity.

Figure 5 illustrates forward recovery of a write operation to a RAID level 5 disk array in which an error has occurred, preventing a small-write operation from reading old data. The array must detect the presence of an error during a disk access, recognize the context of the error as “during read of old data in a small-write operation having read old parity already,” move the array to a consistent operating state, and successfully complete the operation. These actions must all be executed with the system in an erroneous state.

Unfortunately, as already shown, error recovery in redundant disk arrays is required to cope with an unmanageable number of erroneous states. One way to simplify the task of forward error recovery is to reduce the number of erroneous states the array may reach. This involves reducing the amount of concurrency in the array, leading to the undesirable result of diminished performance. For instance, one such reduction would be to increase the number of ordering constraints given in Figure 4. The number of states could easily be reduced by forcing data to be written to disk before parity is read and written. Doing this eliminates the ability of the array to perform these in parallel, reducing concurrency and adversely affecting performance.

Another method of simplification is based on recognizing that errors that have identical recoveries can be grouped and handled by a single error recovery. This has the advantage of reducing the number of distinct corrective procedures which must be constructed; however, the task of identifying all erroneous states remains. For example, errors which a disk may present to array software include: head crash, seek error, and electronics failure. All of these errors can be handled in the same fashion by declaring the disk as “failed” and moving the array to a degraded operating state.

allocate memory
read old parity
~~read old data~~
exclusive-or: new data, old data, old parity
write new data
write new parity
deallocate resources

error induces algorithm change →

place array in “degraded” state
discard old parity
allocate additional memory
read surviving data
exclusive-or: new data, surviving data
write new parity
deallocate resources

Figure 5 **Forward Error Recovery can be Complex.** This figure provides an example of the complexity of forward error recovery in a RAID level 5 small-write operation. In this illustration, the erroneous state characterized by the inability of a small-write operation to read old data has been reached. To proceed from this erroneous state and complete the operation, new parity must still be written. This is accomplished according to Figure 3(b), by exclusive-or’ing the surviving data and the new data. The operation must now read surviving data to compute new parity. To do this, additional memory necessary to hold that larger amount of data must be allocated. Because the update procedure has been abandoned, the previously read “old parity” is discarded.

It is important to note that the complexity of error recovery in this operation is not limited by simply knowing what corrective actions to take. For instance, it is possible that the “allocate additional memory” action may create a deadlock condition. Since the system can no longer accurately predict the amount of resources an operation will require, admittance of operations to the system based upon resource scheduling is unreliable. On the other hand, pre-allocating sufficient resources for worst-case error handling in every normal-mode operation is costly and limits normal-mode performance.

Forward error recovery must be designed specifically for each system. This is a result of the dependence upon knowledge of the context in which an error occurs [Randell78]. Because of this, once a design is created, it can be very difficult to make changes to the design, particularly when new error types are introduced or when existing error types are altered. This property limits the scope of modifications to an existing code base, thereby restricting a designer’s ability to explore the design space, confining experimentation to limited departures from the current code structure.

Finally, researchers are investigating more aggressive redundant disk array architectures to boost performance [Bhide92, Blaum94, Cao93, Menon93, Stodolsky93, Holland94]. The acceptance of these proposals is put at risk due to their further increases in the complexity of error handling and the difficulty of modifying existing code structure.

Forward error recovery has been used with arguable success in the design of single disk systems and filesystems. Single disk systems are not fault tolerant and do not execute operations concurrently; hence, error recovery is relatively simple. Operations in a filesystem are complex and are executed concurrently; however, since filesystems are not fault-tolerant, errors which result in a data loss are acceptable. For instance, when the BSD 4.3 UNIX operating system unexpectedly loses access to a disk, data may be lost [Leffler90].

2.4. A Problem Worth Solving

The demand for redundant disk arrays is growing steadily. The value of RAID systems shipped to customers is expected to be $5.0 billion in 1994, reaching $13.0 billion annually by 1997. This compares to the total volume of rigid disks sold, estimated to be $23.7 billion for 1994. Vendors of RAID equipment are under constant pressure to improve performance and decrease development time. The difficulty of handling errors due to disk failures, introduced by the requirement of single fault tolerance, is a limiting factor in the ability of these companies to innovate. Any technology which alleviates this limitation will be both welcomed and encouraged. Our analysis of error recovery in redundant disk arrays suggests that such an opportunity exists.

2.5. Summary

Before continuing, we briefly summarize the problems we have observed and their symptoms. First, redundant disk arrays must provide transparent recovery from errors due to single disk failures. This error recovery is inherently complex and difficult to manage, meaning that implementation is difficult. Second, forward error recovery, the traditional approach to error recovery in nonredundant disk systems, does not scale as complexity is increased, leaving implementors unable to produce more aggressive redundant disk array architectures. Third, the number of erroneous states the system may reach can be decreased by restricting concurrency (adversely affecting performance). Fourth, forward error recovery measures are system specific, limiting the ability to modify existing code and explore the design space.

Redundant disk arrays will always be required to recover from errors due to single disk failures; it is, by definition, what they are designed to do. What we can do is look for a way of making recovery from these errors less painful. To do this, a context-free method of managing complex error recovery which does not degrade performance is needed.

3. Related Work

3.1. Backward Error Recovery

Backward error recovery removes errors from a system by moving the system to a consistent state which existed prior to the introduction of the error. The point of operation that the system attempts to reach during recovery is known as a *recovery point*. A recovery point is established by storing *recovery data*, information which describes the state of the system, as a part of normal processing. When an error is detected, the system is returned to the recovery point by reinstating the recovery data [Randell78, Stone89]. Previously completed work which is undone as a result of moving backward to a recovery point must be redone.

Backward error recovery does not rely upon the type of error or the error's context in removing the error from the system. Thus, context-free error recovery is possible. Also, backward error recovery does not require enumeration of all the erroneous states. This means that backward error recovery is applicable to complex error recovery tasks. Finally, because the error recovery process consists of saving and restoring state, independent of the error context, backward error recovery can be mechanized.

Unfortunately, backward error recovery can be expensive in terms of performance and resource usage, particularly when *atomicity*, the property that operations either complete successfully or leave the system unchanged, is required. Operations composed of actions which guarantee atomicity have particularly simple error recovery. By recovering to the state which existed prior to an operation, backward error recovery techniques achieve atomicity. As the amount of recovery data or the frequency recovery points are established grows, the overhead required to save recovery data increases. This has a direct impact on performance since recovery data is saved as a part of normal processing.

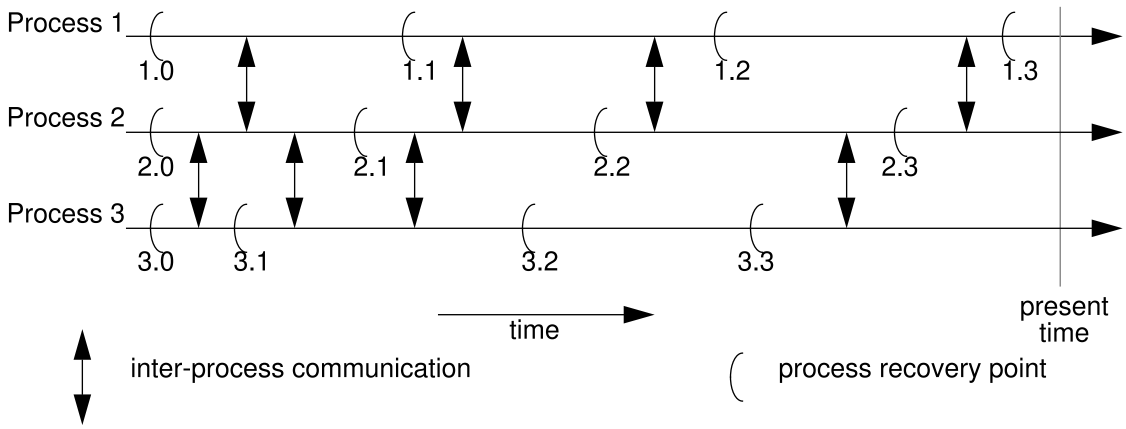

Figure 6 **Effects of Inter-Process Communication on Recovery.** From [Stone89], three communicating processes and the relationship between their recovery points and communications are shown to illustrate the domino effect. Consider recovery, from the present time, of each of the three processes. If Process 1 fails, it recovers to recovery point 1.3 and then continues operation. If Process 2 fails, it recovers to recovery point, 2.3, removing the effects of communication with Process 1, requiring Process 1 to recover to it recovery point 1.2. Finally, if Process 3 fails, it recovers to 3.3, causing Process 2 to recover to 2.2, which in turn causes Process 1 to recover to 1.1. This process continues until processes recover to their initial recovery points, 1.0, 2.0, and 3.0.

Finally, as Randell points out, backward error recovery in systems characterized by communicating processes can lead to disastrous results [Randell75]. The problem, known as the *domino effect*, occurs when communication has taken place between the recovery point and the point in which an error is detected. When recovery is performed, the effects of the communication are undone, requiring recovery of the other processes involved in the communication. An illustration of this problem, taken from [Stone89], is presented as Figure 6. Techniques such as *conversations*, which synchronize communicating processes, are known methods of avoiding the domino effect [Randell75].

A variety of backward error recovery techniques exist, all of which introduce varying degrees of overhead. These techniques fall into three broad classes: checkpointing, recursive caches, and audit trails [Anderson81]. We now examine the applicability of techniques from each of these classes to the domain of redundant disk arrays.

3.2. Checkpointing

Systems employing *checkpointing* establish a recovery point, known as a *checkpoint*, by saving a subset of the system state, known as *checkpoint data* [Chandy72, Siewiorek93]. Erroneous state information is removed by returning the system to a checkpoint which is assumed to be free from error. The process of returning to a checkpoint, referred to as *rollback*, requires the checkpoint data associated with the checkpoint to be reinstated. By returning to a checkpoint, all work performed since the checkpoint is lost and must be performed again.

The overhead of checkpointing depends upon the size of the checkpoint and the frequency of their establishment. The simplest and least effective way to checkpoint a system would be to save the entire state of the system at the start of each process. A more efficient alternative is to save only a subset of the system state. For instance, a technique commonly known as *consistent checkpointing* creates *process checkpoints*, which are checkpoints of the state of a process [Chandy85]. Collectively, these process checkpoints compose a checkpoint of the system.

3.3. Recursive Cache

One solution to the problem of large amounts of recovery data is the *recursive cache*, also known as a *recovery cache* [Horning74]. By monitoring actions which modify the system state, specific state information is saved in a recursive cache, prior to modification. State information is only recorded prior to initial changes from the most recent recovery point, making recursive cache techniques efficient in the sense that the amount of state information in the cache is minimal. Error recovery is performed by restoring the contents of the recursive cache, effectively removing modifications of state and restoring the system to the recovery point. Again, as records are restored, all work which occurred since their entry is lost.

Horning, Lauer, Melliar-Smith, and Randell suggest the use of a recursive cache to implement a *recovery block*, a set of alternate operations, each of which accomplishes the same goal, but through different methods. An *acceptance test* is used to verify correct outcome. When an alternate fails, state is restored from the recovery cache and another alternate is attempted.

A principal difficulty with the recursive cache is the ability to know what state changes an operation will effect upon the system in order that the appropriate information may be entered into the cache. Even with this knowledge, overhead is still introduced when saving recovery data.

3.4. Audit Trails

Finally, *audit trail*, also known as *logging* or *journaling*, techniques provide the ability to record a subset of the system state but, unlike recovery cache techniques, do not require foreknowledge of the state which will be changed by an operation [Bjork75, Verhofstad78, Gray81]. Instead, all changes to the system state are recorded in stable storage. Recovery is performed by applying the inversion of these records in LIFO fashion, thereby removing state changes. As inverted records are applied, work is undone. Once the system is in a consistent state, some records may be applied in FIFO fashion to restore previously completed work. The System R database recovery manager implements such an approach [Gray87].

3.5. Summary

Backward error recovery is well suited for systems in which error recovery is complex. Atomicity is more easily achieved and error recovery is context free. Code modification and enhancement are also simplified. Unfortunately, backward error recovery introduces overhead which degrades normal (error-free) performance. In addition, the process of recovery can remove the effects of previously completed work, therefore requiring a method of reinstating these effects. Furthermore, communicating processes must take special precautions to avoid the domino effect.

4. Approach

Our approach to error recovery is to pursue the advantages of backward error recovery without introducing overhead or effecting previously completed work. It is based upon two assumptions: operations do not guarantee atomicity and operations do not directly communicate with one another.

In the remainder of this section, we examine the details of this approach. We begin by discussing our assumptions. Next, we present our approach for error recovery followed by a description of the error recovery mechanism. We then examine the overhead of this approach and conclude with a discussion of our ability to verify consistent operation of the system.

4.1. Assumptions

First, we assume that filesystems or databases do not expect storage systems to guarantee operational atomicity. We believe this to be reasonable because all storage systems available today can fail and expose partially complete operations. Given freedom from atomicity, we can recover to a convenient consistent state, other than the one which existed prior to the execution of the operation to be recovered and much less expensive to reach.

Second, in our experience, operations in a disk system are independent, only communicating by competing for shared resources. This absence of communication allows us to confine error recovery to the operation which encountered the error, reducing the amount of work undone as a result of backward error recovery. Furthermore, we do not require a method to restore completed operations because only the failing operation is recovered and it is not complete.

4.2. Approach

The goal of the error recovery process is twofold: restore the system to a consistent state and successfully complete operations which encountered an error. Our approach is to use backward error recovery to remove the effects of an error, then move the system to a convenient consistent state and complete recovering operations based on the new operating state. We believe this to be the proper approach for two fundamental reasons. First, by always beginning operations from a consistent state, we greatly reduce the number of paths from starting state to completion state which must be constructed. Second, the error case should not be optimized if it makes normal execution more complex. When an error occurs, consistent operation is more important than minor optimizations. We firmly believe this to be the proper philosophy in highly-concurrent systems such as redundant disk arrays in which error recovery is a complex task which occurs infrequently.

When an error is encountered, our approach requires the following steps be taken:

1. suspend initiation of new operations
2. allow operations already initiated to either complete or reach an error
3. release the resources acquired by operations which encountered an error
4. reconfigure the system
5. using a new strategy, restart operations which encountered an error
6. resume initiation of new operations

In order to transition the system to a consistent state, global state will need to be modified. This is easiest when the system is quiescent. To quiesce the system, incoming operations are queued and operations in the middle of execution are allowed to either complete successfully or execute until an error is encountered. Operations which encounter an error must release all resources which they have acquired. These operations are neither complete nor failed at this point, but are simply suspended until a consistent operating state has been established.

When the system has reached quiescence, the current operating state can be reconciled with the physical state of the system. Once this is done, operations which encountered an error are restarted using a new strategy, appropriate for the current operation state. It is important to understand that the status of these operations during error recovery remains “execution in progress.” The initiation of new operations is also resumed at this time.

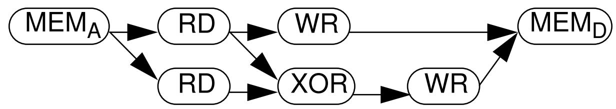

Figure 7 **Antecedence Graph of a RAID Level 5 Small-Write Operation.** Given a block of data, this antecedence graph illustrates the ordering dependencies required to complete a RAID level 5 small-write operation as described in Figure 3(a). Actions include: allocation of memory; reading of old data and old parity; writing of new data; exclusive-or of old data, old parity, and new data to generate the new parity; the writing of new parity; and the deallocation of memory. Since context is not required, disk read operations (RD) do not indicate the logical object type, data or parity, being read. Because of this, error handlers for each of these actions can be constructed and executed independent of operation type.

Finally, it is important to note that some disk systems allow clients to specify the relative ordering of operations [ANSI91]. For example, some filesystems rely upon the ordering of writes to prevent filesystem corruption [Leffler90]. This ordering must be preserved throughout error recovery process.

4.3. Mechanism

The recovery mechanism we present here allows operations to be executed to increase performance during normal operation. Performance is increased by allowing maximal concurrency of actions within an operation and not introducing overhead by saving recovery data. Also, exploration of the design space is enabled by representing operations as a partially-ordered set of actions. We begin the discussion of our mechanism by describing this representation.

To achieve high concurrency during normal operation, we observe that operations perform a specified transformation of state and can be implemented as a partially-ordered set of actions which collectively perform this transformation. An *antecedence graph*, a directed acyclic graph in which the ordering of actions composing an operation is specified, is a natural way to represent an operation, exposes inherent ordering dependencies, and allows maximal concurrency to be achieved. Figure 7 illustrates such a graph for a RAID level 5 small-write operation.

A library of antecedence graphs is constructed from a pre-defined set of actions, such as those found in Table 1. When an operation is initiated in the array, a graph which specifies the work required to complete the operation is selected from this library. The criteria for graph selection includes the type of operation requested and the current operating state. Execution of the graph is dataflow-like, with actions executing when their antecedents have completed. By requiring that all actions return a pass/fail status, the task of satisfying these ordering dependencies becomes straightforward. Obviously, a large variety of graphs can be constructed from a small collection of actions. Because of this, designers are free to experiment with a variety of strategies by simply constructing new graphs and adding them to the library.

Recall from Figure 5 that operations which detect an error are required to alter their strategy to reach completion. Therefore, the antecedence graph currently being executed must be terminated and replaced with a different graph when an error is detected. To do this, forward execution of the current antecedence graph for this operation must be suspended and the resources which it has acquired must be released. This is easily accomplished by allowing actions which have already begun execution to complete and suspending dispatch of further actions in the graph. Once the in-flight actions have completed, we work backward through the graph, releasing resources which were acquired as a part of forward execution. This processes is illustrated in Figure 8 in which a RAID level 5 small-write operation has encountered an error while attempting to write new data to the array.

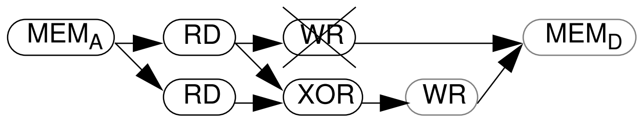

Figure 8(a): Error Encountered During Forward Execution

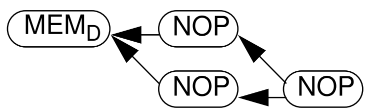

Figure 8(b): UNDO Actions Applied During Backward Execution

Figure 8 **Graceful Termination of a Failed RAID Level 5 Small-Write Antecedence Graph.** In this illustration, a small-write antecedence graph detects an error while attempting to write new data. When the error is detected, execution is suspended, meaning that the XOR operation is allowed to complete, but actions which have not yet been initiated, those in dashed boxes, are not allowed to begin. Once forward execution is halted, backward execution begins, applying the UNDO versions of the actions, found in Table 1. When backward execution completes, all resources allocated by the graph have been released.

To simplify the process of releasing resources we define for every action a corresponding action which releases resources which were acquired. We call these two actions DO and UNDO, respectively. Forward motion through an antecedence graph executes DO actions while backward motion executes UNDO actions. Table 1 summarizes the actions required for implementations of RAID levels discussed in Figure 1.

Error handlers are constructed for each error status that an action might return. For example, a read of a disk (RD in Table 1) could fail due to parity errors, medium failure, or timeout. How these errors are handled is arbitrary as long as they are handled correctly. For example, errors which result from the inaccessibility of a disk which is assumed to be good are obligated to change the array's state information to so that disk is viewed as "inaccessible." By doing this, subsequent operations will not attempt to read the inaccessible device.

Once error handlers have restored the system to a consistent operating state, new graphs are selected for operations which encountered errors and are submitted for execution. These graphs implement different strategies to complete their associated operations, based upon the new operating state. Also, the process of initiating new operations resumes.

Table 1: Actions Required to Implement RAID Operations

DO Action	Description	UNDO Action	Description
RD	read from disk	NOP	no operation
WR	write to disk	NOP	no operation
MEM_A	allocate memory	MEM_D	deallocate memory
MEM_D	deallocate mem.	NOP	no operation
XOR	exclusive-or	NOP	no operation
$LOCK_A$	acquire lock	$LOCK_R$	release lock
$LOCK_R$	release lock	NOP	no operation

4.4. Overhead

As discussed in Section 3., backward error recovery introduces overhead in two ways: resources are required to hold recovery data and work is required to save recovery data during normal processing. Our approach does not introduce overhead since no additional state information is saved as a part of normal processing. The state information we must restore, resources which have been acquired, is already known. The method used to release these resources is determined via a table-lookup during error recovery.

Additionally, since operations do not communicate, our unit of recovery is an operation and we avoid the domino effect. We are not required to undo previously completed operations. Therefore, a log of completed work does not need to be maintained.

Finally, unlike forward error recovery, we do not embed code throughout the forward execution path to identify the state of the system at the completion of each action; rather, we simply assess each action as pass/fail and then continue forward or backward.

4.5. Consistent Operation

By specifying an operation, its antecedence graph, and the actions in the graph, we can reason about the correctness of an operation. This is accomplished by showing a correspondence between the specification of an operation and its implementation which is represented as the antecedence graph.

Consistent operation of a redundant disk array requires that *invariants*, specified relationships between a data object and the redundancy object associated with it, be maintained. Guaranteeing that invariants are maintained is trivial for a nondestructive operation, such as a read, which alters neither data nor redundancy. Destructive operations, such as write, are obligated to modify both data and redundancy. When a write operation completes, these modifications must satisfy invariants between data and parity.

When a failure occurs during a write operation in a redundant disk array, either data or redundancy will be inaccessible. The surviving data or redundancy object will be in an indeterminate, but accessible, state since the error may have occurred either before or after its update was performed. Consistent recovery, therefore, requires the ability to <u>overwrite</u> an indeterminate object, making it correct. This process of resolving determinacy is a key component of the alternative operation strategies of a retry.

4.6. Summary

Our approach to the handling of errors in redundant disk arrays is based upon retry, rather than continuation, of operations which encounter an error. To simplify our approach, we make two assumptions regarding operations: they do not guarantee atomicity and they do not communicate. From these assumptions, we are able to construct an error recovery mechanism which does not introduce overhead during normal processing.

When an error is encountered, we quiesce the system, reconfigure to achieve a consistent state, and retry operations which encountered an error.

Operations are represented as antecedence graphs, allowing clear reasoning about the ordering of actions which compose an operation and the exploit of concurrency. New types of antecedence graphs are easily created and integrated into the system, greatly simplifying the task of exploring new implementation strategies.

Finally, by specifying state transformations of operations, antecedence graphs, and actions, we can demonstrate correctness, either formally or informally.

5. Conclusions and Future Work

5.1. Conclusions

By making the handling of errors independent of the context in which they occur, we allow code modules to be more easily modified. This makes exploration of the design space easier, allowing designers of redundant disk arrays to spend more time formulating an approach and less time implementing it.

By simplifying the design process, we enable production of more aggressive RAID algorithms which, in today's environment, are arguably too complex.

By using antecedence graphs as an execution model for an operation, we expose the inherent ordering of actions which compose an operation. This simplifies the scheduling of these actions, making concurrency easier to implement.

Finally, by structuring our design and error handling process, we enable verification of the correctness of our design. From specifications of operations and error handlers, correctness can be argued either formally or informally.

5.2. Future Work

Work is in progress to verify our approach. We are concentrating on three efforts to validate correctness, performance, and complexity reduction. First, we are specifying RAID in general and a left-symmetric implementation of RAID level 5 in particular. This will allow us to argue correctness. Second, we are implementing a left-symmetric RAID level 5 driver to verify performance and correct operation. Finally, we will modify this driver, employing more aggressive algorithms, to demonstrate code reusability, the ability to implement more aggressive RAID technology, and the ability to explore the design space by simply composing new operations from an existing set of actions.

6. Acknowledgments

We thank Mark Holland and Daniel Stodolsky for enlightening discussions which provided valuable insights into this work. Also, Daniel Stodolsky provided careful proofreading and correcting of early drafts of this paper. Finally, we thank Jeannette Wing for her many hours of patient listening and instruction regarding the applicability of formal specification to the problem redundant disk array design.

7. References

[Anderson79] T. Anderson and B. Randell, Computing Systems Reliability, Cambridge University Press, 1979.

[ANSI91] Small Computer System Interface - 2 (SCSI-2), American National Standard for Information systems, X3T9.2/86-109, Revision 10h, X3T9/89-042, Global Engineering Documents, X3.131-199x, Irvine CA, October 17, 1991.

[Anderson81] T. Anderson and P. A. Lee, Fault Tolerance, Principles and Practice, Prentice-Hall, 1981.

[Anderson82] T. Anderson and P. A. Lee "Fault tolerance terminology proposals." In *Proceedings of the 12th Annual International Symposium on Fault Tolerant Computing (FTCS)*, Santa Monica CA, June 1982, pp. 29-33.

[Bhide92] A. Bhide and D. Dias, “RAID architectures for OLTP.” IBM Computer Science Research Report RC 17879, 1992.

[Bjork75] L. A. Bjork, Jr., “Generalized audit trail requirements and concepts for data base applications.” *IBM Systems Journal*, Vol. 14, No. 3, 1975, pp. 229-245.

[Blaum94] Mario Blaum, Jim Brady, Jehoshua Bruk, Jai Menon, “EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures.” In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, Chicago IL, April 18-21, 1994, pp. 245-254.

[Cao93] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes, “The TickerTAIP parallel RAID architecture.” In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego CA, May 1993, pp. 52-63.

[Chandy72] K. M. Chandy and C. V. Ramamoorthy, “Rollback and recovery strategies for computer programs.” *IEEE Transactions on Computers*, Vol. C-21, No. 6, June 1972, pp. 546-556.

[Chandy85] K. Mani Chandy and Leslie Lamport, “Distributed snapshots: determining global states of distributed systems.” *ACM Transactions on Computer Systems*, Vol. 3, No. 1, Feb. 1985, pp. 63-75.

[Gibson89] Garth A. Gibson, “Performance and reliability in redundant arrays of inexpensive disks (RAID).” In *Proceedings of the 1989 Computer Measurement Group conference (CMG)*, Reno NV, December 1989, pp. 381-391.

[Gibson92] Garth A. Gibson, Redundant Disk Arrays: Reliable, Parallel Secondary Storage, The MIT Press, 1992.

[Gibson93] Garth A. Gibson, David A. Patterson, “Designing disk arrays for high data reliability.” *Journal of Parallel and Distributed Computing*, Vol. 17, No. 1-2, Jan.-Feb. 1993, pp. 4-27.

[Gray81] Jim Gray, “Notes on data base operating systems.” lecture notes from The Advanced Course in Operating Systems, July 28-August 5, 1977, Technical University, Munich, Federal Republic of Germany, published in Operating Systems: An Advanced Course, Vol. 60 of the series “Lecture Notes in Computer Science,” Springer-Verlag, 1981, pp. 393-481.

[Gray87] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, “The recovery manager of the System R database manager.” *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 223-242.

[Holland94] Mark Holland, “On-line data reconstruction in redundant disk arrays.” Ph.D. dissertation, Carnegie Mellon University School of Computer Science technical report CMU-CS-94-164, May 1994.

[Horning74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, B. Randell, “A program structure for error detection and recovery.” Proceedings of an International Symposium held at Rocquencourt, April 23-25 1974, published in *Lecture Notes in Computer Science*, Vol. 16, Springer-Verlag, 1974, pp. 171-187.

[Katz89] Randy H. Katz, Garth A. Gibson, David A. Patterson, “Disk system architectures for high performance computing.” In *Proceedings of the IEEE*, Vol. 77, No. 12, December 1989, pp. 1842-1858. Also published in *CMG Transactions*, issue 74, fall 1991, pp. 27-46.

[Kim86] Michelle Y. Kim, “Synchronized disk interleaving.” *IEEE Transactions on Computers*, Vol. 35, No. 11, November 1986, pp. 978-988.

[Lampson79] Butler W. Lampson and Howard E. Sturgis, “Crash recovery in a distributed data storage system.” Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, April 27, 1979.

[Lee82] P. A. Lee and T. Anderson, “Fundamental concepts of fault tolerant computing: progress report.” In *Proceedings of the 12th Annual International Symposium on Fault Tolerant Computing (FTCS)*, Santa Monica CA, June 1982, pp. 34-38.

[Lee90] Edward K. Lee and Randy H. Katz, “Performance considerations of parity placement in disk arrays.” In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, Palo Alto CA, April 1991, pp. 190-199.

[Leffler90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, <u>The Design and Implementation of the 4.3BSD UNIX Operating System</u>, Addison-Wesley, Reading MA, 1990.

[Melliar-Smith77] P. M. Melliar-Smith and B. Randell, “Software reliability: the role of programmed exception handling.” In *Proceedings of an ACM Conference on Language Design for Reliable Software*, Raleigh NC, March 1977, pp. 95-100.

[Menon93] J. Menon, J. Roche, and J. Kasson, “Floating parity and data disk arrays.” *Journal of Parallel and Distributed Computing*, Vol. 17, No. 1-2, Jan.-Feb. 1993, pp. 129-139.

[Patterson88] David A. Patterson, Garth A. Gibson, and Randy H. Katz, “A case for redundant arrays of inexpensive disks (RAID).” In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, Chicago IL, June 1988, pp. 109-116. Also published in *CMG Transactions*, issue 74, fall 1991, pp. 13-25.

[Randell75] Brian Randell, “System structure for software fault tolerance.” *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[Randell78] B. Randell, P. A. Lee, and P. C. Treleaven, “Reliability issues in computing system design.” *ACM Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

[Reddy89] A. L. Narasimha Reddy and Prithviraj Banerjee, “An evaluation of multiple-disk I/O systems.” *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1680-1690.

[Salem86] K. Salem and H. Garcia-Molina, “Disk Striping.” In *Proceedings of the 2nd International Conference on Data Engineering*, IEEE CS Press, Los Alamitos, CA Order No. 827 (microfiche only), 1986, pp. 336-342.

[Siewiorek92] Daniel P. Siewiorek and Robert S. Swarz, <u>Reliable Computer Systems: Design and Evaluation, Second Edition</u>, Digital Press, 1992

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

[Stodolsky93] Daniel Stodolsky, Garth Gibson, Mark Holland, “Parity logging: overcoming the small write problem in redundant disk arrays.” In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego CA, May 1993, pp 64-75.

[Verhofstad78] Joost S. M. Verhofstad, “Recovery techniques for database systems.” *ACM Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 167-195.