Backward Error Recovery in Redundant Disk Arrays (CMU-CS-94-193)

William V. Courtright II  
*Carnegie Mellon University*

Garth A. Gibson  
*Carnegie Mellon University*

Follow this and additional works at: [http://repository.cmu.edu/pdl](http://repository.cmu.edu/pdl)
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.

The traditional approach to recovering from errors in RAID systems has been 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.
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 all data units in the parity group.

RAID level 5 is block interleaved to optimize throughput and uses parity-based redundancy. Both data and parity are evenly distributed throughout the array. A variety of strategies exist to evenly distribute data units and parity units; this illustration uses the left-symmetric layout [Lee90]. Parity is the bitwise exclusive-or of all data units in the parity group.

The remaining RAID levels, 2 and 4, are not shown. RAID level 2 employs Hamming codes and does not rely on a disk’s ECC logic to report errors. Since commodity drives all have sufficient ECC logic, the space and complexity overhead of Hamming codes is generally considered to be unwarranted. RAID level 4 is similar to RAID level 5 but does not evenly distribute the parity, creating “hot spots” in the array [Patterson88].

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 sum-
2. Motivation

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.

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.

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].
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.

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.
Forward Error Recovery is Inadequate

The traditional approach to error recovery in disk systems, forward recovery, attempts to remove an error by applying corrective actions 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.

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, Stoddsky93, 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].

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.

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

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.
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.

**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]. Errorneous 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.

**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.

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].

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.

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.

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.

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.

**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

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>UNDO Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
<td>read from disk</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>WR</td>
<td>write to disk</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>MEMA</td>
<td>allocate memory</td>
<td>MEMD</td>
<td>deallocate mem.</td>
</tr>
<tr>
<td>MEMD</td>
<td>deallocate mem.</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>XOR</td>
<td>exclusive-or</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>LOCKA</td>
<td>acquire lock</td>
<td>LOCKR</td>
<td>release lock</td>
</tr>
<tr>
<td>LOCKR</td>
<td>release lock</td>
<td>NOP</td>
<td>no operation</td>
</tr>
</tbody>
</table>
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 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.

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.

**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.

**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 overwrite an indeterminate object, making it correct. This process of resolving determinacy is a key component of the alternative operation strategies of a retry.

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

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.

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


[Bjork75] L. A. Bjork, Jr., “Generalized audit trail requirements and


