Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing, 1993.

Fast, On-Line Failure Recovery in Redundant Disk Arrays

Mark Holland
Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3890

Garth A. Gibson, Daniel P. Siewiorek
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3890

Abstract

This paper describes and evaluates two algorithms for performing on-line failure recovery (data reconstruction) in redundant disk arrays. It presents an implementation of disk-oriented reconstruction*, a data recovery algorithm that allows the reconstruction process to absorb essentially all the disk bandwidth not consumed by the user processes, and then compares this algorithm to a previous-proposed* parallel stripe-oriented *approach. The disk-oriented approach yields better overall failure-recovery performance.*

The paper evaluates performance via detailed simulation on two different disk array architectures: the RAID level 5 *organization, and the* declustered parity *organization. The benefits of the disk-oriented algorithm can be achieved using controller or host buffer memory no larger than the size of three disk tracks per disk in the array. This paper also investigates the tradeoffs involved in selecting the size of the disk accesses used by the failure recovery process.*

1. Introduction

Fast, on-line recovery from disk failures is crucial to applications such as on-line transaction processing that require both high performance and high data availability from their storage subsystems. Such systems demand not only the ability to recover from a disk failure without losing data, but also that the recovery process (1) operate without taking the system off-line, (2) rapidly restore the system to the fault-free state, and (3) have minimal impact on system performance as observed by users. A good example is an airline reservation system, where inadequate recovery from a disk crash can cause an interruption in the availability of booking information and thus lead to flight delays and/or revenue loss. With this in mind, the twin goals of the techniques proposed in this paper are to minimize the time taken to recover from a disk failure, i.e. restore the system to the fault-free state, and to simultaneously minimize the impact of the failure recovery process on the performance of the array.

Fault-tolerance in a data storage subsystem is generally achieved either by *disk mirroring* [Bitton88, Copeland89], or *parity encoding* [Gibson93, Kim86, Patterson88]. In the former, one or more duplicate copies of all data are stored on separate disks. In the latter, popularized as Redundant Arrays of Inexpensive Disks (RAID) [Patterson88, Gibson92], a portion of the data blocks in the array is used to store an error-correcting code computed over the remaining blocks. Mirroring can potentially deliver higher performance than parity-encoding [Chen90a, Gray90], but it is expensive in that it incurs a capacity overhead of at least 100%. Furthermore, recent work on overcoming the bottlenecks in parity-encoded arrays [Stodolsky93, Menon92, Rosenblum91] has demonstrated techniques that allow the performance of these arrays to approach that of mirroring. This paper focuses on parity-encoded arrays.

Section 2 of this paper provides background material about disk arrays, and describes the failure-recovery problem. It presents the RAID level 5 and declustered-parity architectures and describes some previous approaches to on-line failure recovery. Section 3 develops the disk-oriented reconstruction algorithm and Section 4 evaluates it using a disk-accurate event-driven simulator. Section 5 provides a summary and some conclusions.

2. Background

This section presents a brief overview of the redundant disk array systems considered in this paper.

2.1. Disk arrays

Patterson et. al. [Patterson88] present a number of possible redundant disk array architectures, which they call RAID levels 1 through 5. Some of these are intended to provide large amounts of data to a single process at high speeds, while others are intended to provide highly concurrent access to shared files. The latter organizations are preferable for OLTP-class applications, since such appli-

This work was supported by the National Science Foundation under grant number ECD-8907068, by the Defense Advanced Research Projects Agency monitored by DARPA/CMO under contract MDA972-90-C-0035, and by an IBM Graduate Fellowship.

The authors can be reached as mark.holland@ece.cmu.edu, garth.gibson@cs.cmu.edu, and dan.siewiorek@cs.cmu.edu.

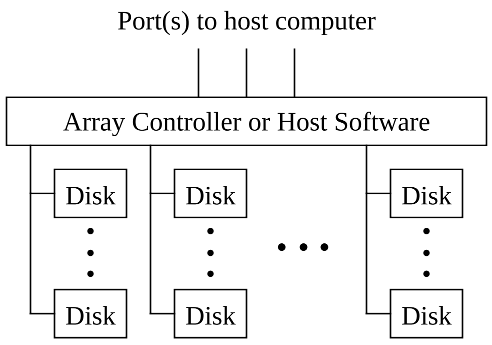
cations are characterized by a large number of independent processes concurrently requesting access to relatively small units of data [TPCA89]. For this reason, this paper focuses on architectures derived from the RAID level 5 organization. Figure 1a illustrates a storage subsystem employing this organization. The array controller is responsible for all system-related activity: communicating with the host, controlling the individual disks, maintaining redundant information, recovering from failures, etc. The controller is often duplicated so that it does not represent a single point of failure. The functionality of the controller is sometimes implemented in host software rather than as dedicated hardware. The algorithms and analyses presented in this paper apply equally well to either implementation.

Figure 1b shows the arrangement of data on the disks comprising the array, using the “left-symmetric” variant of the RAID level 5 architecture [Chen90b, Lee91]. Logically contiguous user data is broken down into blocks and striped across the disks to allow for concurrent access by independent processes. The shaded blocks, labelled *Pi*, store the parity (cumulative exclusive-or) computed over the corresponding data blocks, labelled *Di.0* through *Di.3*. An individual block is called a *data unit* if it contains user data, a *parity unit* if it contains parity, and simply a *unit* when the data/parity distinction is not pertinent. A set of data units and their corresponding parity unit is referred to as a *parity stripe*. The assignment of parity blocks to disks rotates across the array in order to avoid hot-spot contention; every update to a data unit implies that a parity unit must also be updated, so if the distribution of parity across disks is not balanced, the disks with more parity will see more work.

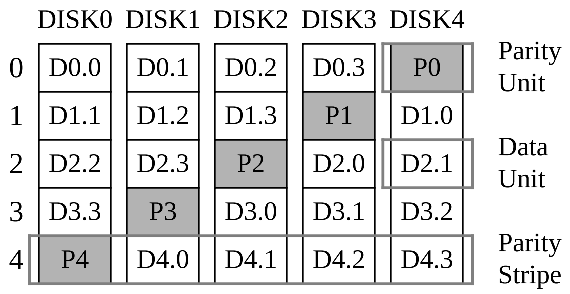
Because disk failures are detectable [Patterson88, Gibson93], arrays of disks constitute an erasure channel [Peterson72], and so a parity code can correct any single disk failure. To see this, assume that disk number two has failed and simply note that

$$(Pi = Di.0 \oplus Di.1 \oplus Di.2 \oplus Di.3) \Rightarrow$$
$$(Di.2 = Di.0 \oplus Di.1 \oplus Pi \oplus Di.2)\,.$$

An array can be restored to the fault-free state by successively reconstructing each block of the failed disk and storing it on a replacement drive. This process is termed *reconstruction*, and is generally performed by a background process created in either the host or the array controller. Note that the array need not be taken off-line to implement reconstruction, since reconstruction accesses can be interleaved with user accesses, and since user accesses to failed data can be serviced via on-the-fly reconstruction. Once reconstruction is complete, the array can again tolerate the loss of any single disk, and so it can



(a) Array subsystem configuration



(b) Data layout over five disks

Figure 1: A system employing the left-symmetric RAID level 5 organization

be considered to be fault-free, albeit with a diminished number of on-line spare disks until the faulty drives can be physically replaced. Gibson [Gibson93] shows that a small number of spare disks suffices to provide a high degree of protection against data loss in relatively large arrays (>70 disks). Although the above organization can be easily extended to tolerate multiple disk failures, this paper focuses on single-failure toleration.

2.2. The reconstruction problem

After a disk failure (1) a RAID level 5 array is vulnerable to data loss due to a second failure, and (2) the load on each surviving disk increases by 100% servicing user-invoked on-the-fly reconstruction, and then still more during reconstruction on to a replacement. The load increase arises because every user access to a unit on the failed disk causes one request to be sent to each of the surviving disks, and so in the presence of failure each surviving disk must service its own request stream, an equivalent stream generated by requests to the failed disk, and the requests generated by the background reconstruction process.

It is now possible to state the problem this paper addresses: the goals of designing disk arrays for on-line reconstruction are to minimize the time taken to reconstruct the entire contents of a failed disk and store it on a spare disk, and to incur the minimum possible user performance degradation while accomplishing this. The figures of merit this paper uses are therefore (1) *reconstruction time*, which is the time from invocation of reconstruction to its completion, and (2) *average user response time during reconstruction*, which is the average latency experienced by a user's request to read or write a block of data while reconstruction is ongoing.

A *reconstruction algorithm* is a strategy used by a background reconstruction process to regenerate data resident on the failed disk and store in on a replacement. The most straightforward approach, which we term the *stripe-oriented* algorithm, is as follows:

```
for each unit on the failed disk
   1. Identify the parity stripe to which
      unit belongs.
   2. Issue low-priority read requests for all
      other units in the stripe, including
      parity unit.
   3. Wait until all reads have completed.
   4. Compute the XOR over all units read.
   5. Issue a low-priority write request to the
      replacement disk.
   6. Wait for the write to complete.
end
```

This stripe-oriented algorithm uses low-priority requests in order to minimize the impact of reconstruction on user response time, since commodity disk drives do not generally support any form of preemptive access. A low-priority request is used even for the write to the replacement disk, since this disk services writes in the user request stream as well as reconstruction writes [Holland92]. Section 3 demonstrates that this simple algorithm has a number of drawbacks, and describes an algorithm to address them.

2.3. The declustered parity organization

The RAID level 5 architecture described above, while viable for applications that can tolerate data unavailability during recovery, presents a problem for continuous-operation systems like OLTP: the greater than 100% per-disk load increase experienced during reconstruction necessitates that each disk be loaded at less than 50% of its capacity in the fault-free state so that the surviving disks will not saturate when a failure occurs. Disk saturation is unacceptable for OLTP applications because they mandate a minimum acceptable level of responsiveness; the TPC-A benchmark [TPCA89], for example, requires that 90% of all transactions complete in under two seconds. The long queueing delays caused by saturation can violate these requirements.

The *declustered parity* disk array organization [Muntz90, Holland92, Merchant92] addresses this problem. This scheme reduces the per-disk load increase caused by a failure from over 100% to an arbitrarily small percentage by increasing the amount of error correcting information that is stored in the array. Declustered parity can be thought of as trading some of the array's data capacity for improved performance in the presence of disk failure. This translates into an improvement in fault-free performance by allowing the disks to be driven at greater than 50% utilization without risking unacceptable user response time after a failure occurs.

To understand parity declustering, begin with Figure 1b, and note that each parity unit protects C-1 data units, where C is the number of disks in the array. If instead the array were organized such that each parity unit protected some smaller number of data units, say G-1, then more of the array's capacity would be consumed by parity, but the reconstruction of a single data unit would require that the reconstruction process read only G-1 units instead of C-1. This would mean that not every surviving disk would be involved in the reconstruction of every data unit; C-G disks would be left free to do other work. Thus each surviving disk would see a user-invoked on-the-fly reconstruction load increase of $(G-1)/(C-1)$ instead of $(C-1)/(C-1) = 100\%$. The fraction $(G-1)/(C-1)$, which is referred to as the *declustering ratio* and denoted by α, can be made arbitrarily small either by increasing C for a fixed G, or by decreasing G for a fixed C.

For $G=2$ the declustered parity scheme is similar to mirroring, except that the parity, which duplicates the data, is distributed over all disks in the array. At the other extreme, when $G=C$ ($\alpha = 1.0$) parity declustering is equivalent to RAID level 5. The performance plots in subsequent sections are presented with α on the x-axis, since α has direct impact on the recovery-mode performance and its associated cost.

Figure 2 shows a portion[1] of a parity-declustered data layout where G, the number of units in a parity stripe, is four, but C, the number of disks in the array, is five, resulting in $\alpha = 0.75$. The figure illustrates that each surviving disk absorbs only 75% of the failure-induced load. If, for example, disk 0 fails and the data unit marked *D0.0* is reconstructed, disk 4 is not involved at all.The technique by which data and parity units are assigned to disks for general C and G is beyond the scope of the discussion

1. Parity is not balanced in this diagram because it shows only a portion of the data layout. A complete figure would show that the layout does actually assign an equal number of parity units to each disk.

Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	P1
1	D1.0	D1.1	D1.2	D2.2	P2
2	D2.0	D2.1	D3.1	D3.2	P3
3	D3.0	D4.0	D4.1	D4.2	P4

Figure 2: Example data layout in the declustered parity organization.

(refer to [Holland92] or [Merchant92]).

We use a parity-declustered architecture for comparing reconstruction algorithms in Section 4, and employ the data layout described in Holland and Gibson [Holland92]. To evaluate each algorithm for RAID level 5 arrays, consider only the points where α = 1.0.

3. The reconstruction algorithm

This section identifies the problems with the simple algorithm presented in Section 2.2, and develops the disk-oriented approach. The main idea behind this technique has been suggested in previous studies [Merchant92, Hou93], but personal communications with disk array manufacturers indicate that both algorithms are currently used in disk array products. This paper provides a detailed analysis of the trade-offs between these algorithms.

3.1. The reconstruction algorithm

The problem with the stripe-oriented reconstruction algorithm is that it is unable to consistently utilize all the disk bandwidth that is not absorbed by users. This inability stems from three sources. First, it does not overlap reads of the surviving disks with writes to the replacement, so the surviving disks are idle with respect to reconstruction during the write to the replacement, and vice versa. Second, the algorithm simultaneously issues all the reconstruction reads associated with a particular parity stripe, and then waits for all to complete. Some of these read requests will take longer to complete than others, since the depth of the disk queues will not be identical for all disks and since the disk heads will be in essentially random positions with respect to each other. Therefore, during the read phase of the reconstruction loop, each involved disk may be idle from the time that it completes its own reconstruction read until the time that the slowest read completes. Third, in the declustered parity architecture, not every disk is involved in the reconstruction of every parity stripe, and so uninvolved disks remain idle with respect to reconstruction since the reconstruction algorithm works on only one parity stripe at a time.

One way to address these problems is to reconstruct multiple parity stripes in parallel [Holland92]. In this approach, the host or array controller creates a set of *N* identical, independent reconstruction processes instead of just one. Each process executes the single stripe-oriented algorithm of Section 2.2, except that the next parity stripe to reconstruct is selected by accessing a shared list of as-yet unreconstructed parity stripes, so as to avoid duplication. Since different parity stripes use different sets of disks, the reconstruction process is able to absorb more of the array's unused bandwidth than in the single-process case, because it allows for concurrent accesses on more than *G*-1 disks.

Although this approach yields substantial improvement in reconstruction time, it does so in a haphazard fashion. Disks may still idle with respect to reconstruction because the set of data and parity units that comprise a group of *N* parity stripes is in no way guaranteed to use all the disks in the array evenly. Furthermore, the number of outstanding disk requests each independent reconstruction process maintains varies as accesses are issued and complete, and so the number of such processes must be large if the array is to be consistently utilized.

The deficiencies of both single-stripe and parallel-stripe reconstruction can be addressed by restructuring the reconstruction algorithm so that it is *disk-oriented* instead of *stripe-oriented*. Instead of creating a set of reconstruction processes associated with stripes, the host or array controller creates *C* processes, each associated with one disk. Each of the *C*-1 processes associated with a surviving disk execute the following loop:

```
repeat
  1. Find the lowest-numbered unit on this
     disk that is needed for reconstruction.
  2. Issue a low-priority request to read the
     indicated unit into a buffer
  3. Wait for the read to complete.
  4. Submit the unit's data to a centralized
     buffer manager for subsequent XOR.
until (all necessary units have been read)
```

The process associated with the replacement disk executes:

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

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

The advantage of this approach is that it is able to maintain one low-priority request in the queue for each disk at all times, which means that it will absorb all of the array's bandwidth that is not absorbed by users. Section 4 demonstrates that disk-oriented reconstruction does substantially reduce reconstruction time when compared to the stripe-oriented approach, while penalizing average response time by only a small factor.

3.2. Implementation issues

There are two implementation issues that need to be addressed in order for the above algorithm to perform as expected. The first relates to the amount of memory needed to implement the algorithm, and the second to the interaction of reconstruction with updates in the normal workload. This section briefly outlines our approach to each.

3.2.1. Memory requirements

In the stripe-oriented algorithm, the host or array controller never needs to buffer more than one complete parity stripe per reconstruction process. In the disk-oriented algorithm, however, transient fluctuations in the arrival rate of user requests at various disks can cause some reconstruction processes to read data more rapidly than others. This data must be buffered until the corresponding data arrives from slower reconstruction processes. It's possible to construct pathological conditions in which a substantial fraction of the data space of the array needs to be buffered in memory.

The amount of memory needed for disk-oriented reconstruction can be bounded by enforcing a limit on the number of buffers employed. If no buffers are available, a requesting process blocks until a buffer is freed by some other process.

In our implementation, this buffer pool is broken into two parts: each surviving-disk reconstruction process has one buffer assigned for its exclusive use, and all remaining buffers are assigned to a "free buffer pool." A surviving-disk process always reads units into its exclusive buffer, but then upon submission to the buffer manager, the data is transferred to a free buffer. Only the first process submitting data for a particular parity stripe must acquire a free buffer, because subsequent submissions for that parity stripe can be XORed into this buffer.

Forcing reconstruction processes to stall when there are no available free buffers causes the corresponding disks to idle with respect to reconstruction. In practice we find that a small number of free buffers suffices to achieve good performance. There should be at least as many free buffers as there are surviving disks, so that in the worst case each reconstruction process can have one access in progress and one buffer submitted to the buffer manager. Section 4 demonstrates that using this minimum number of buffers is in general adequate to achieve most of the benefits of the disk-oriented algorithm, and using about twice as many free buffers as disks reduces the buffer stall penalty to nearly its minimum. This requires about two megabytes of memory for a moderately-sized array (21 320MB disks), which can be used for other purposes (such as caching or logging) when the array is fault-free.

3.2.2. Writes in the normal workload

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

There are a number of approaches to this interlocking problem. The buffered partially-reconstructed units could be flushed when a conflicting user write is detected. Alternatively, buffered units could be treated as a cache, and user writes could update any previously-read information before a replacement-disk write is issued. A third approach would be to delay the initiation of a conflicting user write until the desired stripe's reconstruction is complete.

We rejected the first option as wasteful of disk bandwidth. We rejected the second because it requires that the host or array controller buffer each individual data and parity unit until all have arrived for one parity stripe, rather than just buffering the accumulating XOR for each parity stripe. This would have multiplied the memory requirements in the host or controller by a factor of at least G-1. The third option is memory-efficient and does not waste disk bandwidth, but if it is implemented as stated, a user write may experience a very long latency when it is forced to wait for a number of low-priority accesses to complete. This drawback can be overcome if it is possible to expedite the reconstruction of a parity stripe containing the data unit that is being written by the user, which is what we implemented. When a user write is detected to a data unit in a parity stripe that is currently under reconstruction, all pending accesses for that reconstruction are elevated to the priority of user accesses. If there are any reconstruction accesses for the indicated parity stripe that have not yet been issued, they are issued immediately at user-access priority. The user write triggering the re-prioritization stalls until the expedited reconstruction is complete, and is then allowed to proceed normally.

Note that a user write to a lost and unreconstructed data unit implies an on-the-fly reconstruction, because all written data must be incorporated into the parity, and there is no way to do this without the previous value of the affected disk unit. Thus, our approach to interlocking reconstruction with user writes does not incur any avoidable disk accesses. Also, in practice, forcing the user write to wait for an expedited reconstruction does not significantly elevate average user response time, because the number of parity stripes that are under reconstruction at any given moment (typically less than 50) is small with respect to the total number of parity stripes in the array (many thousand).

4. Performance evaluation

This section presents the results of detailed simulation evaluating the above algorithms. It analyzes reconstruction performance, memory requirements, and the reconstruction unit size.

4.1. Comparing reconstruction algorithms

This section presents the results of a simulation study comparing the two reconstruction algorithms: parallel stripe-oriented and disk-oriented. This and all subsequent performance analyses in this paper were performed using an event-driven disk array simulator called *raidSim* [Lee91, Chen90b, Holland92]. RaidSim contains a realistic disk model, which was calibrated to an IBM Model 0661 (Lightning) drive [IBM0661]. The simulated array consists of 21 spin-synchronized disks, using data units that are one track (24KB) in size [Chen90b]. Our synthetically-generated workload was derived from access statistics taken from a trace of an airline-reservation OLTP system [Ramakrishnan92], and consisted of 80% 4KB reads, 16% 4KB writes, 2% 24KB reads, and 2% 24KB writes. Accesses were randomly distributed in the data space of the array, and the access rate was regulated to present 294 user accesses per second to the array. This rate was selected because it causes the fault-free array to be loaded at slightly less than 50% utilization, and hence is the maximum that a RAID level 5 organization can support in the presence of a disk failure. All reported results are averages over five independently-seeded simulation runs. Our simulations assume that the disks do not support zero-latency reads, since commodity disks rarely do. Given that disk units are track-sized, this feature would substantially benefit the reconstruction process.

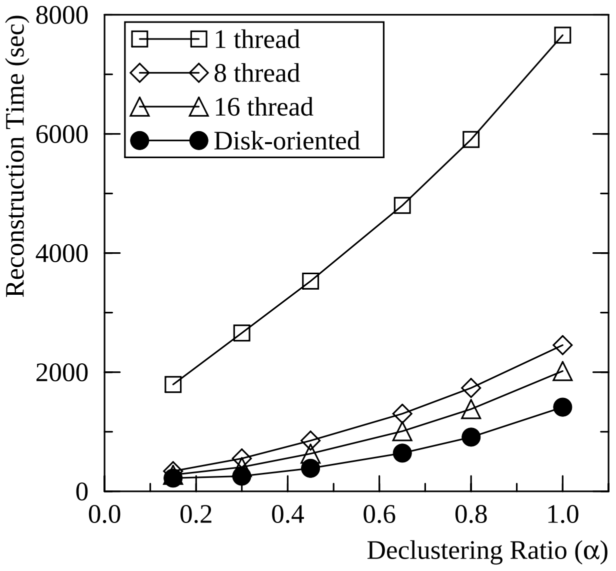
Figure 3a shows the resultant reconstruction time for parallel stripe-oriented reconstruction (1, 8, and 16 way parallel) and for disk-oriented reconstruction. On the x-axis is the declustering ratio, which is varied from 0.15 (G=4) to 1.0 (recall that α = 1.0 is equivalent to RAID level 5)[2]. On the y-axis is the total time taken to reconstruct a failed disk while the array is supporting the user workload of 294 user accesses per second. Figure 3b shows the average response time for a user access while reconstruction is ongoing, for the same configurations and ranges on α. Reconstruction accesses were one track in size, and fifty free reconstruction buffers were used for the disk-oriented runs, for reasons explained below.

Independent of the reconstruction algorithm, Figure 3 shows the efficacy of parity declustering in improving the array's performance under failure. Both reconstruction time and average user response time during recovery steadily improve as the declustering ratio is decreased. The flattening out of the reconstruction time curve at low declustering ratios is caused by saturation on the replacement disk; the simulations show nearly 100% disk utilization at these points. Note, however, that this saturation does not cause excessive user response time (recall from Section 2.2 that the replacement disk services user writes as well as reconstruction writes), because user accesses are strictly prioritized with respect to reconstruction accesses, and the user workload does not by itself saturate the replacement disk. Refer to Holland and Gibson [Holland92] or Merchant and Yu [Merchant92] for more details on the advantages of parity declustering.

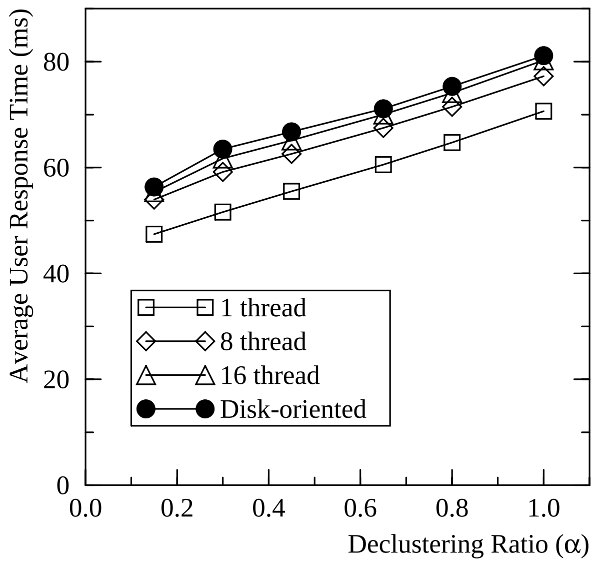
More to the point, Figure 3 shows that the disk-oriented algorithm makes much more efficient use of the available disk bandwidth than the stripe-oriented algorithm; reconstruction time is reduced by 30-40% over the 16-way parallel stripe-oriented algorithm, with essentially the same average user response time. The average user response time is slightly worse for the disk-oriented algorithm than for the 1- and 8-way parallel stripe-oriented approaches, because under disk-oriented reconstruction, disks are so well-utilized that a user request almost never experiences zero queueing time.

For completeness, Figure 4 shows the coefficient of

2. We did not simulate the mirroring case (α = 0.05) because our simulator does not implement mirroring-specific optimizations such as reading the copy that requires the shortest seek. We chose not to make such an unfair comparison.



(a) Reconstruction time



(b) Average user response time during recovery

Figure 3: Comparing reconstruction algorithms

variation (the standard deviation divided by the mean, denoted "COV") for the user response time in the simulations of Figure 3. At high values of α, the surviving disks see a heavy workload, and so the response time shows a large variance. The disk-oriented algorithm yields a slightly smaller variance in user response time because of its more balanced disk utilization.

4.2. Memory requirements analysis

As stated in Section 3.2.1, *C*-1 free reconstruction buffers are sufficient to obtain most of the benefit of the

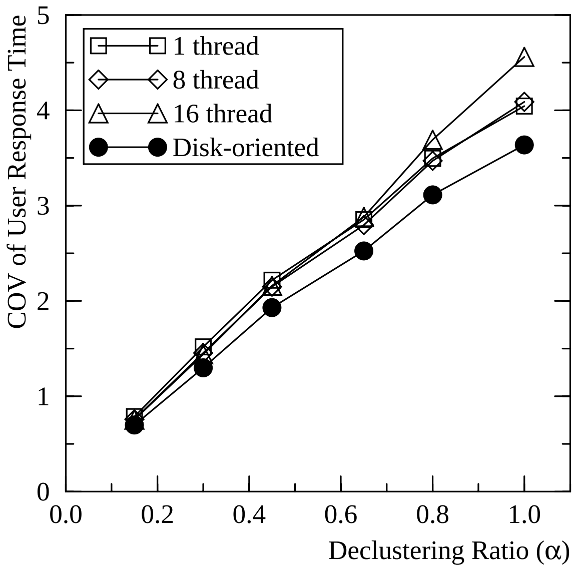
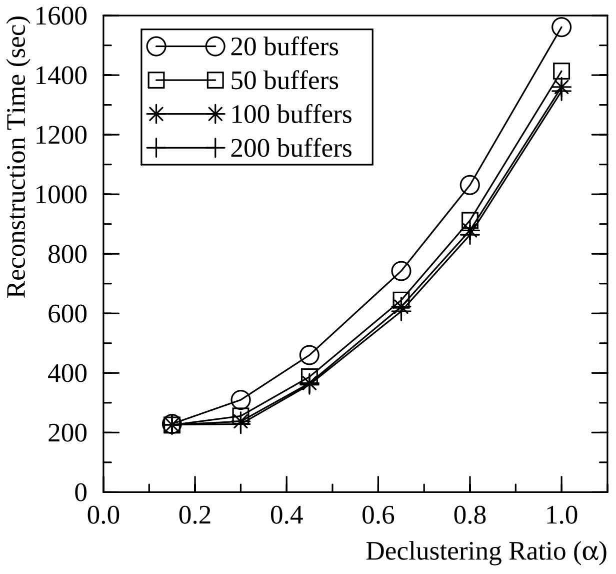


Figure 4: COV of average user response time

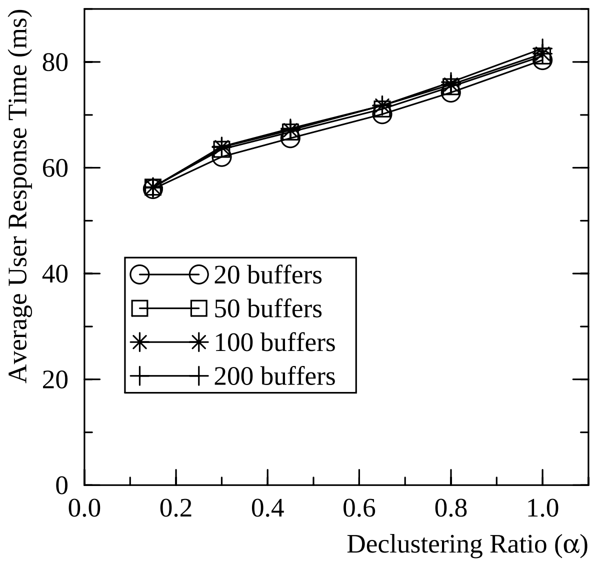
disk-oriented reconstruction algorithm. Figure 5 demonstrates this by showing the reconstruction time and average user response time during reconstruction for a varying number of free reconstruction buffers. Figure 5a shows that the reconstruction time can be slightly improved by using 50 rather than 20 free buffers for a 21-disk array, but further increases do not yield any significant benefit. Figure 5b shows that the number of reconstruction buffers has virtually no effect on user response time.

This effect is explained by noting that at declustering ratios close to 1.0, the reconstruction rate is limited by the rate at which data can be read from the surviving disks, while at low values of α, the replacement disk is the bottleneck. When the surviving disks are the bottleneck, buffer stall time is expected to be small because data cannot be read from the surviving disks as fast as it can be written to the replacement. The simulations bear this out, showing that when α is close to 1.0, each surviving-disk process spends only a few seconds (out of 300-1000) waiting for reconstruction buffers. Thus, little benefit is expected by increasing the number of buffers. At low values for α, data can be read from the surviving disks much faster than it can be written to the replacement, and so all free reconstruction buffers fill up very early in the reconstruction run, no matter how many of them there are. In this case, the rate at which buffers are freed is throttled by the rate at which full buffers can be written to the replacement disk. Since the total number of free buffers has no effect on this rate, there is little benefit in increasing the total number of these buffers.

From this we conclude that using slightly more than twice as many free reconstruction buffers as there are disks is in general sufficient to achieve the full benefits of



(a) Reconstruction time



(b) Average user response time during recovery

Figure 5: The effects of increasing the number of reconstruction buffers

a disk-oriented reconstruction algorithm. For an array of 21 disks using 24 kilobyte data units and 50 reconstruction buffers, the total buffer requirements in the host or controller is 20 exclusive buffers + 50 free buffers = 70 buffers or 1.7 megabytes. Figure 5 shows that this number can be reduced further with relatively small decreases in reconstruction performance.

4.3. Reconstruction unit size

The algorithms presented thus far access disks one unit at a time. Since the rate at which a disk drive is able to deliver data increases with the size of an access, it is worthwhile to investigate the benefits of using larger reconstruction accesses. Using larger units gives rise to an inefficiency in the declustered layout because not all the data on each disk is necessary for the reconstruction of a particular disk. Unnecessary data is interleaved with necessary data on the surviving disks, so reading large sequential blocks accesses some unnecessary data, wasting disk bandwidth and buffer space. This is not a problem in the RAID level 5 architecture, since all surviving units are necessary no matter which disk fails.

This inefficiency can be avoided by modifying slightly the data layout policy in the declustered parity organization. The modified layout replaces the data and parity units in Figure 2 by larger (e.g. track-sized) “reconstruction units”, and then stripes the actual data units across these units, as illustrated in Figure 6.

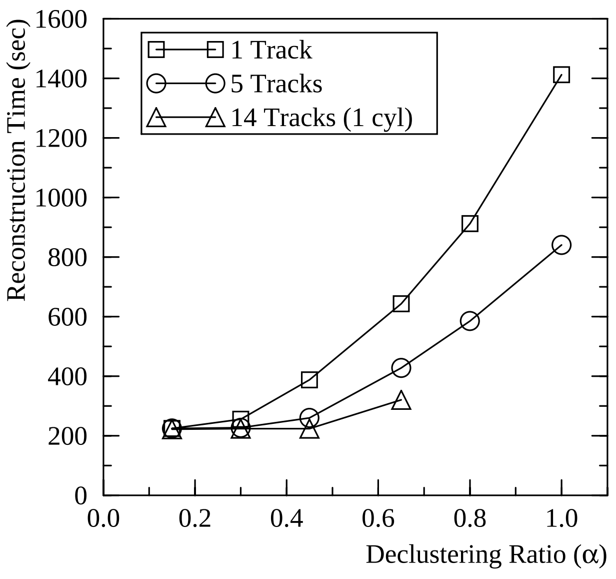
Offset	DISK0	DISK1	DISK2	DISK3	DISK4
0	D0.0	D0.1	D0.2	P0	P1
1	D5.0	D5.1	D5.2	P5	P6
2	D1.0	D1.1	D1.2	D2.2	P2
3	D6.0	D6.0	D6.2	D7.2	P7
4	D2.0	D2.1	D3.1	D3.2	P3
5	D7.0	D7.1	D8.1	D8.2	P8
6	D3.0	D4.0	D4.1	D4.2	P4
7	D8.0	D9.0	D9.1	D9.2	P9

Figure 6: Doubling the size of the reconstruction unit.

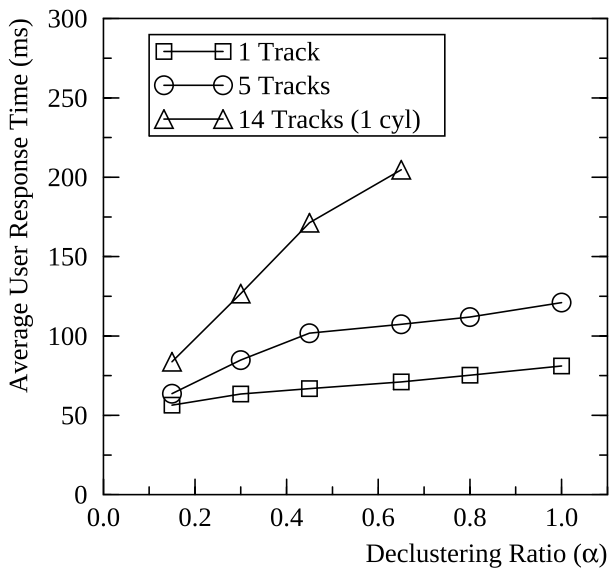
Figure 7a shows the reconstruction time for various sizes of the *reconstruction unit*, which is defined as the size of the accesses that reconstruction processes issue. It shows that except at low values of α, cylinder-sized reconstruction units provide the shortest reconstruction time; about twice as fast as reconstruction using track-sized units. The curves join at low α due to saturation on the replacement disk.

However, this benefit does not come without a cost. Reconstruction accesses take a long time to complete when reconstruction units are large. This causes user accesses to block in the disk queues, increasing the observed user response time. Figure 7b shows that reconstruction using cylinder-sized units more than triples the observed user response time over track-sized reconstruction units. At high declustering ratios ($\alpha > 0.65$), reconstruction using full-cylinder accesses consumes so much of the array’s bandwidth that it is unable to maintain the user workload of 294 accesses per second, as indicated by the missing data points in the figure.

Figure 7 indicates that for the workload we employ,



(a) Reconstruction time



(b) Average user response time

Figure 7: The effects of varying reconstruction unit size

the most appropriate reconstruction unit size is a single or a few tracks. At high declustering ratios, the benefit of larger reconstruction units is outweighed by the increase in response time, and at low declustering ratios, there is no benefit due to saturation of the replacement disk. If, however, the disks supported preemption and subsequent resumption of low-priority accesses by higher priority accesses, then much of the benefit of using the larger reconstruction units might be obtained without large response time penalties. We leave this issue for future work.

5. Conclusions

This paper demonstrates and evaluates an implementation of a fast on-line reconstruction algorithm for redundant disk arrays. The analysis shows that a disk-oriented algorithm results in a 400-800% improvement in failure recovery time when compared to a naive algorithm, with only a small (15%) degradation in user response time during failure recovery. The improvement comes from a much more efficient utilization of the array's excess disk bandwidth.

For our example 21-disk array, the algorithm can be implemented using a moderate amount of memory (1-2 MB), and track-sized reconstruction accesses represent a good trade-off between reconstruction time and user response time.

Acknowledgments

Ed Lee and the Berkeley RAID group supplied us with our original copy of the raidSim disk array simulator. We're also indebted to Hugo Patterson and Daniel Stodolsky for reviewing early drafts of the paper.

References

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

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

[Chen90b] P. Chen and D. Patterson, "Maximizing Performance in a Striped Disk Array", *Proceedings of International Symposium on Computer Architecture*, 1990.

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

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

[Gibson93] G. Gibson and D. Patterson, "Designing Disk Arrays for High Data Reliability," *Journal of Parallel and Distributed Computing*, January, 1993.

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

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

[Hou93] R. Hou, J. Menon, and Y. Patt, "Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array," *Proceedings of the Hawaii International Conference on Systems Sciences*, 1993, pp. 70-79.

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

[Kim86] M. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. 35 (11), 1986, pp. 978-988.

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

[Menon92] J. Menon and J. Kasson, "Methods for Improved Update Performance of Disk Arrays," *Proceedings of the Hawaii International Conference on System Sciences*, 1992, pp. 74-83.

[Merchant92] A. Merchant and P. Yu, "Design and Modeling of Clustered RAID," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1992, pp. 140-149.

[Muntz90] R. Muntz and J. Lui, "Performance Analysis of Disk Arrays Under Failure," *Proceedings of the 16th Conference on Very Large Data Bases*, 1990, pp. 162-173.

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

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

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

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

[Stodolsky93] D. Stodolsky, G. Gibson, and M. Holland, "Parity Logging: Overcoming the Small-Write Problem in Redundant Disk Arrays," *Proceedings of the International Symposium on Computer Architecture*, 1993.

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