

A Redundant Disk Array Architecture for Efficient Small Writes

Daniel Stodolsky, Mark Holland, William V. Courtright II,
and Garth A. Gibson

July 29, 1994
CMU-CS-94-170

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

An abbreviated version of this report will appear in *Transactions on Computer Systems*, September 1994.

Abstract

Parity encoded redundant disk arrays provide highly reliable, cost effective secondary storage. They provide high performance for reads and large writes. Their performance on small writes, however, is much worse than mirrored disks – the traditional, highly reliable, but expensive organizational storage. Unfortunately, small writes are a substantial portion of the I/O workload of many important, demanding applications such as on-line transaction processing. This paper proposes a parity logging a novel solution to the small write problem for redundant disk arrays. Parity logging uses journaling techniques to substantially reduce the cost of small writes. We provide detailed models of parity logging and competing schemes – mirroring, floating storage, and RAID level 5 – and compare these models by simulation. Parity logging provides performance competitive with mirroring, with capacity overhead close to the minimum offered by RAID level 5. Parity logging can exploit data caching more effectively than all three alternative approaches.

This research was supported by the ARPA, Information Science and Technology Office, under the title “Research on Parallel Computing”, ARPA Order No. 7330, the National Science Foundation under contract NSF ECD-8907068, and by IBM and AT&T graduate fellowships. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by ARPA/CMO to CMU. 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 ARPA, NSF, IBM, AT&T, or the U.S. government.

Keywords: Redundant disk arrays, RAID level 5, Parity logging.

1. INTRODUCTION

The market for disk arrays, collections of independent magnetic disks linked together to form a data store, is undergoing rapid growth and has been predicted to exceed 13 billion dollars [DiskTrend94]. This growth has been driven by three factors. First, the growth in processor speed has outstripped the growth in disk data rate. This imbalance transforms traditionally compute-bound applications to I/O-bound applications. To achieve application speedup, I/O system bandwidth must be increased by increasing the number of disks. Second, arrays of small diameter disks offer a substantial cost, power and performance advantages over larger drives. Third, such systems can be made highly reliable by storing a small amount of redundant information on the array. In contrast, the redundancy in large disk arrays has unacceptably low data reliability because of their large component disks. For these three reasons, redundant disk arrays, also known as Redundant Array of Inexpensive Disks (RAID), are strong candidates for nearly all on-line secondary storage [Patterson88, Gibson92].

Figure 1 presents an overview of the RAID systems considered. The most promising variant, RAID level 5, employs distributed parity with data striped on a unit that is one sector. RAID level 5 arrays exploit the low cost of parity encoding to provide high data reliability. Data is striped over all disks so that large files can be fetched with high bandwidth. By distributing parity many small random blocks can also be accessed in parallel without hot spots on any one disk.

While RAID level 5 disk arrays offer performance and reliability advantages for a wide range of applications, they do not tolerate faults. Level 1 is simple mirroring, in which two copies of each data unit are stored on separate disks. Level 4 uses block-interleaved parity, in which parity is stored on a single disk and data is striped across the other disks.

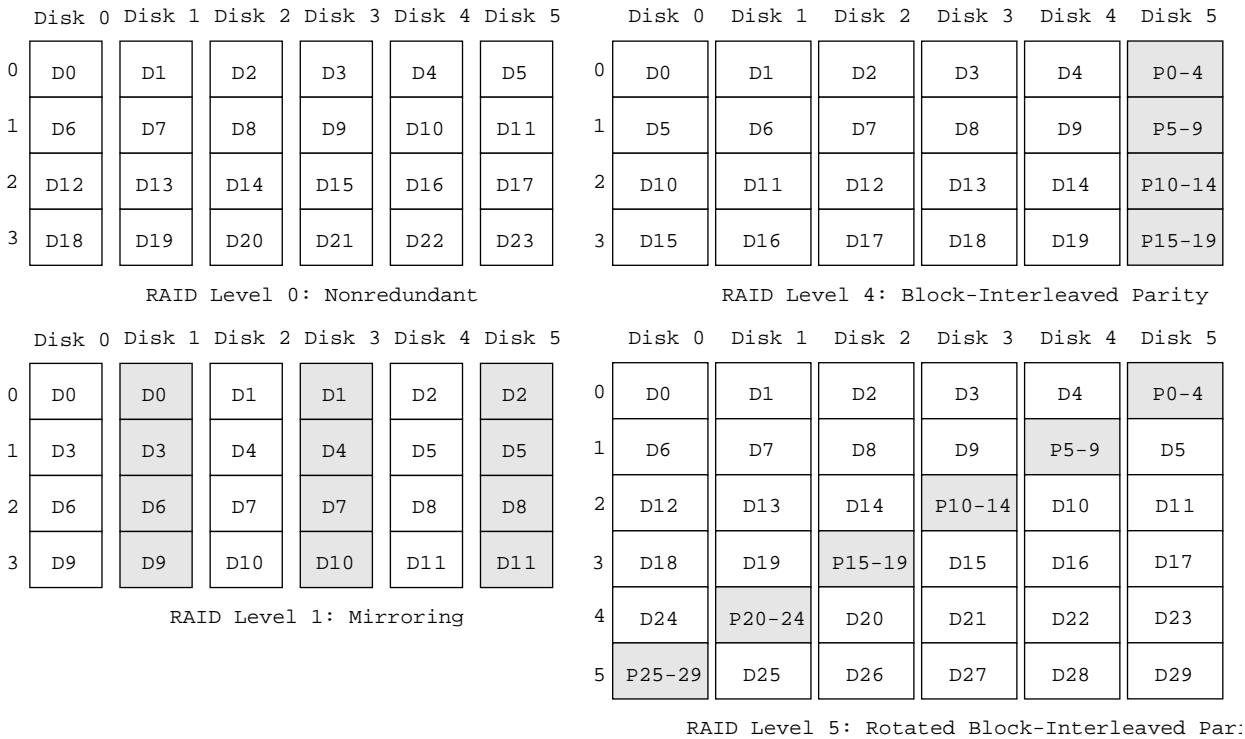


Fig. 1. Data Layout in RAID Levels 0, 1, 4 and 5. This figure shows the first few units on each disk in each RAID level. "D" represents a unit of user data (of unspecified size, but some multiple of one sector) computed over user data units x through y. The numbers on the left indicate the offset into the raw disk units. Shaded blocks represent redundant information, and non-shaded blocks represent user data. RAID level 0 does not tolerate faults. Level 1 is simple mirroring, in which two copies of each data unit are stored on separate disks. Level 4 uses block-interleaved parity, in which parity is stored on a single disk and data is striped across the other disks. Level 5 uses rotated block-interleaved parity. In level 5, the parity blocks rotate through the array rather than being concentrated on a single disk, thus avoiding an access bottleneck.

1. In current industry usage, the "I" in RAID denotes "independent".

TPC Benchmark	Scaling Requirements		
	Record Type	Minimum Quantity per TPS	Record Size (Bytes)
get request from terminal	Account	100K	100
begin transaction	Teller	10	100
update account record	Branch	1	100
write history log	History	30K	50
update teller record	Total		11.5 MB per TPS
update branch record			
commit transaction			
respond to terminal			

Fig. 2OLTP Workload Example. The transaction processing council (TPC) benchmark is an industry benchmark for DB systems stressing update-intensive database services [TPCA89]. It models the customer withdrawals and deposits at a bank. The primary metric for TPC benchmarks is transaction rate. Systems are required to complete 90% of their transactions in under 2 seconds and to meet the scaling requirements. Customer account records are selected at random from the local branch 85% of the time, and from other branches the rest of the time. Because history record writes are delayed and grouped into large sequential writes and are easily cached, the disk I/O from this benchmark is dominated by the random account record update.

applications, they possess at least one critical limitation: their throughput is penalized by a factor of four over nonredundant arrays for workloads of mostly small writes [Patterson88]. This penalty is incurred because a small write request may require the old value of the data to be read (we call this a read-modify-write), overwriting this second disk block with the updated data, then overwriting this second disk block with the updated data. On parity systems based on mirrored disks simply write the data on two separate disks and, therefore, are only penalized by a factor of two. This disparity in accesses per small write instead of two, has been termed the read-modify-write problem.

Unfortunately, small write performance is important. The performance of on-line transaction processing (OLTP) systems, a substantial segment of the secondary storage market, is largely determined by small write performance. The workload described by Figure 2 is typical of nearly the worst possible for RAID level 5, where a single read-modify-write of an account record require five disk accesses. The same operation would require three accesses on mirrored disks and two on a nonredundant array. Because of this limitation, many systems continue to employ the much more expensive option of mirrored disks.

This paper describes and evaluates a powerful technique for eliminating this small write penalty. Parity logging exploits well understood techniques for logging or journaling to transform small random accesses into large sequential accesses. Section 2 of this paper describes the parity logging mechanism. Section 3 introduces a simple model of its performance and cost. Section 4 describes alternative disk system organizations, develops comparable performance models, and contrasts them to parity logging. Section 5 provides an analysis of small-write overheads in parity logging with respect to configuration and workload parameters, and analyzes potential load imbalances in a parity logging system. Section 6 introduces our simulation system, describes implementation details of parity logging and alternative organizations, and contrasts their performance on workload of random writes and an OLTP workload. Section 7 analyzes extensions to multiple-failure tolerant disk arrays. Section 8 discusses how the large write optimization can be accommodated in a parity logging disk array. Section 9 reviews related work. Section 10 closes with a few comments on future work on redundant disk arrays for small write intensive workloads.

2. PARITY LOGGING

This section develops parity logging as a modification to RAID level 5. Our approach is motivated by the fact that disks deliver much higher bandwidth on large accesses than they do on small writes. Parity logging disk array batches small changes to parity into large accesses that are much more efficient. The model is introduced in terms of a simple, but impractical RAID level 4 scheme, then refined to a realistic implementation used in our simulations.

The duration of a disk access can be broken down into three components: seek time, rotational positioning time, and data transfer time. Small disk writes make inefficient use of disk bandwidth.

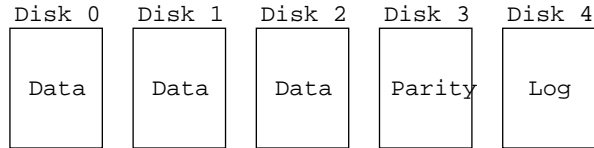


Fig. 4. Basic Parity Logging Model

because the data transfer component is much smaller than the seek and rotational positioning components. Thus a disk servicing a small-access-dominated workload spends the majority of its time positioning instead of transferring data. This figure shows the relative bandwidths of random block, track, and cylinder accesses for a modern small-diameter disk [IBM0661]. This figure largely bears out the folklore of disk bandwidth: random cylinder accesses move data twice as fast as random track accesses, which, in turn, move data ten times faster than random block accesses.

D	Data units per track	12	V	Tracks per cylinder	14
V	Cylinders per disk	949	N	Number of disks in array	22
S	Average seek time	12.5 ms	M	Single track seek time	2.0 ms
R	Average rotational delay (1/2 disk rotation time)	~9.5 ms	H	Head switch time	1.16 ms

B	Number of regions per disk	100	C _D	Cylinders of data per region	200
C _L	Cylinders of log per region	10	C _P	Cylinders of parity per region	20
K	Tracks buffered per region	100	L	Log striping factor	1

Fig. 5. Model Parameters. The bandwidth utilization models of Section 2, 3, and 4 are presented with the parameters listed above. The first table presents common disk parameters and the second, parameters used in the first and fourth columns in each table show the symbol used in the text; the second and fifth symbols denote; and the third and last column show the default value of the parameter as used in the models. A tilde (~) indicates an approximate value.

Logically, we develop our scheme beginning with Figure 4 in which a RAID level 4 disk array is augmented with one additional log disk. Initially, this disk is considered empty. RAID level 4, a small write prereads the old user data, then overwrites it. However, before updating parity with a preread and overwrite, the parity update image (the result of XOR of old parity and new user data) is held in a dedicated block of memory called a parity buffer. When enough parity update images are buffered to allow for an efficient disk transfer (one or more tracks), they are transferred to the end of the log on the log disk.

When the log disk fills up, the out-of-date parity and the log of parity update records are read from memory using sequential cylinder accesses. The logged parity update images are applied to the memory image of the stale parity and the resulting updated parity is written with large sequential writes. When this completes, the log disk is marked empty and the logging cycle begins again.

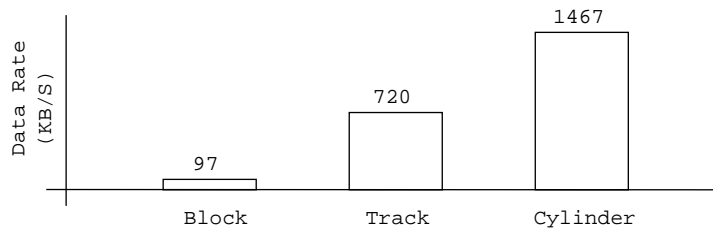


Fig. 3. Peak I/O Bandwidth. This figure shows the sustained data rate in kilobytes per second that can be written to an IBM 0661 drive using random one block (2KB), one track (24 KB), and one cylinder (336 KB) for disk parameters).

Because only parity updates (not data changes) are deferred, this scheme preserves single-disk tolerance. If a data disk fails, the log disk (and any buffered parity updates) are first read from the parity disk, which is then used to reconstruct the lost data in the same manner as is done in RAID level 5. If the log or parity disk fails, the system can simply recover by reconstructing parity from the surviving parity or log disk. The failed drive is then replaced with a new empty disk. If a controller fails, its buffered parity updates are lost, but, after the controller has been replaced, parity can be updated in the same way as if a log disk had been lost.

The addition of a log disk allows substantially less disk time to be devoted to parity updates than in a comparable RAID level 4 or 5. This can be shown by computing the average disk busy time devoted to updating parity. Assume there are D data units per track, T tracks per cylinder, and V cylinders per disk (refer to the glossary in Figure 5). Each user write requires a corresponding data unit, which introduces an overhead of one block (data unit) access per track. In addition, each user write to a data unit consumes buffer memory equal to the size of the user write. A track's worth of small (unit-sized) writes issued to the array causes one track write to occur. Next, a disk's worth of small writes causes the log disk to fill up, which must then be emptied by updating the parity. This update involves reading the entire contents of the parity array (at $2V$ cylinders), and then writing the entire parity array (at cylinder transfer rates). On average, then, for every small user writes the TVD block accesses, sequential track accesses, and cylinder accesses for maintenance of the parity information. Recall track access is D times larger than random small writes but about 10 times more efficient and cylinder access is twice as efficient than track accesses. Thus, parity maintenance for a disk's worth of small user writes consumes about as much disk time as

$$TVD + TV(D/10) + 3V(T/2 \times D/10) = 5TVD/4$$

random small accesses. In a standard RAID level 4 or 5 disk array, maintenance for small writes would consume about as much disk time as random block accesses. The ratio of parity maintenance work performed by parity logging to RAID level 4 or 5 is therefore

$$\frac{5TVD/4}{3TVD} = \frac{5}{12}$$

Thus, by logging parity updates, we have reduced the disk time consumed by parity maintenance by about a factor of two.

In many cases, it may be possible to avoid the pre-read of the user data. For example, in the benchmark (Figure 2), the update of a customer account record is a read-modify-write operation. The account record is read, modified in memory, and written back to disk. In these cases, the old data is usually known (cached) at the time of the write and the pre-read of the data may be avoided [Menon93]. Under these conditions, the overhead for RAID levels 4 or 5 is just two random accesses per small write or random block accesses for small user writes, and the overhead for parity logging is

$$TV(D/10) + 3V(T/2 \times D/10) = TVD/4$$

random small accesses. Therefore, in these cases, parity logging reduces disk time consumed by maintenance by about a factor of eight.

2.1. Partitioning the Log Into Regions

As stated, however, this scheme is completely impractical: an entire disk's worth of random access memory is required to hold the parity during the application of the parity updates. Figure 2 shows that this limitation can be overcome by dividing the array into manageably-sized regions. Figure 3 is a miniature replica of the array proposed above. Small user writes for a particular

2. Our failure model treats disk and controller failures as independent. If concurrent controller and disk failures survived, controller state must be partitioned and replicated [Schulze89, Gibson93, Cao93].

3. Notice that we make no attempt to reduce the cost of the overwrite of the target data block. Additional savings in data writes are deferred and optimally scheduled [Solworth90, Orji93].

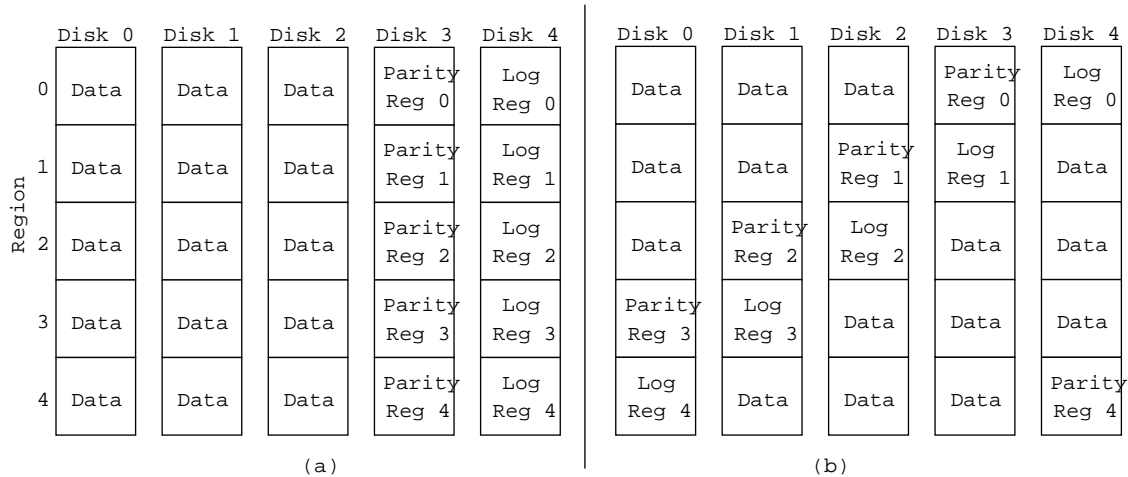


Fig. 6 Parity Logging Regions

journalled into that region. When a region fills up, only that region's parity is updated. This reduces the size of the controller memory buffer needed during reintegration from the size of a disk to a manageable fraction of a disk. Section 2.4 shows that the number of regions is dependent on disk capacity, about 100 in our example 22-disk array.

Each region requires its own log buffer. Each buffer holds a few (typically less than three) parity update images. When one of these buffers fills up, the corresponding region is updated with an efficient track (or multi-track) write. Thus, the sequential track writes of the single-region layout are replaced by random track (or multi-track) writes in the multiple-region layout. While random writes are less efficient than sequential track writes, Section 3 will show that this multiple-region implementation still has dramatically lower parity maintenance overhead than RAID level 4.

2.2. Striping Log and Parity for Parallelism

As in the RAID level 4 case, the log and parity disks may become performance bottlenecks. In a RAID level 4 array, the maximum aggregate bandwidth for log accesses is just the bandwidth of a single disk. This limitation can be overcome by distributing parity and log information across all the disks in the array, as indicated in Figure 6(b). This distribution boosts the aggregate bandwidth to the bandwidth of the array. However, the log and parity bandwidth for a particular region remains that of a single disk.

Following the example of RAID level 5, Figure 7 shows a layout in which the parity for each region is distributed across the array to increase bandwidth. This distribution decreases the number of disks needed to re-integrate parity updates for a particular region. However, the log and parity bandwidth for a particular region remains that of a single disk.

The log bottleneck may also be eliminated by distributing the log for each region over multiple disks. Figure 8 shows a parity logging array with the log for each region striped across two disks. Update records in the log are logically part of the parity and are placed on the same disks as the data they protect. If they were, the failure of that disk would cause both data and parity to be lost, which is an unrecoverable failure in a disk array using a parity-based code. Instead, data and log for each region are restricted to disjoint sets of disks. Thus, log striping reduces the number of disks on which data for a particular region may be placed. If, for example, the log is striped across two disks, data for that region may be placed only on the other two disks.

This reduction in data striping in a region increases the disk space overhead. The overhead follows the number of disks over which each log is striped and the number of cylinders of parity per region. The number of data cylinders per region is related to the size of the parity, according to the standard RAID level 4 and 5 rule for data striping.

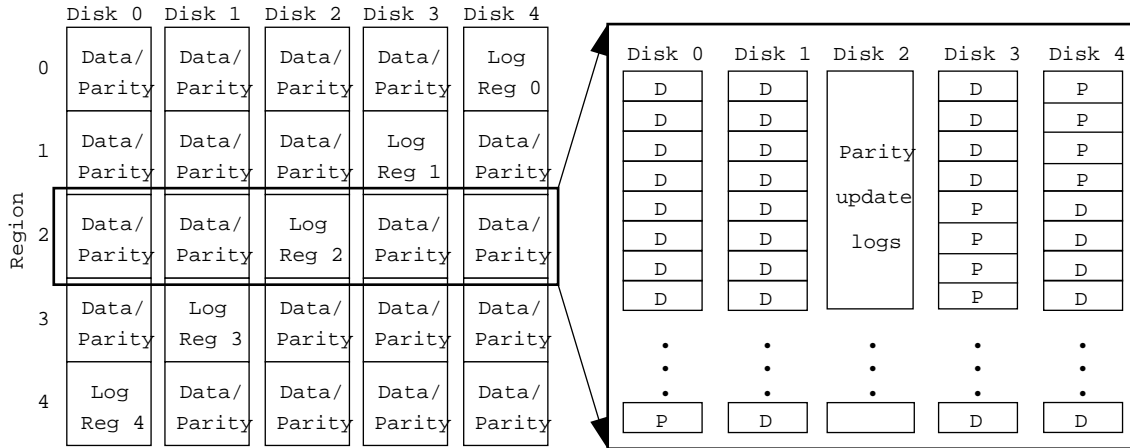


Fig. Block Parity Striping. The inset shows a detailed layout of a sample region.

$$C_D = (N - L - 1)C_P$$

where N is the number of disks in the array, L is the log length, C_L is the number of cylinders of log per region, C_P is the number of cylinders of parity per region, and C_D is the number of cylinders of data per region. The disk space overhead (the fraction of the array containing log and parity) equals

$$(C_P + C_L) / (C_L + C_P + C_D) = 2 / (N - L + 1)$$

and rises as the degree of log striping increases. Figure 9 shows the disk space overhead for different degrees of log striping for an array of 22 disks. However, the performance advantages of log striping are substantial.

2.3. The Impact of Varying Log Length

The previous subsection assumes that the same amount of disk space is used for log and

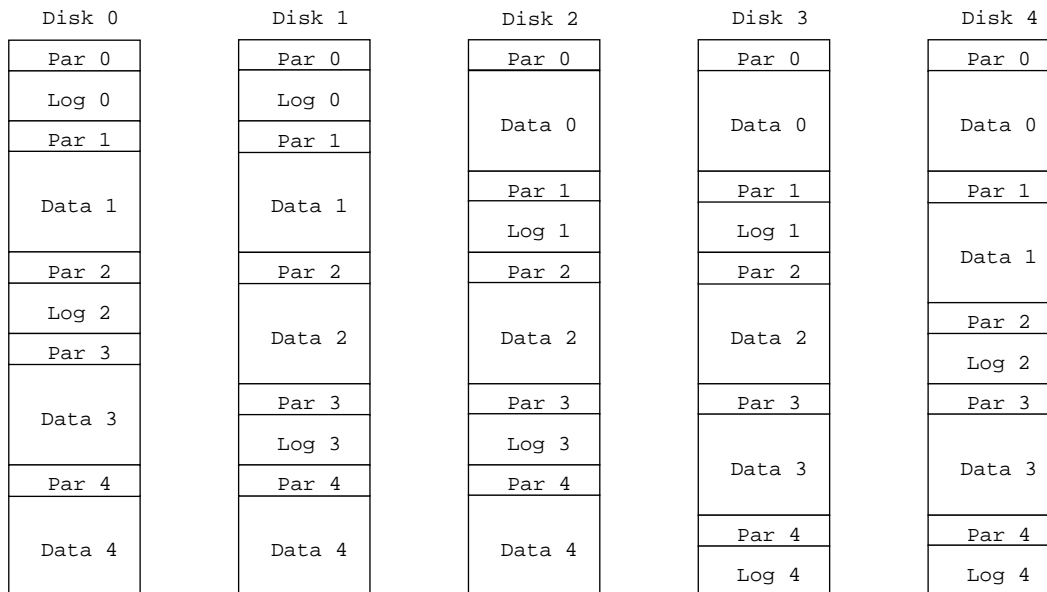


Fig. Distributed Parity Logs

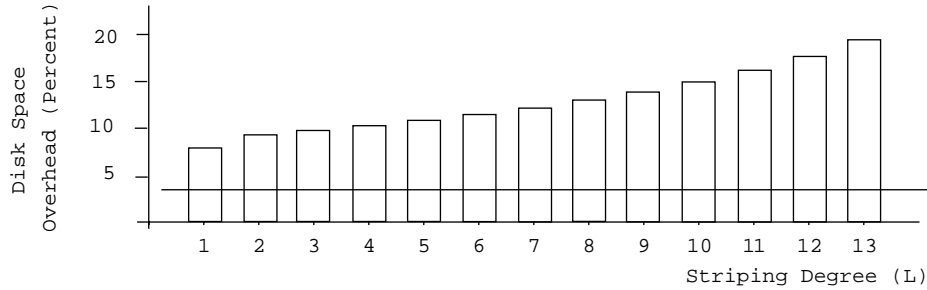


Fig. 9. Disk Storage Overheads The horizontal line shows the capacity overhead of : configuration of the same array

parity (C_p cylinders) is allocated in each region because our introduction adds exactly one an array Given the more flexible striped log and parity model of Figure 8, the efficiency overheads of parity logging can be altered by increasing or decreasing the amount of log region.

Let A be the ratio of total log space to total parity space each region. The disk space overhead then becomes

$$\frac{C_L + C_P}{C_L + C_P + C_D} = \frac{AC_P + C_P}{AC_P + C_P + (N - L - 1)C_P} = \frac{1 + A}{N - L + A}$$

Now the log for each region fills after a small user writes into that region. Updating the parity still requires prereading old data on each small user write (assuming the old data is not cached), writing the log buffers, plus, every time the log fills, reading parity (C_p cylinders), reading the log cylinders, and writing the updated parity cylinders). Thus the parity maintenance work for uncached small user writes is

$$ATC_p D + ATC_p \left(\frac{D}{10}\right) + (2 + A)C_p \left(\frac{T}{2} \times \frac{D}{10}\right) = \left(\frac{23}{20} + \frac{1}{10A}\right)ATC_p D$$

random small accesses, or an overhead of $(23/20 + 1/10A)$ random small accesses per uncached small user write. Performance can therefore be traded for space, as shown in Figure 10. Applying example 22 disk array with logs striped over two disks allocating twice as much log as parity ($A = 2$) increases the space overhead from 9.5% to 13.6% of the total capacity, the parity maintenance overhead from 41.7% to 40% of that of RAID level 5, where three related parity occur for each small user write. Halving the amount of log increases the disk space overhead to 7.3% while only increasing the parity maintenance work to 45% of RAID level 5.

If the old data is cached, RAID level 5 does two parity-related accesses for each small parity logging disk ($20 + 1/10A$). Applying this cached workload to our 22 disk array with striped over two disks does not change the space overheads. However, doubling log size reduces parity maintenance work from 12.5% to 10% of RAID level 5 while halving increases the work to 17.5% of RAID level 5.

2.4. Accounting for and Managing Buffers

The primary benefit of parity logging, that parity maintenance operations access disks in efficient transfers, requires expensive controller memory buffers. This buffer memory is ways. First, each region delays the most recent parity update images until efficient log-appends can be performed. K tracks are transferred in a log-append operation and then, conservatively, K tracks of buffer memory are required to delay log appends. Second, whenever log for a region fills, the parity for that region is read and applied

to it, and the updated parity is written back. This parity reintegration operation requires buffer memory where C_p is the number of cylinders of parity per region and n is the number of tracks per cylinder. Since the number of cylinders of parity per region is the same as the total cylinders N , divided by the number of regions, total buffer memory space B is measured in tracks.

By selecting $B = \sqrt{TV/K}$, the memory buffer space is minimized to $\sqrt{2TV/K}$. If the ratio of the cost of a byte of memory and a byte of disk is the buffer memory space cost, relative to the cost of an array of disks $2X\sqrt{TVK}/(NTV) = 2X/N\sqrt{TV/K}$. If memory costs 30 times as much as disk [Feigel94], then an array of 22 IBM 0661 (Figure 12) disks buffering a single log track ($K = 1$) requires about 5.6 MB of buffering the equivalent of about 2% of the array.

In practice, parameters such as the number of regions must be discrete. If we further require size per region of the log appends, sublogs (the postlog of a region), as well as parity and data, per region, be an integral number of tracks, then a significant fraction of the space may be wasted. We have found that if the number of regions allowed to vary from the optimum by $\pm 10\%$, then a set of integral parameters can be found such that the wasted disk space is less than 1% of the array space.

If, however, the size per region of the sublogs, parity and data, per region, are only required to be an integral number of disk sectors (rather than tracks), substantially less disk space is wasted. The number of regions, R , is selected as an integer near $\sqrt{TV/K}$. Relaxing this discrete-tracks condition will cause additional head switches and single cylinder seeks to occur during log and parity appends, but because these positioning overheads are small relative to track access times, parity performance is only slightly affected (3% for our experiments).

A more significant performance degradation results if small user writes are blocked during reintegration of a region into its parity. Blocking should be minimized by managing the per region buffers as a single global buffer pool. Using this approach, user writes are only blocked if the entire buffer pool is full of parity updates images that have not yet been appended to the logs.

2.5. Summary

In summary, parity logging buffers parity updates until they can be written to the log efficiently. It further delays their reintegration into a redundant parity array. There are enough parity updates in the log to make a complete revision of the parity in a limited memory. For reintegration of parity records, the disk array is partitioned into regions. Then, to avoid bandwidth bottlenecks, parity and log information is striped over multiple regions. This parity logging scheme reduces the extra work done by RAID level 5 arrays for small random writes little more than is done in the much more expensive, traditional mirrored approach even

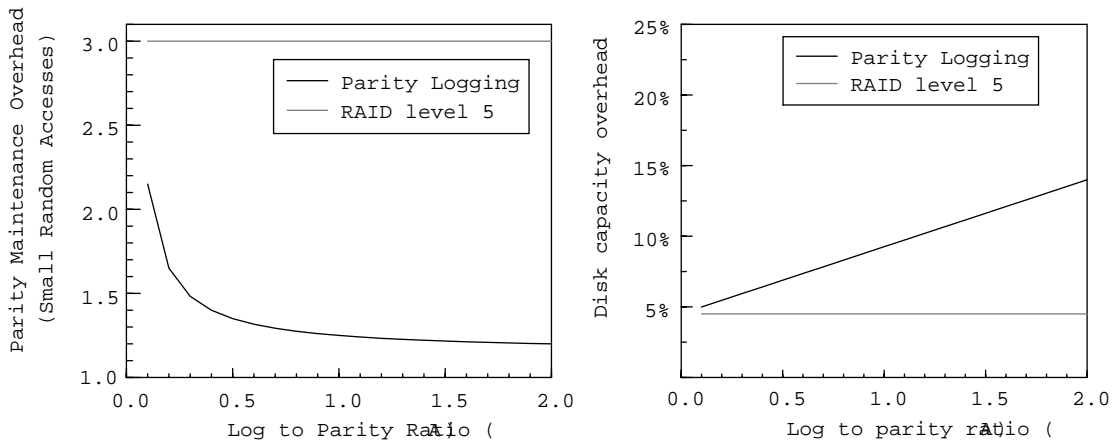


Fig. 10. Log Length and Efficiency

caching is ineffective.

3. MODELING PARITY LOGGING

In this section we present a utilization-based analytical model of a parity logging re-
array. This model predicts saturated array performance in terms of achieved disk utilization,
geometry and access size. The variables used in this model are defined in Figure 5.

Consider a single small user write in a parity logging array. The data must be preread, then
overwritten. This is done in an access which seeks to the cylinder containing the data, waits for the
data to rotate under the head, reads the data, waits for the disk to rotate around once, then
data.⁴ Defining S as the average seek time, the time for one-half of a disk rotation, and recalling
that D is the number of data units per track, the time to perform this operation,

$$t_{m w} = \underbrace{(S+R)}_{\text{Seek and rotational delay}} + \underbrace{2R/D}_{\text{Data preread}} + \underbrace{(2R - 2R/D)}_{\text{Rotational delay}} + \underbrace{2R/D}_{\text{Data write}} = S + (3 + 2/D)R \quad (1)$$

disk seconds, on average.

As mentioned earlier in many cases it may be possible to predictably avoid the preread of
data. Without prereading, the disk busy time needed for a small write access,

$$t_w = S + (1 + 2/D)R \quad (2)$$

disk seconds.

Each region has tracks worth of log buffers. On average, K disks are user writes, one
region's buffers will fill and be written to the region's track write. Defining the
disk's head-switch time, the number of disk seconds required to do this,

$$t_{k \text{ track}} = \underbrace{(S+R)}_{\text{Seek and rotational delay}} + \underbrace{2RK}_{\text{Data transfer time}} + \underbrace{(K-1)H}_{\text{Head switch time}} = S + (2K+1)R + (K-1)H \quad (3)$$

assuming K tracks are on the same cylinder

Finally recall that each region consists of C_L cylinders, each of which has D data
units. Therefore, on average, for every small user writes, one region of logged parity must
reintegrated. Consider the case of an array that does not stripe its log (Figure 7). This
consists of three steps: a sequential read of the log, a striped read of the parity
 $N-1$ disks, and a striped write of the parity back to the log. Defining M as the time taken to
seek one cylinder and the time for the sequential log read (

$$t_{C_L} = \underbrace{(S+R)}_{\text{Seek and rotational delay}} + \underbrace{C_L(2RT + (T-1)H)}_{\text{Read time for 1 Cylinder}} + \underbrace{M(C_L-1)}_{\text{C}_L-1 \text{ single cylinder seeks}} \quad (4)$$

4. This single access could be separated into two accesses, each taking $0.5(1+2/D)R$ seconds. For most modern disks this is about twice as fast as the single access is more efficient.

5. Disks that support zero-latency writes [Salem86] can eliminate the initial rotational delay from the I/O time by up to 26% in drives such as the IBM 0661 (which does not support this feature), if only a single track ($K=1$). However, the impact of zero-latency write support on parity logging is small (under 3%), because the track-switching time is only a small contributor to parity logging (Figure 1).

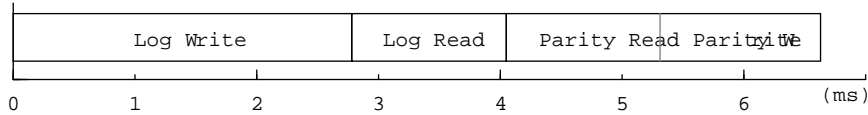


Fig. 11 Parity Logging Overheads The amortized overhead cost of parity and log accesses do parity logging array is shown above. The log writes contribute approximately 40% (1 millisecond), while the cylinder rate log reads, parity reads and parity writes each (1 millisecond). For comparison, the parity accesses done by RAID level 5 cost nearly 35 mi

disk seconds, and may be rewritten as

$$t_{C_L} = S + (2TC_L + 1)R + (T - 1)HC_L + (C_L - 1)M \quad (5)$$

The striped parity accesses each Nonis sequential transfers/(N - 1) cylinders. Each of these sequential transfers takes

$$\frac{(S + R) + (C_p / (N - 1)) (2RT + (T - 1)H) + (C_p / (N - 1) - 1)M}{\text{Cylinders per subaccess}} \quad \text{Single track seeks per subaccess}$$

First seek and rotational delay
Read time for 1 cylinder

disk seconds. The total striped access

$$t_{C_p} = (N - 1) (S + R) + C_p (2RT + (T - 1)H) + M (C_p - N + 1) \quad (6)$$

disk seconds.

Thus, on average, the disk utilization induced by a small write,

$$t_{am w} = t_{rm w} + \frac{1}{KD} [t_{K track}] + \frac{t_{C_L}}{DTC_L} + \frac{2t_{C_p}}{DTC_L} \quad (7)$$

Log write
Log read
Parity read and write

Figure 11 shows the contributions to disk busy time of the three terms in equation 7 for the example disk array given in Figure 12.

The analysis for a parity logging disk array with a striped log. (When a region's log buffer fills, it will be written to one of the regions - sublog write. The rest of this operation is the same as in the unstriped case. Log reintegration to L is made once every writes, but now consists of three striped I/Os: a striped read of the log, and a striped (over N disks) read and write of the parity. The accesses in the striped log read costs

$$\frac{(S + R) + (C_L / L) (2RT + (T - 1)H) + (C_L / L - 1)M}{\text{Cylinders per subaccess}} \quad \text{Single track seeks per subaccess}$$

First seek and rotational delay
Read Time for 1 Cylinder

for a total of

$$t_{C_L(L)} = L(S + R) + C_L (2RT + (T - 1)H) + (C_L - L)M \quad (8)$$

disk seconds. Similarly, striped parity reads and writes will consume

Workload Parameters	
Access size:	Fixed at 2 KB
Alignment:	Fixed at 2 KB
Write Ratio:	100%
Spatial Distribution:	Uniform over all data
Temporal Distribution:	66 closed loop processes Gaussian think time distribution
Array Parameters	
Stripe Unit:	Fixed at 2KB
Number of Disks:	22 spindle synchronized disks.
Head Scheduling:	FIFO
Power/Cabling:	Disks independently powered/cabled
Disk Parameters	
Geometry:	949 cylinders, 14 heads, 48 sectors/track
Sector Size:	512 bytes
Revolution time:	13.9 ms
Seek Time Model:	$2.0 + 0.01 \cdot \text{dist} + 0.46 \cdot \sqrt{\text{dist}}$ (ms) 2 ms min, 12.5 ms avg, 25 ms max
Track Skew:	4 sectors
Head Switch time:	1.16 ms

Fig. 12. Simulation Parameters. The access size, alignment, and spatial distribution are representative workloads, while a 100% write ratio emphasizes the performance differences of the various array disks have independent support hardware, disk failures will be independent, allowing a parity group [Gibson93]. Disk parameters are modeled on the IBM Lightning drive [IBM066]. Note that the seek model is the number of cylinders traversed, excluding the destination. As is commonly done in SC chosen to equal the head switch time, optimizing data layout for sequential multitrack access.

$$t_{C_p} = N(S + R) + C_p(2RT + (T - 1)H) + (C_p - N)M \quad (9)$$

disk seconds. Thus, striping introduces an additional $L(S + R - M)/D$ disk seconds to the log reintegration. This increases the parity maintenance overhead per small write to $L(S + R - M)/D + C_p$ disk seconds. As Section 6 will show, increase in parity maintenance work is worthwhile because it reduces long reintegration periods during which disk systems grow becomes underutilized, and maximum performance falls far short of expectations.

4. MODELING ALTERNATIVE SCHEMES

Only a few array designs have addressed the problem of high performance, parity-based, design for small write workloads. The most notable of these is floating data and parity [Menon92]. reviews and estimates the performance of four designs: nonredundant disk arrays (RAID mirrored disks (RAID Level 1), distributed N+1 parity (RAID Level 5), floating data and parity notation and analysis methodology are the same as used in Section 3.

In nonredundant disk arrays (RAID Level 0), a small write requires a single disk access which consumes

$$\frac{(S + R) + 2R/D}{\text{Seek and rotational delay} \quad \text{Data write}}$$

disk-arm seconds. No long-term storage is required in the controller

In mirrored systems, every data unit is stored on two disks, and all write requests are duplicated. Each access takes as much time as a small write in a nonredundant array. Hence, each small user write utilizes disks $2S + (2 + 4/D)R$ of seconds. While mirrored disks' write operations are more efficient than RAID Level 1, their capacity is devoted to

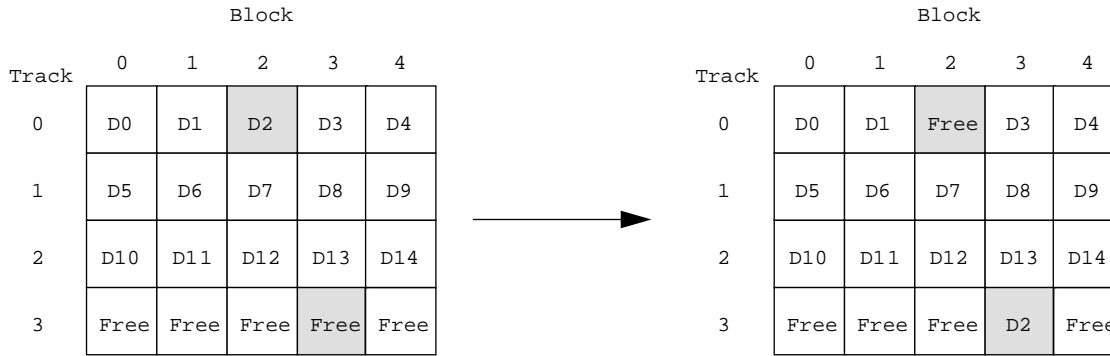


Fig. 13 Floating Data/Parity. This figure shows the movement of data within a cylinder caused by a data and parity array. Each grid represents one cylinder of four tracks, with five blocks per track. the controller searches for a free block within the cylinder that is rotationally close to block at offset 3 into track 3. Immediately following the preread of block D2, the controller writes it and updates mapping tables. The preread of old information and the write of new information ar slightly more than the time of one access.

redundant data. As in the RAID level, controllers for mirrored disk arrays do not require term buffer memory

Small writes in RAID level 5 disk arrays require four accesses: data preread, data write and parity write. These can be combined into two read-rotate-write accesses, each of which

$$\frac{(S + R) + 2R/D + (2R - 2R/D) + 2R/D}{\text{Seek and rotational delay} \quad \text{Data preread} \quad \text{Rotational delay} \quad \text{Data write}}$$

disk seconds for a total disk busy time of $(S + (6 + 4/D)R)$. Again, no long-term controller storage is required.

The floating data and parity modification to RAID levels proposed by Menon and Kasson [Menon92]. In its most aggressive variant, this technique organizes data and parity into contain either only data or parity. As illustrated in Figure 13, by maintaining a single track of space per cylinder for data and parity effectively eliminates the extra rotational delay level 5 read-rotate-write accesses. Instead of updating the data or parity in place, a floating parity array will write modified information into the rotationally nearest free data track. W and parity the rotational delay $2R/D$ in the RAID level 5 disk arm busy time expression above replaced with a head switch and a short rotational delay similar to those in our sample array Menon and Kasson report an average delay of 0.76 data units. So, the expected disk busy each access in a floating data and parity array is

$$\frac{(S + R) + 2R/D + H + 0.76(2R/D) + 2R/D}{\text{Seek and rotational delay} \quad \text{Data preread} \quad \text{Head switch} \quad \text{Rotational delay} \quad \text{Data write}}$$

which may be rewritten as $(S + (1 + 5.52/D)R + H)$. Hence, the total disk busy time for a small random user write in a floating data and parity array is $(S + (2 + 11.04/D)R + 2H)$. Note that if the number of data units per tracks, large and the head-switch time, small, this is close to the performance of mirroring.

Even with a spare track in every cylinder, floating data and parity arrays still have excellent storage overheads. For a disk array with tracks per cylinder floating data and parity has a storage overhead of $(T + N - 1)/(TN)$.⁶ Floating data and parity arrays, however, substantial fault-tolerant storage in the array controller to keep track of the current location of data a

6. Each disk gives T plits capacity for free space and the array gives remaining space for parity the array storage efficiency is $(T + N - 1)/(TN)$ and the array storage overhead is $(T + N - 1)/TN$.

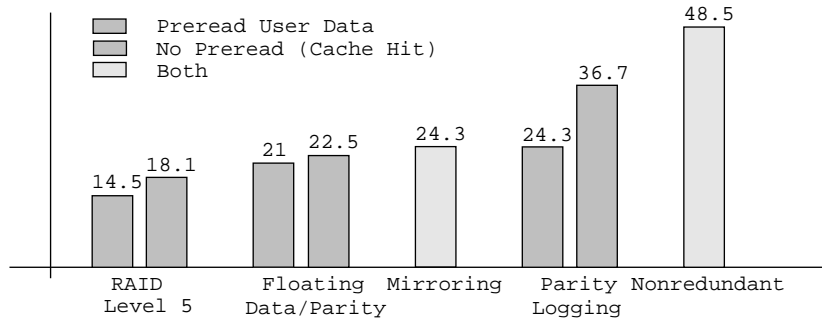


Fig. 1 Model Estimates User writes per second per disk as predicted by the bandwidth models of S predictions assume 100% disk utilization, FIFO disk arm scheduling, and an unbounded number of r and parity logging disk arrays both benefit substantially from not having to preread user data. substantially reduces the overhead of the user preread and therefore achieves less benefit from i nonredundant disk arrays do not need to preread user data. The parity logging estimates assume t

each cylinder an allocation bitmask is maintained. This requires, in addition, a table of current block locations for each cylinder is required. This log(DT) bits per cylinder. Thus, with cylinders per disk, a total of $(T-1) \lceil \log(DT) \rceil$ bits of fault-tolerant controller storage are required. For the disks in Figure 12, this is 1,343,784 bits per disk. The total controller storage in a 22 disk array is about 3,608KB, roughly comparable to parity logging. Note, however, that controller memory in parity logging need not be fault-tolerant.

While floating data and parity substantially improves the performance of small writes on RAID level 5, its performance for other types of accesses is degraded. Logically contiguous user data units are not likely to be physically contiguous. In the worst case, two data units may end up at the same rotational position on two different tracks, requiring two disk rotations to read both. In addition, the average track traversal rate is reduced, on average by $(T-1)/T$.

5. ANALYSIS

Figure 14 compares these models' estimates for maximum throughput of the example array with Figure 12. Throughput at lower utilizations may be calculated by scaling the maximum throughput numbers by the disk utilization. Figure 14 predicts that parity logging and floating data arrays both substantially improve on RAID level 5, approaching the performance of mirroring. Varying the models' parameters from our example 22 disk array does not substantially change the relative performance of parity logging and its alternatives except for the effects of the number of data units per track and the ratio of average seek time to rotational latency. Figure 15 describes the effects of these parameters and the effects of log striping degree on array load balance.

Of the model parameters, the number of data units per track has the greatest impact on performance. Parity logging transfers each data unit two more times than RAID level 5 and 13 times more than mirroring. If the transfer time of a unit is small, parity logging will be effective. Figure 15 shows the relative performance when data caching is ineffective (i.e., a preread is required). Parity logging, mirroring, and RAID level 5 for different values of numbers of data unit per track. The performance of mirroring exceeds that of parity logging with 13 or fewer data units per track ($D \leq 13$), and RAID level 5 performance exceeds that of parity logging with the unlimited number of data units per track. Industry estimates, however, that track capacity within a given form factor is increasing at over 20% a year. It is reasonable to assume that the number of data units per track may not decrease even as database account record sizes grow.

The ratio of average seek time to rotational latency has a substantial impact on the performance of parity logging disk arrays. Figure 16 plots the performance of parity logging on RAID level 5 and mirroring relative to RAID level 0 as this ratio changes. The performance of

7. The nature of fault tolerance in a storage controller depends on failure model. If only power failure is of concern, then nonvolatile storage will suffice, while other failure models require redundant controllers.

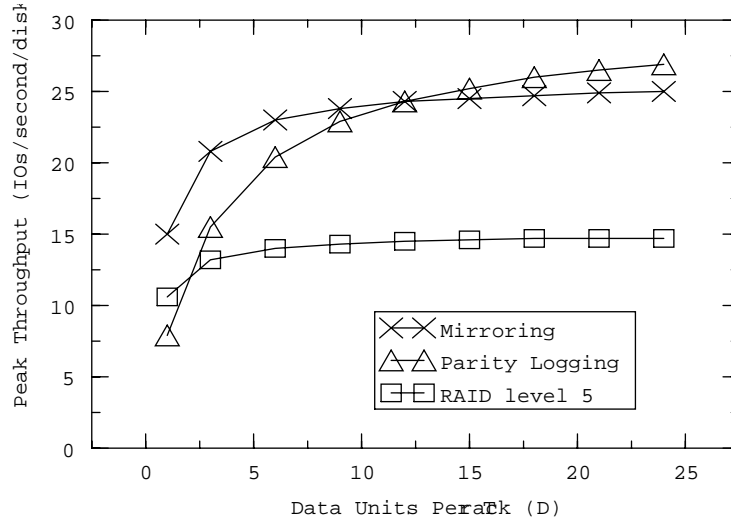


Fig. 13 Effects of track size on throughput. Performance of parity logging is highly sensitive to data units per track. The figure above shows the performance in the example 22 disk array of mirroring level 5 on a workload of 100% blind small writes for varying number of data units per track.

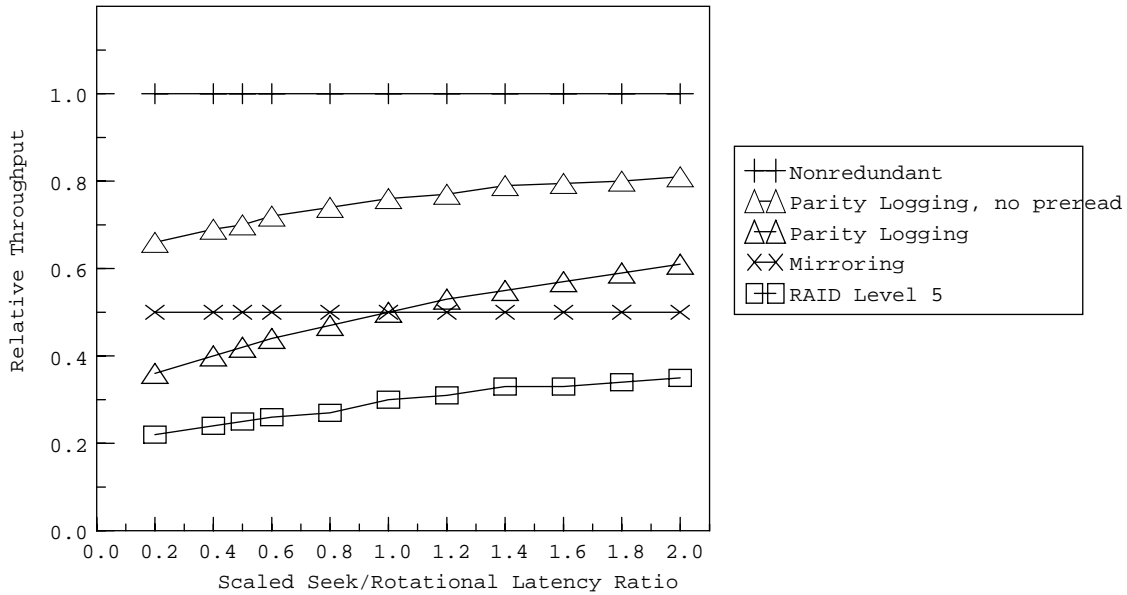


Fig. 14 Peak throughput, normalized to nonredundant array performance, as a function of the ratio of average seek time to rotational latency. Varying the ratio of average seek time (S) to the rotational latency, the relative performance of mirroring, floating data redundancy and parity logging disk arrays. Shows the relative performance of these approaches on the example 22 disk array (Figure 12) as the average seek time is varied from 20% of the Lightning average seek to twice that of the parameter range models a large spectrum of drives, from those with very fast positioning to Light 7200 RPM. The X-axis has been linearly scaled so that 1.0 corresponds to the ratio of average seek to the Lightning drive.

achieves as much benefit from decreased seek time as nonredundant arrays because its two accesses are each equivalent to the single nonredundant access. RAID level 5 and parity logging, however, do more rotational work for each seek so decreasing seek time relative to rotational latency decreases their performance relative to nonredundant arrays. Moreover, parity logging does more rotational work to avoid the parity write seek of RAID level 5. Consequently, the advantage of parity logging over RAID level 5 decreases as the seek time to rotational latency ratio decreases. This ratio is nearly unity for all modern drives, and shows no particular trend in any direction.

Figure 14 assumes the user requests access data uniformly. This assumption is reasonable for huge OLTP databases, other workloads may exhibit substantial non-uniformity. In the average case, all user I/O

is concentrated within one region. Choosing an appropriate data stripe unit [Chen90] will user I/O across the actuators that contain data for this "hot log region" and however traffic are partitioned over non-overlapping disks. If this traffic is not balanced, parity logging per fall short of Figure 14.

The log, parity and data traffic can be balanced by determining the appropriate degree of L. Recall that every small user writes (where T_L and C_L are the number of data units per track, tracks per cylinder and cylinders of log per region, respectively) data to disks of a particular region will cause writes of tracks to the striped log, and then a full log read a full read and full write of the parity for that region to effect parity reintegration. writes are spread out over all disks, so a uniform load is maintained if the work per data the work per sublog disk. That is,

$$\frac{DTC_L t_z}{N - L} = \frac{(TC_L/K) t_{K \text{ track}} + t_{C_L(L)}}{L} \quad (10)$$

where $t_{K \text{ track}}$ (Equation 3), and $t_{C_L(L)}$ (Equation 8) are the service times for a write and a full log read striped over disks, respectively, and N is the number of disks in the array and t_z is the disk service time for a small user write. When data caching is effective (Equation 2). When caching is ineffective equals $t_{m w}$ (Equation 11). Expanding $t_{C_L(L)}$ in Equation 10 yields a quadratic equation in whose solution is omitted here because it is unnecessarily complex because of the terms denoted by transfer time T_L , we approximate this balance equation as a linear equation in whose solution is

$$L = \frac{N}{1 + (KD t_z) / (t_{K \text{ track}} + 2KR)} \quad (11)$$

Using this approximation and the disk array parameters from Figure 12, 20.16N for blind writes (where $t_z = t_{m w}$) and $L \approx 0.11N$ when caching is effective (where $t_z = t_w$). Therefore, to balance the load over all disks in a single region, the example 22 disk array must have sublogs per region.

6. SIMULATION

To validate the analytic models presented in Sections 3 and 4 and to explore response times for disk arrays, we simulated the example array described in Figure 12 under five different configurations: nonredundant, mirroring, RAID level 5, floating data and parity logging. Parity logging was simulated with a single track of log buffer (K per region) over several different degrees of log striping L . The simulations were performed using the RAIDSIM package, a disk array simulator derived from the Sprite operating system disk array driver [Ousterhout88], which was extended to include implementations of parity logging and floating data and parity logging.

In each simulation, a request stream was generated by 66 user processes, an average of one per disk. Each process requests a 2KB write from a disk selected at random, waits for acknowledgment from the disk array, then "thinks" for some time before issuing another request. Process think times are modeled by an exponential distribution, but the mean is dynamically adjusted until the desired system response time is achieved. If the disk array is unable to sustain the offered load, think time is increased. Simulations were run until the 95% confidence interval of the response time became less than 10% of the mean. Because this makes all confidence intervals directly computable, the subsequent performance plots do not show them.

6.1. The Need for Log Striping

Figure 17 shows peak throughput, response time and response time variance as the degree of log striping L varies.

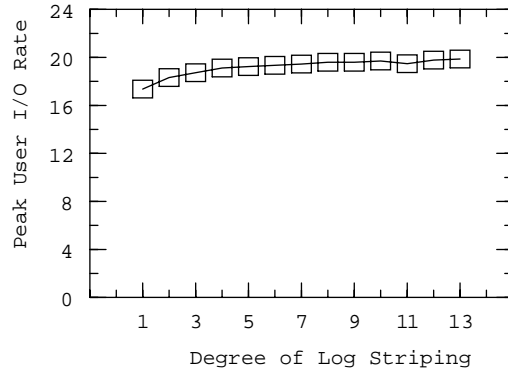


Figure 17(a): Peak user I/Os

Fig. 17. Parity Log Striping Figures 17(a) and (b) show the achieved user I/Os per disk per second, response time, and the standard deviation of the response time under peak load for various degrees of log striping. Performance metrics improve substantially as the striping degree is increased from one (unstriped) to four (striped over 4 disks). The difference between striping over 4 to 13 disks is slight, indicating the robustness of the technique.

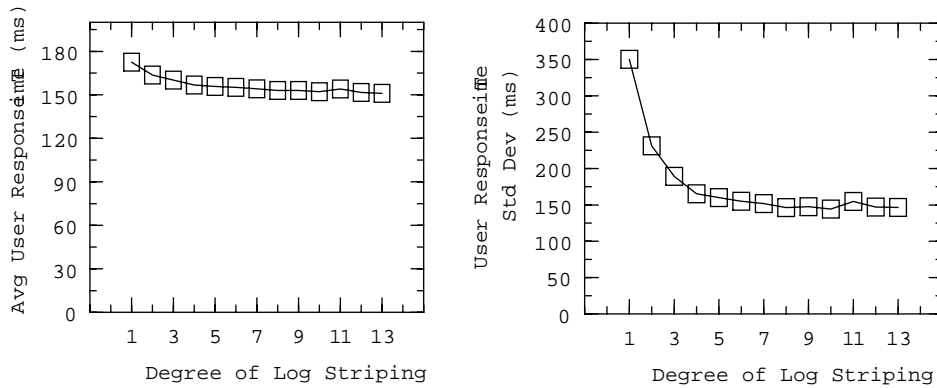


Fig. 17(b): Response time and response time standard deviation at peak load

striping (L) is varied from one (unstriped) to thirteen. As predicted in Section 5, when striped over a small number of disks, performance is substantially lower than in configurations with more widely striped logs. This behavior results from a "convoy effect" in which processing requests is blocked by writes queue behind very long sublog read accesses. Figure 18 shows sublog read times versus degrees of log striping. While these long accesses are efficient, they completely tie up a disk at a time. During this period, any access to the disks involved in the striped log results in a delay, reducing the effective concurrency in the system. This concurrency reduction causes other requests to become idle until the log read completes, reducing peak throughput and utilization. The convoy effect also has a substantial impact on response time; requests that block behind the long reads will have very long response times, leading to an increase in both average response time and response time variance. Fortunately, the least degree of striping eliminates the convoy effect. Figure 17 shows that striping the log over six disks achieves most of the available throughput with minimal increasing disk space overhead.

With convoys avoided by a log striped over six disks, Figure 19 compares the performance of a parity logging array with one track buffered per region against several alternative organizations: nonredundant, mirroring, RAID level 5, and floating data. The primary purpose of this figure is to present performance in terms of response time as a function of throughput. Figures 19(a)-(b) assume that user data must be preread (data cache miss), and Figure 19(c) presents the corresponding performance for the no preread (data cache hit) case.

These simulation response time results may be summarized as follows. Nonredundant disk configurations perform a single disk access per user write, so they have the lowest and most slowly growing response times.

8. The simulations reported herein consider a user write in a parity logging array complete when the user data and parity update record has been buffered. The alternatives (nonredundant, mirroring, floating data, and RAID level 5) consider a user write complete when data and parity are on disk.

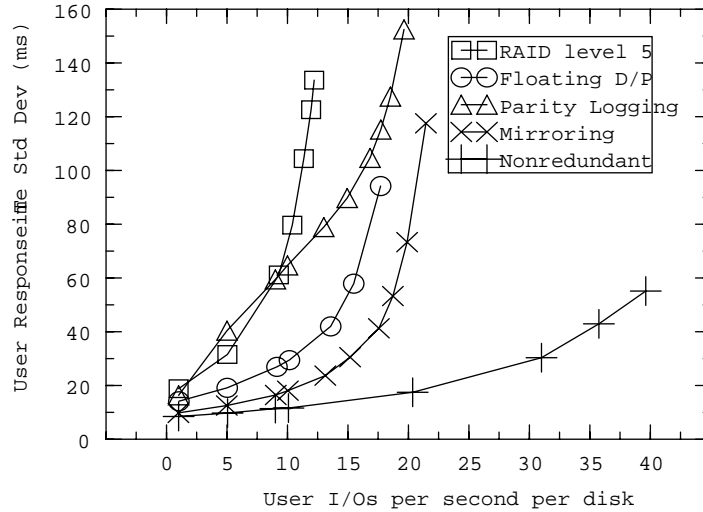


Figure 19(b): Response time standard deviation

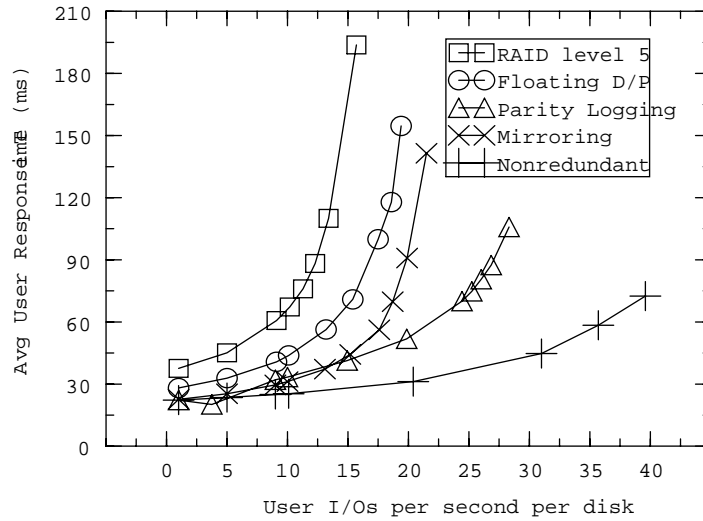


Fig. 19(c): Response times without prereads

nonredundant array. This also reduces the actuator time per access by nearly one third, throughput and response time to improve proportionately.

The variance in user response time, however, is larger with parity logging than with mirroring, floating data and parity. Although it is not as large as with RAID level 5. This results from the structure of parity logging: most accesses are fast because inefficient work is delayed. Fast accesses see long response times as delayed work is (efficiently) completed. In our mind, we conclude that the response time estimates in Figure 19 show that parity logging is a much lower cost, alternative to mirroring for small-write intensive workloads.

	RAID level 5	Floating D/P	Mirroring	Parity Logging	Nonredundant
Preread Required	83.7	82.8	89.7	83.5	81.1
No Preread	86.7	87.0	89.7	81.2	81.1

Fig. 20 Disk Utilization at Peak Load

6.2. Analytic Model Agreement with Simulation

The analytical model estimates in Figure 14 predict the vertical asymptotes (saturation) of Figure 19(a) and (c). A direct comparison will, however, display significant discrepancies because of the relatively small number of simulated processes. With a fixed number of requesting processes, the deep queue of one overloaded disk can periodically go idle. Figure 20 shows the disk utilization load for the configurations simulated. These peak-load disk utilizations differ according to the number of concurrent disk accesses issued by a user write in each configuration. RAID level 5 and parity when user data is not cached, and parity logging and nonredundant disk arrays, require only one disk access request at a time per process. Mirroring and floating data and parity keep the array busier because each user write requires two concurrent disk accesses. Figure 21 shows that, when these differences are accounted for in the model predictions of Figure 14 by the disk utilizations of Figure 20, simulation throughput agrees with analytic predictions to within 5%.

6.3. Performance in More General Workloads

Up to this point, all of the analysis has been specialized for workloads whose accesses are small (2KB) random writes. This section examines a mixed workload, defined in Figure 22, modeled from statistics taken from an airline reservation system [Ramakrishnan92]. In a more general workload, the results of the earlier sections are modified by two important effects: reads and large writes. The issues encountered in extending floating data and parity to handle variable access are beyond the scope of this paper and this technique is omitted from this section. For other array configurations, parity logging, mirroring and RAID level 5 also have a difference in read performance. This will have the effect of compressing the overall performance differences between configurations. Writes that are not small, however, hurt the performance of parity logging as discussed in Section 5.

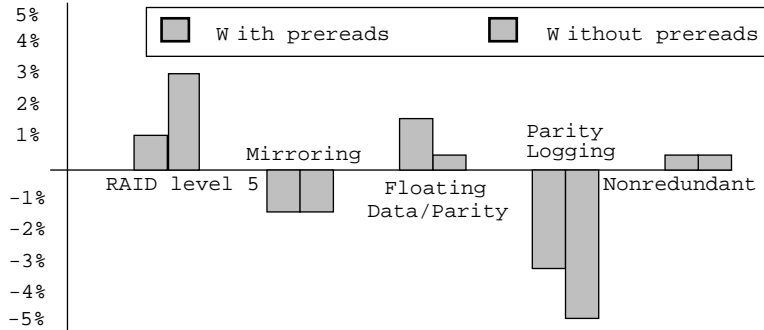


Fig. 21 Model errors. This figure shows the percent error between the models of sections 3 and 4 and Section 6. The model predictions have been scaled by the achieved disk utilizations of Figure 20. The disagreement between the simulation and the models is less than 5 percent. Note that the 95% simulation response time is also the mean.

Type	% of workload	Size (KB)	Type	% of workload	Size (KB)
Read	20	1	Read	20	2
Read	33	4	Read	9	24
Write	9	1	Write	7	8
Write	2	24			

Fig. 22 Airline reservation workload. The I/O distribution shown above was selected to agree with general statistics from an airline reservation system [Ramakrishnan92]. This workload is reported as approximately 82% reads and 18% writes, with a mean read size of 4.61 KB, and a median read size of 3 KB. The mean write size was 1.46 KB, and the median write size was 1 KB. Locality of reference and overwrite percentages were not reported. All accesses are assumed to be within disk boundaries.

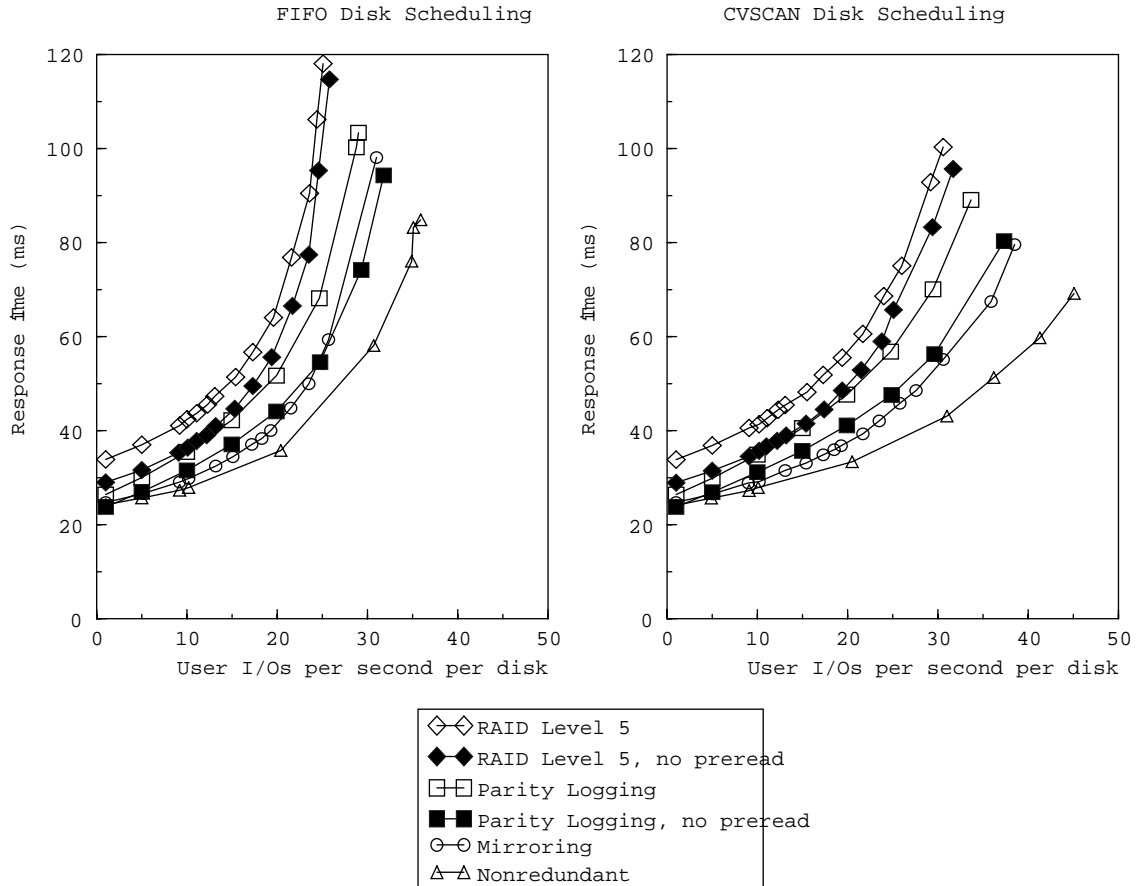


Fig. 23. Airline reservation simulation. Shown above are the results of simulation using the access s Figure 22. The access distribution is uniform throughout the 22 disk array (Figure 12). For all size was 24KB, so no access spans more than a single drive. For RAID level 5 and parity logging the case where all writes are blind, and when the old data for all writes is cached (no pre-read scheduling improves throughput and response of all workloads, mirrored and nonredundant disk array since seek time is a larger proportion of their underlying I/Os.

Figure 23 presents the results of simulations of four of the array configurations – non-mirroring, RAID level 5, and parity logging – on this mixed workload with FIFO disk scheduling, used throughout the rest of this paper, parity logging is always superior to RAID level 5 and is equivalent to mirroring when data caching of writes with CVSCAN is effective. [10].

7. MULTIPLE FAILURE TOLERANT ARRAYS

A significant advantage of parity logging is its efficient extension to multiple failure tolerant arrays. Multiple failure tolerance provides much longer mean time to data loss and greater protection for bad blocks discovered during reconstruction [Gibson92]. Using codes more powerful than RAID level 5 and its variants can all be extended to tolerate multiple failures. Figure 24 gives an example of one of the more easily-understood double failure tolerant disk array organizations, two-dimensional parity and the more familiar one dimensional parity used in the rest of this paper, called ordinary codes, because a particular bit of the parity depends on exactly one bit from a subset of the data disks. If, instead, generalized parity (check information) is computed

10. Our simulations do not explicitly model a disk controller cache. Cache write hits are special-cased because the disk access is modified by the previous data values.

bit symbol, dependent on a multiple-bit symbol from each of a subset of the data disks, through a non-binary code [MacWilliams77, Gibson92]. Non-binary codes can achieve much lower check information space overhead in a multiple failure tolerating parity variant of a Reed-Solomon code called "P+Q Parity" has been used in disk array products to provide double failure tolerance with only two check information disks [A

Disk 0	Disk 1	Disk 2	Parity Row 0
Disk 3	Disk 4	Disk 5	Parity Row 1
Disk 5	Disk 6	Disk 7	Parity Row 2
Parity Column 0	Parity Column 1	Parity Column 2	

Fig. 24 Two dimensional parityOne disk array organization that achieves double failure tolerance parity disks hold the parity for the corresponding row or column. In the example above, disk 2 holds the parity of disks 0, 3 and 5. Similarly, disk 4 holds the parity of disks 0, 1 and 3. In a data disk is written, the corresponding units in both row and column parity disks are also updated. In the example above, would require updating the parity on the shaded parity disks, parity row 0 and parity column 1.

This paper is not concerned with the choice of codes that might be used, except to note that the best of these codes all have one property important to small random write [Gibson89]: each small write updates exactly disks f disks containing check information (generalized parity) and the disk containing the data. This check maintenance work, which scales up with the number of failures tolerated, is exactly the work that parity logging handles more efficiently.

Multiple failure tolerating parity logging disk arrays arise as a natural extension of multiple failure tolerating variants of RAID 5. As with single failure tolerating parity logging, the array is augmented with a log. However, to achieve double failure tolerance, the log itself must be double failure tolerant. One way to achieve double failure tolerance is to replicate the log. Figure 25 shows one region of a double-fault tolerant parity logging disk array based on a nonbinary "P+Q Parity".

The log management cycle is quite similar to that of a single fault tolerant parity logging array. When a region's log buffers fill up, the corresponding parity update records are written once to the log. When these logs fill up, one copy of the log is read into the log's write buffer, the check information for the region. The updated check information is then rewritten, all other logs truncated, and the logging cycle starts again.

Mirroring and floating data and parity also extend to multiple failure tolerance in straightforward manner. Mirroring becomes copy shadowing [Bitton88]. Floating data and parity becomes floating data and check, requiring "rotated" read-rotate-write accesses per blind write.

The overhead associated with maintaining check information can be divided into two components: preread bandwidth overhead and nonpreread bandwidth overhead. The bandwidth needed to preread the old copy of the data is independent of the number of failures to be tolerated. Nonpreread bandwidth, the disk work done to update the check information given a data change, grows with the number of failures to be tolerated. Parity logging has the smallest overhead for this growing component of check maintenance overhead because all check information accesses (for generalized parity) are done efficiently.

Figure 26 shows the maximum rate that small random writes can be completed in zero, one, or two failures.

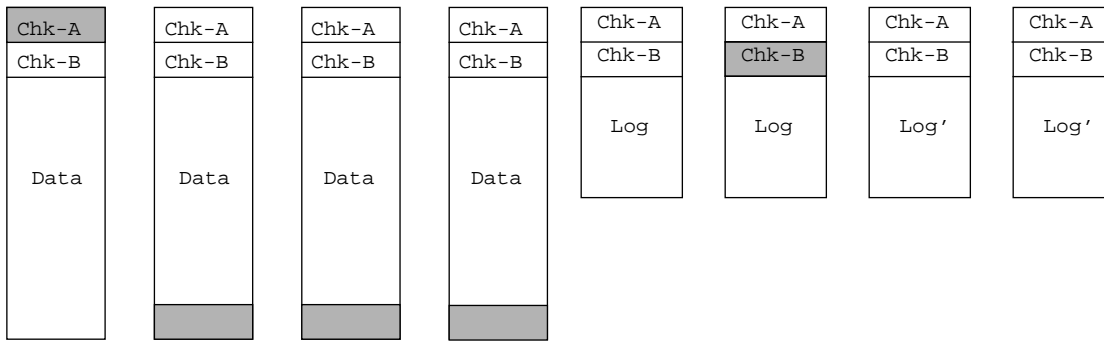


Fig. 25A parity-logging array that uses a nonbinary code to achieve double-fault tolerance. By using nonbinary codes, disk arrays can achieve double failure tolerance with only two disks of check data. Show double fault tolerant parity logging disk array with nonbinary check information. The parity of a replaced with two sets of check information. The shaded area shows an example pair of check information blocks that they protect.

To achieve double fault tolerance in such a parity logging array, a log for each region is duplicated. In the picture above, each log is striped over two disks. Note that the contents of this duplicated log are associated with a particular copy of the check information.

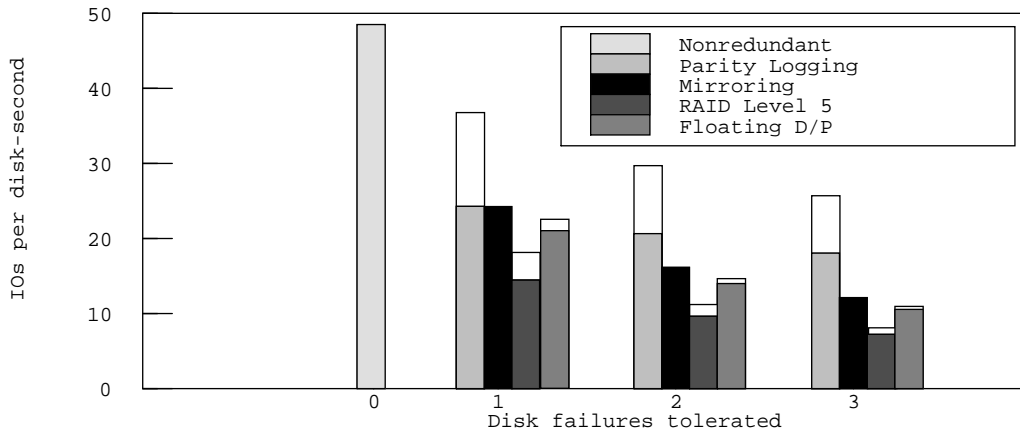


Fig. 26 Performance of multiple failure tolerating arrays. While the performance of all array configurations with the number of failures tolerated, parity logging declines the least, decreasing in performance as failures are tolerated. The highest performing alternative, mirroring, has a huge disk space overhead. The performance of RAID level 5 and floating data and parity both decline, each declining less than 10 user writes per second in the triple failure case. The shaded portion of each bar shows the performance when the data to be rewritten is not cached, while the unshaded portion shows performance when data is cached.

double, and triple failure tolerating arrays using mirroring, RAID level 5, floating data and parity logging. This data is derived from the models of sections 3 and 4 and applied to the array of figure 12.

The maximum I/O rate of the parity logging array declines more slowly than the other configurations because parity logging has a substantially lower nonpre-read overhead. For example, while triple failure tolerating parity logging arrays should sustain about 35% of the I/O rate of non-redundant arrays for random small writes, quadruplicated storage (triple failure tolerating mirroring) will sustain only 25%.

8. ACCOMMODATING THE RAID LEVEL 5 LARGE WRITE OPTIMIZATION

In parity-based disk arrays, a large write operation, which is defined as a write that updates many data units associated with a particular parity unit, can easily be serviced more efficiently than a small write operation. Since all data units in the stripe are updated, the new parity can be calculated in memory from the new data and written directly to the parity unit. This "large write optimization" avoids the pre-read of data and parity associated with small writes, improving write performance.

about a factor of four [Patterson88].

This optimization can not be applied directly to parity logging disk arrays as we have done so far because there may exist outstanding (not yet reintegrated) logged updates for a parity unit at the time when a large write overwrites that parity unit. If these logged updates and a parity overwrite were done, the parity could be erroneously updated with the stale updates when reintegration occurs. This problem can be corrected by placing the new parity instead of writing it directly to disk. A parity displaced in the log by a large write operation is marked as a special "overwrite" record, and the reintegration process, which normally XORs each log record with the corresponding parity unit, now distinguishes between a normal "update" log record and an overwrite record. Update records are XORed into the accumulating parity unit, while overwrite records are simply copied in.

This approach has the disadvantage of forcing the log to be processed sequentially rather than concurrently. If the log were guaranteed to contain only update records, the log records could be processed to the parity image in any order, increasing parallelism. The existence of overwrite records forces the reintegration process to determine the sequence in which the log updates occurred and to process records accordingly.

This new sequentiality constraint potentially lengthens the reintegration time, which, in turn, will show can substantially degrade performance at high loads. In the simple case, a read must be in read in the order they were written and merged to produce a update/overwrite image. If any of the parity is processed. Given sufficient buffer memory for a parity log, full parallelism could be achieved during the log and parity reads, but the application of the log would still have to be deferred until these reads complete. At this point, a sequential reintegration could be performed. However, as log buffers are written to sublogs in a round robin fashion, it is reasonable to assume that parallel sublog reads will return parity records in sequential order. Based on this observation and because overwrite records eliminate all information, the following highly parallel algorithm can be used. Each block in the reintegration is initially zeroed and marked "non-overwrite". Parity and log for the target region are read in parallel. A parity block is applied if the corresponding buffer is marked "non-overwrite," and if the buffer is marked "overwrite." If a logged record is an update and the block is "non-overwrite," the record is XORed in, but is buffered until all earlier log records have been processed. If it is an overwrite, the target block is overwritten and marked as "overwritten by record X." All updates that have already been applied should occur after this overwrite are reapplied. Only update records preceding X are not applied to a block marked "overwritten by X." As long as reads on different sublogs proceeded at nearly the same rate, this algorithm will not require extra buffer space. If buffer exhaustion occurs, the algorithm can simply serialize.

9. RELATED WORK

Bhide and Dias [Bhide92] have independently developed a scheme similar to parity logging. LRAID-X4 organization maintains separate parity and parity-update log disks, and periodically copies the logged updates to the parity disk. In order to allow writes from the user to occur in parallel with reintegration, they duplicate both the parity and the parity log for a total of four disks. LRAID-X4 does not distribute parity or log information. Instead of breaking down the log into regions to reduce the required storage in the LRAID-X4 controller, it buffers parity updates in memory according to the parity block to which they apply. This allows LRAID-X4 to write a "run" of updates for ascending parity blocks to a log disk. When this log disk is full, further updates are written into runs and written to the second log disk while the first log disk reintegrates its parity by reading from one parity disk and writing to the other. The reintegration of a full log disk uses an external sorting algorithm to collect subsequences applying to one area of parity from the log disk. If this area is large, all log reads and parity reads and writes will be efficient.

LRAID-X4 reaches its performance maximum of 34.5 writes per disk per second with 20 disks (10 data, 2 parity, 2 log) for a 100% write workload with 5% write cache memory [Stodolsky93]. Additional disks do not increase performance. In comparison, the parity logging disk array described in Section 6, whose controller requires about 2% write cache memory is predicted to achieve 36.7 I/Os per disk per second in Section 3 on the same workload, and its performance continues to increase with more disks.

11. An alternative way to correct the problem is to write the new parity directly to disk and place a "cancel" record in the log. The reintegration process would then discard all previous log entries for the identified parity unit when it detects the cancel record. This solution has the potential to reduce the log traffic by making cancel records only a few bytes in size.

with increasing numbers of disks.

Less closely related research efforts can be characterized by their use of three techniques frequently exploited to improve throughput in disk arrays: write buffering, write-twice, and floating location.

Write buffering delays users' write requests in a large disk or file cache to achieve deep writes. The cache can then be scheduled to substantially reduce seek and rotational positioning overheads [Solworth90, Rosenblum91, Polyzois93]. Data loss on a single failure is possible in these schemes unless fault-tolerant caches are used.

The write-twice approach attempts to reduce the latency of writes without relying on fault-tolerant caches. Similar to floating data and parity, every disk cylinder are reserved, and an allocation bitmap is maintained. When a write is issued, the data is immediately written (in a self-identifying manner) to a rotationally close empty location in a reserved track, making it durable. The write is then acknowledged, but the data is retained in the host or controller cache and eventually written to its fixed location. When the data has been written the second time, the corresponding bit in the allocation bitmap is cleared. While significant memory may be required for allocation bitmaps, mapping tables, and write buffers, this storage is not required to be fault-tolerant. Write-twice is typically combined with one of the write buffering techniques to improve the efficiency of the second write. This technique has been pursued most fully for file systems [Solworth91, Orji93].

The floating location technique improves the efficiency of writes by eliminating the static mapping of logical disk blocks and fixed locations in the disk array. When a new location is chosen in a manner that minimizes the disk arm time devoted to the write, and a new physical location mapping is established. We have described one such scheme, floating data and parity [Menon92], in this paper. An extreme example of this approach is the log structure filesystem (LFS) in which all data is written in a segmented log, and segments are periodically reclaimed and written to their permanent location [Rosenblum91]. Using fault-tolerant caches to delay data writes, this approach writes into long sequential transfers, greatly enhancing write throughput. However, if nearby blocks may not be physically nearby, the performance of LFS in read-intensive workloads may be degraded if the read and write access patterns differ. The distributed mirror approach [Solworth91] uses the 100% storage overhead of mirroring to avoid this problem: one copy of the data is stored in fixed location, while the other copy is maintained in floating storage, achieving the same throughput while maintaining data sequentiality [Orji93]. However, floating location techniques require substantial host or controller storage for mapping information and buffered data.

10. CONCLUDING REMARKS

This paper presents a novel solution to the small write problem in redundant disk arrays using a distributed log. Analytical models of the peak bandwidth of this scheme and alternative techniques in the literature were derived and validated by simulation. The proposed technique achieves superior performance than RAID level 5 disk arrays on workloads emphasizing small random access. When data must be pre-read before being overwritten (writes miss in the cache), parity logging achieves performance comparable to floating parity and data without compromising sequential access performance or application control of data placement. When the data to be overwritten is small, performance is superior to floating parity and data and mirroring array configuration. Superior performance is obtained without the 100% disk storage space overhead of mirroring. The technique scales to multiple failure tolerating arrays and can be adapted to accommodate the log structure optimization.

While the parity logging scheme presented in this paper is effective, several optimizations remain to be explored. More dynamic assignment of controller memory should allow higher performance to be achieved or a substantial reduction in the amount of memory required. Application of data to the parity log should be very profitable. The interaction of parity logging and parity declustering [Holland92] merits exploration. Parity declustering provides high performance during degradation and reconstruction while parity logging provides high performance during fault-free operation. A combination of the two should provide a cost-effective system for data protection.

1. ACKNOWLEDGEMENTS

We would like to thank Ed Lee for the original version of Raidsim. Brian Bershad, Peter Chen, Hugo Patterson, Jody Prival, and Scott Nettles who reviewed earlier copies of this work.

REFERENCES

- [Bhide92] Bhide, A., and Dias, D. RAID architectures. Computer Science Research Report RC 17879, IBM Corporation, (1992).
- [Cao93] Cao, J., Lim, S., Venkatesh, S., and Mes, J. The RAID parallel RAID architecture. Proceedings of the 20th Annual International Symposium on Computer Architecture and Design, San Diego CA (May 16-19, 1993). published as special issue of Computer Architecture News, 21, 2 (May 1993), 52-63.
- [Chen90] Chen, P.M., and Patterson, D. A. Maximizing performance in a striped RAID. Proceedings of the 17th Annual International Symposium on Computer Architecture and Design, Santa Monica CA (May 28-31, 1990), IEEE Computer Society Press, Los Alamitos CA, (1990), 322-331.
- [DiskTrend94] DISK/TREND, Inc. 1994 DISK/TREND Report: Disk Drive Arrangements. Mountain View CA (April 1994) SUM-3.
- [Feigel94] Feigel, C. Flash memory heads toward mainstream. Microprocessor Report, Vol. 8, No. 7, May 30, 1994, 19-25.
- [Geist87] Geist, R. M., and Daniel, S. A continuum of disk scheduling algorithms. ACM Transactions on Computer Systems 1 (Feb. 1987), 77-92.
- [Gibson92] Gibson, G. Redundant Disk Arrays: Reliable, Parallel Secondary Storage. MIT Press, (1992).
- [Gibson93] Gibson, G., and Patterson, D. Designing disk arrays for high availability and Distributed Computing. 17, 1-2 (Jan. - Feb., 1993), 4-27.
- [Holland92] Holland, M., and Gibson, G. Parity Declustering for Continuous Operation in Redundant Disk Arrays. Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V) Boston MA (Oct. 12-15, 1992) published as special issue of ACM Notices, ACM, 27, 9 (Sept. 1992), 23-35.
- [IBM0661] IBM Corporation. IBM 0661 Disk Drive Product Description, Model 370, First Edition, Low End Storage Products 504/14-2 IBM Corporation, (1989).
- [Menon92] Menon, J., and Kasson, J. Methods for improved update performance. Proceedings of the Hawaii International Conference on System Sciences, Kauai, HI, (Jan. 7-10, 1992), IEEE Computer Society Press 1 of 4, (1991) 74-83.
- [Menon93] Menon, J. Performance of RAID5 Disk Arrays with Read Caching. Computer Science Research Report RJ9485(83363), IBM Corporation, (1993).
- [Orji93] Orji, C. U., and Solworth, J. A. Doubly distributed RAID. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington DC, (May 26-28, 1993) published as special issue of SIGMOD Record, ACM, 22, 2 (June 1993), 307-318.
- [Ousterhout88] J. Ousterhout, et. al. The Sprite network operating system. 2, (Feb. 1988) 23-36.
- [Patterson88] Patterson, D., Gibson, G., and Katz, R. A case for redundant arrays of inexpensive disks (RAID). the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, IL, (June 1-3, 1988) published as special issue of SIGMOD Record, ACM, 17, 3 (Sept. 1988) 109-116.
- [Polyzois93] Polyzois, C. A., Bhide, A., and Dias, D. M. Disk mirroring with alternative update schemes. Proceedings of the 19th Conference on Very Large Databases, Dublin Ireland (Aug. 24-27, 1993). Morgan Kaufmann, Palo Alto CA, (1993), 604-611.
- [Ramakrishnan92] Ramakrishnan, K. K., Biswas, S., and Karedla, R. Analysis of file I/O traces in commercial computing environments. Proceedings of the 1992 ACM SIGMETRICS Conference on Performance Evaluation, Newport RI, (June 1-5, 1992) published as a special issue of Performance Evaluation Review, ACM, 20, 1 (June 1992). ACM, 78-90.
- [Rosenblum91] Rosenblum, R. and Ousterhout, J. The design and implementation of a log-structured file system. the 13th ACM Symposium on Operating System Principles, Pacific Grove CA, (Oct. 13-16, 1991) published as special issue of Operating Systems Review, ACM, 25, 5 (1991), 1-15.
- [Schulze89] Schulze, M. E., Gibson, G. A., Katz, R. H., and Patterson, D. A. How RAID works. Proceedings of the IEEE Computer Society International Conference (COMPCON 89) No. 91TH0394-7, San Francisco CA, (Feb. 27-Mar 3, 1989), IEEE Computer Society Press, Washington DC (1989) 123-123.
- [Seltzer90] Seltzer, Chen, P., and Ousterhout, J. Disk scheduling. Proceedings of the 1990 USENIX Conference, Washington DC, (Jan. 1990) 22-26.
- [Solworth90] Solworth, J. A. and Orji, C. U. Only disk cache. Proceedings of the ACM SIGMOD Conference, Atlantic City NJ, (May 23-25 1990) published as special issue of SIGMOD Record, ACM, 19, 2 (June 1990), 123-132.
- [Solworth91] Solworth, J. A. and Orji, C. U. Distributed RAID. Proceedings of the First International Conference on Parallel and Distributed Information Systems, Santa Monica CA. No. 91TH0393-4, Miami Beach FL, (Dec. 4-6, 1991), IEEE Computer Society Press, Alamitos CA (1991) 10-17.
- [Salem86] Salem, K., and Garcia-Molina, H. Disk scheduling. Proceedings of the 2nd IEEE International Conference on Data Engineering, IEEE (1986).
- [Stodolsky93] Stodolsky, D., Gibson, G., and Holland, M. Parity Logging: Overcoming the Problem in Redundant Disk Arrays. Proceedings of the 20th Annual International Symposium on Computer Architecture and Design, San Diego CA (May 16-19, 1993). published as special issue of Computer Architecture News, 21, 2 (May 1993), 64-75.
- [TPCA89] The TPC-A Benchmark: A Standard Specification. Transaction Processing Performance Council, (1989).