8-1994

A Redundant Disk Array Architecture for Efficient Small Writes (CMU-CS-94-170)

Daniel Stodolsky
Carnegie Mellon University

Mark Holland
Carnegie Mellon University

William V. Courtright II
Carnegie Mellon University

Garth A. Gibson
Carnegie Mellon University

Recommended Citation

.

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 with 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 organization for second ary storage. Unfortunately, small writes are a substantial portion of the I/O workload of many impor tant, demanding applications such as on-line transaction processing. This paper presents parity logging, a novel solution to the small write problem for redundant disk arrays. Parity logging applies journalling 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 verify these models by simulation. Parity logging provides performance competitive with mirroring, but with capacity overhead close to the minimum offered by RAID level 5. Finally, 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.

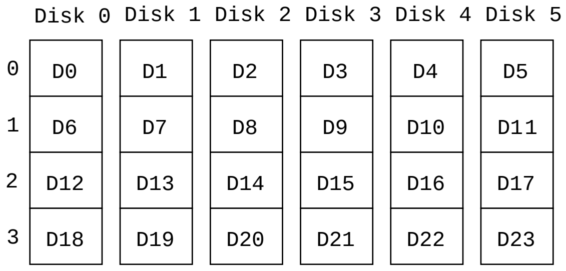
1. INTRODUCTION

The market for disk arrays, collections of independent magnetic disks linked together data store, is undergoing rapid growth and has been predicted to exceed 13 billion doll [DiskTrend94]. This growth has been driven by three factors. First, the growth in processo outstripped the growth in disk data rate. This imbalance transforms traditionally comp 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 substantial cost, power, and performance advantages over larger drives. Third, such systems c made highly reliable by storing a small amount of redundant information in the array. Without this redundancy, large disk arrays have unacceptably low data reliability because of their larg component disks. For these three reasons, redundant disk arrays, also known as Redundant Inexpensive[1] Disks (RAID), are strong candidates for nearly all on-line secondary storage [Patterson88, Gibson92].

Figure 1 presents an overview of the RAID systems considered in this paper. The most promising variant, RAID level 5, employs distributed parity with data striped on a unit that is one sectors.

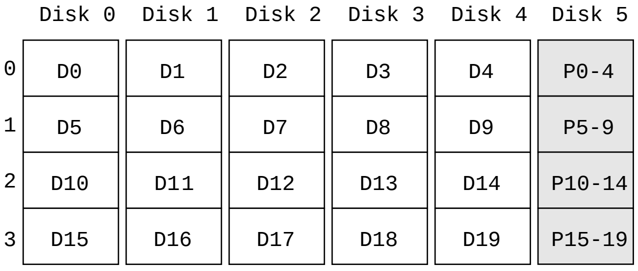
RAID level 5 arrays exploit the low cost of parity encoding to provide high data reliabil Data is striped over all disks so that large files can be fetched with high bandwidth. By di parity, many small random blocks can also be accessed in parallel without hot spots on any d

While RAID level 5 disk arrays offer performance and reliability advantages for a wide



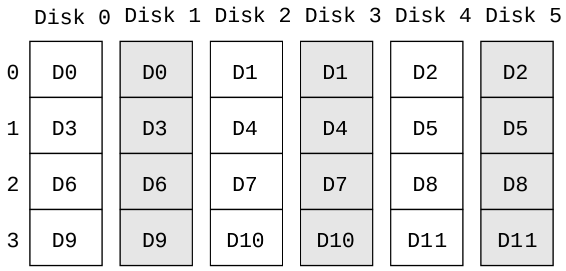
	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	D5
1	D6	D7	D8	D9	D10	D11
2	D12	D13	D14	D15	D16	D17
3	D18	D19	D20	D21	D22	D23

RAID Level 0: Nonredundant



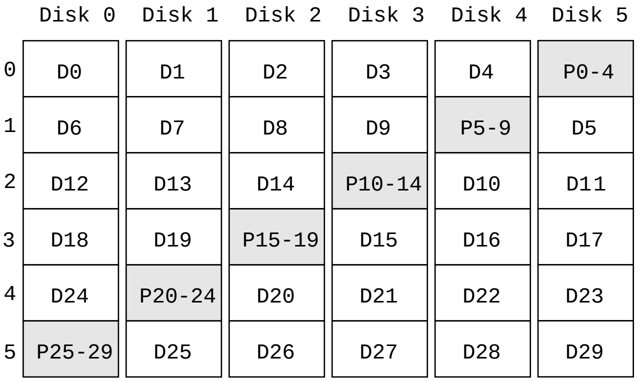
	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	P0-4
1	D5	D6	D7	D8	D9	P5-9
2	D10	D11	D12	D13	D14	P10-14
3	D15	D16	D17	D18	D19	P15-19

RAID Level 4: Block-Interleaved Parity



	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D0	D1	D1	D2	D2
1	D3	D3	D4	D4	D5	D5
2	D6	D6	D7	D7	D8	D8
3	D9	D9	D10	D10	D11	D11

RAID Level 1: Mirroring



	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	D0	D1	D2	D3	D4	P0-4
1	D6	D7	D8	D9	P5-9	D5
2	D12	D13	D14	P10-14	D10	D11
3	D18	D19	P15-19	D15	D16	D17
4	D24	P20-24	D20	D21	D22	D23
5	P25-29	D25	D26	D27	D28	D29

RAID Level 5: Rotated Block-Interleaved Par

Fig. 1. Data Layout in RAID Levels 0, 1, 4 and 5. This figure shows the first few units on each disk in e levels. “D” represents a block, or 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 dis units. Shaded blocks represent redundant information, and non-shaded blocks represent user data. and does not tolerate faults. Level 1 is simple mirroring, in which two copies of each data unit exploit the fact that failed disks are self-identifying, achieving fault tolerance using a s lowering the capacity overhead to only one disk out of six in this example. Levels 4 and 5 diff parity. In level 5, the parity blocks rotate through the array rather than being concentrated on a access bottleneck.

---

[1] In current industry usage, the “I” in RAID denotes “independent”.

TPC Benchmark

```
get request from terminal
begin transaction
  update account record
  write history log
  update teller record
  update branch record
commit transaction
respond to terminal
```

Scaling Requirements

Record Type	Minimum Quantity per TPS	Record Size (Bytes)
Account	100K	100
Teller	10	100
Branch	1	100
History	30K	50
Total		11.5 MB per TPS

Fig. 2 OLTP Workload Example. The transaction processing council (TPC) benchmark is an ind
benchmark for OLTP systems stressing update-intensive database services [TPCA89]. It models the cc
customer withdrawals and deposits at a bank. The primary metric for TPC benchmarks is transacti
Systems are required to complete 90% of their transactions in under 2 seconds and to meet the scal
Customer account records are selected at random from the local branch 85% of the time, and from
the time. Because history record writes are delayed and grouped into large sequential writes and
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 penalize
four over nonredundant arrays for workloads of mostly small writes [Patterson88]. This pen
because a small write request may require the old value of the targeted data be read (we call this
a preread), overwriting this with new user data, prereading the old value of the corresponding,
then overwriting this second disk block with the updated parity. In contrast, systems based on mirrored
disks simply write the user's data on two separate disks and, therefore, are only penalized by a f
two. This disparity, four accesses per small write instead of two, has been termed the small write
problem.

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

This paper describes and evaluates a powerful mechanism, parity logging, for eliminating this small
write penalty. Parity logging exploits well understood techniques for logging or journallin
transform small random accesses into large sequential accesses. Section 2 of this paper
parity logging mechanism. Section 3 introduces a simple model of its performance and cost
describes alternative disk system organizations, develops comparable performance mode
contrasts them to parity logging. Section 5 provides an analysis of small-write overhea
logging with respect to configuration and workload parameters, and analyzes potential load
in a parity logging system. Section 6 introduces our simulation system, describes impleme
parity logging and alternative organizations, and contrasts their performance on workloa
random writes and an OLTP workload. Section 7 analyzes extensions to multiple-failure tole
arrays. Section 8 discusses how the large write optimization can be accomodated in a par
disk array. Section 9 reviews related work. Section 10 closes with a few comments on futur
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 m
the fact that disks deliver much higher bandwidth on large accesses than they do on smal
logging disk array batches small changes to parity into large accesses that are much more e
model is introduced in terms of a simple, but impractical RAID level 4 scheme, then ref
realistic implementation used in our simulations.

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

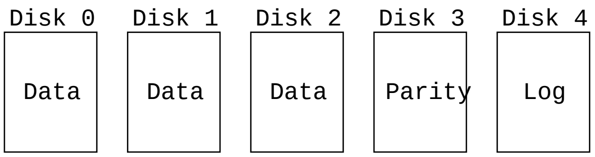


Fig. 4. Basic Parity Logging Model.

because the data transfer component is much smaller than the seek and rotational pos components. Thus a disk servicing a small-access-dominated workload spends the majority of positioning instead of transferring data. Figure 3 shows the relative bandwidths of random block, track and cylinder accesses for a modern small-diameter disk [IBM0661]. This figure largely bear lore of disk bandwidth: random cylinder accesses move data twice as fast as random trac 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)	6.95 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	~0	C_P	Cylinders of parity per region	~0
K	Tracks buffered per region	1	L	Log striping factor	1

Fig. 5. Model Parameters. The bandwidth utilization models of Section 2, 3, and 4 are present parameters list above. The first table presents common disk parameters and the second, parameters The first and fourth columns in each table show the symbol used in the text; the second and fift symbol denotes; and the third and last column show the default value of the parameter as used in for milliseconds and a tilde (~) indicates an approximate value.

Logically, we develop our scheme beginning with Figure 4 in which a RAID level 4 disk a augmented with one additional disk, the log disk. Initially, this disk is considered empty. As in RAID level 4, a small write prereads the old user data, then overwrites it. However, instead of similarly updating parity with a preread and overwrite, the parity update image (the result of XORi and new user data) is held in a dedicated block of memory called a log buffer. When enough parity update images are buffered to allow for an efficient disk transfer (one or more tracks), th 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 memory using sequential cylinder accesses. The logged parity update images are applied memory image of the stale parity and the resulting updated parity is written with large writes. When this completes, the log disk is marked empty and the logging cycle begins aga

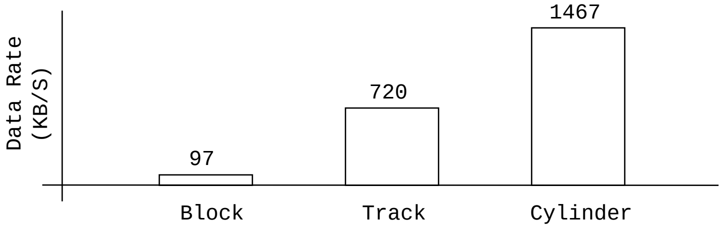


Fig. 3. Peak I/O Bandwidth. This figure shows the sustainted data rate in kilobytes per second that written to an IBM 0661 drive using random one block (2KB), one track (24 KB), and one cylinder (336 for disk parameters).

Because only parity updates (not data changes) are deferred, this scheme preserves sin tolerance.[2] If a data disk fails, the log disk (and any buffered parity updates) are first parity disk, which is then used to reconstruct the lost data in the same manner as is done 5. If the log or parity disk fails, the system can simply recover by reconstructing parit onto the surviving parity or log disk. The failed drive is then replaced with a new empty controller fails, its buffered parity updates are lost, but, after the controller has 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 than in a comparable RAID level 4 or 5 array. 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 p corresponding data unit, which introduces an overhead of one block (data unit) access pe addition, each user write to a data unit consumes buffer memory equal to the size of the u track's worth (D) of small (unit-sized) writes issued to the array causes one track write to to occur. Next, a disk's worth (TVD) of small writes causes the log disk to fill up, which must th emptied by updating the parity. This update involves reading the entire contents of the parity a disks ($2V$ cylinders), and then writing the entire parity disk (V cylinders) at cylinder transfer rates. On average, then, for every TVD small user writes there are TVD block accesses, TV sequential track accesses, and $3V$ cylinder accesses for maintenance of the parity information. Recall track ac D times larger than random small writes but about 10 times more efficient and cylinder ac twice as efficient and T times larger than track accesses. Thus, parity maintenance for a disk's worth (TVD) 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, parity maintenance for TVD small writes would consume about as much disk time as $3TVD$ 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 mainte about a factor of two.[3]

In many cases, it may be possible to avoid the preread of the user data. For example, benchmark (Figure 2), the update of a customer account record is a read-modify-write ope account record is read, modified in memory, then written back to disk. In these cases, the old data is usually known (cached) at the time of the write and the preread of the data may b [Menon93]. Under these conditions, the overhead for RAID levels 4 or 5 is just two ran accesses per small write, or $2TVD$ random block accesses per TVD 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 consume maintenance by about a factor of eight.

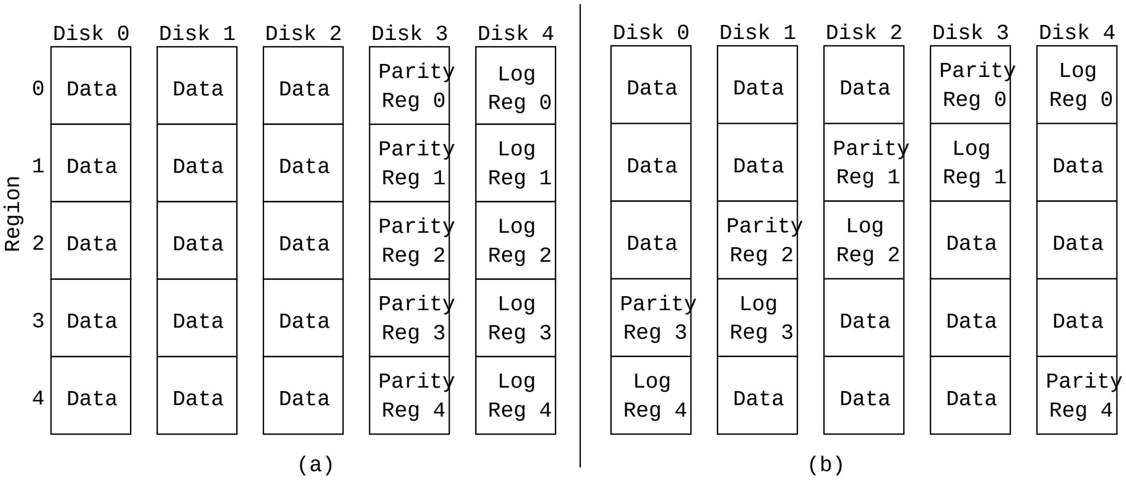
2.1. Partitioning the Log Into Regions

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

---

[2] Our failure model treats disk and controller failures as independent. If concurrent controller and disk 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 savin data writes are deferred and optimally scheduled [Solworth90, Orji93].



(a)

Region	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	Data	Data	Data	Parity Reg 0	Log Reg 0
1	Data	Data	Data	Parity Reg 1	Log Reg 1
2	Data	Data	Data	Parity Reg 2	Log Reg 2
3	Data	Data	Data	Parity Reg 3	Log Reg 3
4	Data	Data	Data	Parity Reg 4	Log Reg 4

(b)

Region	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	Data	Data	Data	Parity Reg 0	Log Reg 0
1	Data	Data	Parity Reg 1	Log Reg 1	Data
2	Data	Parity Reg 2	Log Reg 2	Data	Data
3	Parity Reg 3	Log Reg 3	Data	Data	Data
4	Log Reg 4	Data	Data	Data	Parity Reg 4

Fig. 6 Parity Logging Regions.

journalled into that region's log. When a region's log fills up, only that region's log is required to update that region's parity. 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 show number of regions is dependent on disk capacity, but is about 100 in our example 22-disk array.

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

2.2. Striping Log and Parity for Parallelism

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

Following the example of RAID level 5, Figure 7 shows a layout in which the parity for ea distributed across the array to increase bandwidth. This distribution decreases the reintegrating parity updates for a particular region by using all $N-1$ non-log disks to effect the parity read and write. So that these operations are also efficient, the granularity of distribution is one contiguous set of parity units per disk per region. The log, however, remains a potential bottleneck.

The log bottleneck may also be eliminated by distributing the log for each region over mu 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, they cannot be placed on the same disks as the data they protect. If they were, the failure of that disk would cause both data and pari which is an unrecoverable failure in a disk array using a parity-based code. To avoid data loss, data and log for each region are restricted to disjoint sets of disks. Thus, log striping reduces th on which data for a particular region may be placed. If, for example, the log is striped o data for that region may be placed only on the other $N-3$ disks.

This reduction in data striping in a region increases the disk space overhead as follows. Let L be the number of disks over which each log is striped and C_P the number of cylinders of parity per region. Th number of data cylinders per region, C_D, is related to the size of the parity, C_P, according to the standard RAID level 4 and 5 rule for data striped over $N-L-1$ disks

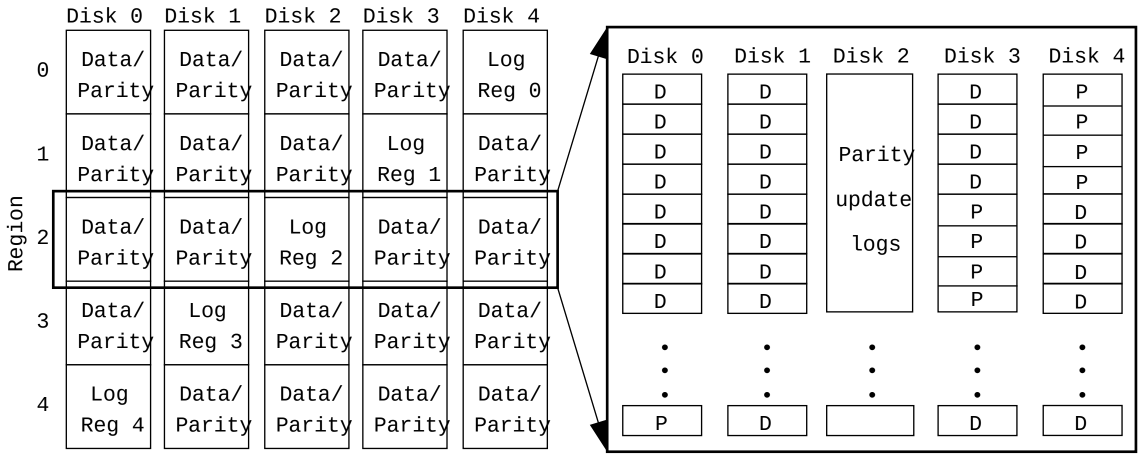


Fig. 7. Block Parity Striping. The inset shows a detailed layout of a sample region.

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

where N is the number of disks in the array. Because the log is equal in size to the parity, C_L, the number of cylinders of log per region, equals C_P. Hence, the disk space overhead (the fraction of the ar 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, L, increases. Figure 9 shows the disk space overhead for dif ent degrees of log striping for an array of 22 disks. However, Section 6 will show that 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 (C_L cylinders) for log and

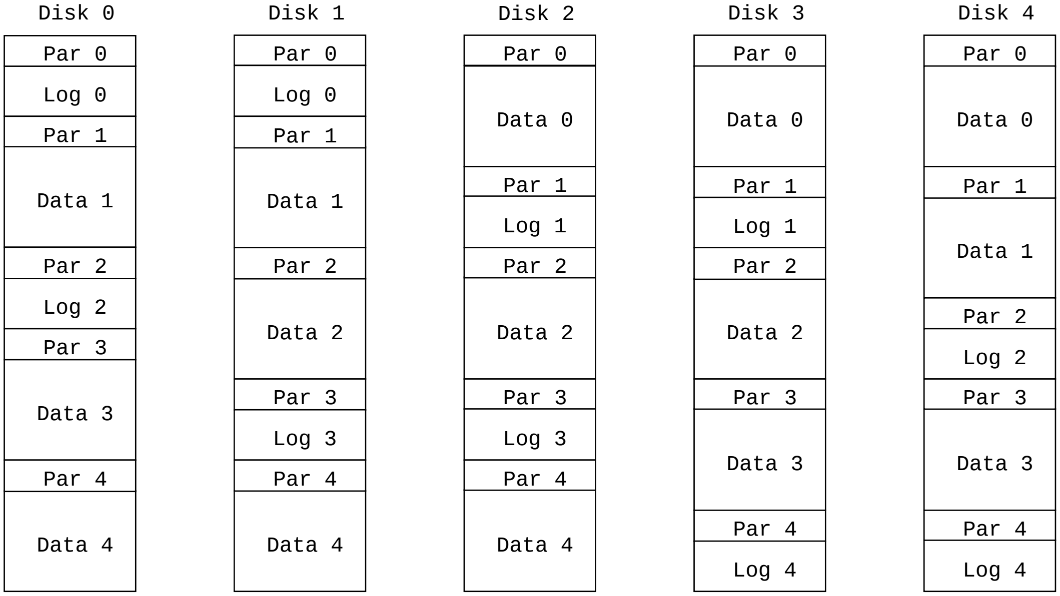


Fig. 8. Distributed Parity Logs

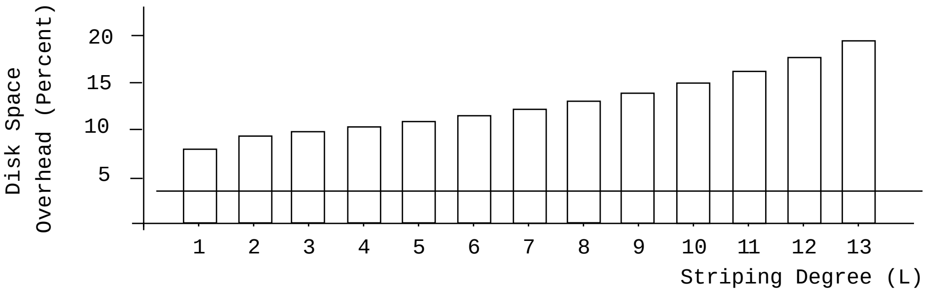


Fig. 9. Disk Storage Overheads. The horizontal line shows the capacity overhead of
configuration of the same array.

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 in each region ($A = C_L/C_P$). 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 up after ATC_PD small user writes into that region. Updating the
parity still requires prereading old data on each small user write (ATC_PD blocks) (assuming the old data is not cached), writing the log buffers (ATC_P tracks), plus, every time the log fills, reading parity (C_P cylinders), reading the log (AC_P cylinders), and writing the updated parity (C_P cylinders),. Thus the parity maintenance work for ATC_PD uncached small user writes is

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

random small accesses, or an overhead of $(23/20 + 1/10A)$ random small accesses per uncached small user write. Performance can therefore by traded for space, as shown in Figure 10. Applyin
example 22 disk array with logs striped over two disks ($L = 2$), allocating twice as much log as parity
($A = 2$) increases the space overhead from 9.5% to 13.6% of the total capacity, but decreases the parity maintenance overhead from 41.7% to 40% of that of RAID level 5, where three related parit
occur for each small user write. Halving the amount of log ($A = 0.5$) decreases 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 does $(3/20 + 1/10A)$. Applying this cached workload to our 22 disk array with
striped over two disks does not change the space overhead. However, in this cached case, 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 u
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-app
can be performed. If K tracks are transferred in a log-append operation and there are B regions, then,
conservatively, KB tracks of buffer memory are required to delay log appends. Second, wheneve
log for a region fills, the parity for that region is read into memory, the newly full log is read and applied

to it, and the updated parity is written back. This parity reintegration operation requires $C_P T$ tracks of buffer memory, where C_P is the number of cylinders of parity per region and T is the number of tracks per cylinder. Since the number of cylinders of parity per region is the same as the total c disk, V, divided by the number of regions, B, the total buffer memory space is $TV/B + KB$, measured in tracks.

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

In practice, parameters such as the number of regions must be discrete. If we further req size per region of the log appends, sublogs (the portion of a region’s log on one disk), 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, B, is allowed to vary from the optimum by ±10%, then a set of integral parameters can be found such that the wasted disk less than 1% of the array’s total space.

If, however, the size per region of the sublogs, parity and data, per region, are only requ integral number of disk sectors (rather than tracks), substantially less disk space is wa number of regions, B, 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 par but because these positioning overheads are small relative to track access times, par performance is only slightly affected (3% for our experiments).

A more significant performance degradation results if small user writes are blocked d reintegration of a region’s log into its parity. This blocking should be minimized by managing the per region buffers as a single global buffer pool. Using this approach, user writes are only 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 a log effici. It then further delays their reintegration into a redundant disk array’s parity until there are enough parity updates in the log to make a complete revision of the parity efficient. To accommodate limited memory for reintegration of parity records, the disk array is partitioned into regions with per-region logging. Then, to avoid bandwidth bottlenecks, parity and log information is striped over multiple parity logging scheme reduces the extra work done by RAID level 5 arrays for small random little more than is done in the much more expensive, traditional mirrored approach eve

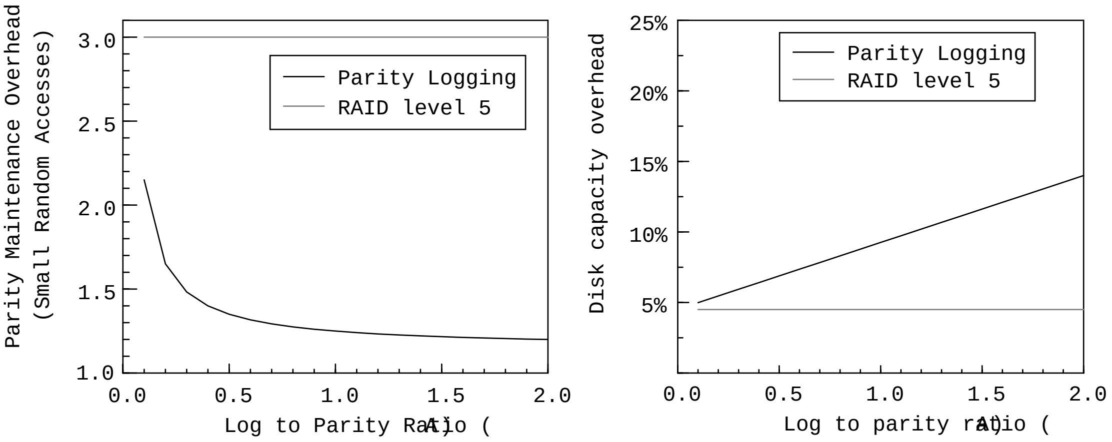


Fig. 10 Log Length and Efficiency

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 utiliza 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. User data must be preread, then overwritten. This is done in an access which seeks to the cylinder with the user's data, waits for the data to rotate under the head, reads the data, waits for the disk to rotate around once, th data.[4] Defining S as the average seek time, R as the time for one-half of a disk rotation, and reca that D is the number of data units per track, the time to perform this operation, t_{rmw}, is

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

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, is

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

disk seconds.

Each region has K tracks worth of log buffers. On average, for every KD small user writes, one region's log buffers will fill and be written to the disk in a single K-track write. Defining H as the disks' head-switch time, the number of disk seconds required to do this, $t_{K\,track}$, is

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

assuming all K tracks are on the same cylinder.[5]

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

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

---

[4] This single access could be separated into two accesses each taking $S+R+2R/D$ disk seconds for a total of $2S+(2+4/D)R$. For most modern disks S is about twice R, so the single access is more efficient.

[5] Disks that support zero-latency writes [Salem86] can eliminate the initial rotational positioning delay. This can reduce 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 ($K=1$). However, the impact of zero-latency write support on parity logging is small (under 3%), because the track-s are only a small contributor to parity logging overhead (Figure 11).

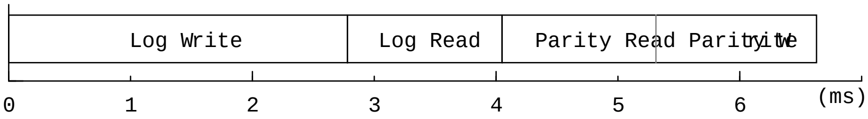


Fig. 11 Parity Logging Overheads The amortized overhead cost of parity and log accesses do
parity logging array is shown above. The log writes contribute approximately 40%
milliseconds), while the cylinder rate log reads, parity reads and parity writes each
milliseconds). For comparison, the parity accesses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

disk seconds, and may be rewritten as

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

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

$$\underbrace{(S+R)}_{\text{First seek and rotational delay}} + \underbrace{(C_P/(N-1))}_{\text{Cylinders per subaccess}}\underbrace{(2RT + (T-1)H)}_{\text{Read time for 1 cylind}} + \underbrace{(C_P/(N-1)-1)M}_{\text{Single track seeks per subaccess}}$$

disk seconds. The total striped access, t_{C_P}, takes

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

disk seconds.

Thus, on average, the disk utilization induced by a small write, t_{amw}, is

$$t_{amw} = t_{rmw} + \underbrace{\frac{1}{KD}[t_{K\,track}]}_{\text{Log write}} + \underbrace{\frac{t_{C_L}}{DTC_L}}_{\text{Log read}} + \underbrace{\frac{2t_{C_P}}{DTC_L}}_{\text{Parity read and write}} \tag{7}$$

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

The analysis for a parity logging disk array with a striped log (Figure 8) is similar. When a region's log buffer fills, it will be written to one of the region's L sublogs in a single K-track write. The cost of this operation is the same as in the unstriped case. Log reintegration still occurs every DTC_L small user writes, but now consists of three striped I/Os: a striped read (over L disks) of the log, and a striped (over N disks) read and write of the parity. Each of the accesses in the striped log read costs

$$\underbrace{(S+R)}_{\text{First seek and rotational delay}} + \underbrace{(C_L/L)}_{\text{Cylinders per subaccess}}\underbrace{(2RT + (T-1)H)}_{\text{Read Time for 1 Cylinder}} + \underbrace{(C_L/L - 1)M}_{\text{Single track seeks per subaccess}}$$

for a total of

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

disk seconds. Similarly, the 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:	$2.0 + 0.01 \cdot dist + 0.46 \cdot \sqrt{dist}$ (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 rep 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 grou [Gibson93]. Disk parameters are modeled on the IBM Lightning drive [IBM0661]. Note that the dist in the seek model is the number of cylinders traversed, excluding the destination. As is commonly done in S chosen to equal the head switch time, optimizing data layout for sequential multitrack access.

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

disk seconds. Thus, striping introduces an additional overhead of $L(S+R-M)$ disk seconds to the log reintegration. This increases the parity maintenance overhead per small wri $L(S+R-M)/DTC_L$ disk seconds. As Section 6 will show, this increase in parity maintenance work is worthwhile because it reduces long reintegration periods during which disk queues grow, the system 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, di 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 0), mirrored disks (RAID level 1), distributed N+1 parity (RAID level 5), and floating data and parity. The 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 whi consumes

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

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 copies. Each access takes as much time as a small write in a nonredundant disk array, $S+(1+2/D)R$. Hence, each small user write utilizes disks for a total of $2S+(2+4/D)R$ seconds. While mirrored disks' write operations are more efficient than RAID level 5, half of their capacity is devoted to

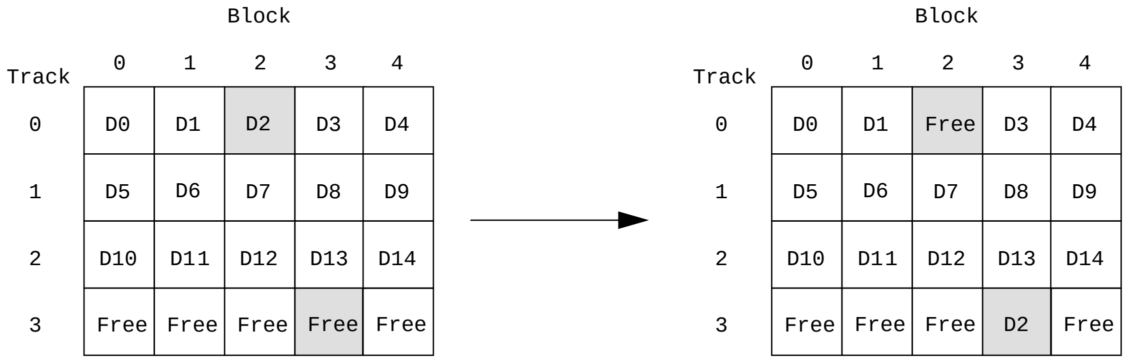


Fig. 13. Floating Data/Parity. This figure shows the movement of data within a cylinder caused by a data and parity array. Each grid represents one cylinder of four tracks, with five blocks per track. the controller searches for a free block within the cylinder that is rotationally close to block at offset 3 into track 3. Immediately following the preread of block D2, the controller writes t 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 5 case, 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

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

disk seconds for a total disk busy time of $2S+(6+4/D)R$. Again, no long-term controller storage is required.

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

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

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 $2S+(2+11.04/D)R+2H$. Note that if the number of data units per track, D, is large and the head-switch time, H, is 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 sto overheads. For an N disk array with T tracks per cylinder, floating data and parity has a storage overhead of $(T+N-1)/(TN)$.[6] Floating data and parity arrays, however, require substantial fault-tolerant storage in the array controller to keep track of the current location of data a [7] For

---

[6] Each disk gives up $1/T$ of its capacity for free space and the array gives up $1/N$ of the remaining space for parity. Thus the array storage efficiency is $(T-1)(N-1)/TN$ and the array storage overhead is $1-(T-1)(N-1)/TN = (T+N-1)/TN$.

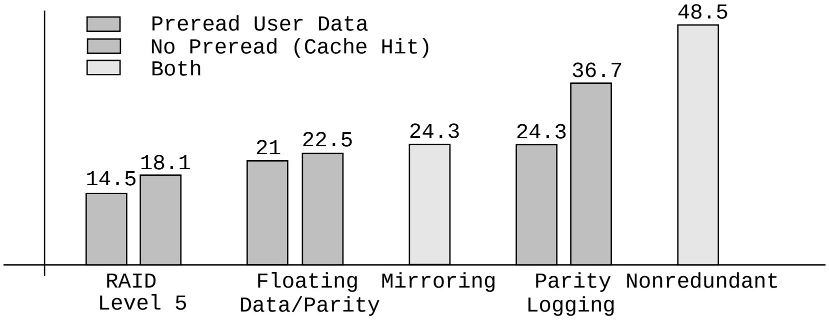


Fig. 14. Model Estimates. User writes per second per disk as predicted by the bandwidth models of S predictions assume 100% disk utilization, FIFO disk arm scheduling, and an unbounded number of r and parity logging disk arrays both benefit substantially from not having to preread user data substantially 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 DT bits per cylinder. In addition, a table of current block locations for each cylinder is required. This consumes $D(T-1)\lceil \log(DT) \rceil$ bits per cylinder. Thus, with V cylinders per disk, a total of $VD(T+(T-1)\lceil \log(DT) \rceil)$ bits of fault-tolerant controller storage are required. For the disks in Figure 12, this is 1,343,784 bi disk. The total controller storage in a 22 disk array is about 3,608KB, roughly comparab 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 r RAID level 5, its performance for other types of accesses is degraded. Within a cylinder, logically contiguous user data units are not likely to be physically contiguous. In the worse case, t data units may end up at the same rotational position on two different tracks, requiring disk rotation to read both. In addition, the average track has only $D(T-1)/T$ valid data units. Thus, even on disks with zero-latency reads, the maximum sequential read bandwidth is reduced, or by $(T-1)/T$.

5. ANALYSIS

Figure 14 compares these models' estimates for maximum throughput of the example array ba Figure 12. Throughput at lower utilizations may be calculated by scaling the maximum thr numbers by the disk utilization. Figure 14 predicts that parity logging and floating data an 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 r performance of parity logging and its alternatives except for the effects of the number of track and the ratio of average seek time to rotational latency. This section describes the effects of thes parameters and the effects of log striping degree on array load balance.

Of the models' parameters, the number of data units per track, D, has the greatest impact on performance. Parity logging transfers each data unit two more times than RAID level 5 and times than mirroring. If the transfer time of a unit is small, parity logging will be eff shows the relative performance when data caching is ineffective (i.e, a preread is requi logging, mirroring, and RAID level 5 for different values of numbers of data unit per tra example array. The performance of mirroring exceeds that of parity logging with 13 or fewer per track ($D \le 13$), and RAID level 5 performance exceeds that of parity logging with the unlik of 1 or 2 data units per track ($D \le 2$). Industry estimates, however, show that track capacity within a given form factor is increasing at over 20% a year. Consequently, 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 (S) to rotational latency (R) has a substantial impact on the performance of parity logging disk arrays. Figure 16 plots the performance of parity lo level 5 and mirroring relative to RAID level 0 as this ratio changes. The performance o

---

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

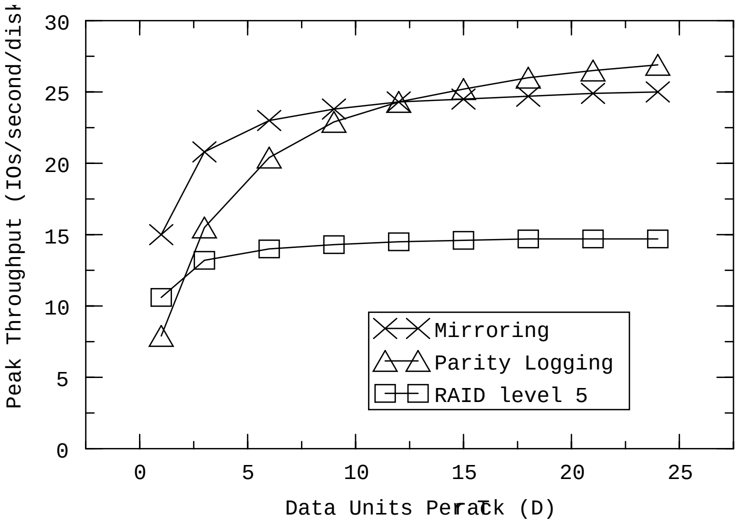


Fig. 15: Effects of track size on throughput. The performance of parity logging is highly sensitive to
units per track. The figure above shows the performance in the example 22 disk array of mirroring
level 5 on a workload of 100% blind small writes for varying number of data units per track.

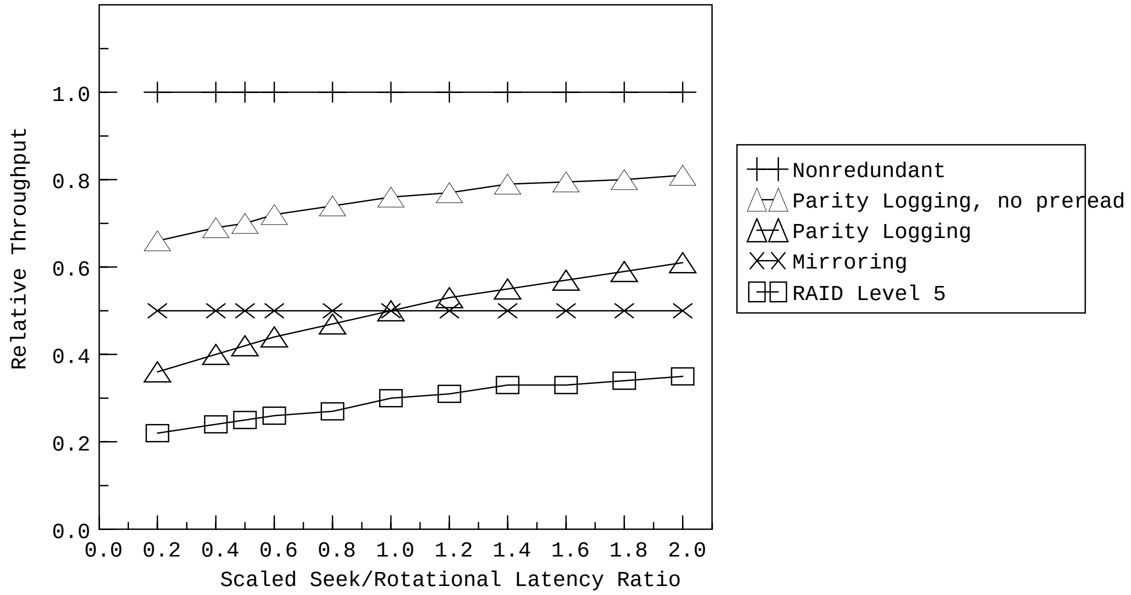


Fig. 16: Peak throughput, normalized to nonredundant array performance, as a function of the ratio
seek time to rotational latency. Altering the ratio of average seek time (S) to the rotational latenc
the relative performance of mirroring, floating data and parity, nonredundant and parity logging disk arrays. Sh
the relative performance of these approaches on the example 22 disk array (Figure 12) as the ave
varied. The average seek time is varied from 20% of the Lightning average seek to twice that of th
parameter range models a large spectrum of drives, from those with very fast positioning to Ligh
7200 RPM. The X-axis has been linearly scaled so that 1.0 corresponds to the ratio of average see
the Lightning drive.

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

Figure 14 assumes the user requests access data uniformly. While this assumption is reasonable for huge OLTP databases, other workloads may exhibit substantial locality. In the worst 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” region; however, log and data traffic are partitioned over non-overlapping disks. If this traffic is not balanced, parity logging pe fall short of Figure 14.

The log, parity and data traffic can be balanced by determining the appropriate degree of L. Recall that every DTC_L small user writes (where D, T, and C_L are the number of data units per track, tracks per cylinder, and cylinders of log per region, respectively) to the $N-L$ data disks of a particular region will cause TC_L/K writes of K 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 da the work per sublog disk. That is,

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

where $t_{K\,track}$ (Equation 3), and $t_{C_L(L)}$ (Equation 8) are the service times for a K-track write and a full log read striped over L disks, respectively, 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, t_z equals t_w (Equation 2). When caching is ineffective, t_z equals $t_{rm\,w}$ (Equation 11). Expanding $t_{C_L(L)}$ in Equation 10 yields a quadratic equation in L whose solution is omitted here because it is unnecessarily complex. Because $t_{C_L(L)}$ is dominated by transfer time ($2RTC_L$), we approximate this balance equation as a linear L equation i whose solution is

$$L = \frac{N}{1+(KDt_z)/(t_{K\,track}+2KR)} \qquad (11)$$

Using this approximation and the disk array parameters from Figure 12, one obtains $L \approx 0.16N$ for blind writes (when $t_z = t_{rm\,w}$) and $L \approx 0.11N$ when caching is effective (when $t_z = t_w$). 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 ti arrays, we simulated the example array described in Figure 12 under five different config nonredundant, mirroring, RAID level 5, floating data and parity, and parity logging. Parity logging was simulated with a single track of log buffer per region ($K = 1$) for several different degrees of log striping (L). The simulations were performed using the RAIDSIM package, a disk array simu derived from the Sprite operating system disk array driver [Ousterhout88], which was exte implementations of parity logging and floating data and parity

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

6.1. The Need for Log Striping

Figure 17 shows peak throughput, response time[8] and response time variance as the degree of log

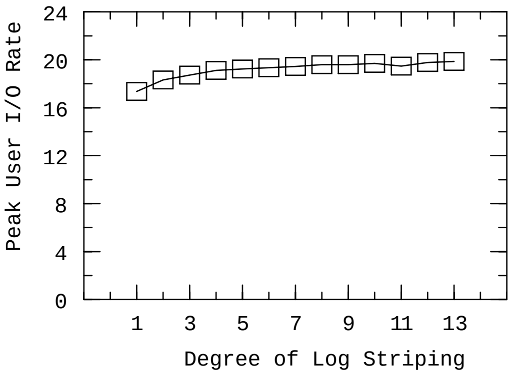


Figure 17(a): Peak user I/Os

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

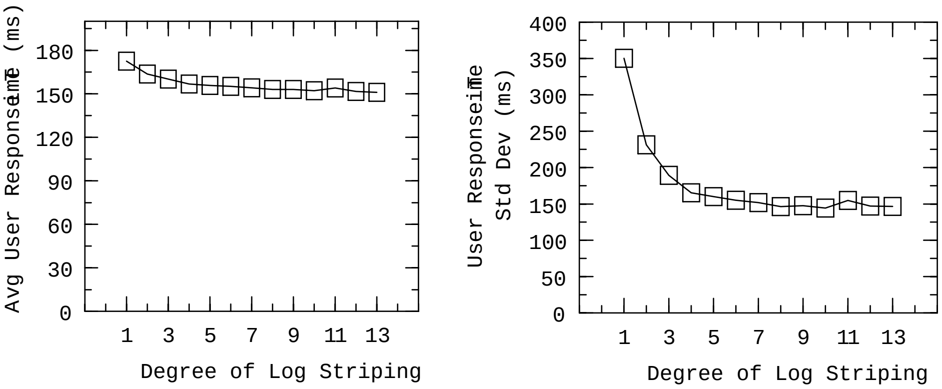


Fig. 17(b): Response time and response time standard deviation at peak load

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 configura more widely striped logs. This behavior results from a “convoy effect” in which proce blocking writes queue behind very long sublog read accesses. Figure 18 shows sublog read ti degrees of log striping. While these long accesses are efficient, they completely tie up a at a time. During this period, any access to the disks involved in the striped log re reducing the effective concurrency in the system. This concurrency reduction causes other array to become idle until the log read completes, reducing peak throughput and utiliz convoy effect also has a substantial impact on response time; requests that block behind th requests will have very long response times, leading to an increase in both average respon response time variance. Fortunately, a modest degree of striping eliminates the convoy effect. Fig shows that striping the log over six disks achieves most of the available throughput wit increasing disk space overhead.

With convoys avoided by a log striped over six disks, Figure 19 compares the performance logging array with one track buffered per region against alternative organizations: in Section 4: nonredundant, mirroring, RAID level 5, and floating data and parity. The graphs of this figure present performance in terms of response time as a function of throughput. Figures 19(a)-(b) assu user data must be preread (data cache miss), and Figure 19(c) presents the corresponding c no preread (data cache hit) case.

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

---

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

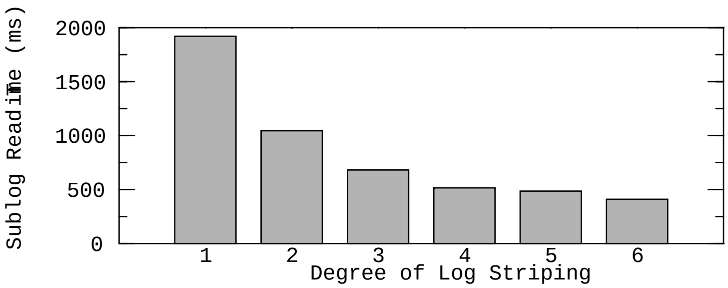


Fig. 18 Sublog Read Times.

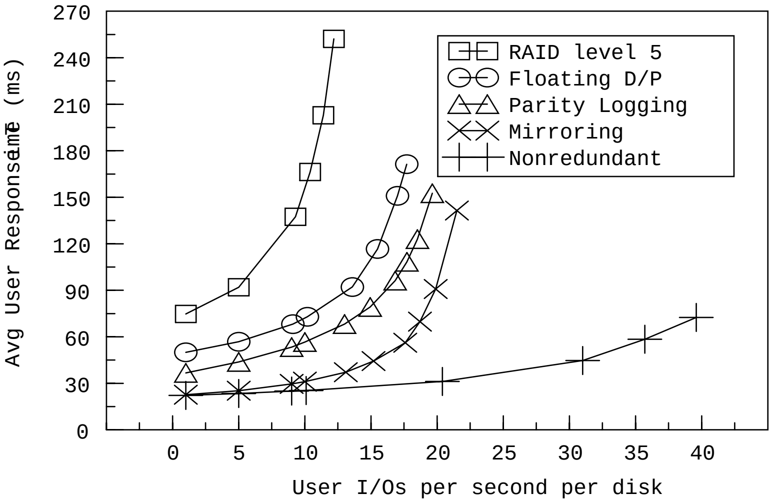


Figure 19(a): Response times

Fig. 19 User Response Times and Disk Utilization. Figures 19(a)-(c) present the average user resp
response time standard deviations as a function of the number of small random writes achieved pe
19(a) and (b) present the results when the user data must be preread, while the results in Figu
was cached, making the preread of the user data unnecessary. In addition to reducing the amount of I/O require
data allows the user write and parity update to occur concurrently, significantly reducing response time for RAID
floating data and parity. The reported times are in milliseconds. The response time standard deviation
is essentially identical to Figure 19(b).

time. Mirroring shows a similar behavior, but is driven into saturation with half as much load.
contrast, each small user write in RAID level 5, when user data must be preread, sequentia
two slow read-rotate-write accesses.[9] Unloaded system response time is thus quite high and queuin
effects cause it to grow quite rapidly with load. While the response time for parity loggi
loaded system is approximately 14 ms (one revolution) higher than mirroring because of the
rotate-write accesses, the peak throughput and response time are quite similar. Similar to RAID level
5, floating data and parity arrays require two read-rotate-write accesses per user wri
minimizing rotational delays, floating data and parity achieves peak throughput similar
logging and mirroring. Response time, however, is significantly longer.

Figure 19(c) shows the performance of all configurations when data cache hits eliminate th
prereads. As expected, this has no effect on mirrored or nonredundant systems, but imp
performance of the other three configurations. RAID level 5 benefits substantially from elim
full rotation delay incurred by a data preread. In addition, a user's data write and parity update can be
issued concurrently, further improving the response time and array utilization. Floating da
parity achieves a lesser benefit from elimination of the preread because its preread overh
less. Response time does drop, however, because of the ability to issue user write and parity u
accesses simultaneously. The response time of parity logging improves by a full rotationa
because of the elimination of the preread rotate, providing an unloaded response time comp

---

[9] In a highly aggressive implementation, it is possible to initiate the parity read-rotate-write access after t
user data completes, but we assume that no status is returned until the entire read-rotate-write access completes

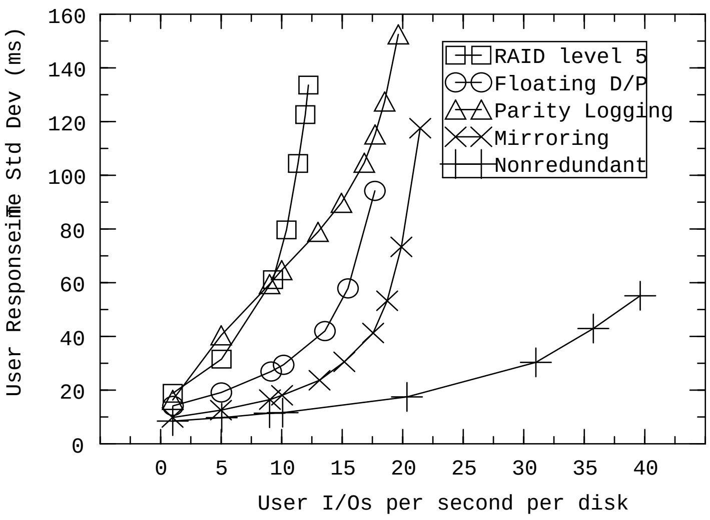


Figure 19(b): Response time standard deviation

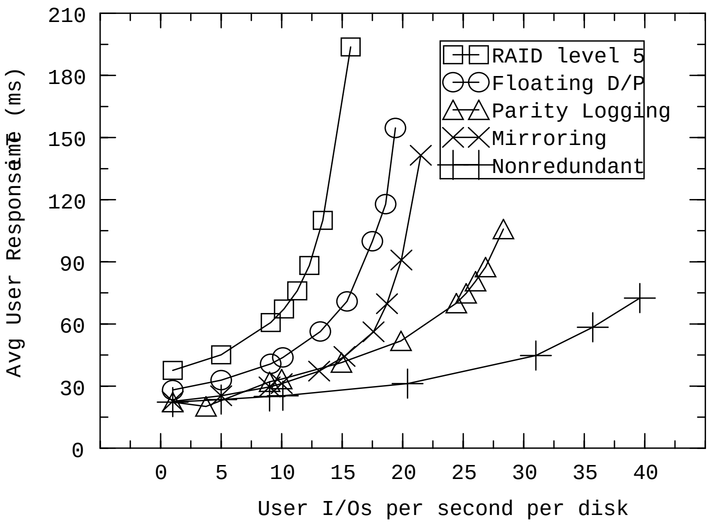


Fig. 19(c): Response times without prereads

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 structure of parity logging: most accesses are fast because inefficient work is delayed. H accesses see long response times as delayed work is (efficiently) completed. With this higher variance in mind, we conclude that the response time estimates in Figure 19 show that parity logging and 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 t of Figure 19(a) and (c). A direct comparison, however, will display significant discrepancies because o 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 utiliz load for the configurations simulated. These peak-load disk utilizations differ according t of concurrent disk accesses issued by a user write in each configuration. RAID level 5 and and parity, when user data is not cached, and parity logging and nonredundant disk arrays, r of caching, present only one disk access request at a time per process. Mirroring and the cases for RAID level 5 and floating data and parity keep the array busier because each user two concurrent disk accesses. Figure 21 shows that, when these difference are accounted fo the model predictions of Figure 14 by the disk utilizations of Figure 20, simulation throu 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 a (2KB) random writes. This section examines a mixed workload, defined in Figure 22, mode statistics taken from an airline reservation system [Ramakrishnan92]. In this more general workload, the results of the earlier sections are modified by two important effects: reads a large writes. The issues encountered in extending floating data and parity to handle var access are beyond the scope of this paper and this technique is omitted from this sectio other array configurations, parity logging, mirroring and RAID level 5, there is no difference in read performance. This will have the effect of compressing the overall performance differenc configurations. Writes that are not small, however, will hurt the performance of parity logging as discussed in Section 5.

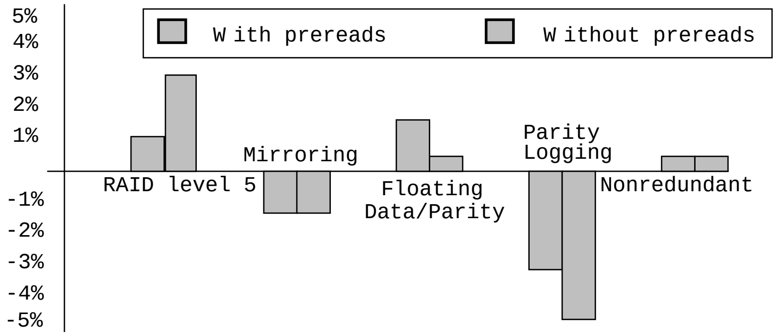


Fig. 21. Model errors. This figure shows the percent error between the models of sections 3 and 4 Section 6. The model predictions have been scaled by the achieved disk utilizations of Fi disagreement between the simulation and the models is less than 5 percent. Note that the 95% simulation response time is also ±5% of 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 gen an airline reservation system [Ramakrishnan92]. This workload is reported as approximately 82% r 4.61 KB, and a median read size of 3 KB. The mean write size was larger, 5.71 KB, but the median write size was 1. KB. Locality of reference and overwrite percentages were not reported. All accesses are assum boundaries.

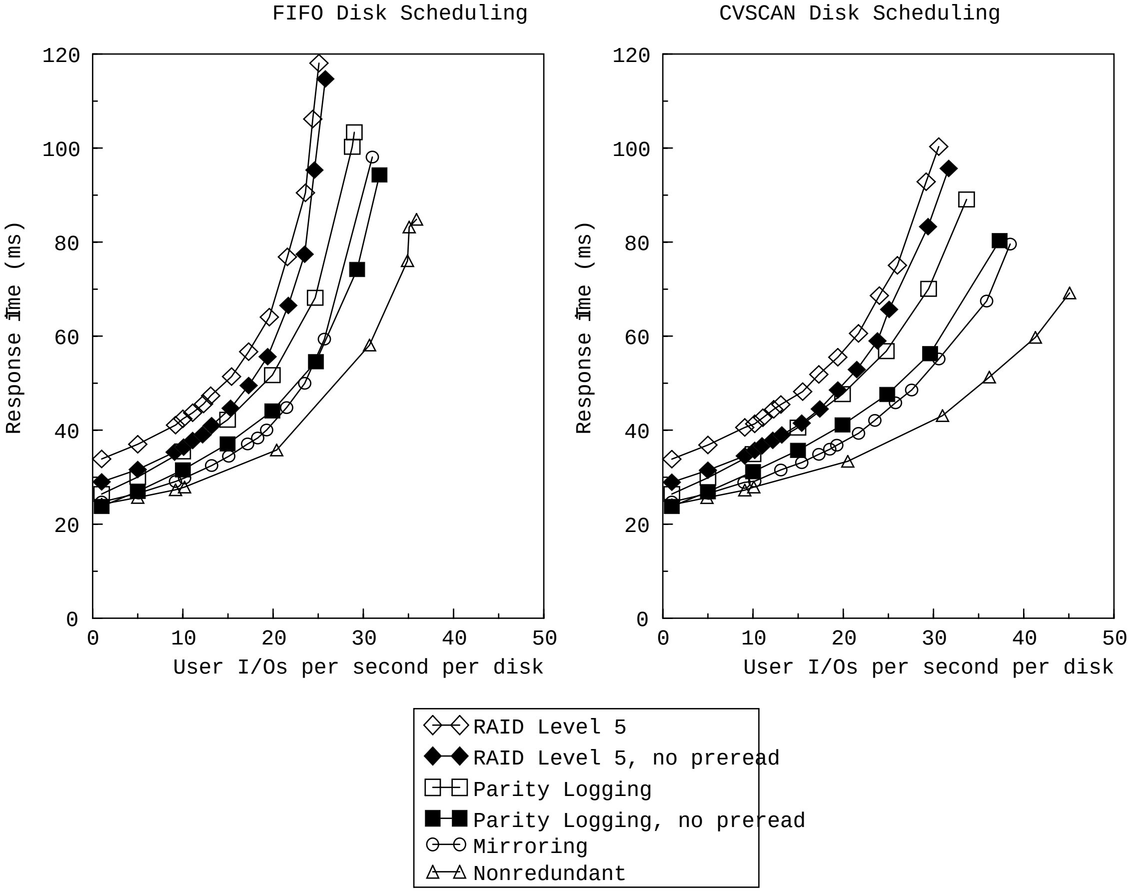


Fig. 23. Airline reservation simulation. Shown above are the results of simulation using the access s
Figure 22. The access distribution is uniform throughout the 22 disk array (Figure 12). For all
size was 24KB, so no access spans more than a single drive. For RAID level 5 and parity logging
the case where all writes are blind, and when the old data for all writes is cached (no prerea
scheduling improves throughput and response of all workloads, mirrored and nonredundant disk ar
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 more realistic OLTP workload. With FIFO disk scheduling, used throughout the rest of this paper, parity logging is always superior to RAID level
and is equivalent to mirroring when data caching of writes is effective[10]. With CVSCAN [Geist87], all configurations deliver higher throughput with lower average response times, but mirror
nonredundant arrays benefit most. Nonetheless, parity logging remains superior to RAID level
comparable to mirroring when data caching of writes is effective.

7. MULTIPLE FAILURE TOLERATING ARRAYS

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

---

[10] Our simulations do not explicitly model a disk or file cache. We consider accesses satisfied in such a cache to not contrib
ute to the disk array workload. Cache write hits are special-cased because the disk access is modified by the av
prior data values.

bit symbol, dependent on a multiple-bit symbol from each of a subset of the data disks, th a non-binary code [Macwilliams77, Gibson92]. Non-binary codes can achieve much lower ch information space overhead in a multiple failure tolerating array. In particular, a variant of a Reed-Solomon code called "P+Q Parity" has been used in disk array products to provide double f tolerance with only two check information disks [ATC90].

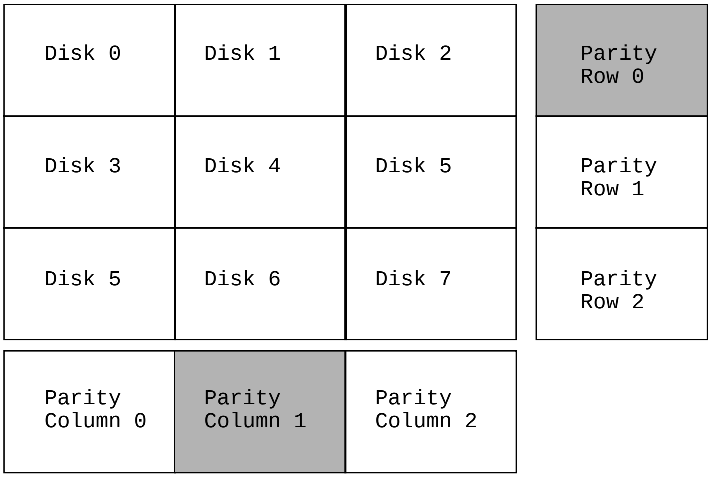


Fig. 24. Two dimensional parity. One disk array organization that achieves double failure tolerance parity. Parity disks hold the parity for the corresponding row or column. In the example above, t holds the parity of disks 0, 3 and 5. Similarly, the parity disk for row 0 holds the parity of disks 0,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

This paper is not concerned with the choice of codes that might be used for f-failure tolerance, except to note that the best of these codes all have one property important to small random write [Gibson89]: each small write updates exactly (f+1) disks — f disks containing check information (generalized parity) and the disk containing the user's data. This check maintenance work, which scales up with the number of failures tolerated, is exactly the work that parity logging handle more efficiently.

Multiple failure tolerating parity logging disk arrays arise as a natural extension of m tolerating variants of RAID 5. As with single failure tolerating parity logging, the un array is augmented with a log. However, to maintain f-failure tolerance, the log itself must be (f−1)-failure tolerant. One way to achieve (f−1)-failure tolerance is to replicate it f times. 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 logg When a region's log buffers fill up, the corresponding parity update records are written once i the f logs. When these logs fill up, one copy of the log is read into the reintegration buf, along with the check information for the region. The updated check information is then rewritten, all truncated, and the logging cycle starts again.

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

The overhead associated with maintaining check information can be divided into two comp preread bandwidth overhead and nonpreread bandwidth overhead. The bandwidth needed to pre the old copy of the user's data is independent of the number of failures to be tolerated. Nonp 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, linear cost for this growing component of check maintenance overhead because all check information accesses (generalized parity) are done efficiently.

Figure 26 shows the maximum rate that small random writes can be completed in zero,

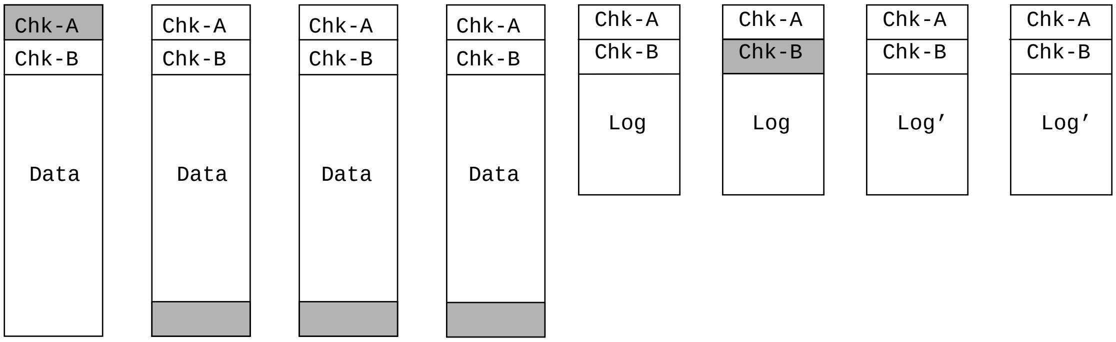


Fig. 25. A parity-logging array that uses a nonbinary code to achieve double-fault tolerance. By using nonbina codes, disk arrays can achieve double failure tolerance with only two disks of check data. Show double fault tolerant parity logging disk array with nonbinary check information. The parity of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

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

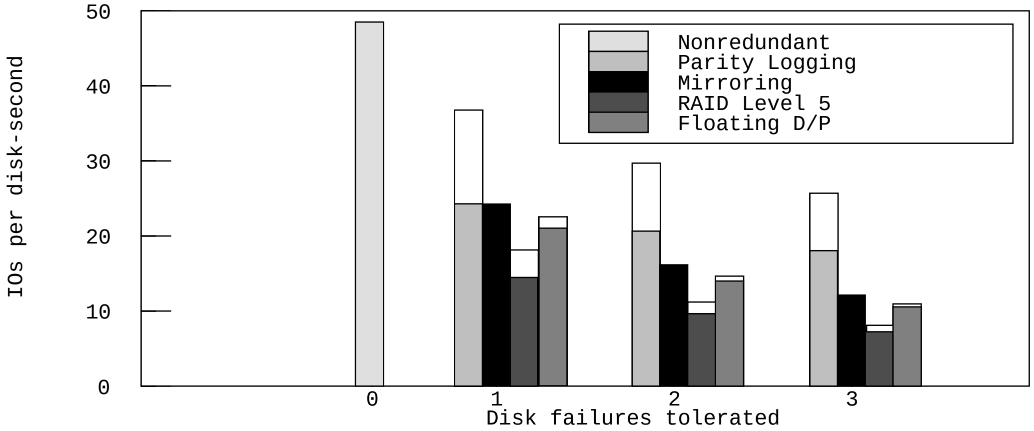


Fig. 26. Performance of multiple failure tolerating arrays. While the performance of all array configurat with the number of failures tolerated, parity logging declines the least, decreasing in performa failure tolerated. The highest performing alternative, mirroring, has a huge disk space overhea disk of user data in the double and triple failure tolerating cases, respectively. The performance of RAID level 5 and data and parity both decline rapidly, achieving less than 10 user writes per second in the triple failu shaded portion of each bar shows the performance when the data to be rewritten is not cached, wh performance when data is cached.

double, and triple failure tolerating arrays using mirroring, RAID level 5, floating data a 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 conf because parity logging has a substantially lower nonpreread overhead. For example, whi failure tolerating parity logging arrays should sustain about 35% of the I/O rate of n 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 up data units associated with a particular parity unit, can easily be serviced more efficientl write operation. Since all data units in the stripe are updated, the new parity can be memory from the new data and written directly to the parity unit. This “large write op avoids the preread of data and parity associated with small writes, improving write perf about a factor of four [Patterson88].

This optimization can not be applied directly to parity logging disk arrays as we have de so far because there may exist outstanding (not yet reintegrated) logged updates for a par 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 st updates when reintegration occurs. This problem can be corrected by placing the new parity instead of writing it directly to disk.[11] Parity placed in the log by a large write operation is marked special "overwrite" record, and the reintegration process, which normally XORs each log rec corresponding parity unit, now distinguishes between a normal "update" log record and overwrite record. Update records are XORed into the accumulating parity unit, while overwri are simply copied in.

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

This new sequentiality constraint potentially lengthens the reintegration time, which, will show, can substantially degrade performance at high loads. In the simplest case, a re sublogs must be in read in the order they were written and merged to produce a update/overwrite im any of the parity is processed. Given sufficient buffer memory for a region's parity and log, full parallelism could be achieved during the log and parity reads, but the application of l would still have to be deferred until these reads complete. At this point, a sequenti reintegration could be performed. However, as long as log buffers are written to sublogs in a ro robin fashion, it is reasonable to assume that parallel sublog reads will return parity re 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 reinte is initially zeroed and marked "non-overwrite". Parity and log for the target region ar parallel. A parity block is applied if the corresponding buffer is marked "non-overwrite," if the buffer is marked "overwrite." If a logged record is an update and the block is "no the record is XORed in, but is buffered until all earlier log records have been processed. an overwrite, the target block is overwritten and marked as "overwritten by record X." A updates that have already been applied should occur after this overwrite are reapplied. O update records preceding X are not applied to a block marked "overwritten by X." As long reads on different sublogs proceeded at nearly the same rate, this algorithm will not co 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 loggi LRAID-X4 organization maintains separate parity and parity-update log disks, and periodical the logged updates to the parity disk. In order to allow writes from the user to occur in p reintegration, they duplicate both the parity and the parity log for a total of four d LRAID-X4 does not distribute parity or log information. Instead of breaking down the log regions to reduce the required storage in the controller, LRAID-X4 sorts buffered 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 upd into runs and written to the second log disk while the first log disk reintegrates its up parity by reading from one parity disk and writing to the other. The reintegration of a full log disk use 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 effi

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

---

[11] An alternative way to correct the problem is to write the new parity directly to disk and place a "cancel" The reintegration process would then discard all previous log entries for the identified parity unit when it detect 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 techni frequently exploited to improve throughput in disk arrays: write buffering, write-twice, location.

Write buffering delays users' write requests in a large disk or file cache to achieve deep 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 s fault-tolerant caches are used.

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

The floating location technique improves the efficiency of writes by eliminating the static of logical disk blocks and fixed locations in the disk array. When a disk block is written, a new location is chosen in a manner that minimizes the disk arm time devoted to the write, and a new ph logical 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 (L which all data is written in a segmented log, and segments are periodically reclaimed collection [Rosenblum91]. Using fault-tolerant caches to delay data writes, this approach writes into long sequential transfers, greatly enhancing write throughput. However, because logically 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 widely. The distorted mirror approach [Solworth91] uses the 100% storage overhead of mirroring to avoid this problem: one copy o is stored in fixed location, while the other copy is maintained in floating storage, achievin throughput while maintaining data sequentiality [Orji93]. However, all 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 distributed log. Analytical models of the peak bandwidth of this scheme and alternative literature were derived and validated by simulation. The proposed technique achieves sub better performance than RAID level 5 disk arrays on workloads emphasizing small random ac When data must be preread before being overwritten (writes miss in the cache), parity loggi performance comparable to floating parity and data without compromising sequential performance or application control of data placement. When the data to be overwritten performance is superior to floating parity and data and mirroring array configuration performance is obtained without the 100% disk storage space overhead of mirroring. The scales to multiple failure tolerating arrays and can be adapted to accommodate the l optimization.

While the parity logging scheme presented in this paper is effective, several optimizati explored. More dynamic assignment of controller memory should allow higher performance 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 [Holland92] merits exploration. Parity declustering provides high performance during degra and reconstruction while parity logging provides high performance during fault-free oper combination of the two should provide a cost-effective system for OLTP environments.

11. 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 earilier copies of this work.

REFERENCES

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

[Cao93] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. The TickerTAIP parallel RAID architecture. *Proceedings of the 20th Annual International Symposium on Computer Architecture*. San Diego CA (May 16-19, 1993). published as special issue of *Computer Architecture News*, ACM, 21, 2 (May 1993), 52-63.

[Chen90] Chen, P. M., and Patterson, D. A. Maximizing performance in a striped disk array. *Proceedings of the 17th Annual International Symposium on Computer Architecture*. (Cat. No. 90CH2887-8), Seattle WA (May 28-31, 1990), IEEE Computer Society Press, Los Alamitos CA, (1990), 322-331.

[DiskTrend94] DISK/TREND, Inc. 1994 DISK/TREND Report: Disk Drive Arrays. 1925 Landings Drive, 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*, 5, 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 data reliability. *Journal of Parallel 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 *SIGPLAN Notices*, ACM, 27, 9 (Sept. 1992), 23-35.

[IBM0661] IBM Corporation. IBM 0661 Disk Drive Product Description, Model 370, First Edition, Low End Storage Produ 504/14-2. IBM Corporation, (1989).

[Menon92] Menon, J., and Kasson, J. Methods for improved update performance of disk arrays. *Proceedings of the Hawaii International Conference on System Sciences*. (Cat. No. 91TH0394-7), 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 and Write Caching. Computer Science Research Report RJ9485(83363), IBM Corporation, (1993).

[Orji93] Orji, C. U., and Solworth, J. A. Doubly distorted mirrors. *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. *Computer*, 21, 2, (Feb. 1988) 23-36.

[Patterson88] Patterson, D., Gibson, G., and Katz, R. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of 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, and A., Dias, D. M. Disk mirroring with alternating deferred updates. *Proceedings of the 19th Conference on Very Large Databases*. Dublin Ireland (Aug. 24-27, 1993). Morgan Kaufmann, Palo Alto CA, (1993), 604-6

[Ramakrishnan92] Ramakrishnan, K. K., Biswas, P., and Karedla, R. Analysis of file I/O traces in commercial computing e ronments. *Proceedings of the 1992 ACM SIGMETRICS Conference on Performance*. 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. *Proceedings of the 13th ACM Symposium on Operating System Principles*, Pacific Grove CA, (Oct. 13-16, 1991) published as special issue *Operating Systems Review*, ACM, 25, 5 (1991), 1-15.

[Schulze89] Schulze, M. E., Gibson. G. A., Katz, R. H., and Patterson, D. A. How reliable is a RAID? *Proceedings of the 1989 IEEE Computer Society International Conference (COMPCON 89)*. (Cat. No. 91TH0394-7), San Francisco CA, (Feb. 27-Mar 3, 1989), IEEE Computer Society Press, Washington DC (1989) 118-123.

[Seltzer90 Seltzer, M., Chen, P., and Ousterhout, J. Disk scheduling revisited. *Proceedings of the Winter 1990 USENIX Conference*. Washington DC, (Jan. 1990) 22-26

[Solworth90] Solworth, J. A. and Orji, C. U. Write-only disk caches. *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. Distorted Mirrors. *Proceedings of the First International Conference on Parallel an Distributed Information Systems*. (Cat. 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 striping. *Proceedings of the 2nd IEEE International Conference on Data Engineering*. IEEE (1986).

[Stodolsky93] Stodolsky, D., Gibson, G., and Holland, M. Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays. *Proceedings of the 20th Annual International Symposium on Computer Architecture*. San Diego CA (May 16-19, 1993). published as special issue of *Computer Architecture News*, ACM, 21, 2 (May 1993), 64-75.

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