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 secondary 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 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 as a single data store, is undergoing rapid growth and has been predicted to exceed 13 billion dollars by 1997 [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 often have 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 in the array. Without this redundancy, large disk arrays have unacceptably low data reliability because of their large number of component disks. For these three reasons, redundant disk arrays, also known as Redundant Arrays of Inexpensive[1] Disks (RAID), are strong candidates for nearly all on-line secondary storage systems [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 or more disk sectors.

RAID level 5 arrays exploit the low cost of parity encoding to provide high data reliability [Gibson93]. Data is striped over all disks so that large files can be fetched with high bandwidth. By distributing the parity, many small random blocks can also be accessed in parallel without hot spots on any disk.

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

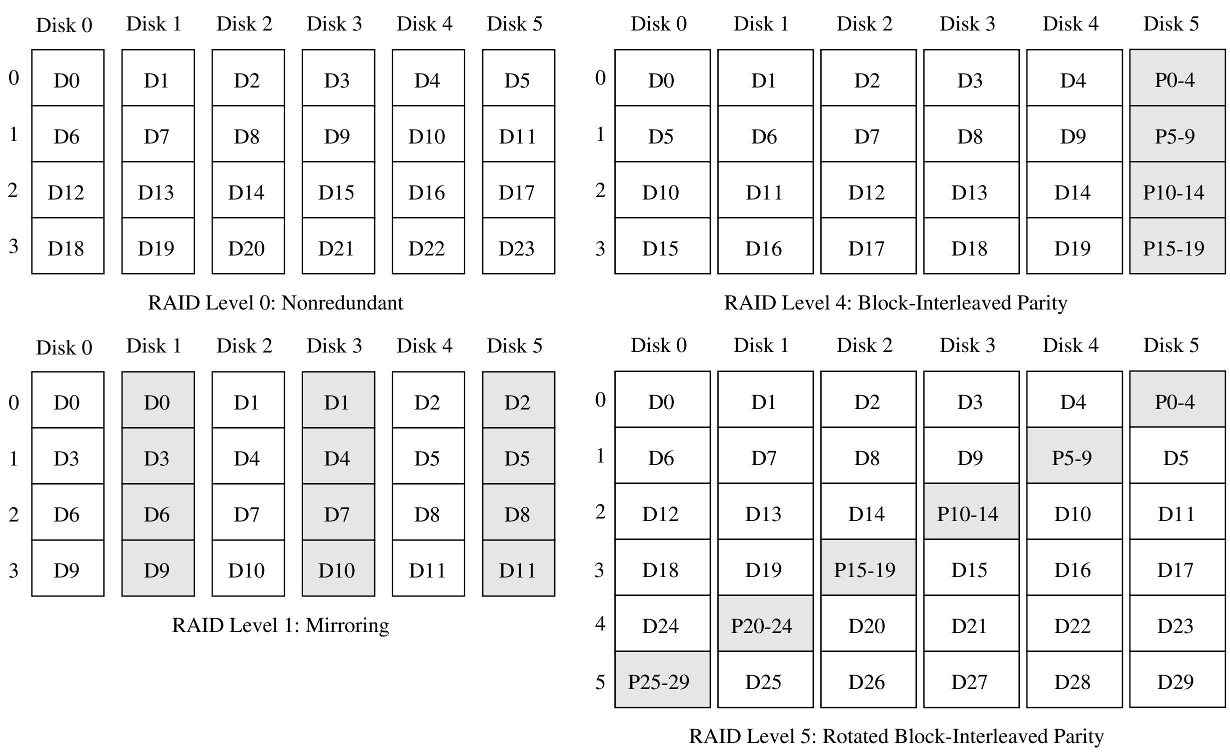

Fig. 1. ***Data Layout in RAID Levels 0, 1, 4 and 5.*** *This figure shows the first few units on each disk in each of the RAID levels. "D" represents a block, or* **unit***, of user data (of unspecified size, but some multiple of one sector) and "Px-y" a parity unit computed over user data units x through y. The numbers on the left indicate the offset into the raw disk, expressed in data units. Shaded blocks represent redundant information, and non-shaded blocks represent user data. Level 0 is nonredundant and does not tolerate faults. Level 1 is simple mirroring, in which two copies of each data unit are maintained. Levels 4 and 5 exploit the fact that failed disks are self-identifying, achieving fault tolerance using a simple parity (exclusive-or) code, lowering the capacity overhead to only one disk out of six in this example. Levels 4 and 5 differ only in the placement of the parity. In level 5, the parity blocks rotate through the array rather than being concentrated on a single disk, avoiding a parity 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 industry standard benchmark for OLTP systems stressing update-intensive database services [TPCA89]. It models the computer processing for customer withdrawals and deposits at a bank. The primary metric for TPC benchmarks is transactions per second (TPS). Systems are required to complete 90% of their transactions in under 2 seconds and to meet the scaling constraints listed above. Customer account records are selected at random from the local branch 85% of the time, and from a different branch 15% of the time. Because history record writes are delayed and grouped into large sequential writes and teller and branch records 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 arises because a small write request may require the old value of the user's targeted data be read (we call this a *preread*), overwriting this with new user data, prereading the old value of the corresponding parity, 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 factor of 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 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 OLTP and nearly the worst possible for RAID level 5, where a single read-modify-write of an account record will require five disk accesses. The same operation would require three accesses on mirrored disks, and only 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 journalling events to transform small random accesses into large sequential accesses. Section 2 of this paper develops 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 overhead 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 implementations of parity logging and alternative organizations, and contrasts their performance on workloads of small random writes and an OLTP workload. Section 7 analyzes extensions to multiple-failure tolerating arrays. Section 8 discusses how the large write optimization can be accomodated in a parity logging disk array. Section 9 reviews related work. Section 10 closes with a few comments on future work in 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. A parity logging disk array batches small changes to parity into large accesses that are much more efficient. Our model is introduced in terms of a simple, but impractical RAID level 4 scheme, then refined to the 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

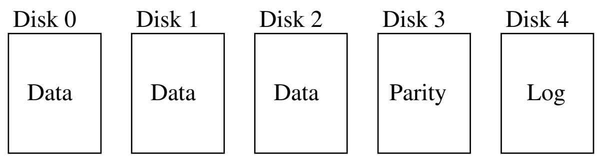

Fig. 4. ***Basic Parity Logging Model*** .

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. Figure 3 shows the relative bandwidths of random block, track and cylinder accesses for a modern small-diameter disk [IBM0661]. This figure largely bears out the lore 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)	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	~9	C_P	Cylinders of parity per region	~9
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 presented in terms of the parameters list above. The first table presents common disk parameters and the second, parameters specific to parity logging. The first and fourth columns in each table show the symbol used in the text; the second and fifth column describe what a symbol denotes; and the third and last column show the default value of the parameter as used in this text. Note "ms" stands 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 array is 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 XORing the old 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), they are written 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 into memory using sequential cylinder accesses. The logged parity update images are applied to the in-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.

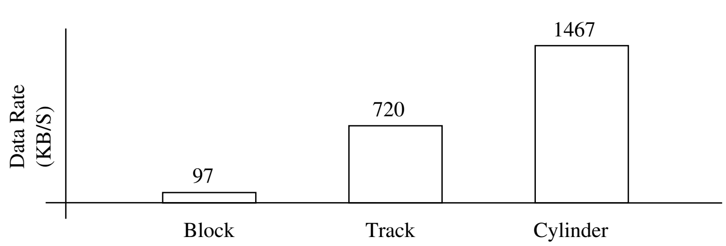

Fig. 3. ***Peak I/O Bandwidth.*** *This figure shows the sustainted data rate in kilobytes per second that can be read from or written to an IBM 0661 drive using random one block (2KB), one track (24 KB), and one cylinder (336KB) accesses (see Figure 12 for disk parameters).*

Because only parity updates (not data changes) are deferred, this scheme preserves single failure tolerance [2]. If a data disk fails, the log disk (and any buffered parity updates) are first applied to 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 its data onto the surviving parity or log disk. The failed drive is then replaced with a new empty log disk. If the controller fails, its buffered parity updates are lost, but, after the controller has been repaired or 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 maintenance 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 preread of the corresponding data unit, which introduces an overhead of one block (data unit) access per write. In addition, each user write to a data unit consumes buffer memory equal to the size of the unit, and so a track's worth (D) of small (unit-sized) writes issued to the array causes one track write to the log disk to occur. Next, a disk's worth (TVD) 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 and log 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 accesses are D times larger than random small writes but about 10 times more efficient and cylinder accesses are 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 maintenance by about a factor of two. [3]

In many cases, it may be possible to avoid the preread of the user data. For example, in the TPC benchmark (Figure 2), the update of a customer account record is a read-modify-write operation; an account record is read, modified in memory, then written back to disk. In these cases, the old data value is usually known (cached) at the time of the write and the preread of the data may be skipped [Menon93]. Under these conditions, the overhead for RAID levels 4 or 5 is just two random block 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 consumed by parity 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. Figure 6(a) shows that this limitation can be overcome by dividing the array into manageably-sized regions. Each region is a miniature replica of the array proposed above. Small user writes for a particular region are

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

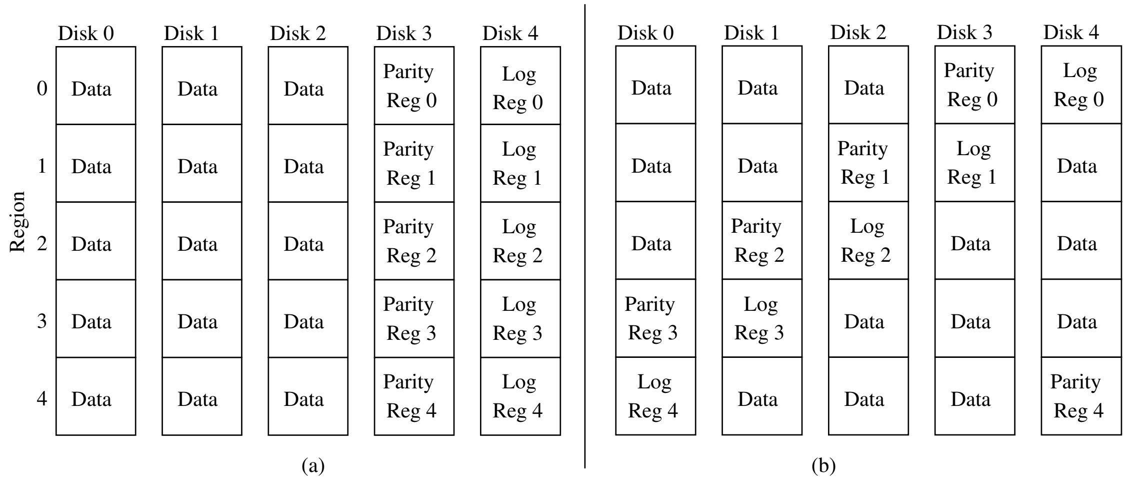

Fig. 6. ***Parity Logging Regions*** .

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 parity reintegration from the size of a disk to a manageable fraction of a disk. Section 2.4 shows that optimal 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) tracks 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 single-log scheme are replaced by random track (or multi-track) writes in the multiple-region layout. While random track writes are less efficient than sequential track writes, Section 3 will show that this more practical implementation still has dramatically lower parity maintenance overhead than RAID level 4 or 5.

2.2. Striping Log and Parity for Parallelism

As in the RAID level 4 case, the log and parity disks may become performance bottlenecks if there are many disks in the array. In particular, the maximum aggregate bandwidth for log accesses is just the bandwidth of 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 log 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 latency of 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 large; there 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 multiple disks. Figure 8 shows a parity logging array with the log for each region striped across two disks. Since parity 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 parity to be lost, 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 the number of disks on which data for a particular region may be placed. If, for example, the log is striped over 3 disks, the 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. The 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 $N-L$ disks:

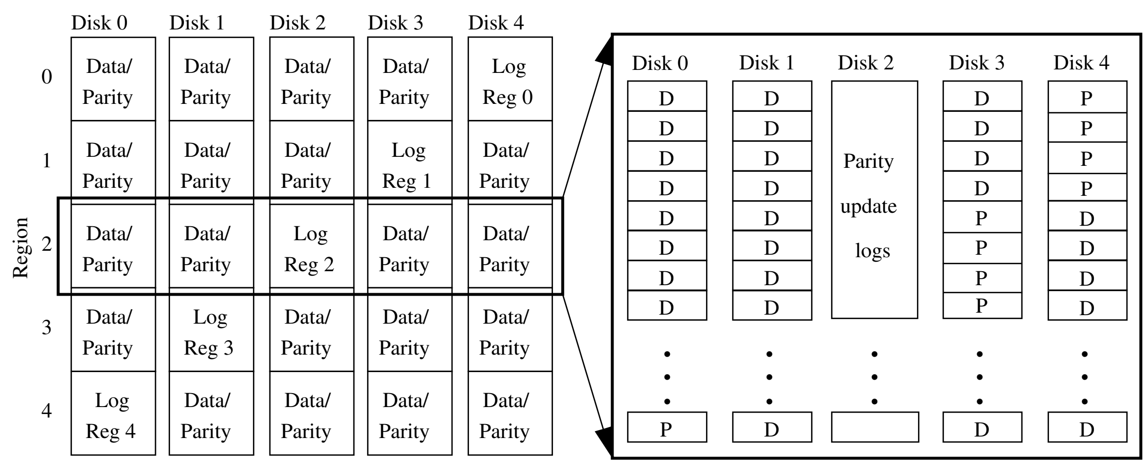

Fig. 7. ***Block Parity Striping****. The inset shows a detailed layout of a sample region.*

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

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 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, L, increases. Figure 9 shows the disk space overhead for different degrees of log striping for an array of 22 disks. Section 6 will show, however, 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 for log (C_L cylinders) and

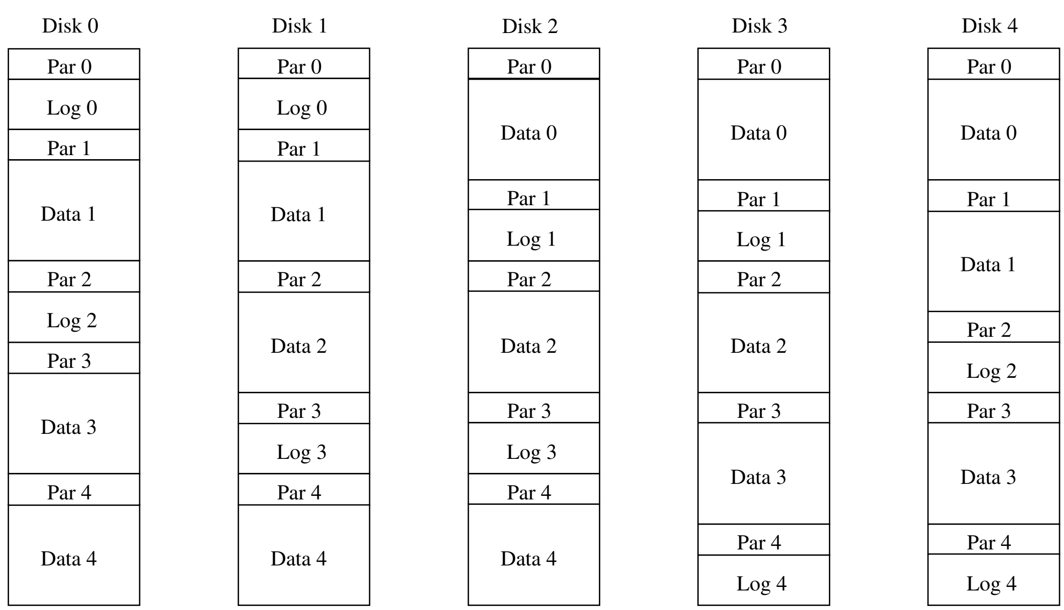

Fig. 8. ***Distributed Parity Logs****.*

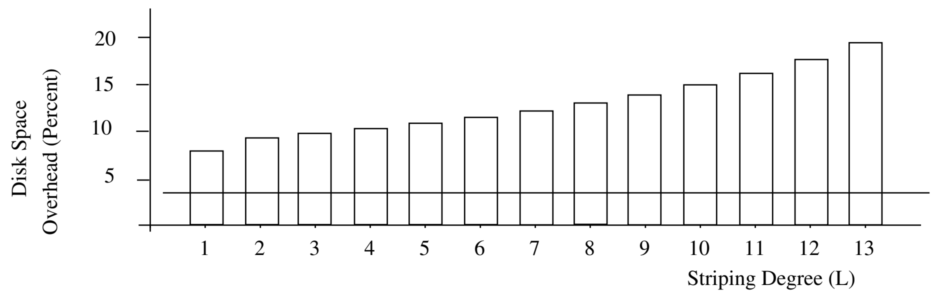

Fig. 9. ***Disk Storage Overheads****. The horizontal line shows the capacity overhead of a RAID level 5 configuration of the same array.*

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

Let A be the ratio of total log space to total parity space ($A = C_L / C_P$) in each region. The disk space overhead then becomes

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

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 the parity (C_P cylinders), reading the log ($A\,C_P$ 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. Applying this to an 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 parity accesses 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 user write and parity logging does ($3/20 + 1/10A$). Applying this cached workload to our 22 disk array with logs striped over two disks does not change the space overheads. However, in this cached case, doubling log size reduces parity maintenance work from 12.5% to 10% of RAID level 5 while halving log size 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 using large, efficient transfers, requires expensive controller memory buffers. This buffer memory is used in two ways. First, each region delays the most recent parity update images until efficient log-append accesses 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, whenever the 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 cylinders per 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 of the 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 track per region ($K = 1$) requires about 5.6 MB of buffer, costing 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 require that the 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 overall disk space may be wasted. We have found that if the number of regions, B, is 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's total 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 when the 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 parity accesses, but because these positioning overheads are small relative to track access times, parity logging performance is only slightly affected (3% for our experiments).

A more significant performance degradation results if small user writes are blocked during the 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 blocked if the entire buffer pool is full of parity updates images that have not yet been appended to their appropriate logs.

2.5. Summary

In summary, parity logging buffers parity updates until they can be written to a log efficiently. 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 disks. This parity logging scheme reduces the extra work done by RAID level 5 arrays for small random writes to little more than is done in the much more expensive, traditional mirrored approach even if write

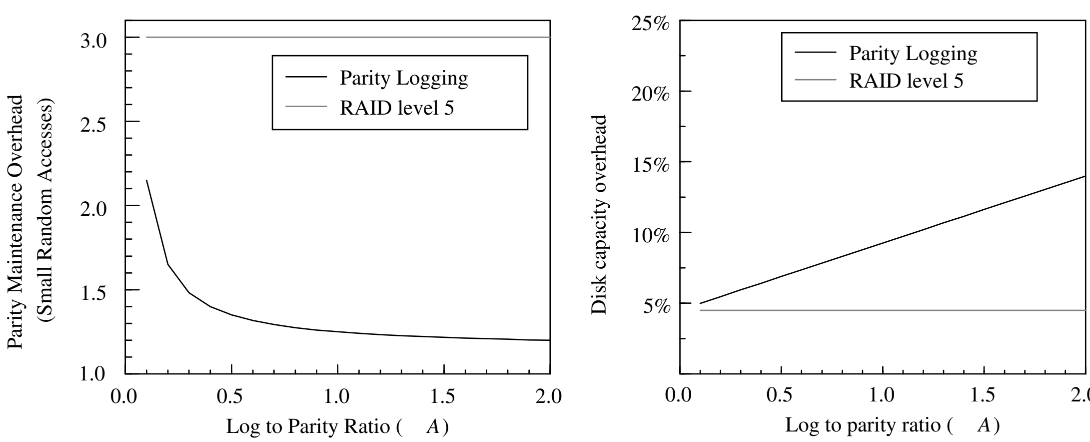

Fig. 10. ***Log Length and Efficiency.***

caching is ineffective.

3. MODELING PARITY LOGGING

In this section we present a utilization-based analytical model of a parity logging redundant disk array. This model predicts saturated array performance in terms of achieved disk utilization, disk 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, then updates the data.[4] Defining S as the average seek time, R as 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_{rmw}, is

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

disk seconds, on average.

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

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

disk seconds.

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

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

assuming 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 DTC_L small user writes, one region of logged parity must be reintegrated. Consider the case of an array that does not stripe its log (Figure 7). The reintegration consists of three steps: a sequential read of C_L cylinders from the log, a striped read of the parity from $N-1$ 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

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

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 track is buffered (K=1). However, the impact of zero-latency write support on parity logging is small (under 3%), because the track-sized log writes are only a small contributor to parity logging's overhead (Figure 11).

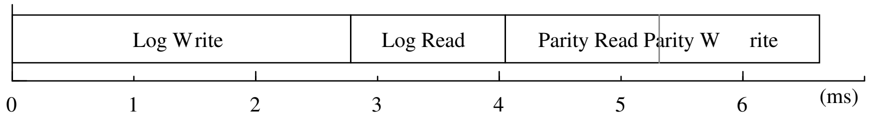

Fig. 11. ***Parity Logging Overheads****. The amortized overhead cost of parity and log accesses done in our example parity logging array is shown above. The log writes contribute approximately 40% of the overhead (2.78 milliseconds), while the cylinder rate log reads, parity reads and parity writes each contribute about 20% (1.29 milliseconds). For comparison, the parity accesses done by RAID level 5 cost nearly 35 milliseconds per small write.*

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

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

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

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

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 regions 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 (over L disks) read of the log, and a striped (over N disks) read and write of the parity. Each of the L accesses in the striped log read costs

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

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 representative of OLTP workloads, while a 100% write ratio emphasizes the performance differences of the various array organizations. Assuming disks have independent support hardware, disk failures will be independent, allowing a parity group to be single fault tolerant [Gibson93]. Disk parameters are modeled on the IBM Lightning drive [IBM0661]. Note that the* dist *term in the seek time model is the number of cylinders traversed, excluding the destination. As is commonly done in SCSI disks, the track skew is chosen to equal the head switch time, optimizing data layout for sequential multitrack access.*

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

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 write by $L(S+R-M)/DTC_L$ 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, disk storage for small write workloads. The most notable of these is floating data and parity [Menon92]. This section reviews and estimates the performance of four designs: nonredundant disk arrays (RAID level 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 which 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 update both 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

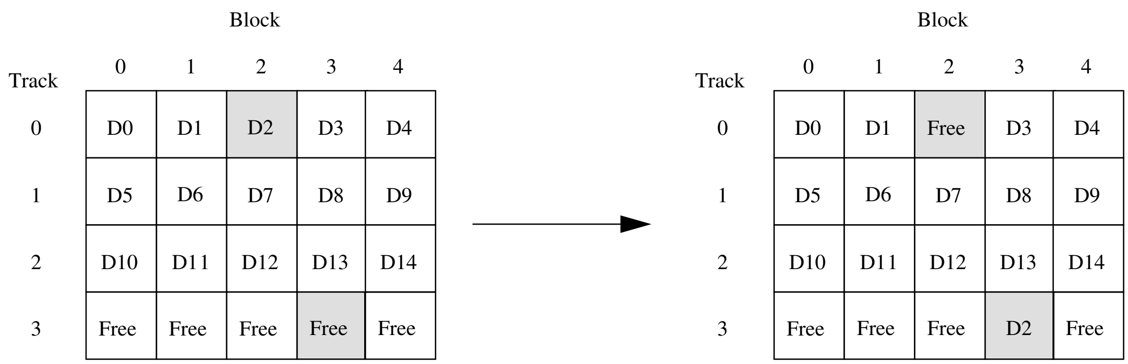

Fig. 13. ***Floating Data/Parity****. This figure shows the movement of data within a cylinder caused by a write in a floating data and parity array. Each grid represents one cylinder of four tracks, with five blocks per track. When updating block D2, the controller searches for a free block within the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

redundant data. As in the RAID level 0 case, controllers for mirrored disk arrays do not require long-term buffer memory.

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

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

disk seconds for a total disk busy time of $2S + (6 + 4/\mathrm{D})R$. 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 cylinders that contain either only data or parity. As illustrated in Figure 13, by maintaining a single track of empty space per cylinder, floating data and parity effectively eliminates the extra rotational delay of RAID level 5 read-rotate-write accesses. Instead of updating the data or parity in place, a floating data and 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 is 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 busy time for 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 $S + (1 + 5.52/\mathrm{D})R + H$. 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 storage 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 and parity.[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.

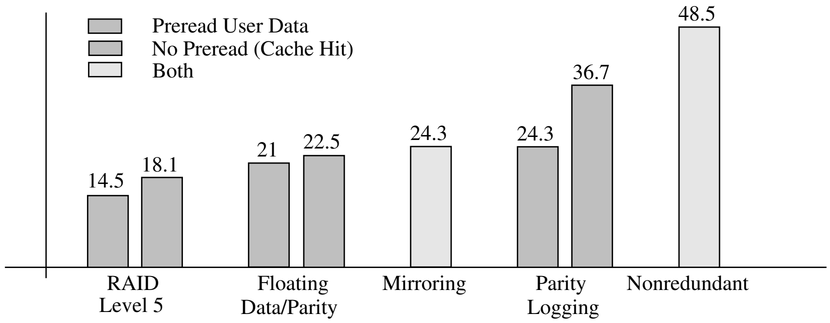

Fig. 14. ***Model Estimates****. User writes per second per disk as predicted by the bandwidth models of Sections 3 and 4. These predictions assume 100% disk utilization, FIFO disk arm scheduling, and an unbounded number of requestors. RAID level 5 and parity logging disk arrays both benefit substantially from not having to preread user data. Floating data and parity substantially reduces the overhead of the user preread and therefore achieves less benefit from its elimination. Mirroring and nonredundant disk arrays do not need to preread user data. The parity logging estimates assume the log is not striped.*

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 bits (164 KB) 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 relative to 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, two consecutive data units may end up at the same rotational position on two different tracks, requiring a complete 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, on average, by $(T-1)/T$.

5. ANALYSIS

Figure 14 compares these models' estimates for maximum throughput of the example array based on 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 and parity will both substantially improve on RAID level 5, approaching the performance of mirroring. Varying the model's 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. This section describes the effects of these parameters and the effects of log striping degree on array load balance.

Of the model's 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 four more times than mirroring. If the transfer time of a unit is small, parity logging will be efficient. Figure 15 shows the relative performance when data caching is ineffective (i.e, a preread is required) of parity logging, mirroring, and RAID level 5 for different values of numbers of data unit per track in for our example array. The performance of mirroring exceeds that of parity logging with 13 or fewer data units per track ($D \le 13$), and RAID level 5 performance exceeds that of parity logging with the unlikely case 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 logging, RAID level 5 and mirroring relative to RAID level 0 as this ratio changes. The performance of mirroring

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

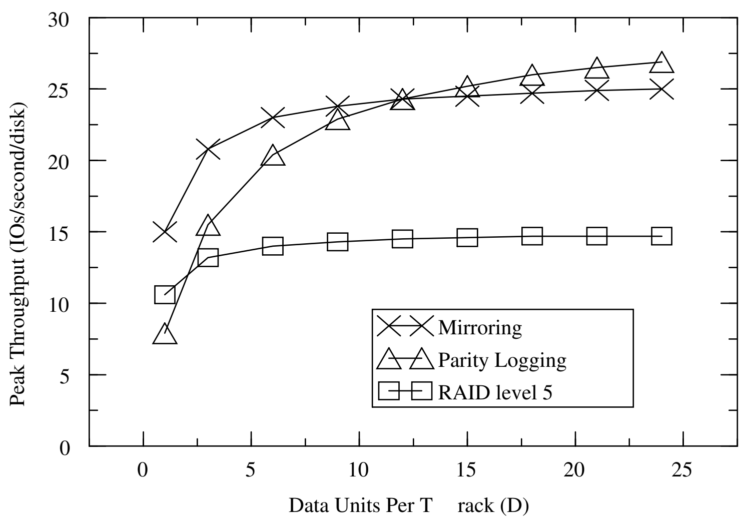

Fig. 15. ***Effects of track size on throughput.*** *The performance of parity logging is highly sensitive to the number of data units per track. The figure above shows the performance in the example 22 disk array of mirroring, parity logging, and RAID level 5 on a workload of 100% blind small writes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

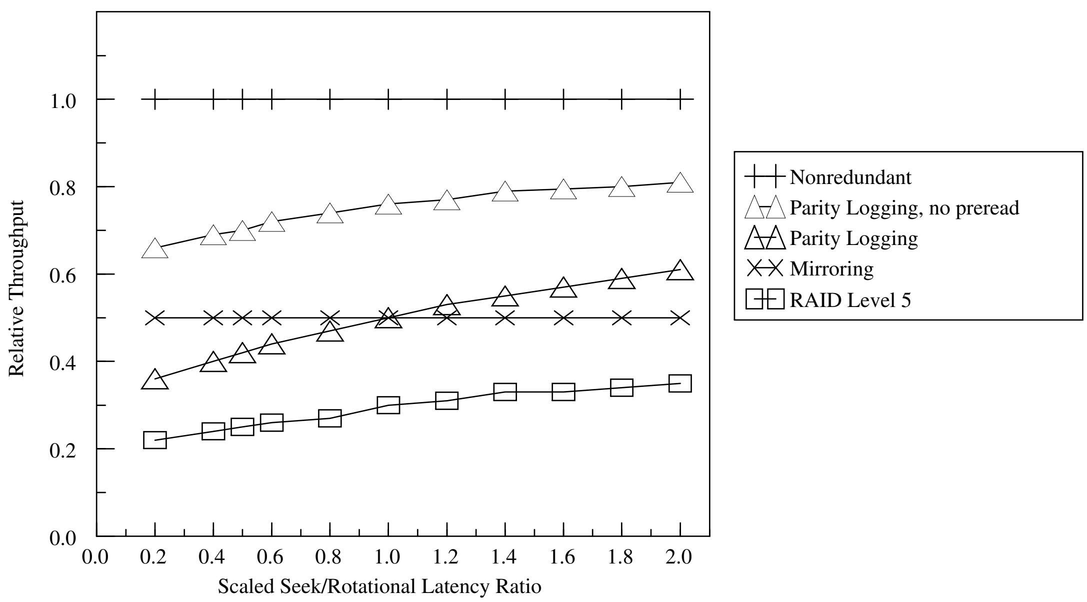

Fig. 16. ***Peak throughput, normalized to nonredundant array performance, as a function of the ratio of average seek time to rotational latency****. Altering the ratio of average seek time (S) to the rotational latency (R) of the disk changes the relative performance of mirroring, floating data and parity, nonredundant and parity logging disk arrays. Shown above is the relative performance of these approaches on the example 22 disk array (Figure 12) as the average seek time for the drive is varied. The average seek time is varied from 20% of the Lightning average seek to twice that of the Lightning average seek. This parameter range models a large spectrum of drives, from those with very fast positioning to Lightning-like drives spinning at 7200 RPM. The X-axis has been linearly scaled so that 1.0 corresponds to the ratio of average seek time to rotational latency of 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 hurts 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 relative advantage of parity logging over RAID level 5 decreases as the seek time to rotational latency ratio decreases. This ratio, however, is 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 worse case, all user I/O is concentrated within one region. Choosing an appropriate data stripe unit [Chen90] will balance the 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 performance will fall short of Figure 14.

The log, parity and data traffic can be balanced by determining the appropriate degree of log striping, 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 and a full read and full write of the parity for that region to effect parity reintegration. Parity reads and writes are spread out over all disks, so a uniform load is maintained if the work per data disk equals the work per sublog disk. That is,

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

where t_{Ktrack} (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 the number of disks in the array, and t_z 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_{rmw} (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 equation in L whose solution is

$$L = \frac{N}{1 + (KDt_z)/(t_{Ktrack} + 2KR)} \tag{11}$$

Using this approximation and the disk array parameters from Figure 12, one obtains $L \approx 0.16N$ for blind writes (when $t_z = t_{rmw}$) 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 two to four sublogs per region.

6. SIMULATION

To validate the analytic models presented in Sections 3 and 4 and to explore response time for these arrays, we simulated the example array described in Figure 12 under five different configurations: 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 simulator derived from the Sprite operating system disk array driver [Ousterhout88], which was extended with implementations of parity logging and floating data and parity.

In each simulation, a request stream was generated by 66 user processes, an average of three 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 time has an exponential distribution, but the mean is dynamically adjusted until the desired system throughput is achieved. If the disk array is unable to sustain the offered load, think time is driven to zero. Simulations were run until the 95% confidence interval of the response time became less than 5% 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 [8] and response time variance as the degree of log

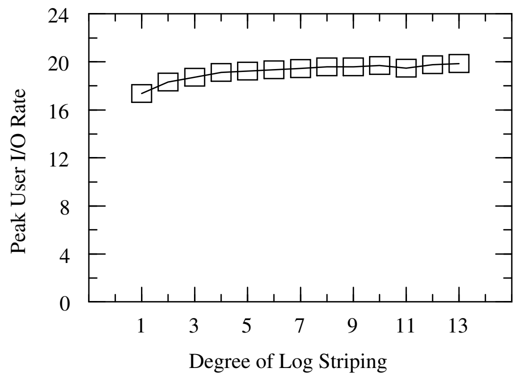

Figure 17(a): Peak user I/Os

Fig. 17. ***Parity Log Striping*** *. Figures 17(a) and (b) show the achieved user I/Os per disk per second, average user response time, and the standard deviation of the response time under peak load for various degrees of parity log striping. All metrics improve substantially as the striping degree is increased from one (no striping) to four . The difference in performance between striping over 4 to 13 disks is slight, indicating the robustness of the technique.*

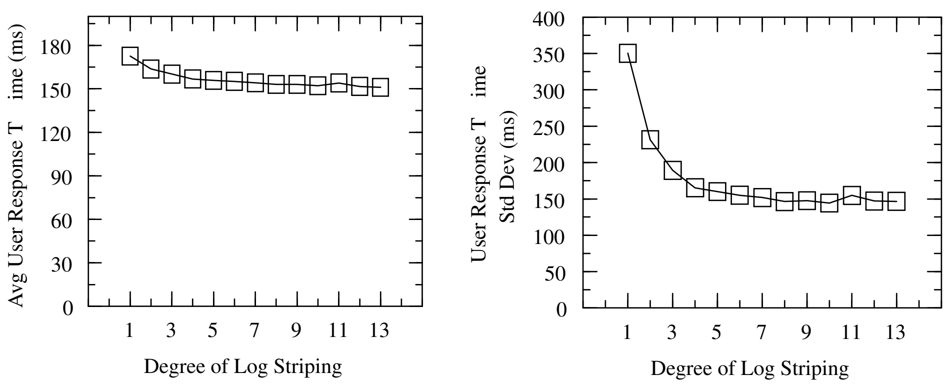

Fig. 17(b): Response time and response time standard deviation at peak load

striping (L) is varied from one (unstriped) to thirteen. As predicted in Section 5, when the log is 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 processes issuing blocking writes queue behind very long sublog read accesses. Figure 18 shows sublog read times for low degrees of log striping. While these long accesses are efficient, they completely tie up a disk for seconds at a time. During this period, any access to the disks involved in the striped log read will block, reducing the effective concurrency in the system. This concurrency reduction causes other disks in the array to become idle until the log read completes, reducing peak throughput and utilization. This convoy effect also has a substantial impact on response time; requests that block behind these long read requests will have very long response times, leading to an increase in both average response time and response time variance. Fortunately , a modest degree of striping eliminates the convoy effect. Figure 17 shows that striping the log over six disks achieves most of the available throughput without greatly 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 Section 4' s alternative organizations: 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) assume that the user data must be preread (data cache miss), and Figure 19(c) presents the corresponding data for the no preread (data cache hit) case.

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

8. The simulations reported herein consider a user write in a parity logging array complete when the user data is on disk and the 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.

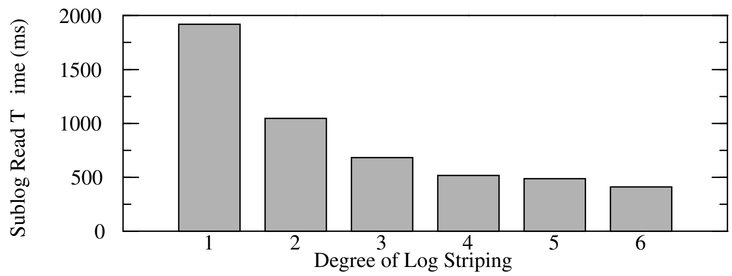

Fig. 18. ***Sublog Read Times*** *.*

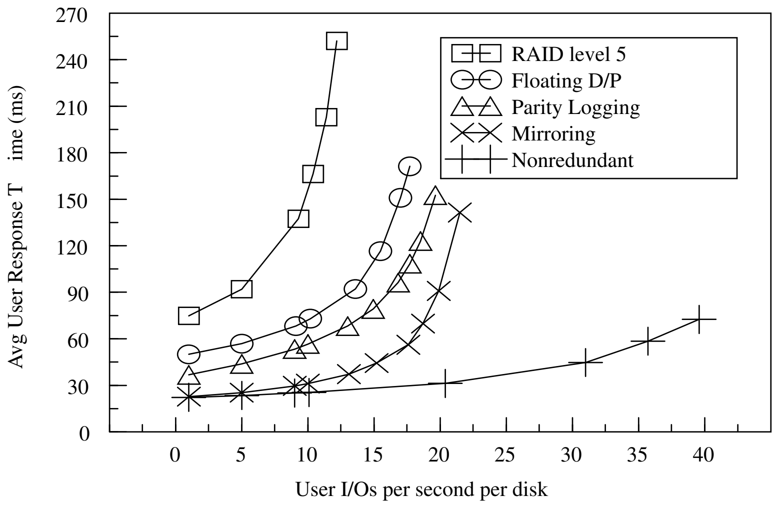

Figure 19(a): Response times

Fig. 19. ***User Response Times and Disk Utilization*** *. Figures 19(a)-(c) present the average user response times and response time standard deviations as a function of the number of small random writes achieved per disk per second. Figures 19(a) and (b) present the results when the user data must be preread, while the results in Figure 19(c) assume the user data was cached, making the preread of the user data unnecessary . In addition to reducing the amount of I/O required, cached user data allows the user write and parity update to occur concurrently , significantly reducing response time for RAID level 5 and floating data and parity . The reported times are in milliseconds. The response time standard deviation for the no preread case is essentially identical to Figure 19(b).*

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

Figure 19(c) shows the performance of all configurations when data cache hits eliminate the need for prereads. As expected, this has no effect on mirrored or nonredundant systems, but improves the performance of the other three configurations. RAID level 5 benefits substantially from eliminating the 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 data and parity achieves a lesser benefit from elimination of the preread because its preread overhead is much less. Response time does drop, however , because of the ability to issue user write and parity update accesses simultaneously . The response time of parity logging improves by a full rotational delay because of the elimination of the preread rotate, providing an unloaded response time comparable to a

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

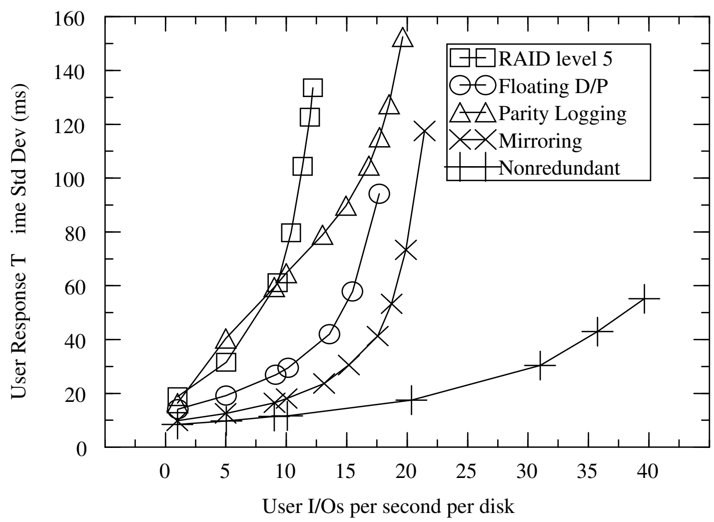

Figure 19(b): Response time standard deviation

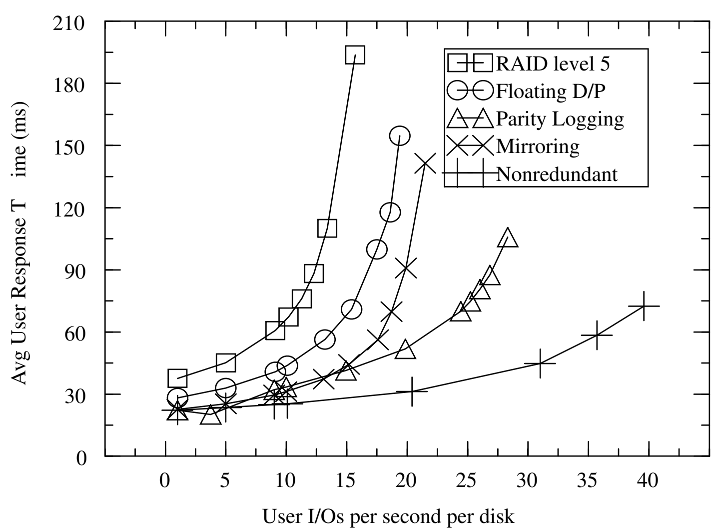

Fig. 19(c): Response times without prereads

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

The variance in user response time, however , is larger with parity logging than with mirroring or floating data and parity , although it is not as large as with RAID level 5. This results from the basic structure of parity logging: most accesses are fast because inefficient work is delayed. However , some accesses see long response times as delayed work is (efficiently) completed. W ith this higher variance in mind, we conclude that the response time estimates in Figure 19 show that parity logging is a viable, 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 throughputs) of Figure 19(a) and (c). A direct comparison, however, will 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 utilizations at peak 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 floating data and parity, when user data is not cached, and parity logging and nonredundant disk arrays, regardless of caching, present only one disk access request at a time per process. Mirroring and the data-cached cases for RAID level 5 and floating data and parity keep the array busier because each user write issues two concurrent disk accesses. Figure 21 shows that, when these difference are accounted for by scaling 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 100% small (2KB) random writes. This section examines a mixed workload, defined in Figure 22, modeled on statistics taken from an airline reservation system [Ramakrishnan92]. With this more general workload, the results of the earlier sections are modified by two important effects: reads and medium to large writes. The issues encountered in extending floating data and parity to handle variable sized access are beyond the scope of this paper and this technique is omitted from this section. Among the 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 differences between configurations. Writes that are not small, however, will hurt the performance of parity logging as is discussed in Section 5.

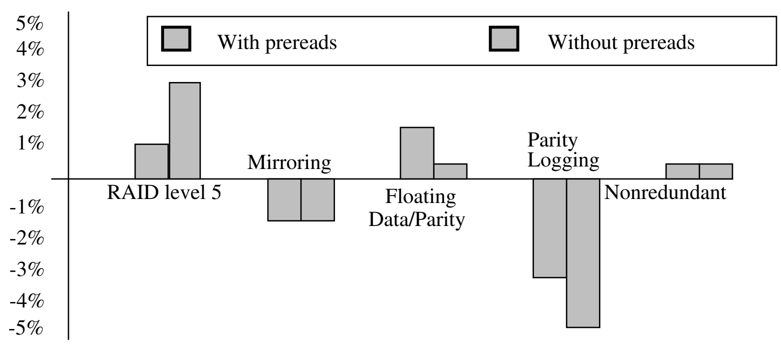

Fig. 21. ***Model errors****. This figure shows the percent error between the models of sections 3 and 4 and the simulations of Section 6. The model predictions have been scaled by the achieved disk utilizations of Figure 20. In all cases, the disagreement between the simulation and the models is less than 5 percent. Note that the 95% confidence interval on the 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 general statistics from an airline reservation system [Ramakrishnan92]. This workload is reported as approximately 82% reads, a mean read size of 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 smaller, 1.5 KB. Locality of reference and overwrite percentages were not reported. All accesses are assumed to occur on their natural boundaries.*

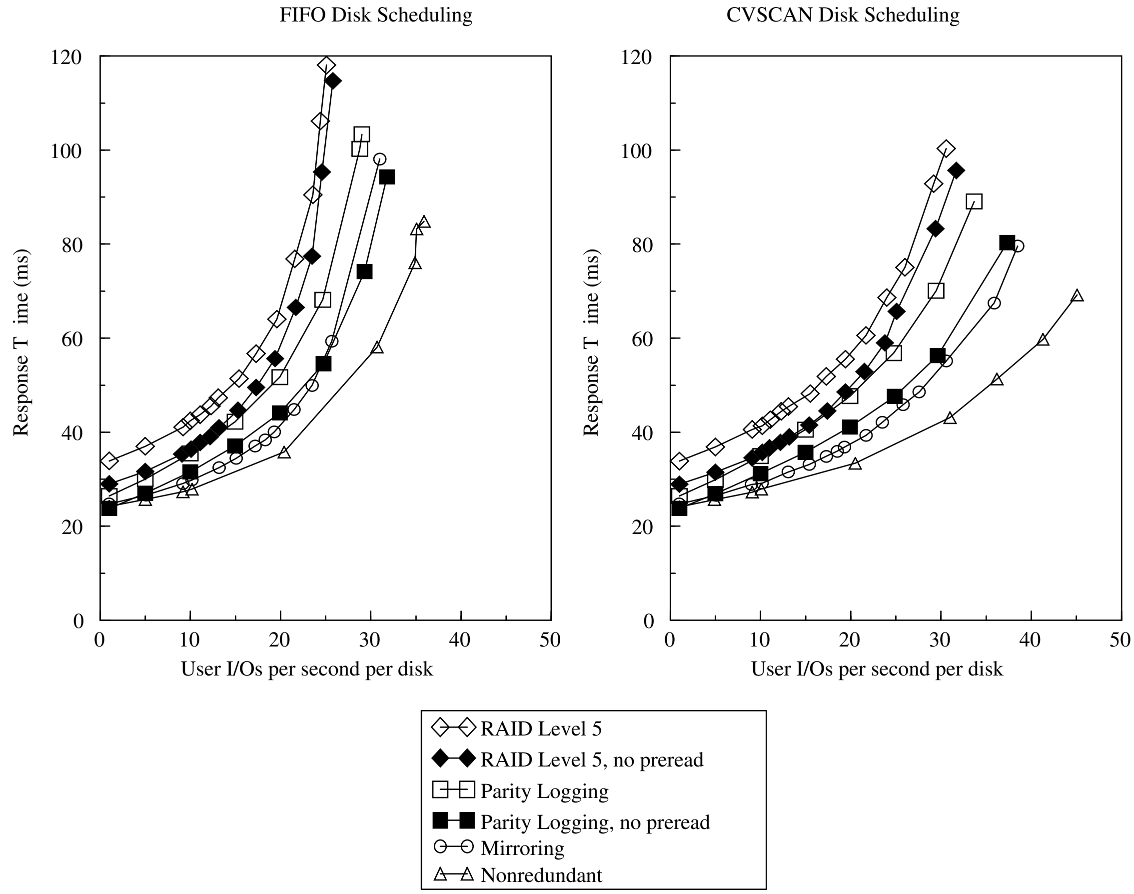

Fig. 23. ***Airline reservation simulation.*** *Shown above are the results of simulation using the access size distribution of Figure 22. The access distribution is uniform throughout the 22 disk array (Figure 12). For all configurations, the data unit size was 24KB, so no access spans more than a single drive. For RAID level 5 and parity logging, results are shown both for the case where all writes are blind, and when the old data for all writes is cached (no preread). While CVSCAN [Geist87] scheduling improves throughput and response of all workloads, mirrored and nonredundant disk arrays improve the most, 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 — nonredundant, 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 5 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 mirroring and nonredundant arrays benefit most. Nonetheless, parity logging remains superior to RAID level 5 and is 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 failure tolerating arrays. Multiple failure tolerance provides much longer mean time to data loss and greater tolerance for bad blocks discovered during reconstruction [Gibson92]. Using codes more powerful than parity, RAID level 5 and its variants can all be extended to tolerate *f* concurrent failures. Figure 24 gives an example of one of the more easily-understood double failure tolerant disk array organizations. This two dimensional parity and the more familiar one dimensional parity used in the rest of this paper are called *binary codes* because a particular bit of the parity depends on exactly one bit from each of a subset of the data disks. If, instead, generalized parity (check information) is computed as a multiple-

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

bit symbol, dependent on a multiple-bit symbol from each of a subset of the data disks, then the code is a *non-binary code* [Macwilliams77, Gibson92]. Non-binary codes can achieve much lower check 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 failure tolerance with only two check information disks [ATC90].

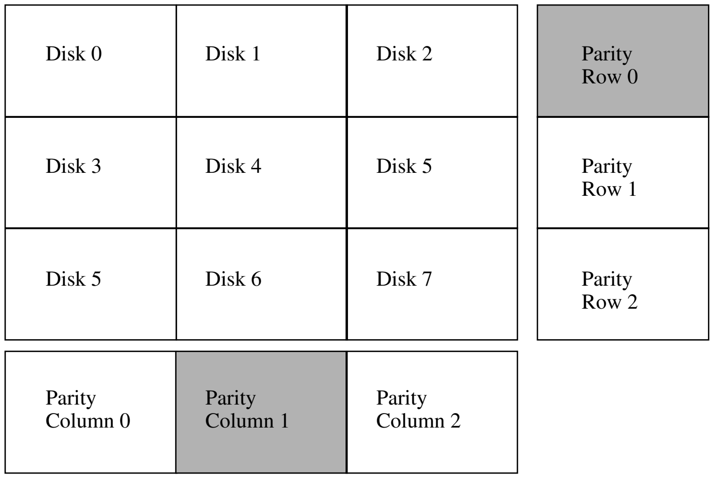

Fig. 24. ***Two dimensional parity****. One disk array organization that achieves double failure tolerance is two dimensional parity. Parity disks hold the parity for the corresponding row or column. In the example above, the parity disk for column 0 holds the parity of disks 0, 3 and 5. Similarly, the parity disk for row 0 holds the parity of disks 0, 1 and 2. Whenever a unit in a data disk is written, the corresponding units in both row and column parity disks are also updated. Thus a write to disk 1, 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 for f-failure tolerance, except to note that the best of these codes all have one property important to small random write performance [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 is designed to handle 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 underlying disk 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 the log f times. Figure 25 shows one region of a double-fault tolerant parity logging disk array based on a nonbinary code such as "P+Q Parity."

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

Mirroring and floating data and parity also extend to multiple failure tolerance in straightforward manner. Mirroring becomes f-copy shadowing [Bitton88]. Floating data and parity becomes floating 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 components: preread bandwidth overhead and nonpreread bandwidth overhead. The bandwidth needed to preread the old copy of the user's 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 linearly with the number of failures to be tolerated. Parity logging has the smallest cost for this latter, linearly growing component of check maintenance overhead because all check information accesses (log and generalized parity) are done efficiently.

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

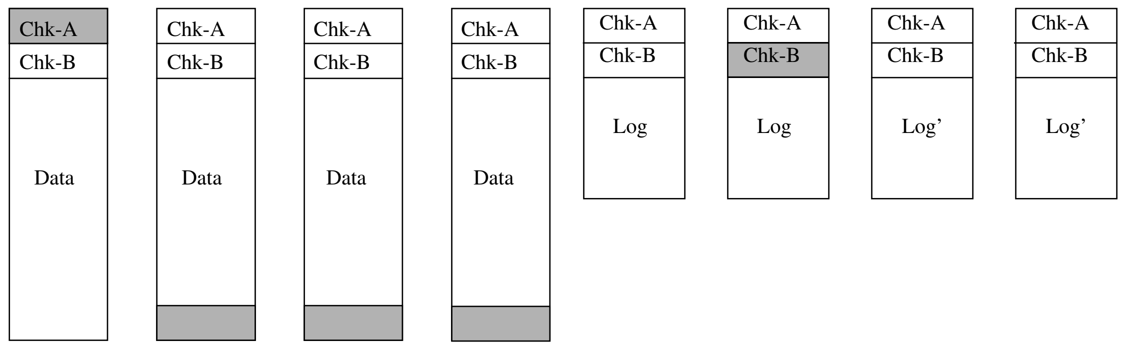

Fig. 25. ***A parity-logging array that uses a nonbinary code to achieve double-fault tolerance.*** *By using nonbinary codes, disk arrays can achieve double failure tolerance with only two disks of check data. Shown above is a single region of a double fault tolerant parity logging disk array with nonbinary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

To achieve double fault tolerance in such a parity logging array , the striped 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 identical and are not associated with a particular copy of the check information.

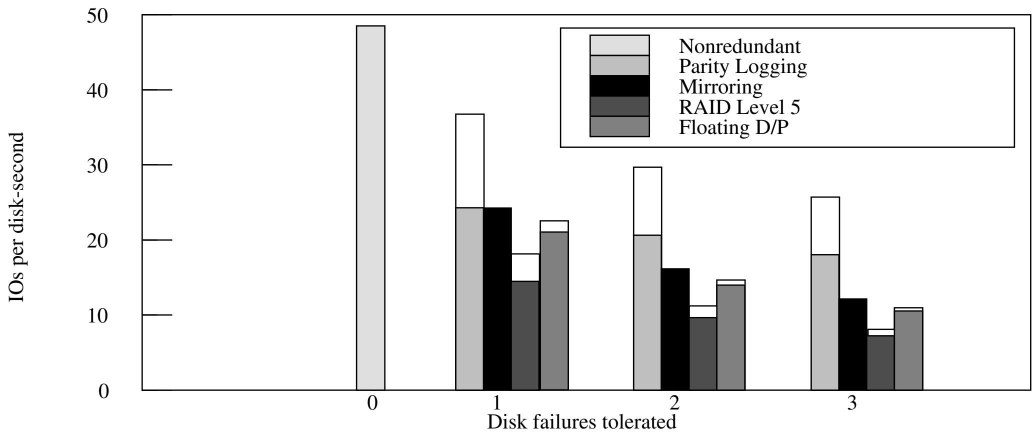

Fig. 26. ***Performance of multiple failure tolerating arrays*** *. While the performance of all array configurations declines with the number of failures tolerated, parity logging declines the least, decreasing in performance by about 15% per degree of failure tolerated. The highest performing alternative, mirroring, has a huge disk space overhead, requiring 3 or 4 disks per disk of user data in the double and triple failure tolerating cases, respectively . The performance of RAID level 5 and floating data and parity both decline rapidly , achieving less than 10 user writes per second in the triple failure tolerating case. The shaded portion of each bar shows the performance when the data to be rewritten is not cached, while the full height indicates performance when data is cached.*

double, and triple failure tolerating arrays using mirroring, RAID level 5, floating data and parity and parity logging. This data is derived from the models of sections 3 and 4 and applied to the example disk 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 nonpreread overhead. For example, while triple failure tolerating parity logging arrays should sustain about 35% of the I/O rate of nonredundant arrays for random small writes, quadruplicated storage (triple failure tolerating mirroring disk arrays) 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 all the 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 computed in memory from the new data and written directly to the parity unit. This "large write optimization" avoids the preread of data and parity associated with small writes, improving write performance by about a factor of four [Patterson88].

This optimization can not be applied directly to parity logging disk arrays as we have described them so far because there may exist outstanding (not yet reintegrated) logged updates for a particular parity unit at the time when a large write overwrites that parity unit. If these logged updates were ignored and a parity overwrite were done, the parity could be erroneously updated with the stale logged updates when reintegration occurs. This problem can be corrected by placing the new parity into the log instead of writing it directly to disk.[11] Parity placed 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 into the corresponding parity unit, now distinguishes between a normal "update" log record and the new 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 applied 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 apply the log records accordingly.

This new sequentiality constraint potentially lengthens the reintegration time, which, as section 6 will show, can substantially degrade performance at high loads. In the simplest case, a region's logs must be in read in the order they were written and merged to produce a update/overwrite image before 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 logs to parity would still have to be deferred until these reads complete. At this point, a sequential in-memory reintegration could be performed. However, as long 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 nearly sequential order. Based on this observation and because overwrite records eliminate all prior information, the following highly parallel algorithm can be used. Each block in the reintegration buffer is initially zeroed and marked "non-overwrite". Parity and log for the target region are all read in parallel. A parity block is applied if the corresponding buffer is marked "non-overwrite," and discarded if the buffer is marked "overwrite." If a logged record is an update and the block is "non-overwritten", the record is XORed in, but is buffered until all earlier log records have been processed. If a log record is an overwrite, the target block is overwritten and marked as "overwritten by record X." Any buffered updates that have already been applied should occur after this overwrite are reapplied. Overwrite or update records preceding X are not applied to a block marked "overwritten by X." As long as parallel reads on different sublogs proceeded at nearly the same rate, this algorithm will not consume much 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. Their LRAID-X4 organization maintains separate parity and parity-update log disks, and periodically applies the logged updates to the parity disk. In order to allow writes from the user to occur in parallel with log reintegration, they duplicate both the parity and the parity log for a total of four overhead disks. LRAID-X4 does not distribute parity or log information. Instead of breaking down the log disk into 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 updates are sorted into runs and written to the second log disk while the first log disk reintegrates its updates with the 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 each run on 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 (16 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 simulated in 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 to increase

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 a 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 that are 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 queues, which can then be scheduled to substantially reduce seek and rotational positioning overheads [Seltzer90, Solworth90, Rosenblum91, Polyzois93]. Data loss on a single failure is possible in these systems 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, several tracks in every disk cylinder are reserved, and an allocation bitmap is maintained. When a write is issued, the data is immediately written (typically in a self-identifying manner) to a rotationally close empty location in a reserved track, making the data durable. The write is then acknowledged, but the data is retained in the host or controller 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 the allocation bitmaps, mapping tables, and write buffers, this storage is not required to be fault tolerant, limiting controller cost. 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 mirroring systems [Solworth91, Orji93].

The floating location technique improves the efficiency of writes by eliminating the static association 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 physical-to-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 (LFS), in which all data is written in a segmented log, and segments are periodically reclaimed by garbage collection [Rosenblum91]. Using fault-tolerant caches to delay data writes, this approach converts all 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 of each block is stored in fixed location, while the other copy is maintained in floating storage, achieving higher write 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 based on a distributed log. Analytical models of the peak bandwidth of this scheme and alternatives from the literature were derived and validated by simulation. The proposed technique achieves substantially better performance than RAID level 5 disk arrays on workloads emphasizing small random accesses. When data must be preread 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 cached, performance is superior to floating parity and data and mirroring array configurations. This 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 large write optimization.

While the parity logging scheme presented in this paper is effective, several optimizations should 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 compression 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 degraded-mode and reconstruction while parity logging provides high performance during fault-free operation. The 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. <u>1994 DISK/TREND Report: Disk Drive Arrays</u>. 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. <u>Redundant Disk Arrays: Reliable, Parallel Secondary Storage</u>. 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. <u>IBM 0661 Disk Drive Product Description, Model 370, First Edition, Low End Storage Products, 504/114-2</u>. 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 (vol. 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), 604-617.

[Ramakrishnan92] Ramakrishnan, K. K., Biswas, P., and Karedla, R. Analysis of file I/O traces in commercial computing environments. *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 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 reliable is a RAID? *Proceedings of the 1989 IEEE Computer Society International Conference (COMPCON 89)* (Cat. No. 91TH0394-7), San Francisco CA, (Feb. 37-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 and Distributed Information Systems*. (Cat. No. 91TH0393-4), Miami Beach FL, (Dec. 4-6, 1991), IEEE Computer Society Press, Los 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).