Proceedings

The 22nd Annual International Symposium on COMPUTER ARCHITECTURE

June 22-24, 1995 Santa Margherita Ligure, Italy

Sponsored by

ACM SIGARCH
IEEE Computer Society, TCCA
With support of SGS-Thomson-Microelectronics and Olivetti
In cooperation with the University of Genoa

SIGARCH

QA76.5
.S84
1995

The Association for Computing Machinery
1515 Broadway
New York, N.Y. 10036

IEEE Catalog Number 95CB35801
ISBN 0-89791-698-0 (softbound)
ISBN 0-7803-3000-5 (casebound)
ISBN 0-7803-3001-3 (microfiche)
Library of Congress Number 85-642899
ISSN: 1063-6897

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 12114
Church Street Station
New York, N.Y. 10257

Phone: 1-800-342-6626
(U.S.A. and Canada)
1-212-626-0500
(All other countries)
Fax: 1-212-944-1318
E-mail: acmhelp@acm.org
acm_europe@acm.org (in Europe)

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
Los Alamitos, CA 90720

Printed in the U.S.A.

Table of Contents

Destage Algorithms for Disk Arrays with Non-Volatile Caches

Anujan Varma and Quinn Jacobson
Computer Engineering Department
University of California
Santa Cruz, CA 95064

Abstract

In a disk array with a nonvolatile write cache, destages from the cache to the disk are performed in the background asynchronously while read requests from the host system are serviced in the foreground. In this paper, we study a number of algorithms for scheduling destages in a RAID-5 system. We introduce a new scheduling algorithm, called *linear threshold scheduling*, that adaptively varies the rate of destages to disks based on the instantaneous occupancy of the write cache. The performance of the algorithm is compared with that of a number of alternative scheduling approaches such as *least-cost scheduling* and *high/low mark*. The algorithms are evaluated in terms of their effectiveness in making destages transparent to the servicing of read requests from the host, disk utilization, and their ability to tolerate bursts in the workload without causing an overflow of the write cache. Our results show that linear threshold scheduling provides the best read performance of all the algorithms compared, while still maintaining a high degree of burst tolerance. An approximate implementation of the linear-threshold scheduling algorithm is also described. The approximate algorithm can be implemented with much lower overhead, yet its performance is virtually identical to that of the ideal algorithm.

I. Introduction

A disk array, in general, consists of a group of disk drives together with an associated controller function, organized logically as a single I/O device [2], [4]. Disk arrays, also known as RAIDs (Redundant Arrays of Inexpensive Disks), are capable of providing improved levels of reliability, availability, and/or performance over single disks. A disk array usually provides protection against loss of data from a disk failure by maintaining redundant information within the array. Moreover, data availability can be maintained on a disk failure by using the redundant information to reconstruct data stored on the failed disk in real time. In addition to improving reliability and availability, disk arrays also improve the performance of the storage system by distributing data across multiple disk drives — this is the result of either concurrency in servicing multiple I/O requests or parallelism in the data transfer for a single I/O request.

Several types of disk array configurations are known in the literature [2], [4], [17]. These configurations vary primarily in the redundancy scheme employed and the data distribution (striping) scheme used to map logical blocks among the individual disks. In a seminal paper, Patterson, Gibson, and Katz [11] introduced a taxonomy of disk arrays, consisting of six different types (or "levels"), which is now widely used in the industry. Among these RAID levels, of particular interest are RAID-4 and RAID-5, which are optimized for transaction processing workloads. These arrays employ coarse-grained striping of data so that small requests can benefit from the concurrency in servicing multiple requests, while a large request can achieve high throughput by transferring data from multiple disks in parallel. RAID-4 organizes the disks in the array into parity groups, with one dedicated parity disk in each group. This has the disadvantage of the parity disks becoming a bottleneck while updating data in the system. RAID-5 eliminates this bottleneck by distributing parity blocks uniformly among all the disks in the group.

Both RAID-4 and RAID-5 provide reliability and availability at a fraction of the storage overhead incurred in disk mirroring. This reduction in storage overhead, however, is achieved at the expense of increasing the number of disk accesses necessary to update data in the system. Every write request to the array that does not update an entire stripe must now update parity by reading the old data and the old parity, and exclusive-ORing them with the new data. This involves a total of four disk accesses — reading the old data and parity, and writing the new data and parity. Menon and Mattson [7] showed that this overhead can degrade the performance of a RAID-5 considerably in a transaction processing environment where small requests dominate the workload. In addition, a small increase in the ratio of writes to reads in the workload can lead to a drastic increase in the response time for both reads and writes.

Several solutions have been proposed to reduce the overhead for small writes in a RAID-5 [8], [9], [15]. One approach is *parity logging* [15], where parity updates are posted into a dedicated log disk instead of updating the parity blocks in the array; updates of parity blocks in the array are performed in the background. This has the advantage of converting small writes of parity into large writes; the scheme also captures some of the temporal locality in parity updates since successive updates of parity can be combined into a single update in the disk array. Another scheme that reduces the overhead in parity updates is *floating parity* [9], where parity blocks are dynamically remapped within disk cylinders to reduce the rotational latency between reading and writing parity.

Both parity logging and floating parity only attempt to reduce the overhead of parity updates. In both schemes, the old data must be read from the disk and the new data written before signaling completion of the I/O transaction.

This research is supported by NSF Young Investigator Award No. MIP-9257103.

A more general scheme is to use a nonvolatile write cache to reduce write latency. Writes can now be deemed complete after writing the new data into the cache, an operation often referred to as *fast write* [8]. Both the data and the parity blocks on the disks can then be updated in the background. The process of updating data or parity in the disks from the write cache is referred to as *destaging*. In addition to the write cache, a larger (volatile) read cache may be used to improve read performance.

A nonvolatile write cache has several advantages [8], the most important of which is the substantially lower service time seen by write requests to the array. In addition, the write cache can also exploit any locality in writes — both temporal and spatial — in the workload. Temporal locality is exploited by capturing successive updates of the same block in the cache. Spatial locality allows many small writes to be aggregated into a single large write to the disk. Finally, the write cache can also lower the response time for *read* requests serviced by the disks because of the reduced contention with writes to the disk.

In spite of the above advantages, a nonvolatile write cache introduces several new problems. First, the write cache must be designed to be at least as reliable as the redundancy scheme used for the disks. The IBM Hagar disk array [8], for example, used duplicate caches with independent power boundaries to protect the contents of the write cache. Second, data losses could still occur while updating the disks from the cache because of a failure in the cache memory, disks, or the datapath between them. The destage algorithm must be designed to leave data in a consistent state should a failure occur while updating the disks. Several such algorithms to prevent data losses while destaging are presented in [8].

A third problem in the use of a write cache is that of scheduling the destages, that is determining *when* to destage data from the cache to the disk, as well as *which* of the dirty blocks is to be destaged next. Ideally, in a disk array with a write cache, the disks see only two types of requests — read requests generated by the workload that miss in the read cache, and (ii) read and write requests originated by the array controller for destaging data from the write cache. Requests of the first category must be serviced as soon as possible while destage requests can usually be serviced in the background. Ideally, destages should appear transparent to the user requests received by the array. In practice, however, because of the non-preemptive nature of disk service, some interference between user requests and destage requests is unavoidable. For example, a read request received while a destage request is being serviced is made to wait until the latter is completed. This interference, however, can be minimized with careful scheduling of the destages.

In this paper, we study a number of algorithms for scheduling destages in a RAID-5 system. We introduce a new scheduling algorithm, called *linear threshold scheduling*, that adaptively varies the rate of destages to disks based on the instantaneous occupancy of the write cache. The performance of the algorithm is compared with that of a number of alternative scheduling approaches such as *least-cost scheduling* and *high/low mark*. The algorithms are evaluated in terms of their effectiveness in making destages transparent to the servicing of read requests, as well as their ability to tolerate large bursts of write requests without causing an overflow of the write cache. Our results show that linear threshold scheduling provides the best read performance of all the algorithms compared, while still maintaining a high degree of burst tolerance.

The rest of this paper is organized as follows: Section II summarizes the tradeoffs involved in the design of the destage algorithm and describes the algorithms studied in the paper. Section III introduces the disk array model and workload model used in our simulations. Section IV presents simulation results for the algorithms and compares the algorithms in terms of read performance, disk utilization, and tolerance to bursts of write requests. Section V concludes the paper with directions for future research.

II. Destage Algorithms

In a disk array with a nonvolatile write cache, destages from the cache to the disk are performed in the background asynchronously, while read requests from the host system are serviced in the foreground. Thus, the workload seen by an individual disk predominantly consists of (i) reads from the host system that cannot be serviced from the read cache or write cache, and (ii) read and write requests generated by the array controller as part of destage operations. We refer to requests of the first category as *host reads* and those of the second category as *destages*. Reads of data or parity information performed as part of a destage are referred to as *destage reads* and the writes as *destage writes*. A write request from the host is never directed to the disk except in the event of a write cache overflow or when the size of the block being written is large enough to justify bypassing the write cache.

If host reads are given higher priority over destages, a destage would never be initiated by the array controller when there are requests in the read queue. Still, considerable flexibility exists in scheduling the destages, as the updates to the disk need not be performed in the order the writes were posted to the cache. In addition, the four disk accesses involved in a disk update — reads of old parity and data, and writes of new parity and data — need not be performed as a single indivisible operation.

A. Algorithm Design Tradeoffs

We first examine the tradeoffs involved in the design of the destage algorithm. Assuming that no overflow of the write cache occurs and none of the write requests bypasses the cache, all the write requests from the host system are serviced by the cache. In this case, the scheduling algorithm employed has no direct effect on the response time for a write to the disk array. However, the scheduling algorithm can still affect the response time seen by the *host reads* serviced by the disks, as well as the maximum sustainable throughput of the array. A scheduling algorithm that optimizes destages reduces the load on the disks due to destages, thus improving their read performance.

There are three ways in which a scheduling algorithm can improve the performance of the disk array: First, the algorithm can reduce the number of destages by capturing most of the re-writes in the write cache, thus exploiting the temporal locality in the workload. This requires maintaining blocks that are likely to be re-written in the near future in the write cache and destaging them only when additional re-writes are unlikely to occur soon. Similarly, if parity is allowed to be cached, successive updates of blocks in the same parity group generate only a single update of the parity block on disk.

Second, the number of destages can also be reduced by aggregating blocks that lie physically close on a disk and destaging them as a large read and/or write. This exploits any spatial locality present in the workload.

Finally, the scheduling algorithm can reduce the average time for a destage by ordering destage requests to the disks such that the service times in the individual disks are minimized. Since the mechanical positioning delays usually dominate the service time, this calls for ordering the requests to minimize the positioning delays in the disk. Many such algorithms for scheduling requests in individual disks taking into account the seek time and the rotational latency of requests have been described in the literature [3], [5], [6], [14], [16]. While similar algorithms could be used to schedule destage requests in a disk array, a basic distinction exists between the problem of scheduling destages in a disk array and that of scheduling host requests in an independent disk. A disk scheduling algorithm for host requests must make a tradeoff between maximizing throughput and providing fairness to user requests; for example, servicing the user requests in FCFS (first-come, first-served) order maximizes fairness, while a scheduling algorithm that reorders requests to minimize the total service time achieves maximum throughput at the expense of increasing the variance of the response time [5]. The destage algorithm in a disk array, on the other hand, does not have to deal with this dilemma: In most cases, the algorithm can schedule destages to maximize throughput without causing starvation of user requests.

All the above objectives in the design of the scheduling algorithm suggest maintaining the occupancy of the write cache close to its capacity. This allows maximum benefit to be obtained from any locality present in the workload and maximizes scheduling flexibility. Maintaining the write cache close to full, however, makes it vulnerable to overflow. Even a short burst of writes in the workload may cause the cache to overflow, forcing subsequent writes to bypass the cache until some destages can be posted to disk. Thus, the scheduling algorithm must strike a compromise between the conflicting goals of being able to exploit maximum locality and scheduling flexibility for destages, and preventing frequent overflows of the write cache.

The scheduling algorithm may be designed to be either work-conserving or non-work-conserving with respect to the disks in the array. A work-conserving algorithm never allows a disk to be idle when one or more destages are pending to the disk. A non-work-conserving algorithm, on the contrary, may refrain from scheduling a destage to the disk even when it is idle, expecting that the same destage could be performed at a lower cost in the future; the goal is to minimize the response time for host reads by reducing the contention between host reads and destages. Both types of algorithms are studied in this paper.

Thus, in summary, the following are the parameters that can be used by the destage algorithm to determine the block to be destaged next:

1. The probability of the block to be re-written in the near future. This factor is usually accommodated in the algorithm by ordering blocks in the destage queue based on the time of their last access.
2. The number of blocks to be read/updated on the same track (or cylinder) to take advantage of spatial locality.
3. Service times of the requests in the destage queue. The service time includes the delay in positioning the head at the beginning of the block to be read/written and the data transfer time. Different levels of approximations could be used to estimate the positioning delays in the disk [16].
4. The current level of occupancy of the cache.

In addition, other factors such as the type of destage request (parity or data, read or write), may also be used in the scheduling process.

In the following, we describe four scheduling algorithms for destaging blocks from the write cache in a RAID-5. In all the algorithms, destaging to each disk is handled independently by maintaining a separate queue of pending destages for each disk. In addition, in all the algorithms, a destage to a disk is initiated only when there are no host reads queued for that disk at that time. However, not all of the algorithms schedule a destage always when the read queue is empty. The algorithms differ in the criteria used to determine *whether* a destage is to be initiated when the disk becomes idle after servicing a request, and in the function used to select the block to be destaged.

B. Least-Cost Scheduling

This algorithm is modeled after the well-known *shortest seek-time first* disk scheduling algorithm [3]. After every disk operation, the queue of eligible destages is examined and the request that takes the shortest access time is performed. In addition, to exploit spatial locality, if there are pending destage requests to other blocks on the same track, these are also performed as one large read or write.

The access time of a destage read or write operation includes the seek time (or head-switch time), rotational latency, and controller delays. Precise estimation of these delays is difficult due to many factors such as the physical-to-logical mapping of sectors on the disk and the use of caching within the disk itself. Considerable improvement can still be obtained using approximate models for estimating the delays involved. We implemented such a scheme in our RAID simulator for comparison with other schemes. Details of implementation of the scheme can be found in Section III.

C. High/Low Mark Algorithm

This algorithm is designed after the cache purging algorithm proposed by Biswas, Ramakrishnan and Towsley [1] for disks with a nonvolatile write cache. In this algorithm, two cache-occupancy thresholds are used to enable and disable destages. Destages to the disk are disabled when the cache occupancy drops below a "low mark" and turned on when it increases over a "high mark." Some hysteresis between the two thresholds is used to prevent short bursts in the workload from causing a cache overflow. We chose the high threshold as 70% of cache capacity and the low threshold as 30%. Destage requests were selected using the least-cost policy to minimize the service time of individual disk accesses. In addition, when a block is selected for destage, all pending destage requests to the same track on the disk are scheduled together.

D. *Linear Threshold Scheduling*

The basic principle of this algorithm is to match the rate of destages from the cache to the current level of occupancy of the cache, accelerating destages gradually as the cache occupancy increases and slowing them as the occupancy falls. The primary difference between this algorithm and high/low mark scheduling is that, instead of on/off thresholds, the rate of destages is changed gradually as a function of cache occupancy.

As in other algorithms, each disk in the array is scheduled independently. Both parity and data destages are treated in a similar manner. The destage queue is examined after each disk operation if the host read queue is empty. As in the case of least-cost scheduling, the block to be destaged is chosen as the one with minimum cost among all the eligible destage requests to that disk. However, unlike in least-cost scheduling, the destage is performed only if its cost is within a certain threshold maintained by the algorithm. Should a read request from the host arrive while the destage is in progress, this policy minimizes its waiting time.

The role of the threshold is to reduce the interference between destages and host reads; the threshold sets an upper bound on the waiting time of a read arriving while a destage in progress. A tradeoff exists in the choice of the threshold: If the threshold is set too low, the destage rate may not be adequate to clear the cache fast enough, causing frequent overflows. A large threshold, on the other hand, may perform destages too soon, degrading both the waiting time for host reads and the hit rate of the write cache, and reducing the level of temporal and spatial localities that may potentially be exploited.

In our implementation, the threshold was selected as a linear function of the instantaneous cache occupancy. The cost of a destage is its service time, computed as in the case of the least-cost scheduling algorithm. As in other algorithms, when a destage request is scheduled, all pending destage requests to the same track are also performed along with it.

The linear threshold scheduling algorithm uses the cache occupancy, service time of destages, and spatial locality as its decision-making criteria, but does not attempt to maximize temporal locality explicitly. Thus, a block that has been written recently may be scheduled for destage if the destage can be performed "cheaply." This may increase the total number of destage accesses to the disk in comparison to servicing the requests in LRU order, but the total service time can still be lower as a result of minimizing the cost of individual destage requests. Note that temporal locality for host reads can still be exploited by keeping the destaged block in the write cache until the block needs to be replaced.

E. *Approximation to Linear Threshold Scheduling*

A major difficulty in implementing both the least-cost scheduling and the linear threshold scheduling algorithms is the need to scan the entire queue of destage requests to select the minimum-cost request. This operation must be performed at the completion of every disk service. Furthermore, the cost of individual destage requests must be computed afresh if the position of the head has changed.

The need for searching the entire destage queue can be overcome by resorting to an approximate implementation of linear threshold scheduling; the approach works by dividing the disk into regions and maintaining a separate queue of destage requests into each region. When the disk becomes idle, the queues are searched in the order of the closest to the farthest region relative to the current position of the head. The first non-empty queue is found and the request at its head is selected as the minimum-cost destage. As in the ideal linear-threshold scheduling algorithm, the request is scheduled if its estimated service time is within the threshold determined by the current cache occupancy level. The requests within each of the destage queues can be ordered using the LRU policy to allow maximum temporal locality. As in the other scheduling algorithms, when a request is selected for destage, all the pending destages to the same track are performed in one disk access.

Both seek time and rotational latency must be taken into account in ordering the search of regions. Therefore, we used a partitioning scheme in which the disk cylinders are partitioned into circular *bands*; each band is then radially subdivided into *regions*. This is illustrated in Fig. 1(a) with respect to the example disk used in our simulation model. The example drive is an HP 97560 disk with a total of 1935 cylinders. The cylinders are partitioned into 15 circular bands, with 129 cylinders per band; each band is radially subdivided into three regions, to provide a total of 45 regions for the scheduling algorithm. This is similar to the disk partitioning scheme proposed by Jacobson and Wilkes [6], except that the cost of each destage is approximated as the *average* positioning delay from the current region to the target region. The average positioning delay from region i to region j is taken as the positioning delay from the center of region i to the center of region j.

The linear threshold scheme is approximated by designing a stepped threshold function that specifies the maximum destage cost for each level of cache occupancy. The destage cost is the average access time to a region expressed as a multiple of the fractional rotational delay of the disk corresponding to a region. Table I shows the threshold function used in our example. The second column in the table provides the maximum allowable destage cost for each step of cache occupancy; instead of expressing as milliseconds, the delay is given as the number of regions that pass under the head within that period (that is, 3 times the number of rotations within the period). For example, for a cache occupancy of 30%, the maximum allowable destage time is 3 regions, or the time for one revolution of the disk.

The following algorithm is used to select the regions that are searched for scheduling destages. Assume that the head is currently over region i and let w be the cache occupancy. Let $Th(w)$ be the maximum allowable destage time from the second column of Table I, in terms of the number of regions passing under the head. Then, region j qualifies for scheduling a destage only if

$$Th(w) \geq cost(i,j), \tag{1}$$

where $cost(i,j)$ is the estimated average cost of the destage to region j. Assuming that the head is exactly at the center of region i, $cost(i,j)$ estimates the delay for the head to be positioned at the center of region j, taking into account the seek time, rotational latency, and controller overhead. The actual equations used in computing the estimates are given in Appendix A. For convenience, the cost is expressed in units of the time taken by the disk to rotate through one region.

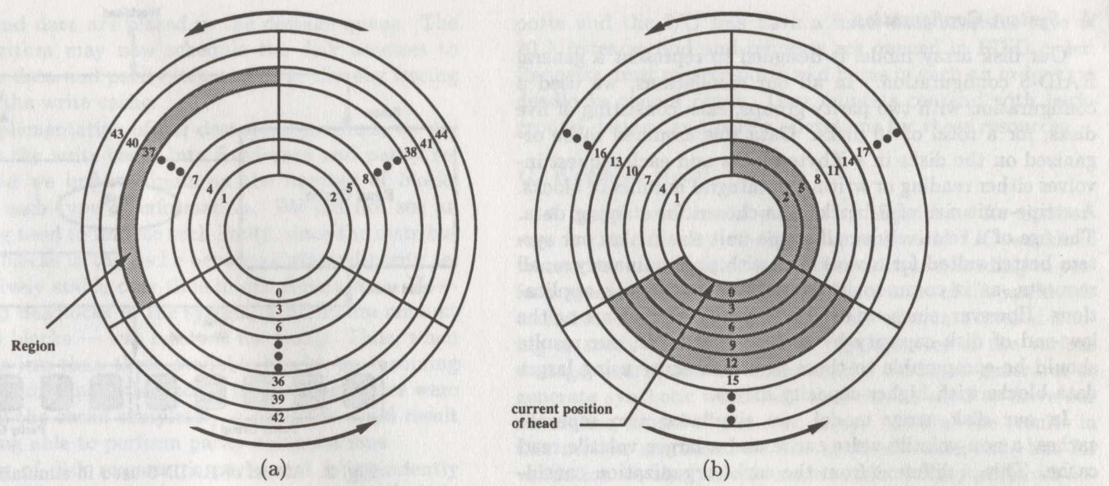


Fig. 1. Illustration of the disk partitioning scheme used for approximate implementation of the linear threshold scheduling algorithm: (a) Regions in the example disk; (b) Regions qualifying for destage when the head is currently on region 0 and the cache occupancy is in the range 25–37.5 percent.

Searches to the qualifying regions are ordered according to their destage cost, from the smallest to the largest. In our simulations, the searches are performed such that all the regions that fall within one unit of positioning delay are examined in parallel first; if no destages are queued in any of these regions, those that fall within two units of delay are checked, and so on. If pending destages are found in multiple regions during any step of the search, the least-recently accessed block among them is chosen for the destage.

The cost of a destage is estimated as the total access time for a block at the center of the target region, assuming the head is at the center of the current region. This requires estimating the seek time, rotational latency, and other overhead incurred in positioning the head in the target region. The average seek time $t_{seek}(i,j)$ between regions i and j can be estimated from a seek-time model of the disk. Our simulation model for the disk drives is based on the HP 97560, obtained from [13]. If two regions i and j fall in different bands, the average seek time $t_{seek}(i,j)$ between them is estimated as the seek time between the middle cylinders of the two regions. When the regions i and j are within the same band, the average seek time is estimated as the time to seek across half the width of the band. A constant controller overhead of 2.2 ms is added to the seek time in both cases.

For example, the shaded area in Fig. 1(b) shows the regions qualifying for a destage in our example disk when region 0 is currently under the head and the cache occupancy is within the range 25–37.5 percent. These regions were determined as follows: The rotation time of the disk was taken as 15 ms. Using the seek-time model from [13] and Eq. (2) from Appendix A, the average seek time from region 0 to to its adjacent band was computed as 9.98 ms, including controller overhead. Since this is within one rotational latency, the destage cost from region 0 to region 3 is estimated as one rotational latency, or 3 units. For the range 25–37.5% of cache occupancy, Table I lists the maximum allowable destage cost as 3 units. Thus, region 3 is allowed to be destaged according to Eq. (1). Similarly, regions 6, 9, and 12 also qualify because their average seek times were estimated to be within 15 ms, satisfying Eq. (1). For region 1, however, the seek time within the band, plus controller overhead, is estimated as 6.45 ms; since this is larger than 1/3 of the rotational latency, the destage cost to region 1 is estimated as 4 units, or 4/3 × rotational latency. The shaded regions in Figure 1(b) all have destage cost within 3 units. Note that blocks within region 0 qualify for destage because the seek time within the region, as computed by Eq. (3) is within 15 ms, so that the average cost of a destage would be estimated as one rotational latency.

Range of cache occupancy (%) w	Maximum destage delay (in units of rotational latency of a region) $Th(w)$
0.0 – 12.5	1
12.5 – 25.0	2
25.0 – 37.5	3
37.5 – 50.0	4
50.0 – 62.5	5
62.5 – 75.0	6
75.0 – 87.5	7
87.5 – 100	8

TABLE I. Threshold function used in the example to illustrate the approximate implementation of linear threshold scheduling.

III. Simulation Model

To evaluate the effectiveness of the scheduling algorithms described in the previous section, we implemented the algorithms in our RAID simulator and performed extensive simulations. In this section, we describe details of our simulation model and the methodology used to generate the synthetic workloads used in our simulations.

A. System Configuration

Our disk array model is designed to represent a general RAID-5 configuration. In all our simulations, we used a configuration with two parity groups, each consisting of five disks, for a total of 10 disks. Data was assumed to be organized on the disks in 4 Kbyte blocks and each request involves either reading or writing an integral number of blocks. A stripe-unit size of 9 blocks was chosen for striping data. The use of a relatively small stripe-unit size makes our system better suited for a workload with predominantly small requests, as is common in transaction processing applications. However, since we used 1 Gbyte disks that are at the low end of disk capacity by today's standards, our results should be comparable to those in a disk array using larger data blocks with higher capacity drives.

In our disk array model, we simulated two separate caches, a non-volatile write cache and a larger volatile read cache. This is different from the cache organization considered by Menon [10], where a single non-volatile cache was used to service both reads and writes. The single cache allows more flexibility in allocating the available cache space between reads and writes, but at a substantially higher cost. Both caches in our model are dual-ported and use LRU replacement algorithm. Since the total storage capacity of the array is approximately 10 Gigabytes, we chose the size of the read cache as 8 Mbytes, approximately 0.1 percent of disk capacity. The size of the write cache used in the simulations vary over the range of 512 Kbytes – 4 Mbytes, with focus on the 1 Mbyte cache size. Both the read and write caches were organized with a fixed block-size of 4 Kbytes.

The read cache always contains unmodified copies of disk blocks. It is used for two primary purposes: (i) to hold copies of the most recently read data so that subsequent reads can be serviced from the cache, and (ii) to hold data or parity that is needed for parity calculations. The write cache is used mostly to hold information that is newer than the copy on the disk, that is, both data written by the host and newly calculated parity blocks. A block in the write cache remains valid until it is updated on the disk by the destage algorithm. Although information can be removed from the write cache immediately after updating it on disk, our simulator simply marks the corresponding cache blocks as "available." These blocks can still be accessed for future reads and parity calculations if they have not been written over.

In our implementations of the scheduling algorithms, we did not explicitly attempt to make the algorithms robust. Instead, our focus is on evaluating their performance. Techniques such as those used by Menon and Cortney [8] could be used to make the write cache and the destage algorithm tolerant to failures.

Requests from the host are queued in the system and scheduled for service in FCFS order. Although requests are initiated in order, they are not always completed in order. Requests that can be completed entirely by the read and write caches and require no immediate disk accesses are completed in order. Requests that involve disk accesses, on the other hand, are handled asynchronously and may complete out of order. The following paragraphs explain the servicing of host requests in more detail.

Host reads are handled in a relatively simple manner. When a read request arrives from the host, both caches are checked for the requested block(s). If all the requested blocks are present in the caches, data will be transferred as soon as the I/O channel becomes available. If one or more blocks are found missing, a read request is generated for every missing block and queued for the corresponding disk to become free. Note that host requests are serviced at a higher priority by the disks in comparison to destage requests. When all the blocks have been transferred into the read cache, and on obtaining access to the I/O channel, data is transferred to the host system as a single block transfer. Thus, the data transfers may not occur in the order in which the requests were received by the disk array.

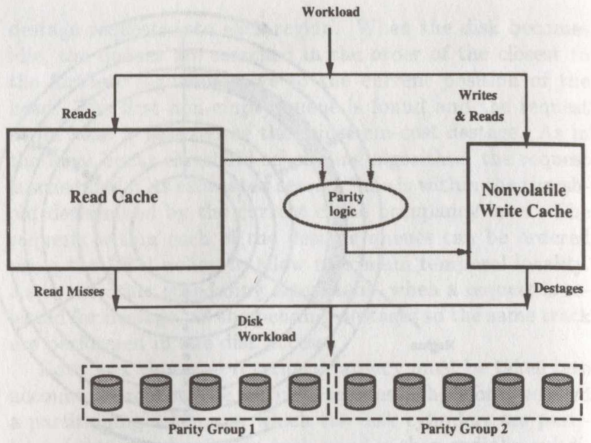


Fig. 2. Model of RAID-5 used in simulations.

Handling of write accesses from the host is substantially more complex, since there are many cases to consider. To keep our model simple, we assume that a write access never bypasses the write cache unless the cache is full. There are three separate cases to consider in handling a write: First, the request may be a re-write of a block already present in the write cache whose parity has not yet been computed. In this case, the block is simply written over with the new data. If the block is not in the write cache, or if its parity has already been computed, an available block is chosen in the write cache and the new data is written to the block. Thus, two updated copies of the block may be present at the same time in the write cache. In this case, the earlier version can be removed after the parity is computed taking into account the newer version. Finally, if the write cache is full at the time the request is serviced, it is handled as in a traditional RAID-5, with all the associated disk accesses occurring before a completion message can be sent out.

Data written to the write cache are eventually updated on the disks in the background by the destage algorithm. Since parity must also be updated, the new parity must be computed using the old copies of data and parity before the new data can be removed from the cache. If the old data and parity are not present in either cache, they must be read from disk. These disk accesses are scheduled by the destage algorithm and the associated data/parity are placed in the read cache when they become available. When all the needed data/parity blocks are available in the caches, a new parity is calculated and written into the write cache. Once the parity computation is complete, requests for updating the new parity and data are placed in the destage queue. The destage algorithm may now schedule the disk accesses to copy the new data and parity items to disk, thereby freeing up blocks in the write cache.

In the implementation of our destage algorithms, we did not partition the write cache into fixed data and parity regions, nor did we impose limits on the number of blocks occupied by each type of information. We did not see an overwhelming need to impose such limits, since the distribution of valid blocks in the cache between data and parity remained relatively stable over time in our simulations. However, to avoid deadlocks in the system, a minimum amount of space — 3 blocks — was reserved for parity. Thus, when the cache has less than three available blocks, an incoming write is forced to bypass the cache. If incoming writes were allowed to fill the cache completely, a deadlock could result from not being able to perform parity computations.

Scheduling of disk requests is performed independently for each disk. There are two separate queues for each disk, one for host reads (and writes, when the write cache becomes full) and the second for destage accesses. Whenever a request is received for a disk that is currently idle, or when a disk that is currently busy completes its access, the scheduling algorithm for that disk is invoked. The scheduling algorithm handles host reads and writes in FCFS order. These accesses are given priority over destage disk accesses. A request is serviced from the destage queue only when the host queue is empty.

B. The RAID Simulator

In designing the simulator we needed to determine the components that are critical to the performance of the system. The key elements simulated were the I/O channel, the two caches, disk drives, the controllers, and some of the data paths. We felt that these elements represented the majority of the system's timing and data transfer constraints.

Our simulation model for the disk drives is based on the HP 97560. These are 5.25-inch, 1.26 Gbyte disks which rotate at 4002 rpm. The average access time of the disk is 23 ms for an 8 Kbyte data transfer. The disk drive model takes into account seek time, rotational latency, head settling time, and data transfer time. The disk drive model is based on the work of Ruemmler and Wilkes [13].

Data transfers from the disk drives are limited by the bandwidth of the write port of the read cache. All data read from disks must be transferred through this port. When disks perform reads they buffer the data in their track buffer, but the disk is marked busy until the data-transfer is completed. For writes we overlap data transfer with the seek, and assume that sufficient bandwidth will be available to transfer the data to the disk buffer before the seek is completed. This does not model datapath contention in estimating the disk service time, and is therefore optimistic; in our simulations, however, the number of cases when the data could not be delivered in time to the disk because of contention was insignificantly small.

The response time is also affected by the implementation of the caches and the I/O channel. Both the read and the write cache can be searched in parallel, but blocks in each cache are searched sequentially. Response times of requests also take into account the data transfer rate and queueing delay of the cache ports and the I/O bus. The cache ports and the I/O bus have a fixed data transfer rate of 20 Mbytes/second and requests are queued in FIFO order. Requests must reserve ports and buses in such an order that deadlocks cannot occur. User requests contend with background destage and parity operations for these resources.

C. Workload Model

A major problem in evaluating algorithms for disk arrays is the difficulty of obtaining real-world I/O workload traces that are capable of providing adequate loading to perform meaningful measurements. Since none of the workloads available to us was able to provide the adequate level of loading to study the differences in the behavior of the various destage algorithms, we constructed a workload generator to generate synthetic workloads that can be used to drive our simulation model of the disk array. Most of the results in this section are based on the synthetic workload, but for validation we also provide some results from a workload obtained by overlaying multiple request streams from a set of real traces.

Instead of using a random spatial distribution of requests to blocks in the disk array, the workload allows to simulate requests from multiple process groups (each representing one or more processes), that accesses subsets of the logical disk space. Within each of the groups, the request rate, read/write ratio, and degree of locality of the accesses are configurable. For obtaining the results reported in this paper, the workload generator was configured to represent ten process groups each generating requests at the same rate. Nine of these process groups accessed a randomly chosen one-percent subset of the disk space, while one process group produced requests distributed across the entire disk space.

Within each process group, locality is modeled by maintaining a history buffer of previous requests. The workload generator chooses the address of a request by either generating a new one randomly from the process group's address space, or from the history buffer randomly. Addresses in the history buffer are chosen according to a normal distribution that favors more recent requests over older ones. When the history buffer is used to generate addresses the target addresses for reads are chosen from old requests (both reads and writes) uniformly, while the target addresses for writes are chosen 90% of the time from old writes and 10% from old reads. The history buffer is larger than either of the caches, so the cache hit ratios would be lower than the percentage of requests generated from the history buffer.

The issue times of the requests were based on a normal distribution. This distribution causes requests to be issued with a pattern of small cyclical bursts. For the most part we set the mean issue rate to a fixed value, but in some of the experiments designed to measure the burst-tolerance of the scheduling algorithm, we configured the system to create sustained bursts in the workload. The bursts were implemented by issuing requests at a fixed background rate and then tripling the issue rate periodically.

A number of default values are implied for parameters in the simulations reported in this paper. The size of the write cache is 1 Mbyte, unless specified otherwise. The total number of process groups used in the workload model is 10, and the write percentage is always 40%. The default size of the history buffer is 10 Mbytes (0.1 percent of disk capacity). The default percentage of requests to be chosen from the history buffer is 65% for reads and 90% for writes. For an 8 Mbyte read cache, this results in a hit ratio of approximately 51%. For all the simulations the mean number of disk blocks per request was 2, with a normal distribution favoring smaller requests.

The basic objective of our evaluation is to study the behavior of the destage algorithms, as opposed to benchmarking. The simplified workload model we used allowed us to study how the parameters of the workload such as locality and burst distribution affect the different algorithms. The model allows certain characteristics of the workload to be exaggerated in a simple manner. In addition, errors introduced by the spatial skew in a real workload are likely to be much less severe in a disk array as compared to a single disk, because of the striping employed in the former.

IV. Simulation Results

Thus far we described the algorithms for scheduling destages and the simulation model of the disk array. In this section, we present results from our simulation of the algorithms and analyze the behavior of the algorithms based on observed results. We first start with a discussion of the performance metrics that are useful in comparing the scheduling algorithms.

A. Performance Metrics

As pointed out in Section II, the average response time for write requests from the host may be little affected by the scheduling algorithm, as most of them may be serviced by the cache. The destage algorithm may, however, have a significant effect on the latency of host reads that are serviced by the disks, particularly at heavy loads. In addition, a number of other metrics are useful in bringing out the tradeoffs involved in the design of the algorithm. We used the following metrics to evaluate our destage algorithms:

1. **Response time of host reads:** This is the average delay experienced by a read request from the host, taking into account the service time and various queueing delays involved. Since the response time of the read requests serviced from the caches is not directly affected by the destage algorithm, we considered only the host reads that miss in the caches and are therefore serviced by the disks. These reads, although serviced at a higher priority as compared to destages, may still undergo queueing delays if a destage is in progress at the time the read request arrives in the queue. A scheduling algorithm that reduces either the number of destage accesses or the average time per destage, or both, can potentially reduce this delay, thus improving the response time of host reads directed to the disks.
2. **Maximum throughput:** The maximum throughput is the maximum sustained rate of host requests that can be serviced by the disk array without causing the response time to grow unbounded. The destage algorithm can increase the maximum throughput by minimizing both the number of destages and the service time of individual destages. The former is achieved by exploiting the temporal locality of writes and the latter by minimizing the total delay for each destage.
3. **Disk utilization:** The disk utilization is defined as the fraction of the time the disk is busy servicing a request. The disk utilization can be broken up into two components, one representing host accesses and the other destage accesses, and further into reads and writes. The load due to host reads is unlikely to change significantly with the different scheduling algorithms; thus, any change in the total utilization is predominantly due to the destage component. This component captures both the average number of destages performed in a given period of time and the average time per destage, and can therefore be taken as a measure of the total amount of work performed by the disks in a unit of time for performing the destage function.
4. **Burst tolerance:** Another important aspect of the destage algorithm is its ability to tolerate short bursts in the workload without causing a write-cache overflow. The degree of tolerance to bursts can be measured in several ways; we chose to use *the minimum length of a burst to cause cache overflow* to express the sensitivity of the algorithm to bursts in the workload. We evaluated this metric as follows: The system is first brought to steady state with a certain steady background workload. After reaching steady state, a burst is injected into the system, increasing the request rate. In our experiments, the burst rate was chosen as three times the background rate. The length of the burst to cause an overflow was then found as the interval from the start of the burst to the time at which the write cache becomes full. The burst is turned off once a cache overflow occurs and the system allowed to reach steady state again with the background workload. The cycle was repeated several times to obtain a mean value of the burst length to cause cache overflow.

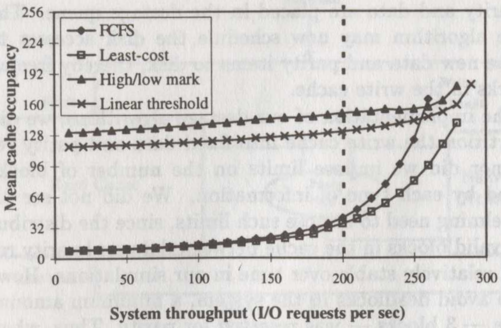
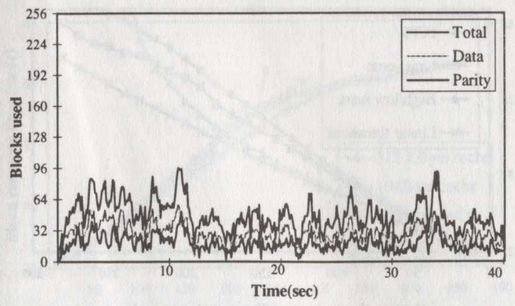


Fig. 3. Mean occupancy of the write cache in the RAID-5 system for the various scheduling algorithms (size of write cache = 1 Mbyte).

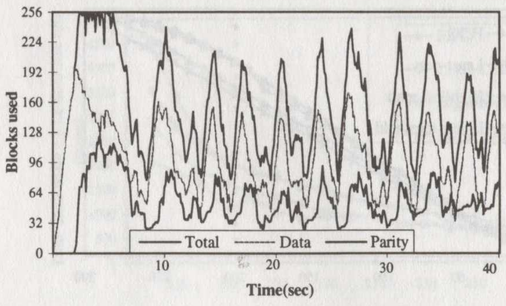
B. Results

We now present the results from simulating the various scheduling algorithms introduced in Section II. We first compare the performance of four scheduling algorithms — least-cost scheduling, high/low mark, linear threshold scheduling, and a fourth algorithm where destages are performed in FCFS order. The approximation to linear-threshold scheduling discussed in Section IIE is evaluated later since its behavior is similar to that of linear threshold scheduling.

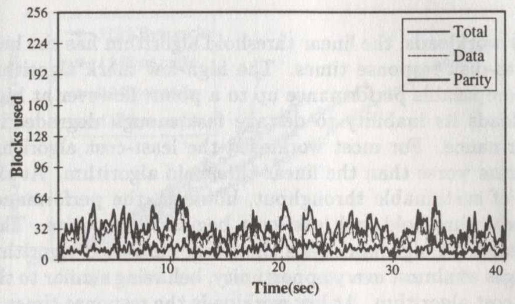
Fig. 3 shows the mean occupancy of the write cache for the four destage algorithms. Fig. 4 shows the variation in the occupancy over time for the same algorithms. As ex-



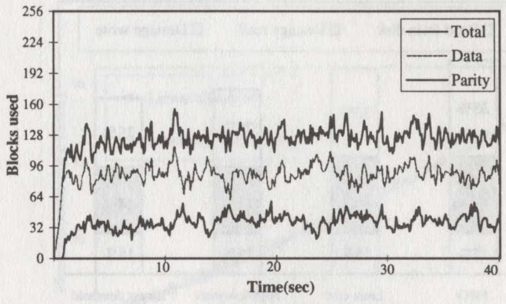
(a) FCFS scheduling.



(b) High/low mark scheduling algorithm.



(a) Least-cost scheduling.



(b) Linear-threshold scheduling algorithm.

Fig. 4. Variation of the occupancy of the write cache with various scheduling algorithms (size of write cache = 1 Mbyte).

pected, the high-low mark algorithm maintains the write cache occupancy at approximately 50%, near the middle of its high and low marks. The inability of the high-low mark algorithm to destage quickly enough once it reaches its high mark results in frequent overflow of the write cache, as seen in Fig. 4(b). This is a serious limitation in that the performance of the system is degraded during the periods when the write cache becomes full. The linear threshold algorithm demonstrated greater stability. Its write cache occupancy has the same mean of approximately 50%, but is less likely to cause an overflow for comparable workloads. The least-cost and FCFS algorithms maintain much lower cache occupancies, until the workload becomes very heavy. At the limits of their sustainable workload intensities, the write cache occupancies of both FCFS and least-cost scheduling approach that of linear threshold scheduling.

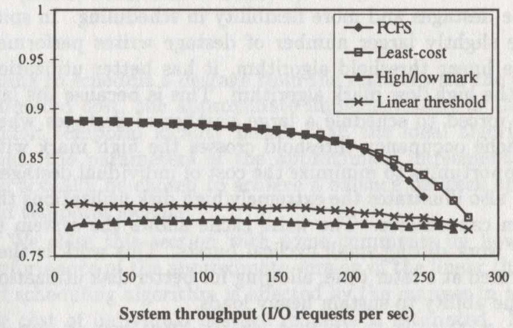


Fig. 5. Ratio of disk writes to host writes for the four scheduling algorithms.

Destage algorithms gain performance by optimizing disk use in terms of both the number and duration of disk accesses. Figures 5, 6, and 7 illuminate this aspect of the algorithms. Figure 5 plots the ratio of the number of writes actually seen by the disks to the number of host writes, both in blocks. Both FCFS and least-cost scheduling perform significantly more writes as compared to the linear threshold scheduling algorithm, because of their inability to exploit locality among the writes. The high/low mark algorithm performs the best in terms of the number of writes performed, because of its higher average cache occupancy.

To evaluate the effectiveness of the algorithms in minimizing the total time taken by disk writes, both the number of disk writes and their service times must be taken into account. Fig. 6 compares the disk utilization of the four destage algorithms. Fig. 7 further provides a breakdown of the utilization into its components, measured at a system throughput level of 200 I/Os per second. The linear threshold and the high-low mark algorithms have noticeably lower disk utilization than the other two algorithms for comparable workloads. The improved disk utilization is the result of spending less time performing destage reads and writes. This is accomplished by maintaining more dirty blocks in their

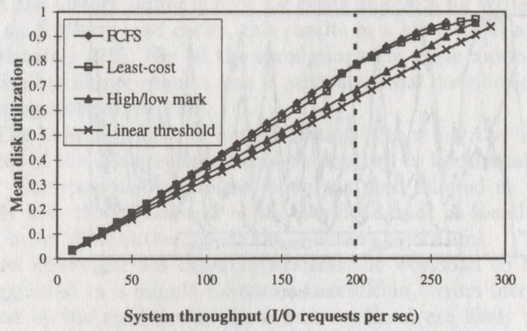


Fig. 6. Disk Utilization in the RAID-5 system for the various scheduling algorithms with a 1 Mbyte write cache.

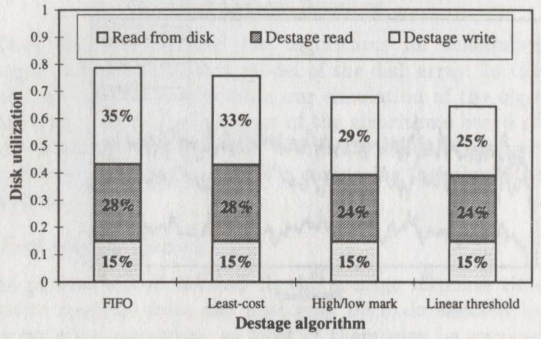


Fig. 7. Breakdown of disk usage in the RAID-5 system for the various scheduling algorithms with a 1 Mbyte write cache.

write caches, which gives them the advantages of fewer duplicate destages and more flexibility in scheduling. In spite of the slightly larger number of destage writes performed by the linear threshold algorithm, it has better utilization than the high/low mark algorithm. This is because the latter is forced to schedule a large number of destages when the cache occupancy threshold crosses the high mark with less opportunity to minimize the cost of individual destages. Fig. 6 also illustrates the extremely high disk utilizations the system can achieve. The write cache allows the system to buffer work during small cyclical bursts. This work is then performed at a later time, allowing for better disk utilization and the ability to sustain heavy loads.

Fig. 7 shows the contribution to the disk utilization from three different components — host reads, destage reads, and destage writes. Host writes occur in our simulations only when the write cache is full; therefore it was an insignificant part of overall disk utilization. The share of host reads is virtually identical for all the algorithms, demonstrating the relative insensitivity of the number and service time of host reads serviced by the disks to the scheduling algorithm used. The contribution of the destage components, however, shows considerable variation among the algorithms. Both the high/low mark and the linear threshold algorithms again perform the minimum amount of work for destaging.

One of the most important performance measures visible to the user is the response time of host reads that are serviced by the disks. Fig. 8 compares the disk read response time for all the destage algorithms. For moderate to

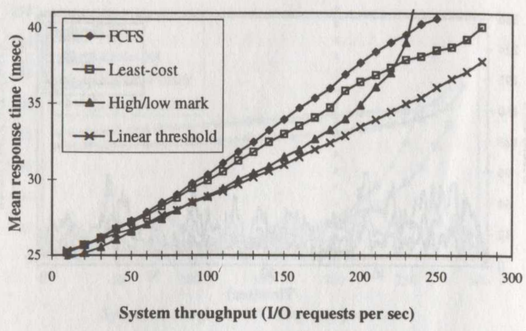


Fig. 8. Performance of disk reads in the RAID-5 system with 1 Mbyte write cache.

heavy workloads, the linear threshold algorithm has the best read-to-disk response times. The high-low mark algorithm has comparable performance up to a point; however at high workloads its inability to destage fast enough degrades its performance. For most workloads the least-cost algorithm performs worse than the linear threshold algorithm. At the limit of sustainable throughput, however, the performance of linear threshold and least-cost begins to converge. This is because at high workloads the linear threshold algorithm destages at almost every opportunity, behaving similar to the least-cost algorithm. At low workloads the response times of all the algorithms converge to the disk access time for reads since there is very little queueing delay.

Up to this point we have focused on systems with a write cache size of 1 Mbyte, but it is important to consider how the algorithms are affected by the size of the write cache. Fig. 9 and Fig. 10 show the delay-throughput plots for various cache sizes for least-cost scheduling and linear threshold scheduling, respectively. The least-cost algorithm is not able to achieve substantial response-time gains from a larger write cache. This is due to the conservative nature of the algorithm. The linear threshold algorithm, on the other hand, consistently improves response times for larger caches as it becomes more selective in scheduling destages, as can be seen in Fig. 10. Both algorithms see comparable increases in sustainable throughput as the cache size is increased. This is due both to being able to achieve better disk utilization and to having the additional cushion for absorbing bursts.

The ability of the destage algorithm to tolerate sustained bursts in the workload without causing an overflow of the write cache is important for the stable operation of the system. The addition of a write cache to a disk array allows it to tolerate occasional overloads by buffering work. However, to sustain a burst in the workload comparable to the size of the cache, the scheduling algorithm must perform as many destages as possible. Fig. 11 compares the ability of the scheduling algorithms to tolerate occasional bursts in the workload, in terms of the minimum duration of the burst to cause a cache overflow. This was measured by bringing the system to steady state with a certain background workload represented by the x-axis, and then injecting a burst by tripling the request rate. The length of the burst to cause an overflow was then found as the interval from the start of the burst to the time at which the write cache becomes full. The burst is turned off once a cache overflow occurs

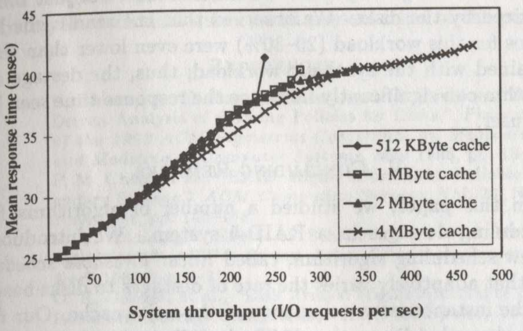


Fig. 9. Performance of disk reads in the RAID-5 system with the least-cost scheduling algorithm.

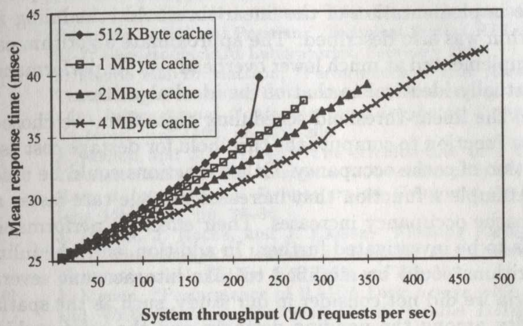


Fig. 10. Performance of disk reads in the RAID-5 system with the linear-threshold scheduling algorithm.

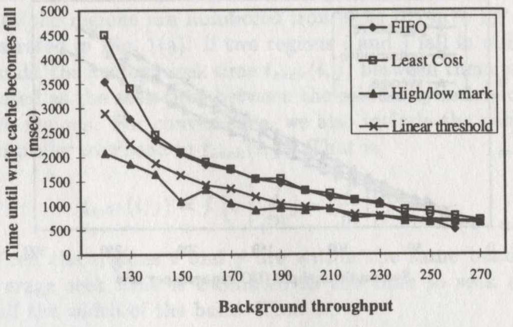


Fig. 11. Minimum burst duration to cause write cache overflow in the RAID-5 system under various scheduling algorithms (size of write cache = 1 Mbyte).

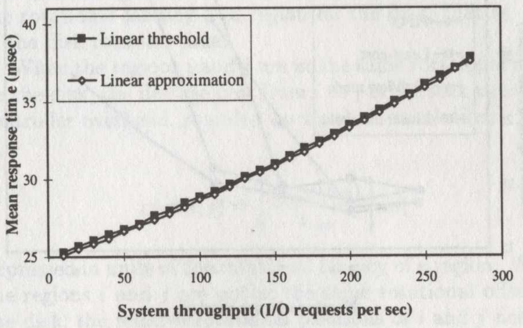


Fig. 12. Performance of disk reads in the RAID-5 system for linear threshold scheduling and its approximate implementation (size of write cache = 1 Mbyte).

and the system allowed to reach steady state again with the background workload. The cycle was repeated several times to obtain a mean value of the burst length to cause cache overflow.

The least-cost algorithm, followed closely by FCFS, showed the best burst tolerance. Both of these algorithms gain their resilience from maintaining a low mean write-cache occupancy before the burst, thus giving them a larger cushion when the burst begins. The linear threshold algorithm is not able to sustain bursts as long as the least-cost or FCFS algorithms because it generally has a higher write-cache occupancy at the beginning of a burst. However, although its burst tolerance is only moderate, the linear threshold algorithm was able to recover from bursts with considerably higher background workloads than the other algorithms.

Fig. 12 shows the performance of the approximate implementation of the linear threshold scheduling algorithm in terms of the response time for host reads. The approximate implementation shows no degradation in response time as compared to the ideal algorithm it is based on; in fact, the performance slightly improved as a result of the approximation. This effect can be explained as follows: The approximate implementation attempts to exploit spatial locality among the destage requests in the same manner as the linear threshold algorithm, but its coarseness in estimating the destage cost makes it slightly less effective. However, this deficiency is more than offset by its increased effectiveness in exploiting the temporal locality. By using the LRU ordering to schedule a destage from the regions with the same estimated cost, the approximate implementation is able to exploit temporal locality better than the ideal algorithm. Thus, the parameters of the approximate implementation can actually be chosen to achieve a balance between spatial and temporal locality.

We close this section with some comments on how the performance of the approximate version of the linear threshold scheduling algorithm is affected by the manner in which the cost of individual destage requests is estimated. In all the simulations so far, the destage cost was computed based on the *average* positioning delay between regions on the disk. This results in the algorithm occasionally under-estimating the delay, causing the actual delay of the destage to be higher than the estimate by as much as one rotational latency. A more conservative approach would be to estimate the cost of the destage based on the *maximum* positioning delay needed to move the head to any part of the target region from any point in the current region. A third alternative is to take an aggressive approach by computing the estimate based on the *minimum* positioning delay from the current region to the target region. The three approaches are compared in Fig. 13. The comparison is based on the the actual values of the mean access time of the destages measured in the simulations. The conservative algorithm performed nearly as well as the average one, missing cheaper accesses in some cases

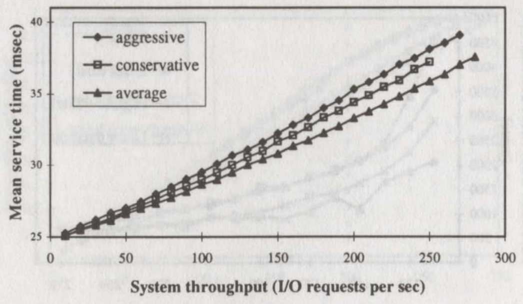


Fig. 13. Comparion of mean destage service time measured under various schemes for estimating disk service time in approximate linear threshold scheduling.

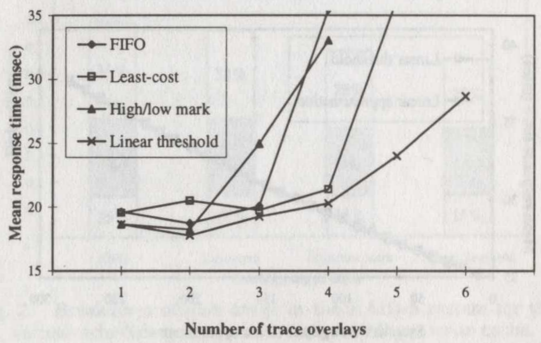


Fig. 14. Response time for host reads from trace-driven simulations.

but compensating for them by avoiding cost over-runs in others. The aggressive algorithm performed poorly compared to the other two implementations, as a result of underestimating the cost of almost every destage.

C. Results from Trace-Driven Simulations

To further validate our results from the synthetic workload, we also ran a set of simulations on a workload based on I/O traces obtained from HP Laboratories. These traces were collected from HP-UX systems during a 4-month period, and are described in detail in [12]. To obtain a workload of adequate intensity, we overlaid multiple trace files corresponding to separate days. The particular trace used was *cello*, over a 6-day period starting on April 20, 1992. The workload for our simulations was generated by first translating the disk accesses in the traces to a single logically contiguous space and mapping them to the physical disks in our RAID model. The workload intensity was varied by varying the number of overlays used.

Fig. 14 plots the average response time seen by all reads in the workload as a function of the number of overlays. The high/low mark algorithm provided nearly identical performance as the liner threshold algorithm at low loads, but diverged considerably as the workload intensity is increased. At high intensities, the bursts in the workload caused the high/low mark algorithm to overflow the cache more often than the linear threshold algorithm. Note that the plots show the average response time for all reads, not just those serviced by the disks. We observed that the read-cache hit ratios for this workload (20–30%) were even lower than that obtained with the synthetic workload; thus, the destage algorithm can significantly influence the response time seen by the user.

V. Concluding Remarks

In this paper, we studied a number of algorithms for scheduling destages in a RAID-5 system. We introduced a new scheduling algorithm, called *linear threshold scheduling* that adaptively varies the rate of destages to disks based on the instantaneous occupancy of the write cache. Our results show that linear threshold scheduling provides the best read performance of all the algorithms compared, while still maintaining a high degree of burst tolerance. An approximate implementation of the linear-threshold scheduling algorithm was also described. The approximate algorithm can be implemented at much lower overhead, yet its performance is virtually identical to that of the ideal algorithm.

In the linear-threshold scheduling algorithm, we chose a linear function to compute the threshold for destage cost as a function of cache occupancy. Other functions could be used, for example a function that increases destage rate faster as the cache occupancy increases. Their effect on performance needs to be investigated further. In addition, the scheduling algorithms could be modified to take into account several criteria we did not consider in our study, such as the spatial locality among the pending destages and the relationships between data and parity destages.

Although most of our simulation results in this paper are based on a synthetic workload, results from simulations based on a workload derived from real I/O traces show that our estimates on the performance of the linear threshold algorithm are in fact conservative. The use of caching in the operating system can cause the percentage of writes in the workload to be higher than the estimate (40%) used in our synthetic workload model. At the same time, the locality among reads can be considerably lower, making the disk read performance more critical.

The scheduling algorithms we studied can also be applied to a RAID-4 system. RAID-4 systems are not as widely used as RAID-5 because of the bottleneck due to the dedicated parity disks. This bottleneck, however, can be made less severe if a nonvolatile cache is used to buffer updates to the disk array. Since parity is stored in a dedicated disk, separate algorithms could be used to destage data and parity in a RAID-4. Since host reads do not access the parity disk during normal operation, there is usually no reason to resort to a non-work-conserving destage algorithm for the parity disk. Thus, the least-cost scheduling algorithm can be used to schedule the parity disk, while any of the algorithms in Section II can be used for the data disks. From simulations with the synthetic workload, we found the behavior of the algorithms to be very similar to that in the RAID-5. This suggests that the potential for performance gains from optimizing destages is not limited to any single disk-array configuration.

Acknowledgement

We thank John Wilkes at HP Labs for providing the I/O traces used in our simulations. Garth Gibson and the anony-

mous reviewers provided many insightful comments for improving the technical quality and presentation of the paper.

References

[1] P. Biswas, K. K. Ramakrishnan, and D. Towsley, "Trace-Driven Analysis of Caching Policies for Disks," *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 13–23.

[2] P. M. Chen, et al., "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, Vol. 26, No. 2, June 1994, pp. 145–188.

[3] P. J. Denning, "Effect of Scheduling on File Memory Operations," *AFIPS Spring Joint Computer Conference*, April 1967, pp. 9–21.

[4] G. R. Ganger, et al., "Disk Arrays: High-Performance High-Reliability Storage Subsystems," *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 30–36.

[5] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems*, February 1987, pp. 77–92.

[6] D. M. Jacobson and J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position," Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, February 1991.

[7] J. Menon and D. Mattson, "Performance of Disk Arrays in Transaction Processing Environments," *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992, pp. 302–309.

[8] J. Menon and J. Cortney, "The Architecture of a Fault-Tolerant Cached RAID Controller," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 76–86.

[9] J. Menon, J. Roche, and J. Kasson, "Floating Parity and Data Disk Arrays," *Journal of Parallel and Distributed Computing*, Vol. 17, No. 1–2, 1993, pp. 129–139.

[10] J. Menon, "Performance of RAID5 Disk Arrays with Read and Write Caching, *Distributed and Parallel Databases*, Vol. 2, No. 3, July 1994, pp. 261–293.

[11] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of ACM SIGMOD*, June 1988, pp. 109–116.

[12] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proceedings of the Winter 1993 USENIX Conference*, January 1993, pp. 405–420.

[13] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 17–28.

[14] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," *Proceedings of the Winter 1990 USENIX Conference*, January 1990, pp. 313–324.

[15] D. Stodolsky, G. Gibson, and M. Holland, "Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 64–75.

[16] B. Worthington, G. Ganger, and Y. Patt, "Scheduling Algorithms for Modern Disk Drives," *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 241–251.

[17] *The RAIDBook*, The RAID Advisory Board, Lino Lakes, Minnesota, June 1993.

Appendix A: Estimating Destage Cost for the Approximate Linear-Threshold Scheduling Algorithm

This section provides the equations we used in calculating the estimated destage cost for the approximate linear-threshold algorithm introduced in Section 2.5. We use the notation $\lfloor x \rfloor$ to denote the largest integer less than or equal to x, and $\lceil x \rceil$ the smallest integer greater than or equal to x.

The seek time, rotational latency, and controller overhead are taken into account in estimating the destage cost. The controller overhead t_{con} is taken as a constant 2.2 ms. Let $f(x)$ be the seek time of the disk between two cylinders at a distance of x. The average seek time between two regions is estimated as follows: Let c be the number of cylinders in each band and m the number of regions per band. Assume that the regions are numbered from 0 to $(c \cdot m - 1)$, as illustrated in Fig. 1(a). If two regions i and j fall in different bands, the average seek time $t_{seek}(i,j)$ between them is estimated as the seek time between the middle cylinders of the two regions. For convenience, we also include the constant controller overhead in $t_{seek}(i,j)$. That is,

$$t_{seek}(i,j) = f\left(c\left|\lfloor\frac{j}{m}\rfloor - \lfloor\frac{i}{m}\rfloor\right|\right) + t_{con}. \qquad (2)$$

When the regions i and j are within the same band, the average seek time is estimated as the time to seek across half the width of the band. That is,

$$t_{seek}(i,j) = f(c/2) + t_{con}. \qquad (3)$$

Let t_{rot} be the rotational latency of the disk in milliseconds. Since the scheduling decisions are made at the granularity of a region, the destage cost is estimated in units of the rotational latency of a region on the disk, that is, $1/m$ of the disk rotation time.

When the regions i and j are at the same rotational offset on the disk, the destage cost from i to j is the seek time plus controller overhead, rounded up to whole revolutions. That is,

$$cost(i,j) = \left\lceil \frac{t_{seek}(i,j)}{t_{rot}} \right\rceil m,$$

expressed in units of the rotational latency of a region. When the regions i and j are not at the same rotational offset on the disk, the relative rotational positions of i and j need to be taken into account. This is accomplished as follows: We can express the seek time $t_{seek}(i,j)$ as

$$t_{seek}(i,j) = \alpha t_{rot} + \beta,$$

where α is an integer and $0 \leq \beta < t_{rot}$. α represents the number of whole revolutions made by the disk during the seek time and β the fractional part. If the fractional part is greater than the rotational offset from i to j, then region j can be accessed only during the $(\alpha+1)$th revolution of the disk. In this case, the total cost of the destage, including both seek time and rotational latency is given by

$$cost(i,j) = (\alpha+1)m + (j-i) \bmod m, \quad \text{if } \frac{m\beta}{t_{rot}} > (j-i) \bmod m. \qquad (4)$$

If the fractional part is less than or equal to the rotational offset, the extra revolution is saved. The cost is then given by

$$cost(i,j) = \alpha m + (j-i) \bmod m, \quad \text{if } \frac{m\beta}{t_{rot}} \leq (j-i) \bmod m. \qquad (5)$$

Thus, the general equation for the estimated cost of a destage from region i to region j is given by

$$cost(i,j) = \begin{cases} \left(\lfloor\frac{t_{seek}(i,j)}{t_{rot}}\rfloor + 1\right) m + (j-i) \bmod m, \\ \quad \text{if } m\left(\frac{t_{seek}(i,j)}{t_{rot}} - \lfloor\frac{t_{seek}(i,j)}{t_{rot}}\rfloor\right) > (j-i) \bmod m; \\ \lfloor\frac{t_{seek}(i,j)}{t_{rot}}\rfloor m + (j-i) \bmod m, \quad \text{otherwise.} \end{cases} \qquad (6)$$