Disk Arrays
High-Performance, High-Reliability Storage Subsystems

Gregory R. Ganger, Bruce L. Worthington, Robert Y. Hou, and Yale N. Patt
University of Michigan

As the performance of other system components continues to improve rapidly, storage subsystem performance becomes increasingly important. Storage subsystem performance and reliability can be enhanced by logically grouping multiple disk drives into disk arrays. Array data organizations are defined by their data distribution schemes and redundancy mechanisms. The various combinations of these two components make disk arrays suitable for a wide range of environments. Many array implementation decisions also result in trade-offs between performance and reliability.

Disk arrays are an essential tool for satisfying storage performance and reliability requirements. Proper selection of a data organization can tailor an array to a particular environment.

Data organization

Data organization for disk arrays can be partitioned cleanly into two orthogonal components: data distribution schemes and redundancy mechanisms. Data distribution defines the translation from externally visible logical addresses to storage locations within the subsystem. Two ways to do this are by independently addressing each disk (the conventional approach) or by disk striping. The latter scheme often provides improved levels of load balancing, data-transfer parallelism, and/or access concurrency. Redundancy mechanisms specify the type, scope, and location of any redundant information within the array. Examples include data replication (disk mirroring) and striped parity. Redundant disk arrays continue to provide full access to data after a disk failure and while lost data is being reconstructed on a replacement disk. Many disk arrays combine striping and redundancy to provide both high performance and high reliability.

Disk array products usually provide software or firmware that lets the customer choose among several data organization schemes. While this improves an array's usability and increases its marketability, system administrators must understand the trade-offs involved to fully exploit the array's capabilities.

Data distribution. Data distribution consists of mapping the logical addresses used by the host onto the disks in the subsystem. Figure 1 shows the three main methods for performing this translation. The conventional approach is to address each disk independently and map logical block numbers to disk block numbers directly. The distribution is performed "manually" by system administrators, application programs, or system software. *Disk striping*, also called disk interleaving, folds multiple disk address spaces into a single, unified space seen by the host. This is accomplished by distributing consecutive logical data units (called *stripe units*) among the disks in a round-robin fashion, much like interleaving in a multi-

0018-9162/94/$4.00 © 1994 IEEE

bank memory system. We distinguish between two distinct types of disk striping. *Fine-grained* striping (Figure 1b) distributes the data so that all of the array's disks cooperate in servicing every request. *Coarse-grained* striping (Figure 1c) allows the disks to cooperate on large requests and service small requests independently.

Independent addressing. If each disk is independently addressed (see Figure 1a), users and/or host applications must explicitly distribute the data. Generally, system administrators are responsible for deciding which data sets to place on each disk. Load balancing, or distributing the data so as to balance the workload among the disks, is more an art than a science and often requires specialists. Biased data-reference patterns (disk skew) create "hot spots," making the task complex. Many corporations are looking to software products to automate load balancing. Given the problem's intractability, however, a more fundamental change may be necessary.

Fine-grained striping. In fine-grained striping, all N disks in the array contain a fraction of each accessible block (see Figure 1b). The number of disks and the stripe unit size are generally chosen such that their product evenly divides the smallest accessible data unit (from the host's point of view). Popular stripe unit sizes for fine-grained striping include one bit, one byte, and one disk sector (often 512 bytes). The size used is not really important as long as each accessible unit is spread among all of the disks. The load is perfectly balanced, since all disks receive identical workloads. The effective transfer rate approaches N times that of an individual disk, as each disk transfers $1/N$ of the requested data. However, the positioning components of the access time — seek and rotation — are either increased or unaffected (if the disks are synchronized). Also, only one request can be serviced at a time, because all N disks work on each request. These limitations usually restrict fine-grained striping to environments in which transfer times dominate service times.

Coarse-grained striping. With coarse-grained striping (see Figure 1c), it isn't necessary for all disks to cooperate on every request. This technique exploits data-transfer parallelism for large requests while allowing separate disks to handle small requests concurrently. Several effects combine to provide automatic load balancing. First, a hot file will tend to have its component blocks distributed over multiple disks, thereby spreading requests for its data. Course-grained striping, like hashing, essentially randomizes the first disk accessed for each request,[1] with additional disks identified in a round-robin fashion. This statistically balanced load remains intact under highly biased reference patterns and rapidly changing access distributions.[2] For the above reasons, coarse-grained striping has the potential to provide high performance with a minimum of maintenance.

Choosing the stripe unit size. Achieving the performance potential of coarse-grained striping requires correct selection of the stripe unit size. This choice largely determines the number of disks over which each request's data is spread. Large stripe units allow separate disks to handle multiple requests concurrently, reducing overall positioning delays. This often results in reduced queuing delays and higher throughput. On the other hand, small stripe units cause multiple disks to access data in parallel, reducing data transfer times for individual requests. This trade-off between transfer parallelism and access concurrency is governed by the stripe unit size.

For workloads with highly variable request sizes and no sequentiality or locality of reference, the key parameter is the amount of access concurrency in the workload. The stripe unit size should increase with the arrival rate of requests, thereby decreasing the average number of disks accessed per request.[3]

For workloads consisting primarily of aligned, fixed-size requests, the stripe unit size can be chosen such that the number of disk accesses per request is a constant. In particular, using any multiple of the fixed request size allows every request to be serviced by a single disk. Examples of such workloads are those generated by traditional Unix file systems.

For workloads containing some degree of sequentiality and/or locality of reference, load balancing must also be considered. When requests are large, load balancing does not significantly affect the choice of stripe unit size. For light workloads, larger stripe units exploit the positioning-time benefits of sequentiality and locality of reference.

A more complex trade-off exists for general-purpose, commercial workloads, which are often characterized by small requests, bursts of high activity, and biased data-reference patterns. With such workloads, the ability of coarse-grained striping to load-balance an array improves as the stripe unit size decreases. Larger stripe units tend to suffer from the formation of short-term hot spots. Bates suggests that the stripe unit should be approximately 10 times the average request size, rounded up to a multiple of the track size (based on studies using real VAX/VMS workloads).[1] The choice of stripe unit size can be refined by considering additional information, such as request size distributions and locality characteristics.

One interesting question is whether the stripe unit size should be a multiple of the track size. On the one hand, a

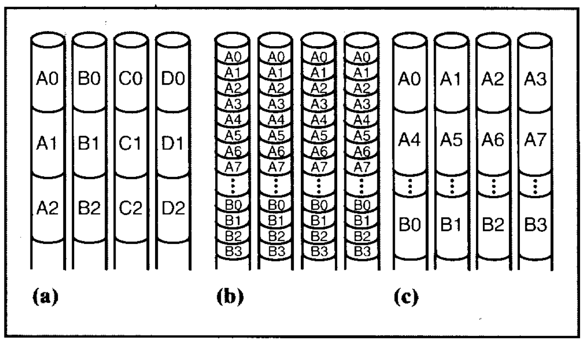


Figure 1. Alternative data distribution schemes: (a) independent disks, (b) fine-grained striping, (c) coarse-grained striping. Each column represents physically sequential blocks on a single disk. In the case of independent disks, A0 and B0 represent blocks of data in separate disk address spaces.

request for a full stripe unit can take advantage of a disk supporting zero-latency access. On the other, the track size may correspond poorly to request alignment and size, resulting in excessive multidisk requests. Many workloads consist of block requests of a specific size, say, 4 Kbytes or 8 Kbytes. Current track sizes are multiples of the sector size — often 512 bytes — chosen to maximize capacity rather than match alignment boundaries. Most state-of-the-art disk drives use multiple-zone recording to increase storage density.[4] When using such disks, matching the stripe unit to the track size requires a different stripe unit for each disk zone. More performance studies are necessary before track-based striping can be identified as a good design choice in general.

Redundancy mechanisms. Adding redundancy to a storage system invariably increases its cost, reduces its capacity, and/or degrades its performance. Instead, many systems make periodic backup copies of critical data, usually on magnetic tape. Any data corrupted or lost from the primary system is restored from the latest backup copy, and any updates that occurred after the backup are irrecoverable. Fortunately, individual disk failures are infrequent, and total subsystem failure is quite rare. The mean time to failure (MTTF) for a modern disk drive is in the range of 200,000 to 1 million hours, or roughly 20-100 years.[4]

However, as the number of disk drives continues to grow, the MTTF of a typical nonredundant subsystem shrinks to months or weeks. Assuming that the MTTF of each disk is independent and exponentially distributed,[5] the MTTF of a group of disks is inversely proportional to the number of disks. This threatens the feasibility of nonredundant disk systems.

In addition, the number of files affected per failure increases dramatically with the use of disk striping. Rather than losing $1/N$ of the files in an array of N disks, as occurs if the disks are independently addressed, a striped subsystem could lose $1/N$ of every file. A subsystem incorporating redundancy into the data organization can survive the failure of one or more components. However, additional (critical) failures result in the loss or inaccessibility of data. As storage subsystems grow, online redundancy will be required for crucial data, possibly becoming standard for general storage as well.

Data replication. The simplest way to protect data is through replication. Separate copies of each data block are maintained on D (two or more) separate disks. Data becomes unavailable only if all disks containing a copy fail. This form of redundancy, widely used for many years, is known as disk mirroring, disk shadowing, disk duplexing, and (recently) RAID1. The cost of replicating data is essentially proportional to $D - 1$, as disk cost currently dominates the price of large storage subsystems. Since two copies provide high reliability, additional copies are rarely used. In fact, the high cost of full data replication often compels the use of other redundancy mechanisms that do not provide the same level of performance and reliability.

An array data organization has two orthogonal components: a data distribution scheme and a redundancy mechanism.

Maintaining multiple copies of data affects performance in several ways. First, writes must be performed on all copies. Depending on the implementation, this may create opportunities for data loss or increase observed response times (discussed later). Read performance, on the other hand, benefits from additional copies. First, D read requests can be serviced simultaneously, increasing throughput and decreasing queue times in most environments. In addition, a given read request can be scheduled to the disk on which it will experience the smallest access delay. The improved read performance can actually make the cost/performance ratio lower than that of nonredundant disk systems, even though the absolute cost is multiplied by D.

The conventional method of implementing data replication "mirrors" each disk (see Figure 2a). Every set of D identical disks can survive $D - 1$ failures without data loss. In an array with multiple replicated sets, many failures can occur without causing data loss, as long as D of them do not occur within a single replicated set. However, each disk failure reduces the performance of a set relative to other sets. This imbalance can reduce the array's overall performance if a damaged set becomes a bottleneck.

Alternately, sets may overlap such that this performance degradation is shared equally by all surviving disks. The term *declustering* refers to a method for combining multiple replicated sets. *Chained* declustering[6] divides each disk into D sections. The first section contains the primary copy of some data, with each disk having different primary data. When we view the disks as points on a circle, the secondary copy of this data appears in the second section of the next disk, and so on (see Figure 2b). With clever scheduling, chained declustering can maintain a balanced load after a disk failure, though the probability of surviving multiple failures is reduced.

A second type of declustering, known as *interleaved* declustering, also partitions each disk but stripes the nonprimary copies across all the disks instead of keeping them in contiguous chunks (see Figure 2c). This balances the additional load caused by a disk failure but further reduces reliability (losing any D disks results in critical failure).

Parity-based protection. Disk subsystem architects have taken advantage of results from extensive studies of error-detecting and -correcting codes in other fields. From work in coding theory, we know that a single erasure (a lost bit) can be recovered from a parity bit, the other protected bits, and knowledge of the erasure. Because current disk system control logic can generally detect and identify disk failures, parity can be used to protect a set of disks from a single failure. Each bit of parity information is calculated from an associated data bit on each protected disk and stored on a separate disk. The locations of the particular data bits protected by a given parity bit may vary from scheme to scheme, as may the placement of the parity. The additional capacity (one full disk) required to maintain redundancy via parity is considerably less than what is needed for data replication.

Although the cost is reasonable, parity maintenance can reduce array performance significantly. Read requests are handled as in a nonredundant array. If a disk is added to provide storage capacity for parity, it provides an additional server for read requests (in most parity placement schemes). Write requests require additional disk accesses to update the parity, often leading to very poor performance. If all data bits associated with a parity bit are updated by a write request, the new parity bit is calculated and included as part of the total request. If only a subset of the data bits is being written, updating the parity bit requires additional work.

This additional work consists of reading bits from the disks to construct the new parity bit. The two main approaches are denoted Read-Modify-Write and Regenerate-Write. With RMW, the new parity bit is constructed from the old value of the parity bit and the old and new values of the data bits being written. The other approach, Regenerate-Write, generates the new parity bit from the values of all data bits (that is, the new values of the data bits being updated and the current values of the unchanged data bits). Therefore, RMW requires initial read accesses to all disks being updated (including the disk containing the parity), while Regenerate-Write requires read accesses to all disks not being updated. The performance penalty for maintaining parity can be reduced by dynamically choosing between these options.

Because of the need to perform disk reads before updating the parity, performance can be significantly lower than in an unprotected system or a system protected by data replication. Removing or reducing this performance degradation is an open research topic. In a subsequent section we discuss some of the options being explored.

Three major schemes exist for spreading the parity information among the disks in the subsystem. The straightforward approach is to place all parity on a dedicated *parity disk* (see Figure 3a). While this simplifies the mapping of logical addresses to disk addresses, every write must update the associated bits on the single parity disk. Arrays that use fine-grained striping are well suited for this method, since they handle only one request at a time. With other data distribution schemes, however, the parity disk can limit performance, especially when the old parity is required to participate in RMW operations. To avoid this potential bottleneck, the parity information can be striped among the disks (see Figure 3b). By using *striped parity*, the array can perform multiple parity updates in parallel.

A third option has been proposed: *declustered parity*.[7,8] One way to view this scheme is to imagine combining (into one large array) several smaller arrays protected by striped parity. The data and parity of each array are distributed throughout the entire set of available disks (see Figure 3c). With a clever distribution algorithm, the performance impact of losing a disk can be shared equally by all disks. Although the array can survive only a single failure, it should recover from such a disk failure more quickly. Alternatively, one can view parity declustering as a way to achieve higher reliability and failure-mode performance by increasing the quantity of parity data maintained.

Other schemes. Maintaining parity protects an array only against the loss of a single disk. An array with two copies of all data can survive up to one failure per pair of copies. In both cases, data may be lost if a second failure occurs before the first is repaired. Providing unconditional protection against multiple failures requires additional redundancy. In the most promising approach, two different Reed-Solomon erasure-correcting codes (for example, parity) are maintained. With such redundancy, an array can reconstruct all data after losing access to any two disks. The performance trade-

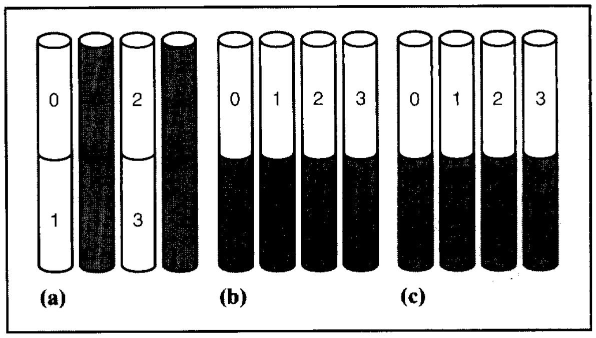


Figure 2. Alternative dual-copy data replication schemes: (a) traditional mirroring, (b) chained declustering, (c) interleaved declustering. Shaded regions represent secondary copies. Numerals 0-3 each represent a region equal in size to half of each disk's available storage. For interleaved declustering, 0a, 0b, and 0c represent the three component pieces of the second copy of 0.

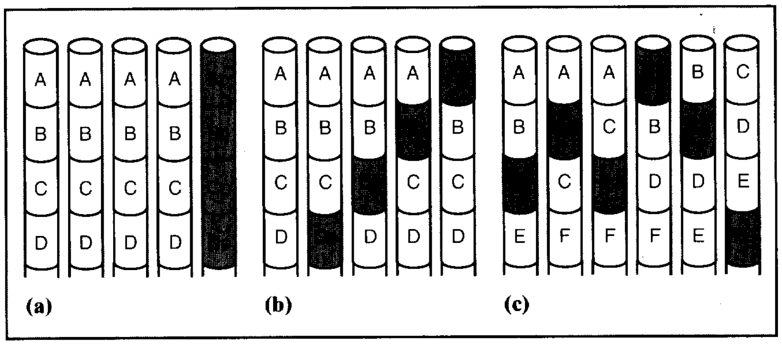


Figure 3. Alternative parity protection schemes: (a) parity disk, (b) striped parity, (c) declustered parity. Ap represents the block of parity bits protecting the data bits in the blocks labeled A (size unspecified).

offs and redundancy placement options match those for parity-protected arrays, except that two redundant bits must be maintained rather than one.

Other schemes, such as multidimensional parity and more aggressive error-correcting codes, can protect against greater numbers of failures.[5] However, the cost and performance implications, combined with the fact that storage device reliability is increasing dramatically, may preclude their use.

The sidebar below summarizes various data organization schemes.

Performance versus reliability

Write requests may encounter lengthy delays between their initiation and the actual writing of new data to stable storage. Until the host accepts a request completion message, any associated system resources (for example, memory pages) cannot be reused. Also, application processes may wait for such acknowledgments, possibly resulting in idle processor cycles. By accepting confirmation as early as possible, the host can better utilize its resources and have fewer idle cycles, resulting in higher system performance. However, if request completion is signaled before the data

A summary of data organization schemes

The table shows a matrix of alternative data organizations, with the columns representing data distribution schemes and the rows representing redundancy mechanisms. Each matrix entry contains names or references for previously proposed and/or implemented disk array organizations. Unlikely combinations of data distributions and redundancy mechanisms are labeled U/C.

The acronym RAID (for redundant array of inexpensive disks) was introduced in 1988 with a partial taxonomy of logical data organizations.[1] Industry has augmented this taxonomy. In many circles, the acronym's meaning has been redefined to redundant array of *independent* disks, since the disks used in such arrays are often state of the art and seldom inexpensive. In some sense, the disks are really *interdependent.* A RAID "level" simply corresponds to a specific combination of data distribution scheme and redundancy mechanism. The ordering of the levels carries no consistent meaning and does not represent a measure of goodness. The data organizations corresponding to current RAID levels are identified in the table.

We should also point out that several "new (?)" RAID levels that do not correspond to new data organizations have recently been introduced by manufacturers to identify their products. We have not included these in the table.

References

1. D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD, Int'l Conf. Management of Data,* ACM Press, New York, May 1988, pp. 109-116.

2. J. Gray, B. Horst, and M. Walker, "Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput," *Proc. 16th VLDB Conf.,* Morgan Kaufmann, Palo Alto, Calif., Aug. 1990, pp. 148-161.

3. M. Kim, "Synchronized Disk Interleaving," *IEEE Trans. Computers,* Vol. C-35, No. 11, Nov. 1986, pp. 978-988.

4. M. Livny, S. Khoshafian, and H. Boral, "Multidisk Management Algorithms," *Proc. SIGMetrics,* ACM Press, New York, 1987, pp. 69-77.

5. N.K. Ouchi, "System for Recovering Data Stored in Failed Memory Unit," US Patent #4,092,732, May 30, 1978.

6. M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," *Proc. Fifth ASPLOS Conf. (Architectural Support for Programming Languages and Operating Systems),* ACM Press, New York, 1992, pp. 23-35.

7. S.W. Ng and R.L. Mattson, "Maintaining Good Performance in Disk Arrays During Failure via Uniform Parity Group Distribution," *Proc. First Int'l Symp. High-Performance Distributed Computing,* IEEE CS Press, Los Alamitos, Calif., Order No. 2970, 1992, pp. 260-269.

A matrix of data organizations.

Data Organizations		Independent Addressing	Fine-Grained Striping	Coarse-Grained Striping
No redundancy		Conventional	Reference 3	RAID0; Reference 4
Replication	Two copies	Mirroring; RAID1	—	RAID0+1
	N copies	Shadowing	—	—
	Declustered	Chained; Interleaved	U/C	—
Parity	Disk	—	RAID3; Reference 3	RAID4; Reference 5
	Striped	Reference 2	U/C	RAID5
	Declustered	—	U/C	References 6 and 7
Two Reed-Solomon codes		—	—	RAID5+; RAID6
Hamming code (ECC)		—	RAID2	—

reaches permanent media, the system is vulnerable to data loss/corruption due to power failure or component breakdown. When the host relaxes reliability constraints to improve performance, other precautionary measures must be taken or the consequences must be accepted.

Trading reliability for performance. An array controller with a cache (or buffer) can signal completion to the host as soon as the last byte of a write request has been copied from main memory. Such a scheme dramatically improves write performance as observed by the host, though the data is vulnerable until written to disk. The disk scheduler, which is responsible for determining the order of service for pending disk accesses, now has a problem. Reliability goals require the scheduler to expedite disk accesses that transfer such cached data to stable storage, while performance goals compel the scheduler to postpone such "background" activity.

In arrays using data replication, reporting completion after only a fraction of the copies have been updated can similarly improve performance with little vulnerability. Data is lost only if multiple failures occur before the remaining writes are completed. For parity-protected arrays, caching parity may improve performance by accumulating the "new" parity for multiple writes and performing the parity updates in a single write. However, some data remains unprotected until this parity update is completed.

In parity schemes requiring an update cycle, the system is vulnerable to data corruption if the set of disk writes do not occur simultaneously. If power fails at an inopportune moment, only some of the accesses will complete. This would make the parity incorrect, causing further corruption if left uncorrected and subsequently used in data reconstruction. On the other hand, enforcing simultaneous writes necessitates synchronous behavior from multiple disks. This may produce suboptimal request schedules and increased disk idle time. If the controller schedules the individual write requests independently, disk utilization can be maximized.

Improving performance and reliability. Most users find any possibility of data loss or corruption unnerving, if not totally unacceptable. Several techniques exist to improve both write performance and reliability. They accomplish this by reducing write latencies while guaranteeing the safety of old data and ensuring that new data is secure before signaling completion. These schemes can be implemented independently by the host system or within the disk array. One simple (but expensive) solution is to use nonvolatile RAM (NVRAM) to cache write data and/or parity. NVRAM maintains state even when the power supply is interrupted. The NVRAM's organization should provide reliability guarantees equivalent to the array's redundancy scheme.

Write remapping, a form of shadow paging, has been proposed for many different data organizations. Write remapping consists of dynamically altering the logical-to-physical mappings so that write request data is always placed at new (free) locations. By maintaining the previous copy of the logical block until the request completes, write remapping improves reliability. Of course, the mappings must also be maintained reliably to prevent data loss. One approach is to store this information in NVRAM. A more complex but less expensive option is to augment each physical data block with the corresponding logical address and a time stamp, which can be used to reconstruct the mappings.

By choosing free locations near the current position of the disk's read-write head, write remapping can also reduce costly seek and rotation delays. The algorithm for choosing among available locations must prevent this greedy approach from consuming all free locations in some regions of the disk and leaving other regions unused. Also, over time, logically sequential data can become scattered throughout the physical storage space. As a result, subsequent read requests may suffer extensive positioning delays unless some type of relocation is performed (during periods of low activity) to restore physical sequentiality.

A third option, logging, writes information about updates to a log area before performing them. Log writes are sequential and usually experience low response times, particularly when using a dedicated log disk. After writing the log entry, the array controller can safely send a completion message to the host and perform the disk accesses in the background. In case of failure, the contents of the log allow any incomplete updates to be repeated. The information written to the log may consist of a brief description of the update, a copy of the new data, or copies of both the new and old data. Generally, reliability increases with the amount of information saved. Some form of checkpointing is normally used to reduce recovery time and allow reuse of log space.

Dealing with failures. By maintaining redundancy, a disk array can survive disk or path failures that make one or more disks inaccessible. While the array continues to provide access to all data, its performance is almost always affected by failures. In addition, reliability is reduced, since additional failures may be critical. In recovering from a disk failure, the lost data must be reconstructed from the surviving disks and written to a replacement disk, a process referred to as *rebuild*. Until this process completes, the disk array is vulnerable. Extra disks, called *hot spares*, are often included in arrays to allow rebuild to begin immediately.

In most redundancy schemes, losing access to a disk reduces the number of concurrent read requests that can be serviced. Arrays using data replication handle requests as usual, albeit with one less copy to service reads. Also, write requests update one less copy. In parity-protected arrays, a read request to a failed disk requires regeneration of missing data. Each lost data bit can be reconstructed from the associated parity bit and the other data bits contributing to the parity. Therefore, handling the read request necessitates accesses to all of the disks containing these bits. Write performance is also affected in parity-protected arrays. Because the old data cannot be accessed, Regenerate-Write must be used for all updates to data on the failed disk and Read-Modify-Write for all updates to surviving disks.

The simplest form of rebuild sequentially reconstructs the unavailable data and writes it to the replacement disk. In a replicated array, the "lost" data is copied from another disk containing the same data. In a parity-protected array, the data must be regenerated from the parity and data bits (as described above). If reliability requirements specify that repair times be minimized, host requests can be denied while rebuild proceeds at full speed; otherwise, rebuild and host requests compete for service, resulting in another trade-off between reliability and performance. This trade-off is particularly important in rebuilding parity-protected arrays, where data regeneration requires access to multiple disks. The choice of a rebuild algorithm, the size of the individual rebuild requests, and the multidisk coordination overhead must be considered when tuning the rebuild process.[9]

The disk array market is expanding as the cost of disk drives decreases and typical storage capacity needs grow. As a result, disk arrays are becoming standard equipment on a wider range of machines. Someday an average user's secondary storage requirements may be satisfied without the use of moving mechanical parts. Until then, disk arrays can provide the necessary levels of reliability and performance.

If storage subsystems continue developing as they have been, the level of intelligence incorporated into their various components will continue to increase. Communication between these components will increase dramatically, resulting in global optimizations, more distributed control, and new algorithms that respond dynamically to the constantly changing flow of disk requests. New protocols and interconnection schemes will be required to maintain coherency between multiple caches and buffers, as well as to coordinate the request streams from multiple hosts sharing the subsystem's resources. Recent improvements in magnetic media and sensing technologies add to the life expectancy of magnetic disk drives, and therefore of disk arrays.

In short, we expect disk arrays to be the implementation mechanism of choice for secondary storage subsystems for some time to come. ■

Acknowledgments

We wish to thank Jim Browning, John Fordemwalt, Jim Gray, David Jaffe, Richie Lary, Mike Leonard, Dick Mattson, Jai Menon, Spencer Ng, Joe Pasquale, Rusty Ransford, Chris Ruemmler, and John Wilkes for their insights on various I/O issues. We also appreciate our daily interaction with the other members of our research group at the University of Michigan. Finally, our group is very fortunate to have the financial and technical support of several industrial partners, including NCR, Intel, Motorola, SES, HaL, DEC, Micro Technology Inc., and Hewlett-Packard.

References

1. K. Bates, *VAX I/O Subsystems: Optimizing Performance*, Professional Press Books, Horsham, Penn., 1991.

2. G. Ganger et al., "Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement," *Proc. 26th Hawaii Int'l Conf. System Sciences*, Vol. 1, IEEE CS Press, Los Alamitos, Calif., Order No. 3230, 1993, pp. 40-49.

3. P. Chen and D. Patterson, "Maximizing Throughput in a Striped Disk Array," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 322-331.

4. C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, Vol. 27, No. 3, Mar. 1994, pp. 17-28.

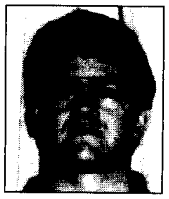
5. G. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, PhD dissertation, TR UCB/CSD 91/613, Univ. of California, Berkeley, Dec. 1990.

6. H. Hsiao and D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines," *Proc. Sixth Int'l Conf. Data Engineering*, IEEE CS Press, Los Alamitos, Calif., Order No. 2025, 1990, pp. 456-465.

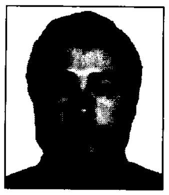
7. M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," *Proc. Fifth ASPLOS Conf. (Architectural Support for Programming Languages and Operating Systems)*, ACM Press, New York, 1992, pp. 23-35.

8. S.W. Ng and R.L. Mattson, "Maintaining Good Performance in Disk Arrays During Failure via Uniform Parity Group Distribution," *Proc. First Int'l Symp. High-Performance Distributed Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2970, 1992, pp. 260-269.

9. R. Hou and Y. Patt, "Comparing Rebuild Algorithms for Mirrored and RAID5 Disk Arrays," *Proc. ACM SIGMOD, Int'l Conf. Management of Data*, ACM Press, New York, May 1993, pp. 317-326.



Gregory R. Ganger is a PhD candidate at the University of Michigan. He is currently on leave from the university, working in DEC's Digital Storage Laboratories. His research interests include the design and performance evaluation of storage subsystems, from the file system(s) to the individual devices. He is also interested in overall system performance issues, including operating systems and their interaction with various hardware resources. Ganger received BS and MS degrees in computer science and engineering from the University of Michigan.



Bruce L. Worthington is a PhD candidate at the University of Michigan. He is currently on leave from the university, working in Hewlett-Packard's Storage Systems Division in Boise, Idaho. His research interests include distributed disk scheduling and storage subsystem design. He received a BS in computer science, computer engineering, and mathematics in 1987 from Graceland College, Lamoni, Iowa, and an MS in computer science and engineering in 1990 from the University of Michigan.

Robert Y. Hou is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include high-performance computer architectures and computer system performance analysis, especially for database and file servers. He has spent summers at NCR Columbia and the IBM Almaden Research Center. He received a BS in electrical engineering from the University of California, Berkeley, in 1987 and an MS in electrical engineering (solid-state electronics) from the University of Michigan in 1991.

Yale N. Patt is the guest editor of this special issue. His biography and photo appear on page 16.

The authors can be contacted at the Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122.

ARTICLE SUMMARIES

An Introduction to Disk Drive Modeling, pp. 17-28

Chris Ruemmler and John Wilkes

If I/O systems are to keep pace with current microprocessor technology, disk drive performance becomes more critical and must be better understood. This means that high-quality disk drive models are needed, and these models must be based on correct assumptions about disk drive behavior.

The authors describe the various components of modern disk drives, emphasizing how each affects performance. This overview, which includes recording components, positioning components, and the disk controller, covers how each component and its functions contribute to the current state of the art in disk drive technology.

Having evaluated the many performance factors, the authors go on to show how they create accurate disk drive simulation models. They present graphs and tables that show how adding features to the model leads to more accurate performance figures. Data caching characteristics turn out to be the most important feature to model, followed by the data transfer model and the seek-time and head-switching costs.

Disk Arrays: High-Performance, High-Reliability Storage Subsystems, pp. 30-36

Gregory R. Ganger, Bruce L. Worthington, Robert Y. Hou, and Yale N. Patt

Storage subsystem performance and reliability can be enhanced by logically grouping multiple disk drives into disk arrays. However, many combinations of data distribution schemes and redundancy mechanisms are possible in an array data organization, so trade-offs between performance and reliability must be considered.

The authors explain how *disk striping* is used to distribute data in the array and how the choice of stripe unit size affects performance. They also discuss data replication and parity-based protection, focusing on their implementation in disk arrays as a way to deal with disk failure. Should a failure occur, lost data can be reconstructed from the surviving disks. Of course, performance is usually affected while this *rebuild* is taking place, and an additional failure at this time could be critical.

Striking the right balance between performance and reliability is an important consideration for users, and techniques are available for improving both. These include the use of nonvolatile RAM, write remapping, and logging. The promise of additional developments in disk drives suggests that disk arrays will become the implementation mechanism of choice for secondary storage subsystems in the near future.

Caching Strategies to Improve Disk System Performance, pp. 38-46

Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry

Processor speeds have increased dramatically over the last few years and are expected to continue to double every year, with main memory density doubling every two years. At the same time, our appetite for I/O continues to grow, especially with the emergence of applications such as multimedia and scientific modeling.

Although rapid technological advances have doubled disk capacity every 1-1/2 years since 1990, no similar advances are expected to reduce mechanical latency and, thus, the access times of storage devices; access times of main memory and storage devices will likely remain many orders of magnitude apart. As a result, I/O subsystems limit overall system response time and leave the CPU underutilized.

This article examines the use of caching as a means to increase system response time and improve the data throughput of I/O subsystems. The authors describe a number of cache parameters that affect cache design and performance, including cache size, various popular line-replacement algorithms, read-ahead and write-to-cache strategies, and cache location.

Using simulations, the authors analyze the effectiveness of three cache replacement algorithms for workload traces from four different systems. Using an algorithm with low implementation overhead, known as segmented least recently used (SLRU), the authors find that in certain circumstances SLRU halves the cache size required by other popular caching strategies.

A Systematic Approach to Host Interface Design for High-Speed Networks, pp. 47-57

Peter A. Steenkiste

Optical fiber has made it possible to build networks with link speeds exceeding a gigabit per second; however, these networks are pushing end systems to their limits. For high-speed networks (100 Mbits per second and up), network throughput is typically limited by software overhead on the sending and receiving hosts.

This article describes how software optimization and hardware support on the network adapter can reduce overhead and improve throughput. The author begins with a typical network interface and systematically adds software optimizations, such as buffered communication primitives and checksum during copy, as well as hardware