A Tutorial on RAID Storage Systems

Sameshan Perumal and Pieter Kritzinger

CS04-05-00
May 6, 2004

Data Network Architectures Group
Department of Computer Science
University of Cape Town
Private Bag, RONDEBOSCH
7701 South Africa
e-mail: dna@cs.uct.ac.za

Abstract

RAID storage systems have been in use since the early 1990's. Recently, however, as the demand for huge amounts of on-line storage has increased, RAID has once again come into focus. This report reviews the history of RAID, as well as where and how RAID systems fit in the storage hierarchy of an Enterprize Computing System (EIS). We describe the known RAID configurations and the advantages and disadvantages of each. Since the focus of our research is on the performance of RAID systems we devote a section to the various factors which affect RAID performance. Modelling RAID systems for their performance analysis is the topic of the next section and we report on the issues as well as briefly describe one simulator, RAIDframe, which has been developed. We conclude with section which describes the current open research questions in the area.

1 Introduction

The concept of Redundant Arrays of Inexpensive Disks (RAID) was introduced about two decades ago and a RAID taxonomy was first established by Patterson [Patterson et al. 1988] in 1988. The taxonomy outlines the theory of RAID systems and provides several schemes to utilize the capabilities in varying ways. These RAID levels describe data placement and redundancy strategies for different applications, as well as the benefits of each level. RAID overcomes the capacity limitations of commodity disks by treating an array of such low-capacity disks as one large Single Large Expensive Disk (SLED). Such arrays offer flexibility over SLED, in that capacity can be increased incrementally, and as desired, by simply adding more disks to the array.

Commodity storage disks currently offer a number of benefits:

1. They are inexpensive, with approximately the same or lower cost per byte than SLED's;
2. Individual disks has a Mean Time To Failure (MTTF) rate comparable to most SLED's;
3. They require far less power;
4. They conform to a uniform access standard, typically SCSI or IDE, and usually
5. they have built in controller logic which performs both error detection and correction functions.

Since each individual disk in a RAID system consumes little power, and the cost of replacement of a single disk is small compared to the overall cost, RAID systems have low maintenance costs. Finally, given that each disk in the array, depending on the logical arrangement, can perform transfers independently of the others the potential for faster transfers through the exploitation of parallelism exists.

2 Overview

The two main hardware architecture factors Dominating the performance of RAID systems are

- *bandwidth* and *parallelism*, both of which are influenced heavily by storage caches.
- Data placement on the disks, and
- The way blocks belonging to a single file are places across the various disks, know as *striping policies*.
- Workload or the storage access pattern of an application as we discuss in Section 4.1.

The following figure illustrates a typical EIS secondary storage hierarchy which includes a RAID system and which we shall refer to later in this report.

2.1 EIS Storage Hierarchy

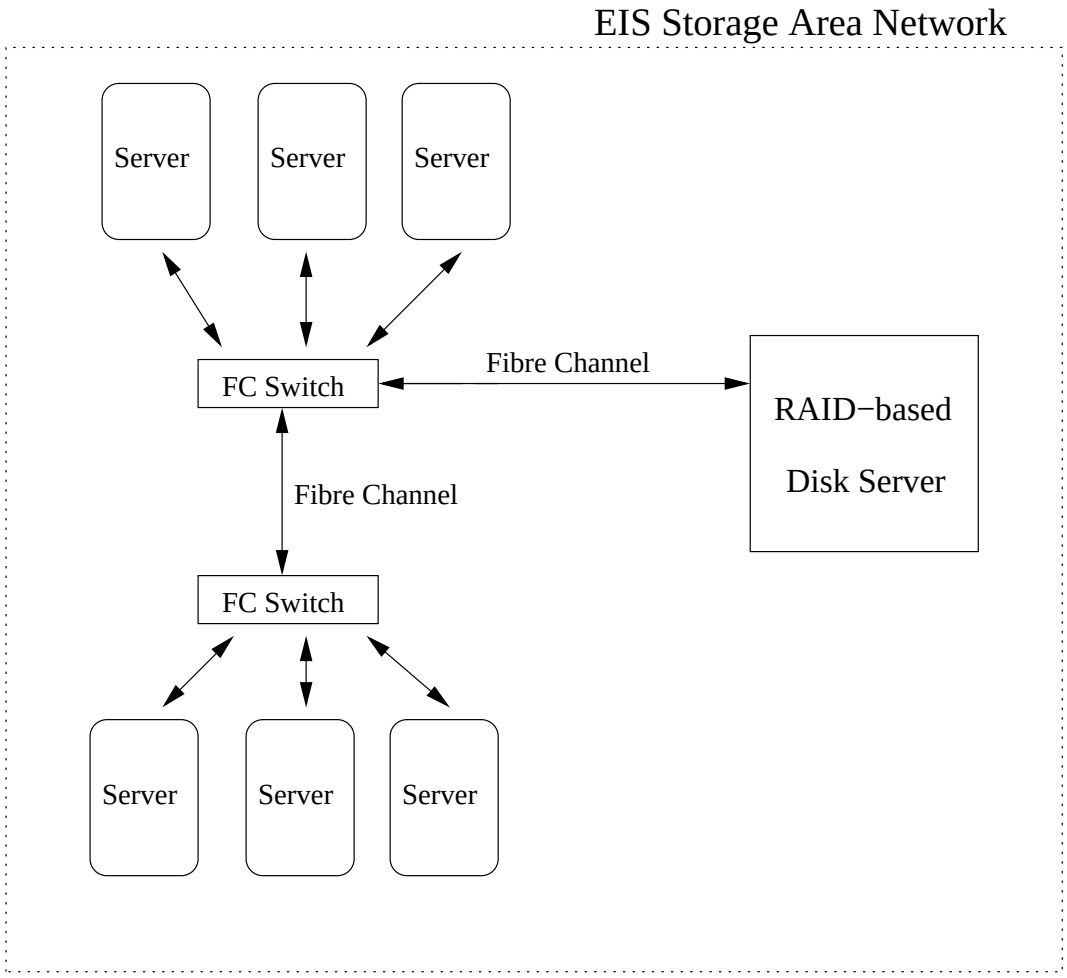

Figure 1: A typical EIS secondary storage hierarchy which includes a RAID system

2.2 RAID Taxonomy

A possible approach is to place each bit on a different disk - however reads occur in multiples of sector sizes, hence this approach is not commonly used. The more common case divides the file into blocks and then distributes these blocks among the disks in rotation. A *stripe* then consists of the same sector from each disk in the array, and is effectively equivalent to single sector in a stand alone disk.

2.2.1 Non-redundant Disk Arrays - RAID Level 0

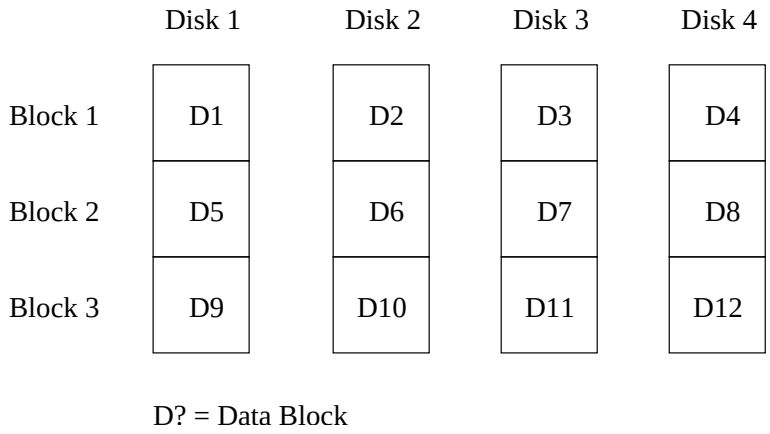

Figure 2: Data Layout in RAID 0 scheme

The above scheme has come to be known as RAID 0, though it is generally acknowledged that is not a true RAID. A single failure will render the data across the entire array useless, since each disk stores part of every file. It is thus necessary to store error correcting information, so that it is possible to recover from at least one disk failure. This is achieved by using parity information, similarly to Error Correcting Codes (ECC) used in disks. Parity is calculated by applying the XOR operator to data across every disk in a stripe. Since each disk has a controller capable of detecting a read error, it is not necessary to store information on which disk in the array failed. Thus, the parity information is sufficient to correct a single error, i.e., a single disk failure.

2.2.2 Mirroring - RAID Level 1

In the case of mirroring[1], reliability is achieved by simply duplicating all data across two or more disks. This provides complete reliability with minimal repair and recovery time - in the event of a single failure, any of the duplicates can be used for reads, while writes can be mirrored across the remaining disks. This scheme offers the possibility of greater bandwidth through parallelism, since two reads of different sectors can be assigned to two different disks - both reads occur at the same time, effectively doubling the bandwidth. With more than two disks, multiple reads can be scheduled simultaneously.

[1]Initially referred to as disk shadowing in [Britton and Gray 1988]

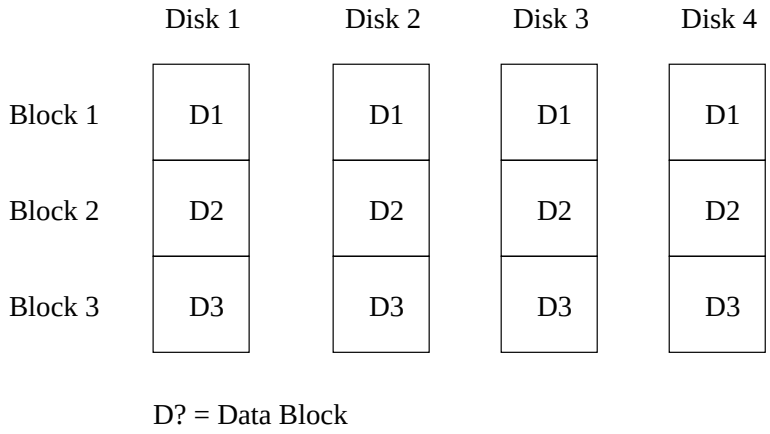

Figure 3: Data Layout in RAID 1 scheme

Mirrored disks also suffer the smallest write penalty, since the cost of any write is simply the maximum cost for any of the individual disk writes. If the disk spindles and heads are synchronized[2], there is no write penalty, since all disks move in unison, acting as one large disk.

Mirroring has the highest storage overhead of all (100%), however, since each disk other than the main one is used solely for redundancy - none of its capacity is available for useful storage. Another issue is that recovering a failed disk involves copying an entire disk to a replacement - this is not only time consuming, it also reduces the performance of the RAID during reconstruction, which may not be acceptable in certain real-time applications. However, if more than 2 disks are used, one of the clones can simply be taken off line and used to recover the failed disk in a short period of time.

2.2.3 Striped Data with Parity - RAID Levels 2 - 4

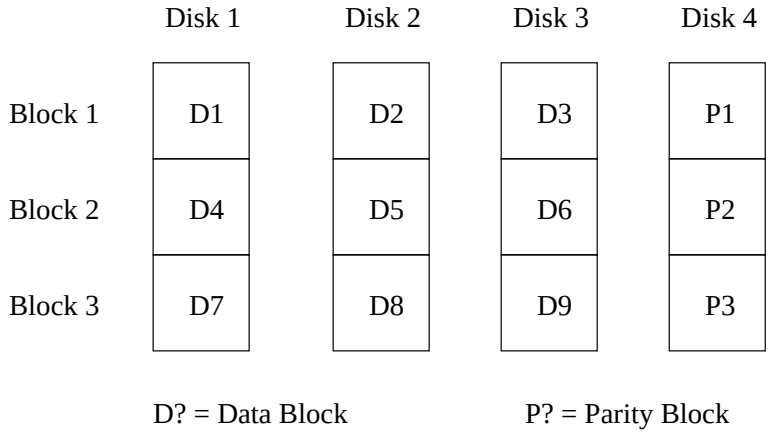

Figure 4: Data Layout in RAID 4 scheme

In order to prevent data loss in RAID systems, other than level 1, it is necessary to incorporate

[2]At any given moment, the heads of each disk are over the same logical sector

some sort of redundancy into the system. The simplest, and most widely adopted, system uses single error correcting parity [Patterson et al. 1988], using XOR operations, and is able to prevent single disk failures. This technique forms the basis of RAID 2-4.

RAID level 2 uses additional disks to store Hamming Code data, which is used to determine which disk in the array failed. However, most modern drive controllers can detect which disk in the array has failed, thus eliminating the need for these extra redundancy disks. This allows more of the total capacity to be utilized for data storage rather than redundancy.

Since disk failure can now be detected by the controller, parity can be stored on a single separate disk, and a single failed disk (parity or data) can be reconstructed from the remaining disks. This is referred to as RAID 3 in the taxonomy. The two possible failure scenarios and an obvious reconstruction scenario are illustrated in Figure 5and 6.

Data	*Parity*
0 1 0	1
1 1 1	1
1 1 0	0

Normal Data Layout

Data	*Parity*
0 1 0	?
1 1 1	?
1 1 0	?

Failed Parity Disk

Data	*Parity*
0 1 ?	1
1 1 ?	1
1 1 ?	0

Failed Data Disk

Figure 5: Illustration of the 2 possible failure scenarios in RAID 3

The reconstruction simply involves reading the data from all the undamaged disks for each stripe, calculating the parity of that data, and then writing this value to the replacement disk. If the failed disk was the parity disk, the recovery is done by simply recomputing the parity. If the failed disk was a data disk, the scheme still works since the XOR operator is commutative. Figure 6 illustrates this using the state represented by the Failed Data Disk scenario in Figure 5.

OldData			*Parity*		*RecoveredData*
0	1	$\oplus$	1	=	0
1	1		1		1
1	1		0		0

Figure 6: Reconstruction of lost data

RAID 4 still distributes data across disks, with a parity disk for redundancy, but now individual files are kept on a single disk. This is meant to assist in the performance of small writes, but the increase is minimal, and speedup through parallelism is sacrificed. RAID 4 is thus not commonly used.

To complete a write request, the new parity must be computed and written to disk. Hence, each write must access the parity disk before it completes. Since multiple write requests will be queueing for this single parity disk, a bottle neck is created in RAID levels 3 and 4.

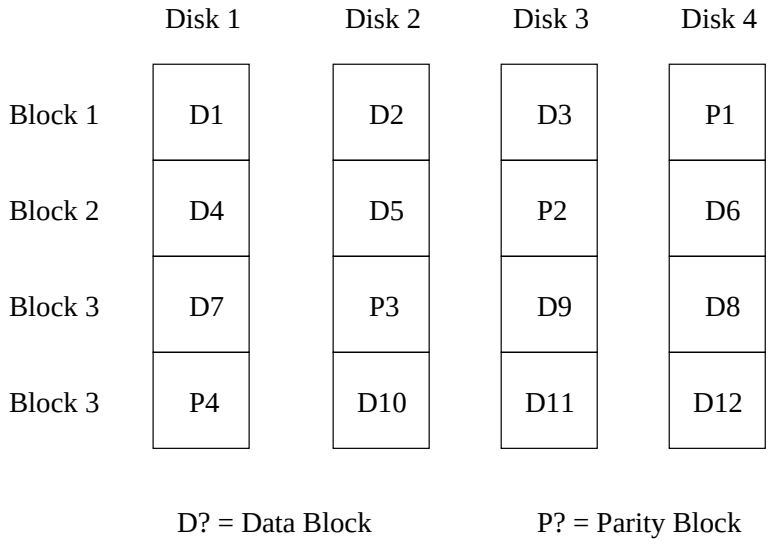

Figure 7: Data Layout in RAID 5 scheme

2.2.4 Rotating Parity with Striped Data - RAID Level 5

RAID 5 improves on this "parity bottleneck" by distributing the parity (check) disk across all disks in the array, thus reducing the bottleneck created by the check disk. This scheme also allows simultaneous writes if the writes are to different stripes, and there are no common clusters between the writes.

2.3 Other Redundancy Schemes

EVENODD [Blaum et al. 1994] is an alternate scheme that guards against two disk failures, and is efficiently implementable in hardware. The layout described to prevent bottlenecks restricts the array to a maximum of 259 disks, however.

Other schemes include balanced incomplete block designs (BIBD) [Holland and Gibson 1992] (see Section 4.3.2), which attempt to uniformly distribute data and parity across a disk, and coding methods proposed in [Hellerstein et al. 1994] which protect against arbitrary numbers of failures, but have overheads that increase exponentially w.r.t. prevented failures. Additionally, the schemes are fairly restrictive on array dimensions for optimal redundancy usage.

Finally, a scheme proposed in [Alvarez et al. 1997], DATUM, allows for recovery from an arbitrary number of disk failures, using an optimal amount of redundant storage, whilst still allowing flexible array configurations, and ensuring both parity and data are evenly distributed across the disks.

Further explanation of these schemes may be found in Section 4.2

3 RAID Implementation

This section overviews some of the key issues of a typical RAID implementation. It covers: the physical location and architecture; possible opportunities for parallelism; the method by which a RAID is connected to a host; and the operation of the RAID controller.

3.1 Controller Location

When designing a RAID storage system, there exist several options regarding the architecture. The array can be completely implemented in hardware as a black box that appears as a single disk to the host machine. Alternatively, the drives in the array can be directly connected to the host and an operating system driver performs the necessary controller functions, again presenting the array as a single disk. Finally, hybrid systems exist which attempt to take advantage of both approaches.

3.1.1 Hardware RAID Systems

All processing and management of RAID is offloaded to a dedicated processor on the hardware, referred to as the RAID controller. This includes parity checking, management of recovery from disk failures, and the physical striping of data across multiple disks. Internally drives are attached using IDE, SATA or SCSI Interfaces. Connection to server is also via one these interfaces. The hardware presents the RAID to the host system as a single, large disk. The configuration of various RAID parameters[3] is handled by the hardware. Most commercial implementations adopt this approach. One possible configuration is illustrated in Figure 8

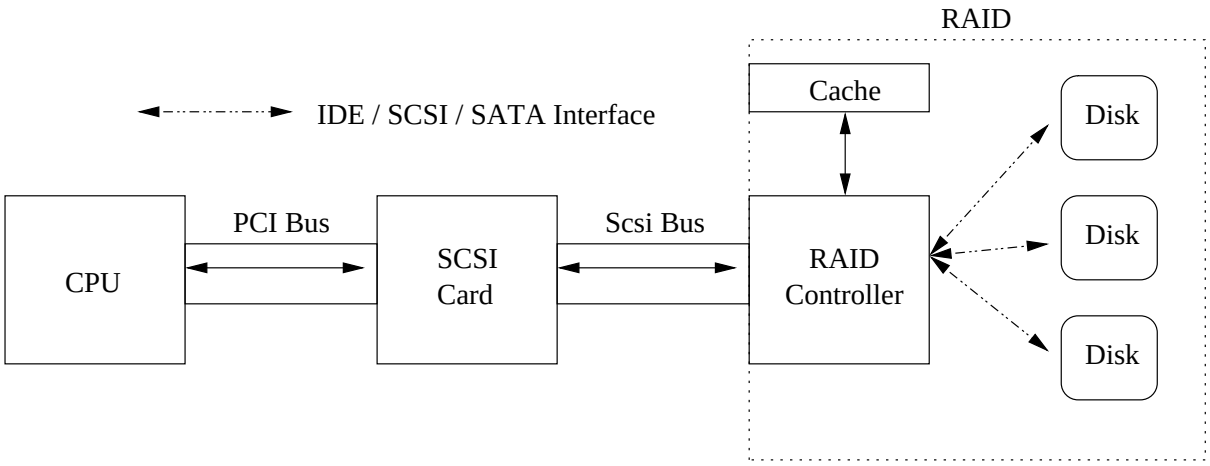

Figure 8: Typical architecture of a hardware RAID implementation

3.1.2 Software RAID Systems

Rather than have dedicated hardware to handle the operation of the RAID, this approach uses normal disk controllers (like IDE and SATA interfaces available on most modern motherboards) and software to achieve the same functionality. This is a less expensive option, as specialized

[3]stripe unit size, RAID level, cache policy

hardware is unnecessary, but CPU Load increases because of the overhead the software processing which can become a bottleneck.

Software RAID Systems use a software driver, that communicates directly with the disk drives present in the machine, which acts as a RAID controller. This driver appears to be a single disk drive to the kernel. When requests are issued, the software driver intercepts these requests and formulates the appropriate actions to service it (see Figures 11 and 12). The driver is also responsible for managing and coordinating the operation of the individual drives in the array. RAID parameters can be configured via a software interface. The Linux RAID *Enterprize Volume Management System (EVMS)* takes this approach.

3.1.3 Hybrid Approaches

Some hybrid approaches that have been attempted include the following:

1. Utilizing IDE Controllers on a motherboard for RAID storage, with controller operation undertaken by an onboard chip. Basically, this operates very similarly to software RAID, except that a dedicated hardware chip handles the tasks of the RAID controller.

   This allows slightly better performance than the software approach, but since all the hard drives are sharing bandwidth on the PCI bus, it is not the best solution. Many motherboards are now available with this technology.

2. Intel RAIDIOS technology, which enables the use of the onboard SCSI/SATA controller for RAID via a PCI RAID Card. This card utilizes the onboard controller to communicate with the connected hard drives. It overrides the default controller operation to make it appear at a hardware level as though the array is a single disk. It is almost identical to the former solution, with the exception that the RAID controller is an optional add-on. If not present, the disk controller (usually SCSI) operates normally. If the PCI Card is present, the controller on the Card overrides the onboard controller and takes on the relevant RAID responsibilities.

3.2 Parallelism in RAID Storage Systems

Access to a RAID storage system is normally via a serial channel, usually SCSI or FibreChannel, although IDE and SATA implementations exist. Some implementations, such as EZRAID, offer multiple access channels, but each channel can only access a designated part of the RAID. This is achieved by partitioning the RAID and allowing each channel exclusive access to a subset of the partitions. If the partitions are on physically separate volumes, true parallel access is supported, since each channel can independently access its associated partitions as exclusive disk access is possible. If all disks are shared by all partitions, this is not possible.

When considering RAID Storage System access there are 8 different combinations of factors that must be investigated:

Access Path		File Size		Read/Write	
serial	parallel	large	small	read	write

where large indicates a transfer of more than one stripe.

However all large accesses must be performed in serial since all disks are necessarily involved in the transfer Hence, no disk can be accessed until the transfer is complete. Figure 9 illustrates this.

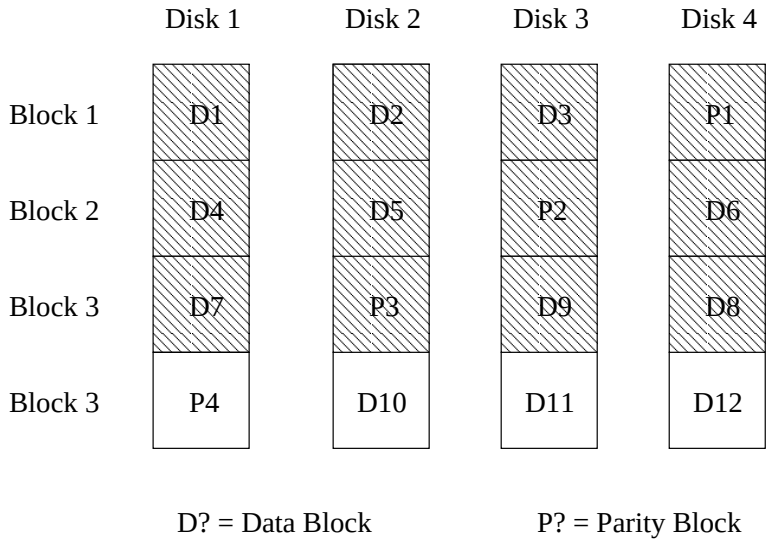

Figure 9: Large access in a RAID spanning multiple stripes

By contrast, most small accesses can be however be parallelized since each access may require different disks, as illustrated in Figure 10. The extent to which small accesses can be parallelized is a measure of the performance of the RAID under a workload typical of Online Transaction Processing systems.

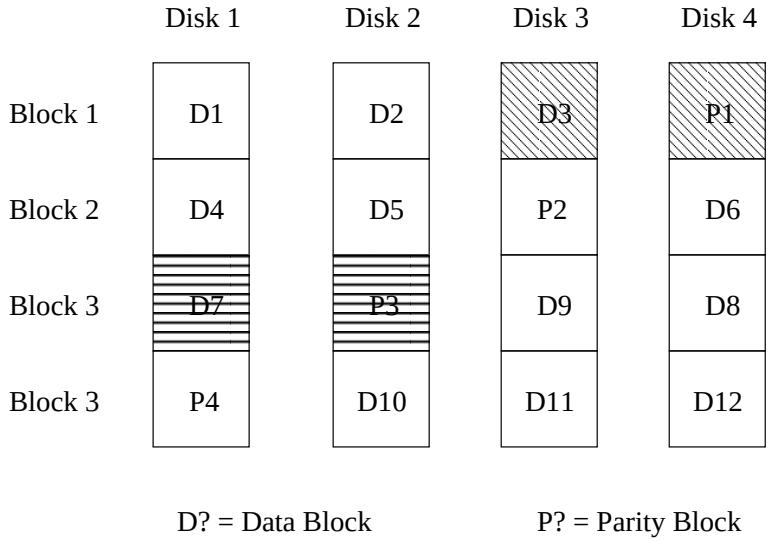

Figure 10: Small accesses requiring disjunct sets of disks can be scheduled and completed in parallel

When a write or read request is issued, the sequence of events illustrated in Figures 11 and 12occur. The SCSI specification states that communication on the bus is asynchronous, and up to 256 SCSI commands can be relayed to any given device at once. This basically allows for multiple requests to be passed to the RAID controller, which is then free to service them in whatever order is most efficient. This reordering does not allow requests to be served in parallel, however, since they

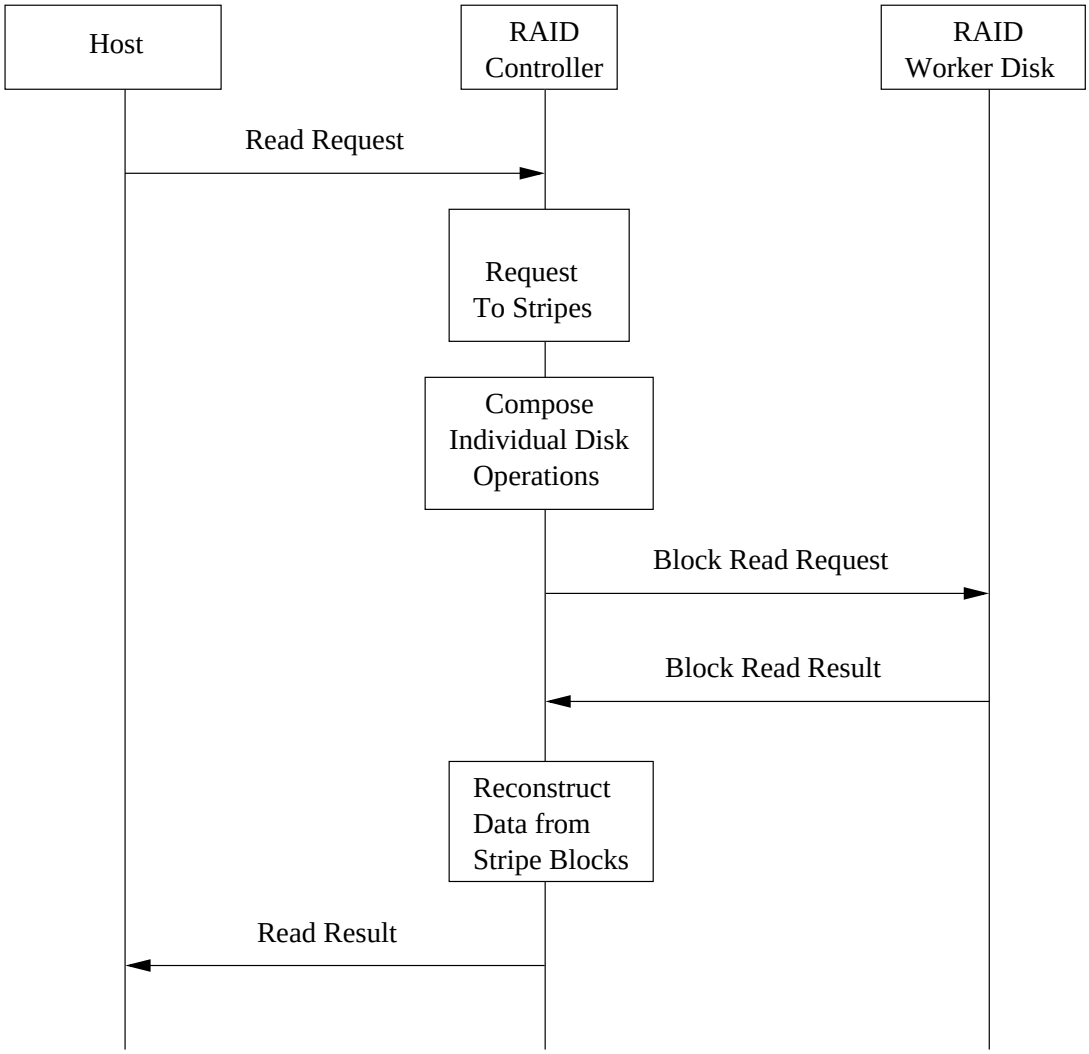

Figure 11: Time sequence of events for a `Read-request` in a RAID system in normal mode

are treated as atomic, although it can reduce overall service time. Similar functionality is available in SATA but not in IDE.

3.3 RAID System Connection to Host

A read/write request is passed to the RAID controller as a set of IDE or SCSI commands, depending on the bus used. These commands are then translated by the RAID controller, using its knowledge of the array configuration, which sends a stream of commands to the individual disks in the array. These disks are also connected to the controller using IDE or SCSI interfaces. This allows commodity disks to be attached to the array, without need for special alterations or hardware, thus complying with the *Inexpensive Disks* part of the name. Having completed the relevant operations necessary to serve the request, the controller sends a series of responses back along the connection interface. At this level, access is identical to that for a single disk.

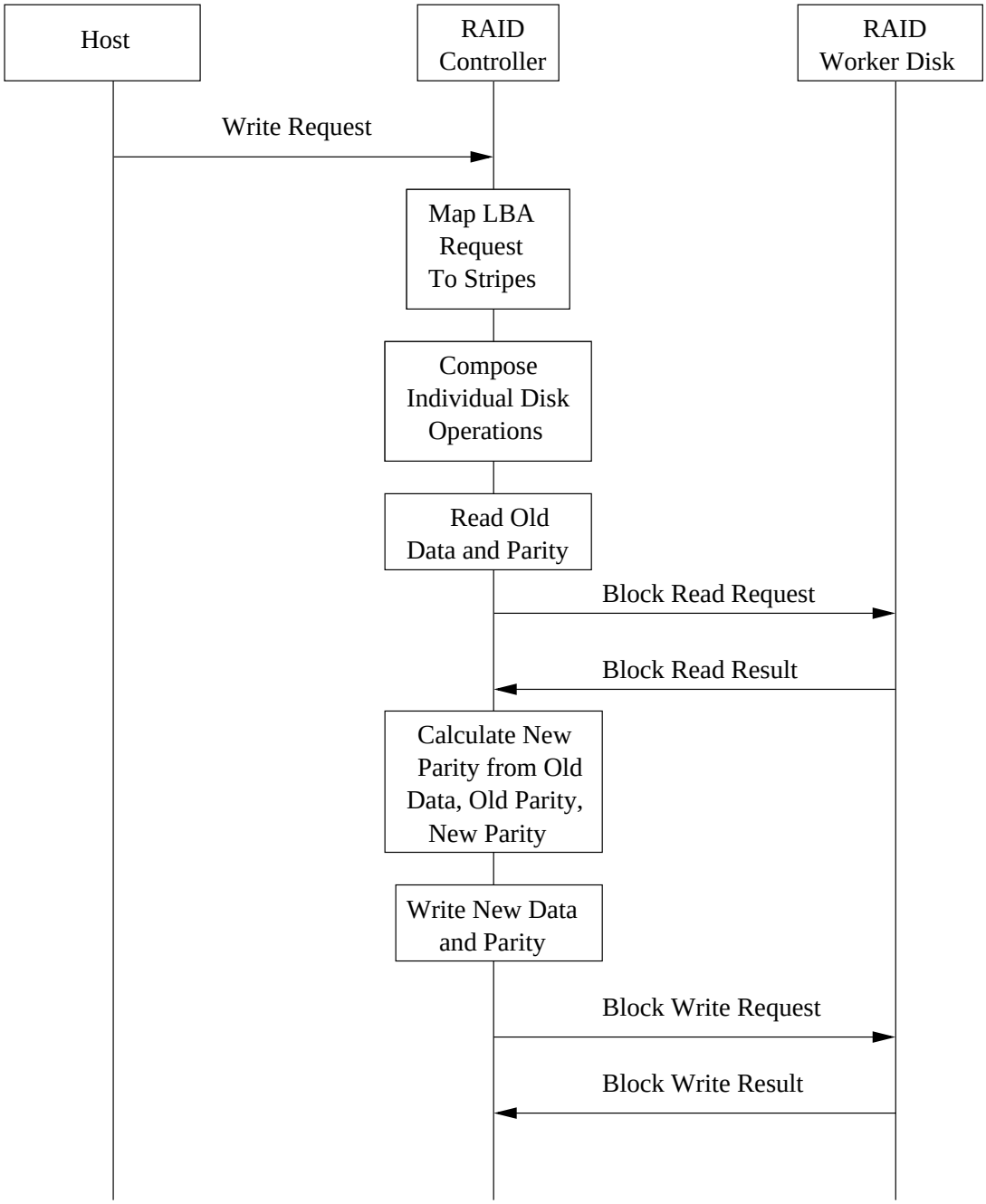

Figure 12: Typical events and communications for a Write request in a RAID in normal mode

3.4 Operation of the RAID Controller

In order for the RAID Storage System to appear like a single disk to the operating system, it is necessary that the disk array be mapped onto a virtual disk. This can either be achieved by creating specific parameters (capacity, number of cylinders, tracks and heads) describing the disk, or by using the Logical Block Addressing (LBA) scheme. LBA is the preferred method, and maps data on a disk to a sequential series of equal sized blocks. Thus, requests to the RAID storage are for a specific sequence of blocks on the virtual disk.

It is the responsibility of the RAID storage controller to take these requests and translate them

to the physical storage. Each logical block requested may map to one or more stripe block and the entire set of blocks requested may map to multiple stripes. To handle this, the controller must:

- calculate the correct mapping from logical blocks to physical blocks;
- issue commands to the drives requesting reads or writes;
- collate the received data and return it to the host over the communication channel.

The commands issued to the drives are of the same type issued to the RAID (they both use one of 2 interfaces) and are automatically handled by the drives.

In addition, the RAID controller is responsible for the following operations:

3.4.1 Caching

The RAID controller maintains a cache of instructions for each drive in the array. These caches are filled when requests are translated, and are sent to each drive sequentially as execution of the previous instruction completes. Additionally, a buffer is present where data to be written can be temporarily stored or data that is being read from different disks can be reassembled. There have also been implementations where caches have been used in a similar strategy to virtual memory systems, with predictive pre-fetching of stripes and so on.

3.4.2 Scheduling

Scheduling of accesses is also the responsibility of the RAID controller. It is necessary to establish what order the outstanding requests to the RAID should be executed in for maximal efficiency. This is particularly true of SCSI RAID controllers, which can accept multiple, parallel requests. However, it is also necessary to decide how the actions necessary for a given request should be scheduled. For instance, 2 reads from a RAID 1 can be scheduled simultaneously if each read is directed to a different drive. Additionally, 2 (or more) small, non-overlapping reads from a RAID 5 can also be scheduled simultaneously (see Figure 10). These accesses can be scheduled and completed in parallel.

3.4.3 Parity Calculation

Parity calculations are the responsibility of the RAID controller. For read requests, parity calculations are unnecessary unless the controller detects that one of the disks has failed. In this case, the array operates in degraded mode, and all parity and data on the failed disk is calculated on the fly using the remaining disks. For write requests, parity calculation is a necessity. Depending on the size of the request, different strategies can be used:

- If the write request is a full stripe, the new parity can simply be calculated from the new data, and written to disk at the same time as the data.
- Otherwise, the blocks to be written, as well as the parity block, are read in. The old data is XORed with the new data, then XORed with the old parity, and then both new data and parity are written to disk.

– If more than half the blocks in the stripe are to be written, it is more efficient to read the blocks that are not going to be written, XOR them with the new data and old parity, and then write new data and parity to disk. This reduces the total number of disk accesses slightly.

4 RAID Performance

There are a number of areas where the different RAID systems have different behavior. These areas usually include: Reliability; recovery and repair after disk failure; design and correctness of RAID controllers and architectures; and performance of RAID architectures under specific workloads areas. The following is a summary of the most important aspects of these areas and related work.

4.1 RAID Workloads

A severe problem with RAID systems arises in their applicability to On-Line Transaction Processing (OLTP) systems. These systems have disk access patterns that typically consist of *read-modify-write cycles*. With the exception of RAID 1, this causes several problems for a RAID system. Firstly, a write in a striped array requires reads of both data and parity blocks, computation of a new parity, and writes of both new data and new parity . . . 4 times more accesses than for a single disk. Another problem is that these accesses are small, and hence only a few blocks within a specific stripe are altered, yet the parity disk for the entire stripe is unavailable during the update - this effectively reduces the performance of the array, by reducing the parallelism possible.

This problem was initially tackled by [Seltzer et al. 1993], who proposed a system wherein writes were buffered until a sufficiently deep queue had developed to minimize the write penalty. The problem with this approach is that a disk failure could lead to data loss unless the buffers used are themselves fault tolerant. The work of [Menon and Mattson 1992] tries to solve this problem using a technique referred to as *Floating Data and Parity*. Each cylinder in a disk is set to contain either data or parity, and for each such cylinder, an empty track is set aside. During the update cycle, rather than overwrite the old data, the new data is instead written to the rotationally closest free block. This allows the read-compute parity-write to be executed without an extra rotational delay. The main problem with this approach is that undermines large block reads, since logically sequential data need not necessarily be stored sequentially on the disk.

The overhead required by this scheme is very low, but fault tolerant array controller storage is required to track data and parity locations. Variations on this technique are the log structure filesystem (LFS) [Rosenblum and Ousterhout 1992] and the distorted mirror approach [Solworth and Orji 1991], which uses the 100% overhead of mirroring to maintain a copy of each lock in both fixed and floating storage. All the above schemes require significant amounts of controller memory to handle buffering and storage location information.

The most promising solution thus far seems to be the work of [Stodolsky et al. 1993; Stodolsky et al. 1994]. They present a system wherein parity updates are buffered, then written to a log when sufficient are accumulated to allow for efficient disk transfer - data updates are written immediately. This log is then periodically purged, with all parity updates in it being written to disk. This scheme ensures data reliability, since if a data disk fails it can be recovered from the parity and remaining data disks; if the parity or log disk fails, then the parity disk can be reconstructed from the remaining data disks, and the log disk can be emptied.

One problem with this approach is that unless the array controller has fault tolerant buffers, a failure could result in data loss. Another is that the log disk could easily become a bottleneck in the system - this can be solved by distributing the log disk across all disks, much as is done with the parity disk in RAID 5. The most problematic aspect of this approach, and one that does not appear to have been addressed, is how user response will degrade during reconstruction of a failed disk. Various schemes to address this for other RAID configurations are presented in Section 4.3.

RAID Configuration	Workload			
	Large Read	Large Write	Small Read	Small Write
RAID 0	+++	+++	+++	+++
RAID 1	+	+	++	++
RAID 2 - 4	++	++	-	–
RAID 5	++	++	+	+

Table 1: Performance of RAID Variations under various Workloads

A fairly recent optimization for short transaction workloads presented in [Haruo 2000], *Fault-tolerant Buffering Disks*, uses disks to provide double buffering, which increases write speed, as well as backup for fault tolerance and read speed improvements. The system does not scale well, and has a higher overhead than other systems, but for small arrays increases both throughput and response time.

4.2 Encoding

Encoding is used to guard against single disk failures using parity. The scheme uses the XOR $\oplus$ operator on all blocks in a stripe to determine their parity, which is then stored to enable error recovery. The XOR operator is typically applied to corresponding bits in each block as follows:

$$10011 \oplus 01101 = 11110 \quad \text{(parity)}$$

EVENODD [Blaum et al. 1994] encoding builds on the simple parity encoding above. It applies the same technique, but in 2 different passes. The first pass calculates a horizontal parity value across all blocks in a stripe. The second pass calculates a diagonal parity across stripes and disks. The combination of these 2 parities allow EVENODD to recover from 2 disk failures in the array. Later extensions to this technique allow recovery from any number of disk failures, but the greater the redundancy, the greater the number of parity disks required.

DATUM [Alvarez et al. 1997] uses an Information Dispersal Algorithm to allow a RAID array to survive n failures using exactly n redundant parity disks. It uses a transformation of the form:

$$(d_1, d_2, \ldots, d_m) \cdot T = (e_1, e_2, \ldots, e_{m+n})$$

where m is the number of data blocks to be written, n is the number of redundant disks (and the number of failures survived) and T is a transformation matrix. d_n represent blocks of data and e_n the transformed data which is written back to disk. One possible problem with this method is that finding T for any general case is not simple, especially if the actual data is required to be unchanged when written to disk (ie. $d_i = e_i, 1 \leq i \leq m$). However, it does scale well and uses the minimal number of redundant disks.

4.3 Reliability

Due to their decreased MTTF, preventing data loss in RAID systems is a very important consideration. Redundancy is the primary fail safe, and many schemes exist to achieve this. Immediately after a failure, it is necessary to perform some form of recovery. Finally, the failed disk needs to be replaced and reconstructed to restore the system to full working efficiency. This section covers advances in these areas.

4.3.1 Recovery

Recovery is necessary after a disk failure, to ensure that operations in progress at that time are not lost completely. There are currently three approaches to this problem. The first and most prevalent solution, forward error correction (FEC), attempts to handle errors dependent on the current state of the system and the state of execution of the requested disk operation. This method requires enumeration of all possible states of the system, and handcrafting execution paths for each. This is both time-consuming and error prone.

An alternate approach, backward error recovery (BEC)[Courtright II and Gibson 1994], uses an approach popular in transactional systems which are required to support atomicity of operations. The approach is to set out a small number of execution paths for each of the possible states of the system such as error-free, disk failed, etc. Each execution path is composed from a set of simple, reversible operations, which are logged as they execute. When a failure occurs during execution of one of these paths, the execution is rolled-back by executing the inverse operations in reverse order. When the original state is recovered, an alternate execution path, appropriate to the current state of the system, is used to retry the failed operation.

The final alternative, roll-away error recovery (REC)[Courtright II 1997], uses a similar scheme to backward error recovery, but adds commit barriers to simulate two-phase commit in transactional systems. The idea is that when a failure occurs, execution is allowed to continue up to a commit barrier, but not beyond. If the completed state is not reached by this, then an alternate execution path is chosen, and the operation is re-attempted. This style of error recovery is popularized by the RAIDframe system [Courtright II et al. 1996a] (see Section 5).

4.3.2 Reconstruction

A major consideration when recovering from a failed disk is the effect on user response times, i.e., does the reconstruction of the failed disk degrade the performance of the RAID? Normally this is very true, since reconstructing any single disk requires access to all the other disks to reconstruct every stripe, effectively rendering the RAID inaccessible during each such access. While stripe reconstruction can be interleaved with user requests to allow the RAID to continue operation, access latency will rise unacceptably and the Mean Time to Repair (MTTR) will increase. The longer it takes to repair the RAID, the more likely it is a second disk will fail before the first is recovered, resulting in data loss.

An innovative solution to this is presented in [Holland and Gibson 1992; Holland et al. 1993; Holland et al. 1994; Holland 1994]. The authors suggest that performance degradation during reconstruction can be reduced by sacrificing some of the data capacity of the RAID toward redundancy. The crux of their work is the idea of a virtual topology that is mapped to the physical disk topology of the system. Assuming that there are 7 disks available, the normal, intuitive solution would be to treat all 7 as a large array with striping across all 7, and distributed parity. An alternative would be to treat this as an array of 4 disks, in a RAID 5 configuration. To allow this, the authors discuss ways of mapping such a virtual topology to the physical one.

The benefit of this approach is that if a single disk fails, only particular stripes in the virtual array will be affected. Importantly, since each stripe only consists of 4 virtual disks, reconstructing affected stripes only requires read accesses to 3 other real disks. This greatly diminishes the bandwidth required for reconstruction, and allows accesses to the other 3^4 disks to occur simultaneously.

[4]Remember that one disk has failed, and is in the process of being reconstructed

Additionally, the mapping scheme mentioned above ensures that the virtual sectors are uniformly distributed, thus ensuring that no single disk gets overburdened during reconstruction. The only requirement is increased redundancy overhead, since instead of a ratio of 6:1 of data:parity sectors, the ratio is now 3:1.

The latest development in this field is something called *data-reconstruction networks* [Haruo 2000]. The idea is that disks are connected in subnetworks which are then interconnected to form one large network. Reconstruction of a single failed disk is localized to a subnetwork, hence reducing the impact on the whole network. Additionally, the overlapping of subnetworks allows the recovery of more than one disk failure, depending on the architecture.

Other considerations during reconstruction are how to handle user requests. If a read is requested of an as yet unreconstructed sector, the sector can recovered on the fly using parity and the remaining data disks. However, once this data is calculated, the option arises to store it (which requires buffering), write it to disk (which could upset the scheduling algorithm in use) or discard it (which entails having to recalculate it later during the reconstruction). The choice of which approach to take is dependent on the implementation.

If a write is requested during reconstruction, then this new value is simply written to replacement disk, and the appropriate stripe is excluded from the rebuild list, which reduces reconstruction time. Alternatively, all stripes are reconstructed in order, and new data is simply overwritten - this has the advantage of significantly less bookkeeping, but there is a lot of unnecessary work performed.

5 RAID Performance Modelling

As RAID architectures are being developed, their performance is analyzed using either simulation or analytical methods. Simulators are generally handcrafted, which entails code duplication and potentially un reliable systems as outlined in [Holland and Gibson 1992]. Equally problematic is that RAID architectures, once defined, are difficult to prove correct without extensive simulations.

One general solution is presented in [Courtright II et al. 1996b] and [Courtright II et al. 1996a]. These papers outline the design of RAIDframe, an application with a graphical interface that enables structured specification of RAID architectures, as well as detailed testing using built-in disk simulators or running the same code as real-world disk drivers. The system uses Directed Acyclic Graphs (DAG) to define RAID architectures in terms of primitive operations, together with the roll-away error recovery scheme, to enable quick development and testing. The system has shown exceptional code reuse, and has been utilized by other, independent researchers, eg. [Alvarez et al. 1997].

As a first step towards systematic proofs of correctness, the work of [Vaziri et al. 1997] identifies key properties of RAID algorithms that need to be verified. Additionally, I/O automata and key invariants are used to illustrate how RAID architectures can be proven or errors detected.

5.1 Analytical Modelling

Analytical models of RAID operations generally utilize drive characteristics to model typical operations [Stodolsky et al. 1993]. Key factors such as *Seek Time* (time to move to a specified track), *Rotational Latency* (time to rotate disk to required sector) and *Transfer Time* (time to read or write to disk) are used to derive mathematical expressions of the time taken for different operations. Such expressions are usually independent of any particular hardware, and are derived from analysis of typical actions performed during a given RAID operation. Such an approach is taken in the work of [Lee and Katz 1993]. Most analytical models of RAID are based on other models of the underlying disks driving the array, such as the work of [Ruemmler and Wilkes 1994]. Using these drive characteristics, queuing models can be introduced to model the arrival of tasks at each disk in a RAID, as such tasks are handed out by the RAID controller. Combining these queued disks with arrival rates for a single disk in normal operation provides a starting point from which more complicated RAID systems can be modelled.

5.2 Simulation

The most notable simulation tool for RAID systems is the previously mentioned RAIDframe system [Courtright II et al. 1996a]. The system offers multiple ways to test any RAID architecture using the previously mentioned, generic DAG representation. The simulator is able to use utilization data gathered from real world systems to feed in to the simulator as well as accepting real requests from outside systems. It can also be used to generate a Linux software driver to test an architecture in a real world system. One downside to this approach is that, since all RAID operations are performed in software, the controller software can easily become a bottleneck (up to 60% of total processing due to software) and hence skew the results.

Another simulation tool presented in the literature (and available for academic use) is the RAIDSim application [Courtright II et al. 1996b]. It is built on top of a simulator for single disks, known as DiskSim, developed at the University of Michigan. Unlike RAIDframe, RAIDSim is labelled as

very difficult to use, and has no generic interface. One other tool the authors are aware of is SimSAN, an environment for simulating and analyzing Storage Area Networks in which RAID usually serves as the underlying storage.

References

Alvarez, G. A., Burkhard, W. A., and Cristian, F. 1997. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, 62–72.

Blaum, M., Brady, J., Bruck, J., and Menon, J. 1994. Evenodd: An optimal scheme for tolerating double disk failures in raid architectures. In *Proceedings of the 21st Symp. on Computer Architecture*, 245–254.

Britton, D., and Gray, J. 1988. Disk shadowing.

Courtright II, W. V., and Gibson, G. A. 1994. Backward error recovery in redundant disk arrays. In *Proceedings of the 1994 Computer Measurement Group Conference (CMG)*, vol. 1, 63–74.

Courtright II, W. V., Gibson, G., Holland, M., and Zelenka, J. 1996. Raidframe: rapid prototyping for disk arrays. In *Proceedings of the 1996 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, vol. 24, 268–269.

Courtright II, W. V., Gibson, G., Holland, M., and Zelenka, J. 1996. A structured approach to redundant disk array implementation. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS)*.

Courtright II, W. V. 1997. *A Transactional Approach to Redundant Disk Array Implementation.* Master's thesis, Carnegie Mellon University.

Haruo, I. P., 2000. Performance and reliability of secondary storage systems.

Hellerstein, L., Gibson, G. A., Karp, R. M., Katz, R. H., and Patterson, D. A. 1994. Coding techniques for handling failures in large disk arrays. *Algorithmica 12*, 2/3, 182–208.

Holland, M., and Gibson, G. A. 1992. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, 23–35.

Holland, M., Gibson, G. A., and Siewiorek, D. P. 1993. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing.*

Holland, M., Gibson, G. A., and Siewiorek, D. P. 1994. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Journal of Distributed and Parallel Databases 2*, 3 (July), 295–335.

Holland, M. C. 1994. *On-Line Data Reconstruction In Redundant Disk Arrays.* Master's thesis, Carnegie Mellon University.

Lee, E. K., and Katz, R. H. 1993. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ACM Press, 98–109.

Menon, J., and Mattson, D. 1992. Performance of disk arrays in transaction processing environments. In *12th International Conference on Distributed Computing Systems*, 302–309.

Patterson, D. A., Gibson, G., and Katz, R. H. 1988. A case for redundant arrays of inexpensive disks (raid).

Patterson, D. A., Chen, P., Gibson, G., and Katz, R. H. 1989. Introduction to redundant arrays of inexpensive disks (raid).

Rosenblum, M., and Ousterhout, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

Ruemmler, C., and Wilkes, J. 1994. An introduction to disk drive modeling. *IEEE Computer 27*, 3, 17–28.

Seltzer, M. I., Bostic, K., McKusick, M. K., and Staelin, C. 1993. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, 307–326.

Solworth, J. A., and Orji, C. U. 1991. Distorted mirrors. In *Proceedings of the first international conference on Parallel and distributed information systems*, IEEE Computer Society Press, 10–17.

Stodolsky, D., Gibson, G., and Holland, M. 1993. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th annual international symposium on Computer architecture*, ACM Press, 64–75.

Stodolsky, D., Holland, M., Courtright II, W. V., and Gibson, G. A. 1994. A redundant disk array architecture for efficient small writes. Tech. Rep. CMU-CS-94-170.

Vaziri, M., Lynch, N. A., and Wing, J. M. 1997. Proving correctness of a controller algorithm for the RAID level 5 system. In *Symposium on Fault-Tolerant Computing*, 16–25.