Proceedings

The 20th Annual International Symposium on COMPUTER ARCHITECTURE

May 16 –19, 1993 San Diego, California

Sponsored by

IEEE Computer Society
Technical Committee on Computer Architecture

Association for Computing Machinery
SIGARCH

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

HPE, Exh. 1010, p. 1

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by the
IEEE Computer Society Press
10662 Los Vaqueros Circle
PO Box 3014
Los Alamitos, CA 90720-1264

© 1993 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 27 Congress Street, Salem, MA 01970. For other copying, reprint, or republication permission, write to IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331.

IEEE Computer Society Press Order Number 3810-02
IEEE Catalog Number 93CH3284-7
Library of Congress Number 85-642899
ISBN 0-8186-3810-9 (paper)
ISBN 0-8186-3811-7 (microfiche)
ISBN 0-8186-3812-5 (case)
ISSN 0884-7495
ACM Order Number 415930
ACM Library Series ISBN 0-89791-579-8 (Hardcover)
ACM SIGARCH ISSN 0163-5964

Additional copies can be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
PO Box 3014
Los Alamitos, CA 90720-1264

IEEE Service Center
445 Hoes Lane
PO Box 1331
Piscataway, NJ 08855-1331

IEEE Computer Society
13, avenue de l'Aquilon
B-1200 Brussels
BELGIUM

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN

ACM Order Dept., PO Box 64145, Baltimore, MD 21264

Production Editors: Mary E. Kavanaugh and Edna Straub
Cover art: Joseph Daigle / Schenk-Daigle Studios
Printed in the United States of America by Braun-Brumfield, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

HPE, Exh. 1010, p. 2

Table of Contents

General Chair's Message v
Program Chair's Message vi
Organizing Committee vii
Referees viii

Session 1: Opening Session

Session 2: Architectural Characteristics of Scientific Applications

Architectural Requirements of Parallel Scientific Applications with Explicit Communication 2
R. Cypher, A. Ho, S. Konstantinidou, and P. Messina

Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors 14
E. Rothberg, J.P. Singh, and A. Gupta

Session 3: TLBs and Memory Management

Design Tradeoffs for Software-Managed TLBs 27
D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown

Architectural Support for Translation Table Management in Large Address Space Machines 39
J. Huck and J. Hays

Session 4: Input/Output

The TickerTAIP Parallel RAID Architecture 52
P. Cao, S.B. Lim, S. Venkataraman, and J. Wilkes

Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays 64
D. Stodolsky, G. Gibson, and M. Holland

The Architecture of a Fault-Tolerant Cached RAID Controller 76
J. Menon and J. Cortney

Session 5: Multiprocessor Caches

The Detection and Elimination of Useless Misses in Multiprocessors 88
M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström

Adaptive Cache Coherency for Detecting Migratory Shared Data 98
A.L. Cox and R.J. Fowler

An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing 109
P. Stenström, M. Brorsson, and L. Sandberg

Session 6: Panel Session I — High-Performance Computing from the Applications Perspective
D. Kuck (Chair)

HPE, Exh. 1010, p. 3

Session 7: Multithreading Support

Register Relocation: Flexible Contexts for Multithreading 120
C.A. Waldspurger and W.E. Weihl

Multiple Threads in Cyclic Register Windows 131
Y. Hidaka, H. Koike, and H. Tanaka

Session 8: Mechanisms for Creating Shared Memory

Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology 144
S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel

Mechanisms for Cooperative Shared Memory 156
D.A. Wood, S. Chandra, B. Falsafi, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, S.S. Mukherjee, S. Palacharla, and S.K. Reinhardt

Session 9: Cache Design

A Case for Two-Way Skewed-Associative Caches 169
A. Seznec

Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches 179
A. Agarwal and S.D. Pudar

Cache Write Policies and Performance 191
N.P. Jouppi

Session 10: Evaluation of Machines I

Hierarchical Performance Modeling with MACS: A Case Study of the Convex C-240 203
E.L. Boyd and E.S. Davidson

The Cedar System and an Initial Performance Study 213
D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner

The J-Machine Multicomputer: An Architectural Evaluation 224
M.D. Noakes, D.A. Wallach, and W.J. Dally

Session 11: Processor Architecture and Implementation

16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors 237
J. Bunda, D. Fussell, R. Jenevein, and W.C. Athas

Register Connection: A New Approach to Adding Registers into Instruction Set Architectures 247
T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W.-M. Hwu

A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History 257
T.-Y. Yeh and Y.N. Patt

HPE, Exh. 1010, p. 4

Session 12: Multiprocessor Memory Systems

The Performance of Cache-Coherent Ring-Based Multiprocessors 268
L.A. Barroso and M. Dubois

Limitations of Cache Prefetching on a Bus-Based Multiprocessor 278
D.M. Tullsen and S.J. Eggers

Transactional Memory: Architectural Support for Lock-Free Data Structures 289
M. Herlihy and J.E.B. Moss

Session 13: Panel Session II — Experimental Research: How Do We Measure Success?
Y. Patt and R. Iyer (Co-Chairs)

Session 14: Evaluation of Machines II

Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5 302
E. Spertus, S.C. Goldstein, K.E. Schauser, T. von Eicken, D.E. Culler, and W.J. Dally

Improving AP1000 Parallel Computer Performance with Message Communication 314
T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata

Session 15: Memory Systems and Interconnection

Performance of Cached DRAM Organizations in Vector Supercomputers 327
W.-C. Hsu and J.E. Smith

The Chinese Remainder Theorem and the Prime Memory System 337
Q.S. Gao

Odd Memory Systems May Be Quite Interesting 341
A. Seznec and J. Lenfant

A Comparison of Adaptive Wormhole Routing Algorithms 351
R.V. Boppana and S. Chalasani

Author Index 361

HPE, Exh. 1010, p. 5

This material may be protected by Copyright law (Title 17 U.S. Code)

The Architecture of a Fault-Tolerant Cached RAID Controller

Jai Menon and Jim Cortney

IBM Almaden Research Center
San Jose, California 95120-6099
Telephone: (408) 927-2070 *E-Mail: menonjm@almaden.ibm.com*

Abstract— RAID-5 arrays need 4 disk accesses to update a data block -- 2 to read old data and parity, and 2 to write new data and parity. Schemes previously proposed to improve the update performance of such arrays are the Log-Structured File System [10] and the Floating Parity Approach [6]. Here, we consider a third approach, called Fast Write, which eliminates disk time from the host response time to a write, by using a Non-Volatile Cache in the disk array controller. We examine three alternatives for handling Fast Writes and describe a hierarchy of destage algorithms with increasing robustness to failures. These destage algorithms are compared against those that would be used by a disk controller employing mirroring. We show that array controllers require considerably more (2 to 3 times more) bus bandwidth and memory bandwidth than do disk controllers that employ mirroring. So, array controllers that use parity are likely to be more expensive than controllers that do mirroring, though mirroring is more expensive when both controllers and disks are considered.

1. Introduction

A ***disk array*** is a set of disk drives (and controller) which can automatically recover data when one (or more) drives in the set fails by using redundant data that is maintained by the controller on the drives. [8] describes five types of disk arrays called RAID-1 through RAID-5 and [2] describes a sixth type called a parity striped disk array. In this paper, our focus is on RAID-5 and/or parity striped disk arrays which employ a parity technique described in [1,8]. This technique requires fewer disks than mirroring and is therefore more acceptable in many situations.

The main drawback of such arrays are that they need four disk accesses to update a data block -- two to read old data and parity, and two to write new data and parity. [5] showed that the performance degradation can be quite severe in transaction processing environments. Two schemes that have been previously proposed to improve array update performance are the Log-Structured File System [10] and the Floating Parity Approach [6]. In this paper, we consider a third approach, called ***Fast Write***, which eliminates disk time from the host response time to a write, by using Non-Volatile Storage (NVS) in the disk array controller. A block received from a host system is initially written to NVS in the disk array controller and a completion message is sent to the host system at this time. Actual destage of the block from NVS to disk is done asynchronously at a later time. We call a disk array that uses the Fast Write technique a ***Cached RAID***.

The rest of this paper is organized as follows. We first review the parity technique. Then, we describe Fast Write. Next, we give an overview of the architecture of Hagar, a disk array controller prototype developed at the IBM Almaden Research Center. Hagar uses Fast Write. In the last sections of this report, we then analyze several alternatives for destaging blocks from NVS to disk. We show that destage algorithms must be carefully developed because of complex trade-offs between availability and performance goals.

2. Review of Parity Technique

We illustrate the parity technique on a disk array of six data disks and a parity disk. In this diagram, Pi is a parity block that protects the six data blocks labelled Di. Pi and the 6 Dis together constitute a ***parity group***. The Pi of a parity group must always be equal to the parity of the 6 Di blocks in the same parity group as Pi.

```
Data Disk 1     D1 D2 D3 D4
Data Disk 2     D1 D2 D3 D4
Data Disk 3     D1 D2 D3 D4
Data Disk 4     D1 D2 D3 D4
Data Disk 5     D1 D2 D3 D4
Data Disk 6     D1 D2 D3 D4
Parity Disk     P1 P2 P3 P4
```

We show only one track (of 4 blocks) from each of the disks. In all, we show four parity groups. P1 contains the parity or exclusive OR of the blocks labeled D1 on all the data disks, P2 the exclusive OR

0884-7495/93 $3.00 © 1993 IEEE

HPE, Exh. 1010, p. 6

D2s, and so on. Such an array is robust against single disk crashes; if disk 1 were to fail, data on it can be recreated by reading data from the remaining five data disks and the parity disk and performing the appropriate exclusive OR operations.

Whenever the controller receives a request to write a data block, it must also update the corresponding parity block for consistency. If D1 is to be altered, the new value of P1 is calculated as:

```
new P1 = (old D1 XOR new D1 XOR old P1)
```

Since the parity must be altered each time the data is modified, these arrays require four disk accesses to write a data block - two to read old data and parity, two to write new data and parity.

3. Overview of the Fast Write Technique

In this technique, all disk array controller hardware such as processors, data memory (memory containing cached data blocks and other data buffers), control memory (memory containing control structures such as request control blocks, cache directories, etc..) are divided into at least two disjoint sets, each set on a different power boundary. The data memory and the control memory are either battery-backed or built using NVS so they can survive power failures. When a disk block to be written to the disk array is received, the block is first written to data memory in the array controller, in two separate locations, on two different power boundaries. At this point, the disk array controller returns successful completion of the write to the host. In this way, from the host's point of view, the write has been completed quickly without requiring any disk access. Since two separate copies of the disk block are made in the disk array controller, no single hardware or power failure can cause a loss of data.

Disk blocks in array controller cache memory that need to be written to disk are called ***dirty***. Such dirty blocks are written to disk in a process we call ***destaging***. When a block is destaged to disk, it is also necessary to update, on disk, the parity block for the data block. This may require the array controller to read the old values of the data block and the parity block from disk, XOR them with the new value of the data block in cache, then write the new value of the data block and of the parity block to disk. Since many applications first read data before updating them, we expect that the old value of the data block might already be in array controller cache. Therefore, the more typical destage operation is expected to require one disk read and two disk writes.

3.1. Overview of Destage

Typically, the disk blocks in the disk array controller (both dirty and clean disk blocks) are organized in Least-Recently-Used (LRU) fashion. When space for a new disk block is needed in the cache, the LRU disk block in cache is examined. If it is clean, the space occupied by that disk block can be immediately used; if it is dirty, the disk block must be destaged before it can be used. While it is not necessary to postpone destaging a dirty block until it becomes the LRU block in the cache, the argument for doing so is that it could avoid unnecessary work. Consider that a particular disk block has the value d. If the host later writes to this disk block and changes its value to d', we would have a dirty block (d') in cache which would have to be destaged later. However, if the host writes to this disk block again, changing its value to d", before d' became LRU and was destaged, we no longer need to destage d', thus avoiding some work.[1]

When a block is ready to be destaged, the disk array controller may also decide to destage other dirty blocks in the cache that need to be written to the same track, or the same cylinder. This helps minimize disk arm motion, by clustering together many destages to the same disk arm position. However, this also means that some dirty blocks are destaged before they become the LRU disk block, since they will be destaged at the same time as some other dirty block that became LRU and that happened to be on the same track or cylinder. Therefore, the destage algorithm must be carefully chosen to trade-off the reduction in destages that can be caused by overwrites of dirty blocks if we wait until dirty blocks become LRU versus the reduction in seeks that can be achieved if we destage multiple blocks at the same track or cylinder position together. An example compromise might be along the following lines: when a dirty block becomes LRU, destage it and all other dirty blocks on the same track (cylinder) as long as these other blocks are in the LRU half of the LRU chain of cached disk blocks.

In a practical implementation, we may have a background destage process that continually destages dirty blocks near the LRU end of the LRU list (and others on the same track or cylinder) so that a request that requires cache space (such as a host write that misses in the cache) does not have to wait for destaging to complete in order to find space in the cache. Another option is to trigger destages based on the fraction of dirty blocks in the cache. For example, if the fraction of dirty blocks in the cache exceeds some threshold (say 50%), we may trigger a destage of dirty blocks that are near the LRU end of the LRU chain (and

---

1 On the other hand, there are two copies of every dirty disk block in the cache. The longer we delay destaging the dirty blocks, the longer they occupy two cache locations.

HPE, Exh. 1010, p. 7

of other dirty blocks on the same tracks as these blocks). This destaging may continue until the number of dirty blocks in cache drops below some reverse threshold (say 40%).

Since read requests to the disk are synchronous while destages to the disk are asynchronous, the best destage policy is one that minimizes any impact on read performance. Therefore, the disk controller might delay starting a destage until all waiting reads have completed to the disk and it may even consider preempting a destage (particularly long destages of many tracks) for subsequent reads.

3.2. Summary of Fast Write Benefits

To summarize, Fast Write: will eliminate disk time from write response time as seen by the host; will eliminate some disk writes due to overwrites caused by later host writes to dirty blocks in cache; will reduce disk seeks because destages will be postponed until many destages can be done to a track or cylinder; and can convert small writes (single block) to large writes (all blocks in parity group) and thus eliminate many disk accesses. Work done by Joe Hyde [3] indicates that, for high-end IBM 370 processor workloads, anywhere from 30% to 60% of the writes to the disk controller cause overwrites of dirty blocks in cache. His work also indicates that even though the host predominantly issues single block writes, anywhere from 2 to 7 dirty blocks can be destaged together when a track is destaged. Together, these results indicate that Fast Write can be an effective technique for improving the write performance of disk arrays that use the parity technique.

4. Overview of Hagar

The Hagar prototype is designed to support very large amounts of disk storage (up to 1 Terabyte); to provide high bandwidth (100 MB/sec); to provide high IOs/sec (5000 IOs/sec at 4 Kbyte transfers); and to provide high availability. It provides for continuous operation through use of battery-backed memory, duplexed hardware components, multiple power boundaries, hot sparing of disks, on-the-fly-rebuilding of data lost in a disk crash to a hot spare and by permitting nondisruptive installation and removal of disks and hardware components.

Hagar is organized around checked and reliable control and data buses on a backplane. The structure of Hagar is shown in Figure 1. The data bus is optimized for high throughput on large data transfers and the control bus is optimized for efficient movement of small control transfers. The Hagar data bus is a multi-destination bus; a block received from the host system or from the disks can be placed in multiple data memory locations even though only one copy of the data block travels on the data bus.

In the idealized Hagar implementation, we would have processor cards; host interface cards; global data memory cards; global control memory cards and disk controller cards attached to the reliable data and control buses. Cards of each type are divided into at least 2 disjoint sets; each set is on a different power boundary. The disk controller cards would attach to multiple disk strings over a serial link using a logical command structure such as SCSI. For availability reasons, the disks would be dual-ported and would each attach to two serial links originating from two different disk controllers. The data memory cards would provide battery-backed memory, accessible to all processors, for caching, fast write and data buffering. The control memory cards also provide battery-backed memory, accessible to all processors, used for control structures such as cache directories and lock tables. Unlike the data memory, the control memory provides efficient access to small amounts of data (bytes) and supports atomic operations necessary for synchronization between multiple processors.

The XOR hardware needed for performing parity operations is integrated with the data memory. We chose to integrate the XOR logic with the data memory to avoid bus bandwidth during XOR operations to a separate XOR unit such as that used in the Berkeley RAID-II design ([9]). The data memory in Hagar supports two kinds of store operations: a regular store operation and a special store & XOR operation. A store & XOR to location X, takes the incoming data, XORs it with data at location X, and stores the result of the XOR back into location X.

5. Data Memory Management Algorithms

5.1. Four Logical Regions of Data Memory

The data memory in the disk array controller is divided into four logical regions: the free pool, the data cache, the parity cache and the buffer pool. When a block is written by the host, it is placed in the buffer pool, in two separate power boundaries. Subsequently, the two data blocks are moved into the data cache (this is a logical, not a physical move; that is, the cache directories are updated to reflect the fact that the disk block is in cache). After this logical move of the blocks into the data cache, the array controller returns "done" to the host system that did the write. At some subsequent time, the block D is destaged to disk. The data cache region of the data memory contains data blocks from disk and the parity cache region of the data memory contains parity blocks from disk. The parity blocks are useful during destage, since the presence of a parity block in the

HPE, Exh. 1010, p. 8

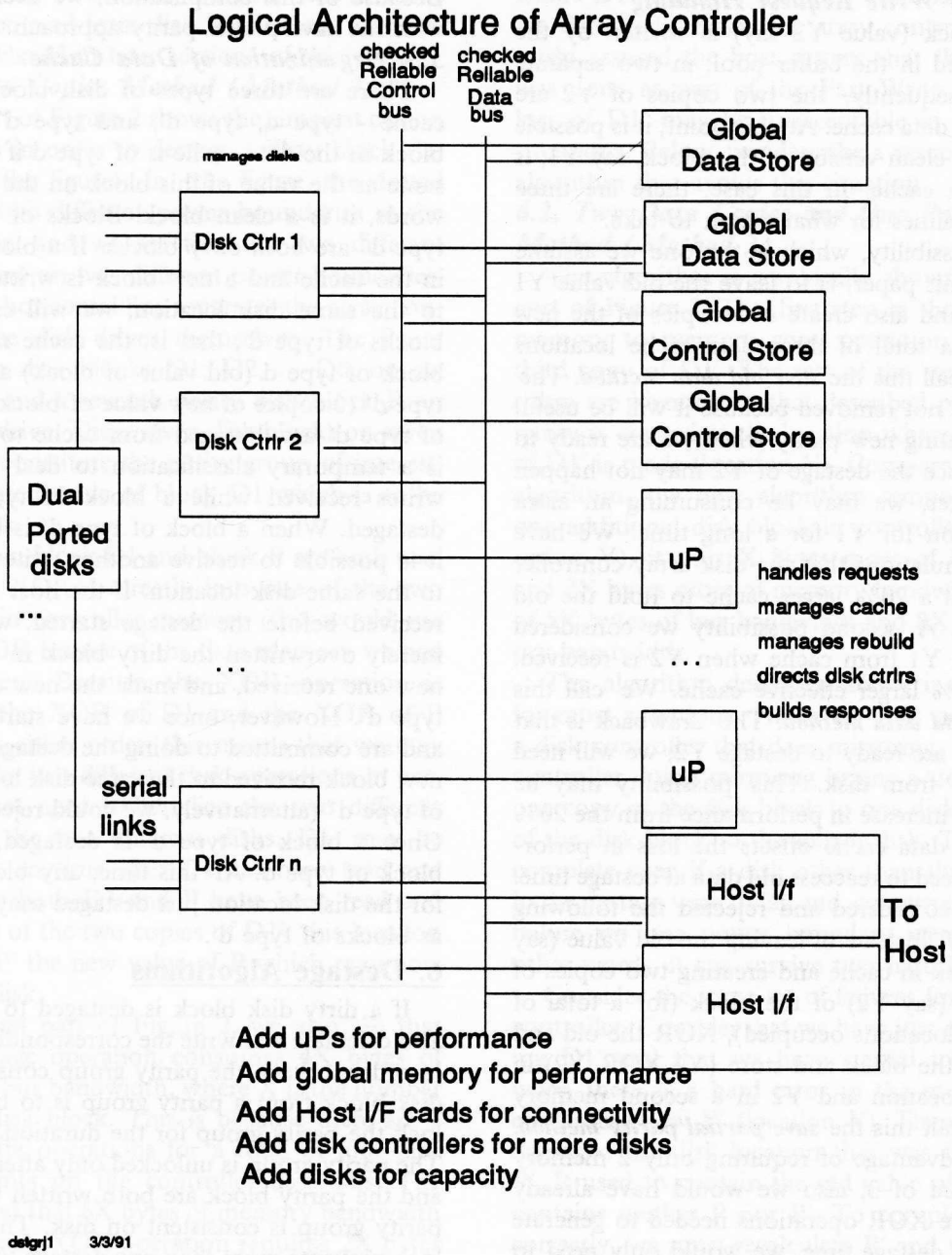

Figure 1: Hagar Array Controller

parity cache would eliminate the need to read it from disk at destage time. There is some argument for not having a parity cache at all and to make the data cache larger. This is because parity blocks in the parity cache only help destage performance, whereas data blocks in the data cache can help both read performance (due to cache hits) and destage performance (by eliminating the need to read old data from disk). Furthermore, data blocks are brought into the data cache naturally as a result of host block requests; parity blocks, on the other hand, must be specially brought into the cache when a particular data block is read in the hope that the host will subsequently write the data block.

HPE, Exh. 1010, p. 9

5.2. *Details of Write Request Handling*

When a block (value Y2 say) is written by the host, it is placed in the buffer pool, in two separate locations. Subsequently, the two copies of Y2 are moved into the data cache. At this point, it is possible that a previous clean version of this block, say Y1, is already in data cache. In this case, there are three different possibilities for what action to take.

The first possibility, which is the one we assume in the rest of this paper, is to leave the old value Y1 in data cache and also create two copies of the new value Y2, for a total of three data cache locations occupied. We call this the ***save old data method.*** The old value Y1 is not removed because it will be useful to us in calculating new parity when we are ready to destage Y2. Since the destage of Y2 may not happen until much later, we may be consuming an extra memory location for Y1 for a long time. We have found from simulations that the disk array controller will need about a 20% larger cache to hold the old values of data. A second possibility we considered was to remove Y1 from cache when Y2 is received, giving us a 20% larger effective cache. We call this the ***overwrite old data method.*** The drawback is that now, when we are ready to destage Y2, we will need to reaccess Y1 from disk. This possibility may be attractive if the increase in performance from the 20% larger effective data cache offsets the loss in performance due to need to reaccess old data at destage time.

Finally, we considered and rejected the following third possibility. Instead of leaving the old value (say Y1) of the block in cache and creating two copies of the new value (say Y2) of the block (for a total of three memory locations occupied), XOR the old and new values of the block and store (Y1 XOR Y2) in one memory location and Y2 in a second memory location. We call this the ***save partial parity method.*** This has the advantage of requiring only 2 memory locations instead of 3; also we would have already done one of the XOR operations needed to generate new parity. At destage time, we would only need to read old parity, XOR it with (Y1 XOR Y2) to generate new parity, then write new parity to disk. However, the results of [3] indicate that there is a very high probability of receiving another write (say Y3) to the same disk location before we have had a chance to destage Y2. With our currently assumed approach (save old data method), we would merely overwrite the 2 memory locations containing Y2 with the new value Y3. However, if we went with an approach in which we had already XORed Y1 with Y2, we would need to first XOR Y2 to this result to get back Y1, then XOR the new value Y3 to get (Y1 XOR Y3). Because of this complication, we decided not to go with the save partial parity approach.

5.3. *Organization of Data Cache*

There are three types of disk blocks in the data cache - type d, type d′, and type d″. A particular block in the data cache is of type d if its value is the same as the value of this block on the disk - in other words, it is a clean block. Blocks of type d′ and of type d″ are both dirty blocks. If a block of type d is in the cache and a new block is written by the host to the same disk location, we will create two new blocks of type d′; that is, the cache now contains a block of type d (old value of block) and 2 blocks of type d′ (2 copies of new value of block). Only blocks of type d′ are destaged from cache to disk. Type d″ is a temporary classification to deal with new host writes received while a block of type d′ is being destaged. When a block of type d′ is being destaged, it is possible to receive another write from the host to the same disk location. If the host write had been received before the destage started, we would have merely overwritten the dirty block in cache with the new one received, and made the new one received of type d′. However, once we have started the destage and are committed to doing the destage, we mark any new block received to the same disk location as being of type d″ (alternatively, we could reject the request). Once a block of type d′ is destaged, it becomes a block of type d. At this time, any blocks of type d″ for the disk location just destaged may be reclassified as blocks of type d′.

6. Destage Algorithms

If a dirty disk block is destaged to disk, we must also calculate and write the corresponding parity block in order to keep the parity group consistent. When a disk block from a parity group is to be destaged, we lock the parity group for the duration of the destage. The parity group is unlocked only after the disk block and the parity block are both written to disk and the parity group is consistent on disk. The parity group lock prevents more than one destage to be in progress simultaneously to any one parity group. While not explicitly referred to in the algorithms that follow, a parity group is locked before a destage begins and is unlocked after the destage completes.

We begin by considering the case where only one of the data blocks of a parity group is dirty in the data cache and needs to be destaged; later we will also consider cases where more than one block of a parity group needs to be destaged. To simplify the discussion, we assume that when a dirty block is to be destaged, other blocks of the parity group are not in the data cache even in clean form. We also assume

HPE, Exh. 1010, p. 10

that the old value of the dirty block is not in cache and needs to be read from disk. Both these assumptions will be relaxed in later sections of this paper.

6.1. Two Data Copies Method (Method 1)

The first part of Figure 2 shows the simplest option available to us in order to destage a dirty block (labelled D1′ in the figure). In this figure, the dotted line separates two different power boundaries in the array controller, and we see that the two different copies of D1′ are on two different power boundaries. Also, the solid horizontal line separates the array controller from the disk drives themselves. The figure shows six data disk blocks D1, D2, ... D6, on six different disks and a seventh parity disk block P on a seventh disk drive. These seven disk blocks on seven different disks constitute the parity group of interest. D1′ is an updated value of block D1 which is to be destaged to disk.

In this option, block D1 and block P are both read from disk and XORed directly into one of the two D1′ locations in controller memory (this would use the store & XOR feature of the data memory we had described earlier). Because the XOR operation is commutative, the XOR of D1 and the XOR of P may happen in either order; this means that we may actually start the two different disk operations in parallel and do not need to serialize the two different disk seeks on the two different disks. D1′ may be written to disk anytime after D1 has been read and XORed. When both D1 and P have been read and XORed to one of the two copies of D1′, this location now contains P′ the new value of P which may now be written to disk.

From the first part of Figure 2, we also see that the entire destage operation consumes 4X bytes of controller data bus bandwidth, where X is the number of bytes in a disk block. This is because there are 2 read and 2 write operations for a total of four disk block movements on the controller data bus. The figure also shows that 6X bytes of memory bandwidth is consumed (each XOR operation requires 2X bytes of memory bandwidth, X to read and X to write). On the other hand, a disk controller that does mirroring which only needs 2X bytes of bus bandwidth and 2X bytes of memory bandwidth.

The simple destage algorithm described above is robust in that no single error can cause it to fail. However, it would not be considered robust enough for many situations, since there are multiple failures that can cause loss of data. For example, a transient error during the process of XORing D1 into one of the two D1′ locations, coupled with a hard failure or loss of the other copy of D1′ results in a situation where D1′ is lost by the array controller (both copies are damaged). Since the array controller had previously assured the host system that the write of D1′ was done as part of the Fast Write operation, this loss of D1′ may be unacceptable in many kinds of situations. Below, we describe a more robust destage algorithm that avoids this situation.

6.2. Two Data Copies and One Parity Copy Method (Method 2)

The algorithm is graphically shown in the second part of Figure 2. The first step in the algorithm is a memory to memory copy operation that creates a third copy of D1′. The rest of the steps of the algorithm are identical to that described previously. New parity is created at the location where the third copy of D1′ is made (location Y). Compared to the earlier algorithm, the new algorithm temporarily occupies one additional disk block in controller memory (location Y), it uses X bytes more of bus bandwidth and 2X bytes more of memory bandwidth, for a total of 5X bytes of bus bandwidth and 8X bytes of memory bandwidth.

The algorithm described above is robust enough for most situations. However, it is not as robust as a disk controller that does mirroring. When the disk controller doing mirroring begins a destage, it writes one copy of the disk block to one disk, another copy of the disk block to the mirror disk. The destage can complete even if a disk other than the two involved in the destage were to fail and, concurrently, a memory failure on one power boundary were to occur. In other words, it can survive two hard failures.

Consider the same set of failures for the disk array controller. Consider that we have just completed writing D1′ and that we have started to write new P′ when there is a hard error in the memory location containing new P′ (location Y). Therefore, we have damaged the disk location that was to contain new P′. It used to contain the old value of P, but it now contains neither P nor P′. To complete the destage correctly, we must recalculate P′ and write P′ to this disk location. Since we already wrote D1′ to disk, we can no longer calculate P′ the way we did before, which was by reading D1 and using D1 to calculate P′. Since D1 on disk has already been overwritten with D1′, we must recalculate P′ by reading D2, D3, ..., D6 and XORing them all together and with D1′. If one of the disks containing D2, D3, ..., D6 also fails, we are unable to recalculate new P′. Therefore, a set of failures that did not prevent a mirrored disk controller from destaging could not be handled by the array controller using the destage algorithm we have described in this section. In the next section, we de-

HPE, Exh. 1010, p. 11

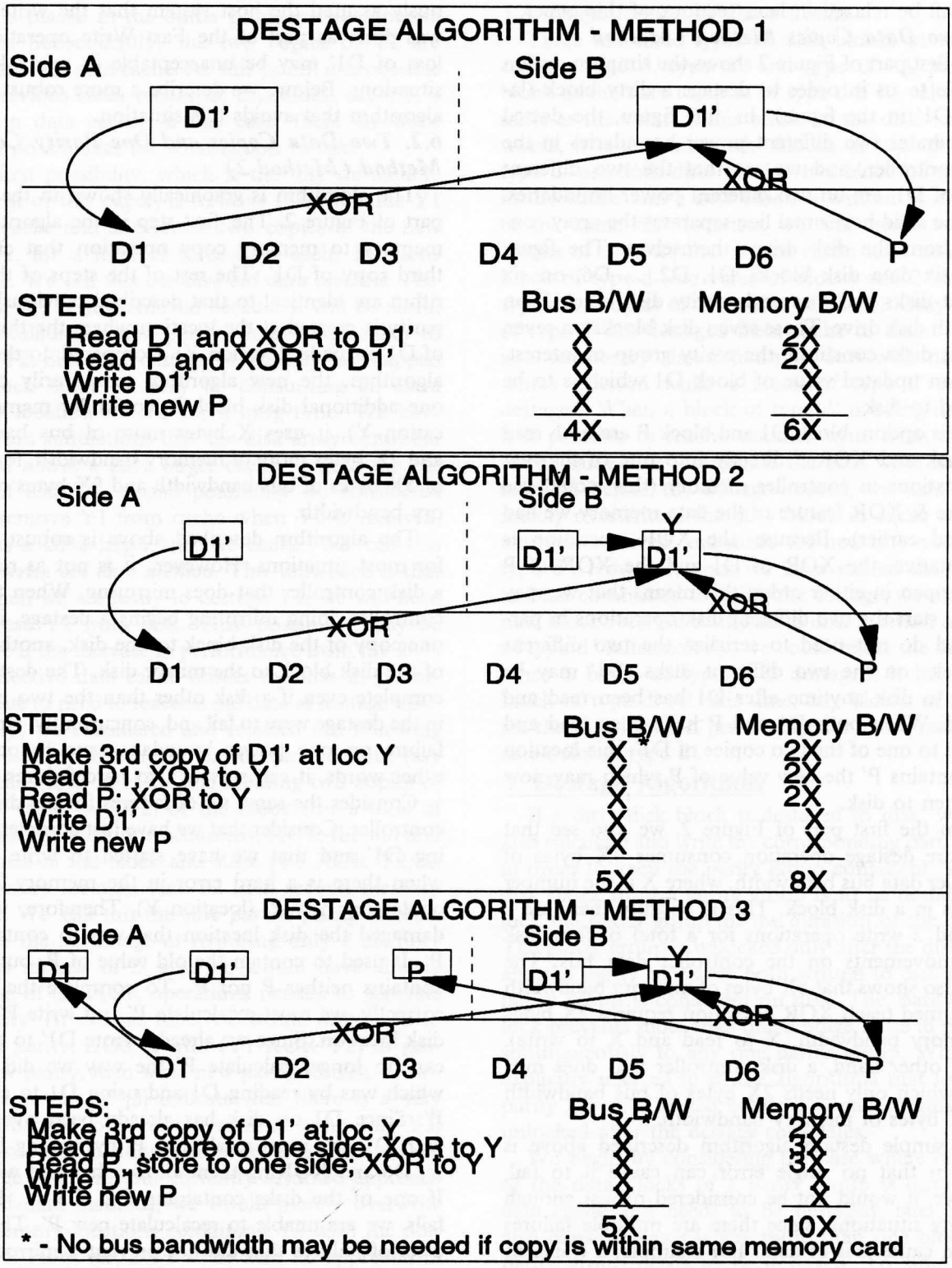

Figure 2: Hierarchy of Destage Algorithms

HPE, Exh. 1010, p. 12

scribe a destage algorithm that makes the array controller as robust as a disk controller that uses mirroring.

6.3. Two Data Copies and Two Parity Copies Method (Method 3)

The third part of Figure 2 graphically demonstrates the most robust of our destage algorithms. (See [7] for other robust algorithms.). The steps are: make a third copy of D1′ at location Y; in any order, read D1 from disk and XOR it to Y and also make a copy of D1 on the other power boundary, read P from disk and XOR it to Y and also make a copy of P on the other power boundary; after all reads and XORs are done, write D1′ and new P′ (from location Y) to disks in any order. By waiting for all reads and XOR operations to complete before beginning any writes, this algorithm is robust against a combination of three failures; the hard failure of one of the two memory cards, the failure of one of the disks containing D2, D3, ..., D6, and a transient failure while reading and XORing D1 or P. Key to achieving this robustness is ensuring that old values of D1 and P are read into a different power boundary than location Y which contains the third copy of D1′. This, in effect, means that two copies of new parity are present in cache before we begin writing to the disks; one at location Y and one which can be created on the other power boundary by XORing D1′, D1 and P. The price to be paid for the increased robustness of the destage algorithm is performance (since writes must wait until all reads are done) and resource consumption (since it now needs two more temporary locations in memory, uses 10X bytes of memory bandwidth and 5X bytes of bus bandwidth).

6.4. Arrays Versus Mirroring Comparison

We compare a disk controller that performs mirroring to one that implements a RAID-5 array using one of the three different destage algorithms described in the previous section. The comparison is in terms of resources consumed (internal bus bandwidth, internal memory bandwidth and number of internal memory locations occupied) for write operations. It is assumed that all disk controllers use the fast write technique so that write operations proceed in two stages; one stage in which the write is received and buffered and a second stage in which the dirty pages are destaged.

Type of ctlr	Stage 1 Bus B/W	Stage 1 Mem B/W	Stage 2 Bus B/W	Stage 2 Mem B/W	Total Bus B/W	Total Mem B/W	Mem Locs
Mirror	X	2X	2X	2X	3X	4X	2
Method 1	X	2X	4X	6X	5X	8X	2
Method 2	X	2X	5X	8X	6X	10X	3
Method 3	X	2X	5X	10X	6X	12X	5

From the above table, we see that the simplest parity array controllers require 67% more bus bandwidth and twice as much memory bandwidth as disk controllers that employ mirroring. The most robust parity array controllers need twice the bus bandwidth and thrice the memory bandwidth of disk controllers that perform mirroring. Furthermore, during the destage process, the most robust parity array controllers require 2.5 times as much temporary cache space as disk controllers that perform mirroring.

6.5. Other Destage Cases

It turns out that we have only considered one of four possible destage situations that may arise. Figure 3 shows all the four cases and indicates that which case applies depends on how many data blocks of the parity group are to be destaged and how many of them are in cache (by definition, all the blocks to be destaged are in cache in two separate locations). In the figure, all blocks in cache that are dirty are designated by Di′. These are the blocks to be destaged. The four cases are:

- Destage entire parity group
- Destage part of parity group; entire parity group in cache
- Destage part of parity group; read remaining members of parity group to create new parity
- Destage part of parity group; read old values of data and parity to create new parity

These four cases are described below. In general, we describe the most robust forms of the destage algorithms to be used in each case.

6.5.1. Destage Entire Parity Group

In this case, we first allocate a buffer (P1) to hold parity and initialize it to zero. Each block in the parity group is written to disk and simultaneously XORed with P1. After all data blocks have been written, write P1 (which contains the new parity) to disk.

6.5.2. Destage Part of Parity Group; Entire Parity Group in Cache

We first make a copy of one of the data blocks in the parity group that is not to be destaged at location P1. P1 will eventually contain the new parity to be written to disk. Each dirty block in the parity group is written to disk and simultaneously XORed with

HPE, Exh. 1010, p. 13

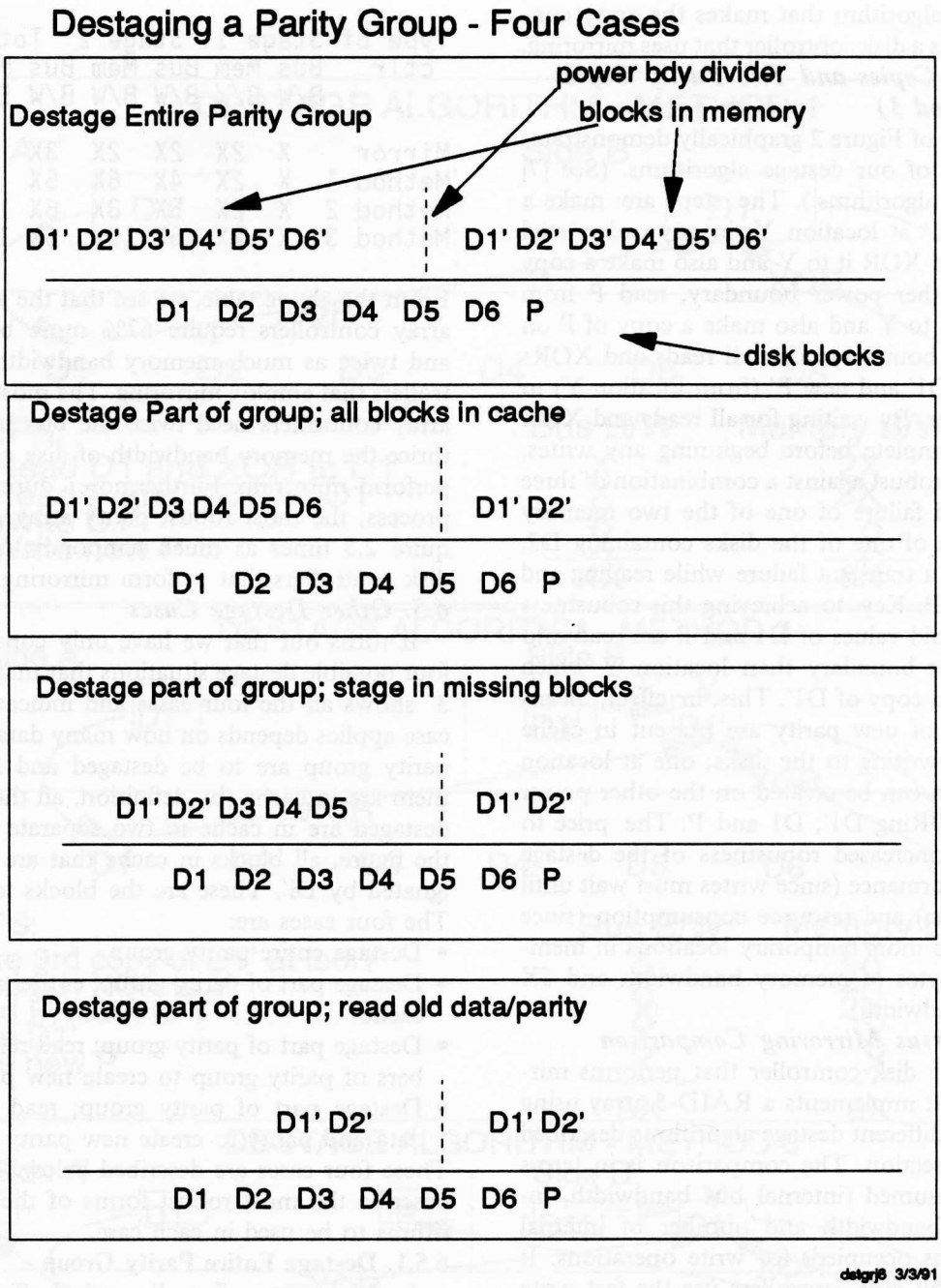

Figure 3: Cases for Destaging a Parity Group

P1. The other blocks of the parity group are only XORed with P1. After all XORing is completed, write P1 (which contains the new parity) to disk.

The above approach has a small exposure. Consider that we have completed writing one or more of the dirty blocks to disk, but have not yet completed generation of new parity in P1. Now, consider that we lose a memory card that contains a clean data block that was going to be used to generate the new parity in P1. We will now need to read this block from disk, and an exposure arises if we cannot do so. The exposure is small, since the fact that this block was in the data cache most likely implies that we were able to either read or write this disk block in the recent

HPE, Exh. 1010, p. 14

past. If the exposure is considered large, we have the following alternative destage policy.

First make a copy of one of the data blocks in the parity group that is not to be destaged at location P1. XOR all non-dirty data blocks of the parity group into P1. Make copy of result in P1 in other power boundary at P2. Now, write each dirty data block to disk while XORing simultaneously with P1. After all XORing is complete, write P1 which contains the new parity. If we lose a memory card during destage, the copy of the result we saved in P2 can be used to complete the generation of new parity without need to read any disk block.

6.5.3. Destage Part of Parity Group; Read rest from disk

The assumption here is that only a very few of the blocks of the parity group are not in cache, so that it is faster to read these missing members in to generate the new parity than it is to read the old values of the blocks to be destaged.

In this case, we first allocate and zero out a buffer P1. Every data block of the parity group that is missing in cache is read in from disk and XORed into location P1. After all reads have completed, each dirty block in the parity group is both written to disk and XORed with P1 simultaneously. Other blocks of the parity group that were neither dirty, nor missing in cache originally, are XORed with P1 but not written to disk. Eventually, write new parity in P1 to disk.

The reason for first completing the reads of the data blocks missing in cache before allowing any writes to take place is to ensure that all such missing data blocks are readable. If one of these data blocks is unreadable, a different algorithm (the one to be described next) would be used for destage.

6.5.4. Destage Part of Parity Group; Read Old Values from Disk

We first create a third copy of one of the data blocks (say D) to be destaged (say at location C). The old value of every data block to be destaged to disk is read in from disk to a location on a different power boundary from C, and it is also simultaneously XORed into location C. The old value of parity is also read in from disk to a location on a different power boundary from C and simultaneously XORed with C. As before, the reading of old data blocks and the reading of the old parity block can proceed in parallel. After the old value of a block has been read and XORed, its new value can be written to disk and XORed with C (if needed; block D does not need to be XORed with C since we started with a copy of block D in location C) at any subsequent time. After all data blocks have been written and the old parity block has been read, write C which contains the new parity.

7. Conclusions

In this paper, we have described a technique called Fast Write to improve the performance of disk arrays that use the parity technique. This technique involves use of battery-backed or Non-Volatile Store in the array controller to hold blocks written by the host system. These host-written blocks are destaged to disk asynchronously. Fast Write is expected to have four advantages: it can eliminate disk time from the write response time as seen by the host; it can eliminate some disk writes due to overwrites caused by later host writes to dirty blocks in cache; it can reduce disk seeks because destages will be postponed until many destages can be done to a track or cylinder; it can convert small writes to large writes.

We used an array controller organization which places the XOR logic (needed for parity generation) close to the cache memory in the controller and not as a separate XOR unit as has been proposed for other array controller designs ([9]). We showed that such an approach can reduce internal bus bandwidth requirements for array controllers. We described an organization of the data memory in the disk controller to support Fast Write which involved caching both data and parity blocks. We proposed that the data cache needs to support three different kinds of disk data blocks for efficiently handling Fast Writes. We articulated three alternatives for handling Fast Write hits - save old data, overwrite old data, save partial parity - and examined their pros and cons. For what appears to be the preferred alternative, we estimated that the disk controller would need a 20% larger cache than traditional or mirrored disk controllers that use Fast Write (to achieve the same hit ratios). We showed that parity group locking is an effective technique to avoid incorrect calculation of parity during concurrent destage and rebuild activity. Finally, we described the destage of disk blocks from the data cache in great detail. Four different destage cases were identified. By using one of the destage cases as an example, we described a hierarchy of three different destage algorithms of increasing degrees of robustness to failures in the disk subsystem. These three algorithms were the ***two data copies method***, the ***two data copies and one parity copy method*** and the ***two data copies and two parity copies method***. These destage algorithms were compared against those that would be used by a disk controller employing mirroring instead of the parity technique. We were able to show that the least robust array controllers require 67% more bus bandwidth and twice as much memory

HPE, Exh. 1010, p. 15

bandwidth as disk controllers that employ mirroring. The most robust parity array controllers, on the other hand, need twice the bus bandwidth and thrice the memory bandwidth of disk controllers that perform mirroring. These results indicate that while mirroring is more expensive overall (because of the need for more disks), disk array controllers are likely to be somewhat more expensive than controllers that do mirroring.

We also posed the following questions for future research:

- How much of the cache should be devoted to hold parity blocks instead of data blocks? Parity blocks are useful during destage, but data blocks can help both read performance (through read hits in the cache) and destage performance (by eliminating the need to read old data from disk at destage time). Furthermore, data blocks are brought into the data cache naturally as a result of user requests; parity blocks, on the other hand, must be specially brought into the cache when a particular data block is read in the hope that the host will subsequently write the data block.
- When a particular data block is selected for destage, should we also destage other blocks on the same track? or on the same cylinder? If these other blocks were only recently received from the host, then it may be better not to destage them immediately, since we might expect the host to write these blocks again. Therefore, the destage policy must be carefully chosen to trade-off the reduction in destages that can be caused by overwrites of dirty blocks if we wait until dirty blocks become LRU versus the reduction in seeks that can be achieved if we destage multiple blocks at the same track or cylinder position together. Should we also take into account the utilization of devices so that destages are begun to devices that are currently under-utilized?
- Since every dirty block in the controller cache occupies two memory locations until the block is destaged, the sooner we destage the dirty block, the sooner we can reclaim two memory locations. How do we trade-off this requirement for a quick destage of dirty blocks versus the requirement to hold off the destage in the expectation of overwrites that reduce the number of destages needed?
- What is the appropriate method for handling write hits? Should we leave the old data in cache since it is needed at destage time and take the attendant drop in effective cache size, or should we overwrite the old data in cache and reaccess it from disk at destage time?
- What is the appropriate granularity at which to do locking? We have proposed parity group locking be used, but is either a coarser or finer granularity more reasonable? What should the duration of locking be? Is it better to hold the lock until both data and parity are written to disk as proposed in this paper, or should we release the lock sooner.

8. Acknowledgements

Jim Brady originated the idea that we build the XOR hardware close to the memory in the controller.

9. References

1. Clark, B. E. et. al., Parity Spreading to Enhance Storage Access, *United States Patent* **4,761,785** (Aug. 1988).
2. Gray, J. N. et. al., Parity Striping of Disk Arrays: Low-Cost Reliable Storage With Acceptable Throughput, *Tandem Computers Technical Report* **TR 90.2** (January 1990).
3. Hyde, J., Cache Analysis Results, *Personal Communication* (1991).
4. Menon, J. M. and Hartung, M., The IBM 3990 Disk Cache, *Compcon 1988* (San Francisco, June 1988).
5. Menon, J. and Mattson, D., Performance of Disk Arrays in Transaction Processing Environment, *12th International Conference on Distributed Computing Systems* (1992) pp. 302–309.
6. Menon, J., Roche, J. and Kasson, J., Floating Parity and Data Disk Arrays, *Journal of Parallel and Distributed Computing* (Jan. 1993).
7. Menon, J. and Cortney, J., The Architecture of a Fault-Tolerant Cached RAID Controller, *IBM Research Report* **RJ 9187** (Jan. 1993).
8. Patterson, D. A., Gibson, G. and Katz, R. H., A Case for Redundant Arrays of Inexpensive Disks (RAID), *ACM SIGMOD Conference* (Chicago, Illinois, June 1988).
9. Lee, Ed, Hardware Overview of RAID-II, *UC Berkeley RAID Retreat* (Lake Tahoe, Jan 1991).
10. Ousterhout, J. and Douglis, F., Beating the I-O Bottleneck: Case for Log-Structured File Systems, *UC Berkeley Research Report* **UCB-CSD-88-467** (Berkeley, CA, October 1988).

HPE, Exh. 1010, p. 16

SESSION 5:

Multiprocessor Caches