NASA-CR-195759

Febuary 1994

UILU-ENG-94-2208
CRHC-94-07

Center for Reliable and High-Performance Computing

REDUNDANT DISK ARRAYS IN TRANSACTION PROCESSING SYSTEMS

Antoine Nagib Mourad

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-94-2208 CRHC-94-07	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (*If applicable*) N/A	7a. NAME OF MONITORING ORGANIZATION National Aeronautics Space Administration Office of Naval Research
6c. ADDRESS (*City, State, and ZIP Code*) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (*City, State, and ZIP Code*) Moffet Field, CA 95406 Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613 / N00014-91-J-1283
8c. ADDRESS (*City, State, and ZIP Code*) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Redundant Disk Arrays in Transaction Processing Systems

12. PERSONAL AUTHOR(S) MOURAD, Antoine Nagib

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1994- 02-08	125

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	databases, redundant disk arrays, recovery, performance analysis, algorithms, heuristics

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

We address various issues dealing with the use of disk arrays in transaction processing environments. We look at the problem of transaction undo recovery and propose a scheme for using the redundancy in disk arrays to support undo recovery. The scheme uses twin page storage for the parity information in the array. It speeds up transaction processing by eliminating the need for undo logging for most transactions. The use of redundant arrays of distributed disks to provide recovery from disasters as well as temporary site failures and disk crashes is also studied. We investigate the problem of assigning the sites of a distributed storage system to redundant arrays in such a way that a cost of maintaining the redundant parity information is minimized. Heuristic algorithms for solving the site partitioning problem are proposed and their performance is evaluated using simulation. We also develop a heuristic for which an upper bound on the deviation from the optimal solution can be established.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

REDUNDANT DISK ARRAYS IN TRANSACTION PROCESSING SYSTEMS

BY

ANTOINE NAGIB MOURAD

Ingénieur, Ecole Polytechnique, 1986
M.S., University of Texas at Austin, 1989
Ingénieur, Ecole Nationale Supérieure des Télécommunications, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

REDUNDANT DISK ARRAYS IN TRANSACTION PROCESSING SYSTEMS

Antoine Nagib Mourad, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1993
W. Kent Fuchs and Daniel G. Saab, Advisors

Disk arrays are a cost-effective approach for building large, reliable and high performance storage subsystems. They provide high transfer rates by striping data over multiple disks and use a parity scheme for recovering from any single disk failure. Transaction processing is a large and growing segment of commercial computing. There is a major need in that environment for large, highly available and fast I/O subsystems.

In this thesis, we address various issues dealing with the use of disk arrays in transaction processing environments. We look at the problem of transaction undo recovery and propose a scheme for using the redundancy in disk arrays to support undo recovery. The scheme uses twin page storage for the parity information in the array. It speeds up transaction processing by eliminating the need for undo logging for most transactions. The use of redundant arrays of distributed disks to provide recovery from disasters as well as temporary site failures and disk crashes is also studied. We investigate the problem of assigning the sites of a distributed storage system to redundant arrays in such a way that the cost of maintaining the redundant parity information is minimized. Heuristic algorithms for solving the site partitioning problem are proposed and their performance is evaluated using simulation. We also develop a heuristic for which an upper bound on the deviation from the optimal solution can be established.

Another part of the thesis focuses on the performance of various disk array organizations in transaction processing environments. Trace data from large scale commercial transaction processing sites are used to evaluate and compare the performance of those organizations. We investigate the use of a nonvolatile cache in the disk array controller to reduce the effect of the high cost of small writes. For noncached systems, we evaluate two redundant disk array organizations and compare them to mirrored disks and nonredundant, nonstriped organizations. For cached systems, we consider the above four organizations as well as a disk array organization that uses a dedicated disk for parity in each array and buffers parity updates in the controller cache before spooling them to the parity disk.

DEDICATION

Dedicated to my parents.

ACKNOWLEDGMENTS

I gratefully acknowledge the advice and guidance rendered by my co-advisors Professors W. Kent Fuchs and Daniel G. Saab during the course of my work at the University of Illinois. I would also like to thank the members of my doctoral committee: Jane Liu, Prithviraj Banerjee, and Wen-mei Hwu for their support and helpful advice.

I am grateful to Robert Horst, Jim Carley, Srikanth Shoroff, Robert van der Linden, and Susan Whitford of Tandem Corporation for providing the TMF audit tapes and for answering numerous questions. Kumar Goswami helped with the CSIM simulation package. His efforts are greatly appreciated. I would like to express special thanks to Professor Weiping Shi of the University of North Texas for his suggestions on the site partitioning problem. Thanks are also due to Kent Treiber, Ted Messinger, Honesty Young and Robert Morris of the IBM Almaden Research Center for providing the DB2 traces and for clarifying several issues. My sincere appreciation to A.L. Narasimha Reddy, Arun Swami and Jai Menon, also from the IBM Almaden Research Center, for some very useful discussions. I am especially grateful to Dr. Joseph Rahmeh for his support and encouragement throughout my graduate studies.

My work at the Center for Reliable and High Performance Computing was supported by NASA under Contract NAG 1-613 and by the Office of Naval Research under Grant N00014-91-J-1283.

I would also like to thank my friends Jalal Wehbeh, Bob Janssens, Paul Chen, Shyh-kwei Chen, Neal Alewine, Yi-min Wang, Nancy Warter, Kumar Goswami, Luke Young, Gwan Choi, Aloke Gupta, Vicki McDaniel, Carolin Rouse and Sunitha Bellino for making my experience at CRHC a pleasant one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The I/O Bottleneck

Processor speeds have been improving by almost a factor of two every year. Memory densities have been doubling every two years. Memory access speeds have also been improving rapidly. Disks, however, have moving parts. Their access speeds have not kept pace with improvements in processor and memory technologies. Seek times have only improved by about 7% a year [1]. This mismatch between processor/memory speeds and disk speeds has created a bottleneck for most applications. The CPU-I/O gap is expected to widen in the future and affect even more applications. Another important trend is the significant decrease in small disk prices due to the high volumes in the PC market. The above trends have led to the development of disk array systems.

Striped disk arrays have been proposed and implemented for increasing the transfer bandwidth in high performance I/O subsystems [2–5]. In order to allow the use of a large number of disks in such arrays without compromising the reliability of the I/O subsystem, redundancy is included in the form of parity information [6].

1.2 Media Recovery

Reliable storage is a necessary feature in transaction processing systems requiring high availability. Media failure in such systems is traditionally dealt with by periodically generating archive copies of the database and by logging updates to the database performed by committed transactions between archive copies into a redo log file. When a media failure occurs, the database is reconstructed from the last copy and the log file is used to apply all updates performed by transactions that committed after the last copy was generated. In such a case, a media failure causes significant downtime and the overhead for recovery is quite high. For large systems, e.g., with over 150 disks, the mean time to failure (MTTF) of the permanent storage subsystem can be less than 28 days.[1]

Mirrored disks have been employed to provide rapid media recovery [7]. However, disk mirroring incurs a 100% storage overhead which is prohibitive in many cases. Redundant disk array organizations [6, 8] provide an alternative for maintaining reliable storage.[2]

1.3 Redundant Disk Arrays

1.3.1 Data striping

Patterson et al. [6] have presented several possible organizations for Redundant Arrays of Inexpensive Disks (RAID). One interesting organization for transaction processing environments is RAID with rotated parity (RAID5) in which blocks of data are inter-

[1] Assuming an MTTF of 100,000 hours for each disk.

[2] However, even when disk mirroring or redundant disk arrays are used, archiving and redo logging may still be necessary to protect the database against operator errors or system software design errors.

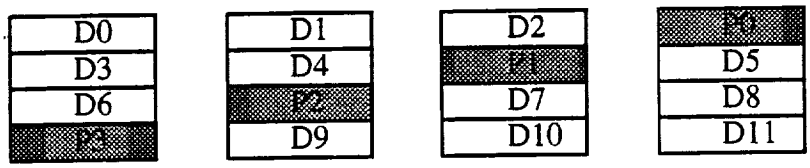


Figure 1.1 RAID5 with four disks.

leaved across N disks while the parity of the N blocks is written on the $(N+1)$st disk. The parity is rotated over the set of disks in order to avoid contention on the parity disk. Figure 1.1 shows the data and parity layout in a RAID5 organization with four disks. This pattern is repeated for the next set of blocks. An important parameter in the RAID5 organization is the striping unit, which can be defined as the "maximum amount of logically contiguous data stored on a single disk" [9].

The RAID5 organization allows both large (full stripe) concurrent accesses or small (individual disk) accesses. For a small write access, the data block is read from the relevant disk and modified. To compute the new parity, the old parity has to be read, XORed with the new data and XORed with the old data. Then the new data and new parity can be written back to the corresponding disks.

The RAID4 organization shown in Figure 1.2 is similar to the RAID5 organization except for the fact that the parity for the N data disks is written on one parity disk. One disadvantage of the RAID4 organization is that the parity disk may become a bottleneck. The RAID4 organization becomes attractive when a nonvolatile cache is used in which case parity blocks can be cached and written asynchronously to the parity disk. This will be discussed further in Chapter 4.

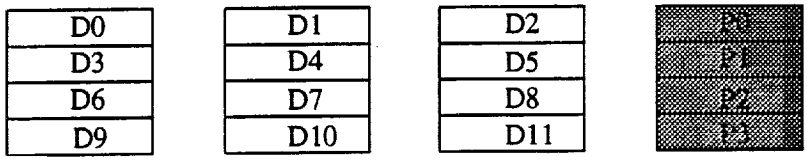


Figure 1.2 RAID4 with four disks.

1.3.2 Parity striping

Gray et al. [8] studied ways of using an architecture such as RAID in transaction processing systems. They argued that because of the nature of I/O requests in OLTP systems, namely, a large number of small accesses, it is not convenient to have several disks servicing the same request. In other words, since in transaction processing systems I/O time is dominated by seek time and rotational latencies rather than by transfer time, it is not advantageous to have a request spread over multiple disks because that will make all those disks spend a significant amount of time seeking and rotating in order to decrease an already small transfer time. Hence, the organization shown in Figure 1.3 was proposed. The shading in the figure indicates the areas that belong to the same parity group. It is referred to as parity striping, which consists of reserving an area for parity on each disk and writing data sequentially on each disk without interleaving. For a group of $N + 1$ disks, each disk is divided into $N + 1$ areas; one of these areas on each disk is reserved for parity and the other areas contain data. N data areas from N different disks are grouped together in a *parity group* and their parity is written on the parity area of the $(N + 1)$st disk.

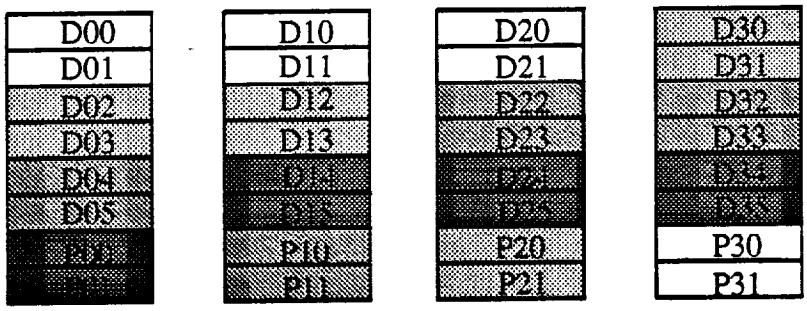


Figure 1.3 Parity striping of disk arrays.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2 we propose a technique for using disk arrays to support transaction UNDO recovery in database systems. A detailed analytical model is developed to evaluate the performance of the proposed scheme and describe some simulations results that validate our findings. Chapter 3 deals with issues involved in using a disk array scheme in a distributed setting to provide for disaster recovery as well as recovery from disk crashes and temporary site failures. We investigate the problem of partitioning the sites of a geographically distributed storage system into redundant arrays in such a way that the cost of the communication needed to update the remote parity information is minimized. Several heuristic algorithms for solving this NP-hard problem are proposed and their performance is evaluated. In Chapter 4, the performance of redundant disk array organizations in transaction processing environments is studied. The RAID5 and parity striping organizations are compared to mirrored disks and nonredundant, nonstriped disk subsystems. We also study the use of nonvolatile caches for buffering data and parity in order to reduce the effect of the small write penalty. Finally, Chapter 5 contains our conclusions.

CHAPTER 2

RECOVERY ISSUES IN DATABASES USING REDUNDANT DISK ARRAYS

2.1 Introduction

In a database system, rapid recovery may be necessary for restoring the database to a consistent state after a failure. Several types of failures can occur. The most typical are transaction aborts which can be due to program errors, deadlocks, or can be user initiated. When a transaction aborts, the recovery manager has to restore all database pages modified by the transaction to their previous states. The second type of failure is a system crash. In this case system tables maintained in main memory are lost. The recovery mechanism has to UNDO all updates made to the database by transactions that were active when the crash occurred and to REDO modifications performed by complete transactions and not yet reflected in the database at the time of the crash.

In this chapter, we present a technique that exploits the redundancy in disk arrays to support recovery from transaction and system failures in addition to providing fast media recovery. This is achieved by using a twin page scheme for storing the parity information making it possible to keep the old version of the parity along with the new version. The old version of the parity is used to undo updates performed by aborted transactions or

by transactions interrupted by a system failure. The proposed scheme works for both the RAID5 or Parity Striping disk array organizations.

In Sections 2.2 and 1.3 we briefly review several techniques for transaction recovery in database systems. In Section 2.3, we present our database recovery scheme. The results of our performance analysis are detailed in Section 2.4. Finally, in Section 2.4.3.2, we show some simulation results using data from a real application. Section 2.6 presents some conclusions.

2.2 Recovery Techniques

Recovery algorithms typically use some form of logging or shadowing. In the logging approach [10], before a new version (*after-image*) of a record or page is written to the database, a copy of the old version (*before-image*) is placed into a sequential log file. If a transaction aborts or the system crashes, the log file is analyzed and the state of the database is restored. In the shadowing approach, the update of a page is placed into a new physical page on disk [11, 12]. The physical pages containing the old versions are released after all updates of the committing transaction have been written to disk. One problem with the shadowing approach is dynamic mapping since it requires maintaining a very large page table which leads to high I/O overhead during normal processing. Another problem is the disk scrambling effect which decreases the sequentiality of disk accesses.

In describing and in analyzing our method, we will use the following taxonomy of database recovery algorithms introduced by Haerder and Reuter [13]. They classify recovery algorithms with respect to the following four concepts:

Propagation[1] of updates. The propagation strategy can be *ATOMIC* in which case any set of updated pages can be propagated to the database in one atomic action. In the $\neg ATOMIC$ case, propagation of updates can be interrupted by a system crash and database pages are updated-in-place.

Page replacement. Two policies can be used: the *STEAL* policy allows pages modified by uncommitted transactions to be propagated to the database before end-of-transaction (EOT); the opposite policy is referred to as $\neg STEAL$. No UNDO recovery is necessary with a $\neg STEAL$ policy.

EOT processing. Two categories exist: the *FORCE* discipline requires all pages modified by a transaction to be propagated before EOT; the opposite discipline is called $\neg FORCE$.

Checkpointing Schemes. Checkpointing is used to propagate updates to the database in order to minimize the number of REDO recovery actions to be performed after a crash. In the Transaction Oriented Checkpointing (TOC) scheme, a checkpoint is generated at the end of each transaction. This is equivalent to using the *FORCE* discipline in EOT-processing. Two other types of checkpoints can be used: Transaction Consistent Checkpoints (TCC) are generated during quiescent periods where no transac-

[1] Propagation to the database means that the new version is visible to higher level software. Updates can be written to disk without being propagated (e.g., shadowing).

tions are being processed, Action Consistent Checkpoints (ACC) are less restrictive and require that no update statements are processed during checkpoint generation.

2.3 RDA-Based Recovery

In the remainder of this chapter, we consider an I/O subsystem that is a collection of redundant disk arrays. The organization of the arrays is either parity striping or data striping (RAID with rotated parity). In the case of data striping, we assume that a large striping unit is used to ensure that I/O requests will typically be serviced by a single data disk. We also make the following assumptions: Communication between the main memory and the I/O subsystem is performed using fixed-size pages; Database pages are updated in place which implies that propagation is $\neg ATOMIC$; A *STEAL* policy is used, thus allowing modified pages to be propagated before EOT.

2.3.1 General description of the approach

The RDA-based recovery scheme makes use of the parity information present in the disk arrays to undo updates performed by aborted transactions. However, the parity is not sufficient by itself to undo all updates performed by an aborted transaction. Updates that cannot be undone using the parity are dealt with using one of the traditional recovery schemes.

A *page parity group* is the set of pages that share the same parity page. In the following, unless there is ambiguity, we will use the term parity group to denote a page

parity group. A parity group can be in one of two states: *clean* or *dirty*. A parity group is dirty when one of its data pages has been modified by a transaction and the modified version has been written back to the database before the transaction modifying it commits (using the notation of Haerder and Reuter, the page has been *stolen* from the buffer). Otherwise, the parity group is called clean. Only one modified data page per parity group can be written back to the database by uncommitted transactions without UNDO logging. If additional pages in the parity group have been modified and have to be written back to the database, then their before-images must be logged first. A dirty parity group goes back to the clean state when the transaction that caused it to become dirty commits. Figure 2.1 shows the state transition diagram of a parity group. A table in main memory contains the numbers of all parity groups that are in the dirty state. It also contains the number of the data page within the group that caused the group to be in the dirty state and the number of the parity page holding the updated parity. Only $\log N$ bits have to be used to store the data page number and one bit for the parity page number. The table is used to check whether a page updated by an active transaction can be written back to disk without UNDO logging.

When a transaction updates a page, that page can be written back to the database without UNDO logging if its parity group is clean or if its parity group is dirty and the update is for the same page that caused the group to move into the dirty state, i.e., the same page has been updated, stolen from the buffer then rereferenced by the

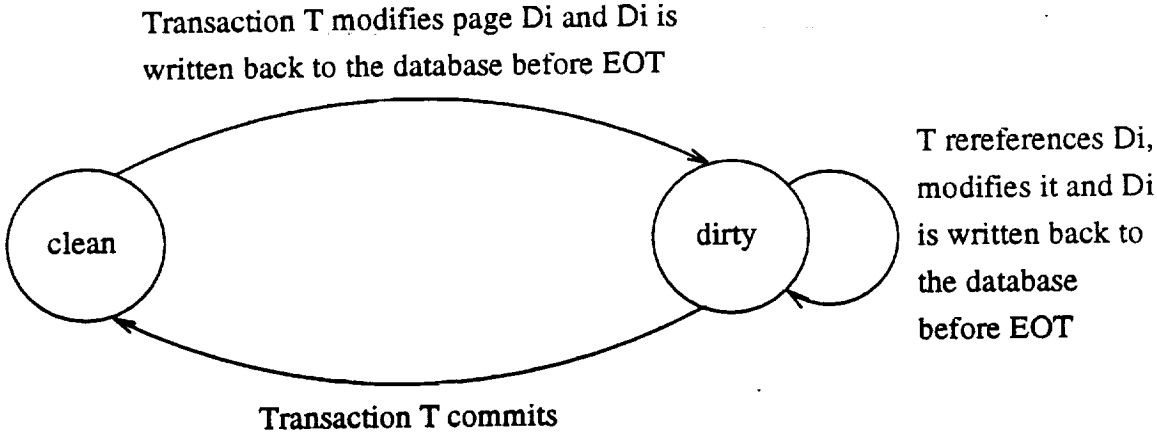


Figure 2.1 State transition diagram of a page parity group.

same transaction, updated and stolen again from the buffer before EOT.[2] Note that this does not affect the degree of concurrency or interfere with the locking policy used in the system. We do not specify when a transaction can or cannot modify a page. We only specify when a modified page can be written back to disk without UNDO logging.

If a single parity page is used, then when a group becomes dirty, the old parity information has to be kept in the parity page to be able to recover in case of a transaction failure. That would mean that when the transaction commits, the new parity has to be recomputed in order to update the parity page. That would require reading all of the data pages in the group in order to compute the new parity. To avoid that problem, a twin page scheme is used for the parity pages. The basic mechanism of the twin page scheme is as follows: one of the parity pages always contains the valid parity of the group while the other page contains obsolete parity information. When a data page is modified in a parity group, the obsolete parity page (P for example) is updated with the new parity

[2]Normally such an event should not occur often since buffer management algorithms are not supposed to replace a page that will be referenced again in the near future.

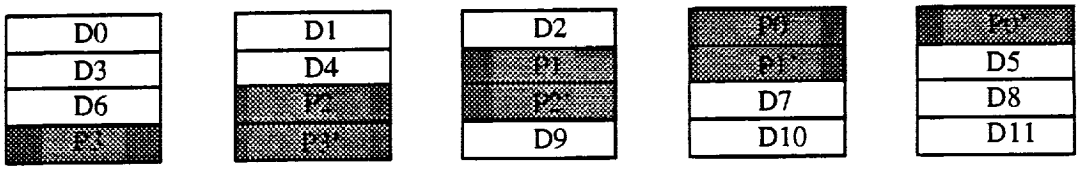


Figure 2.2 Data striping organization with the twin page scheme for the parity.

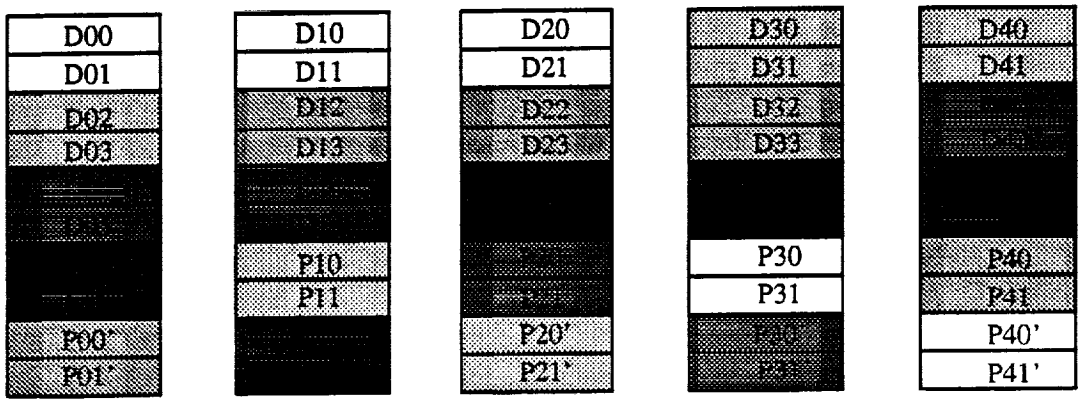


Figure 2.3 Parity striping organization with the twin page scheme for the parity.

of the array. If the transaction performing the update commits, then the modified parity page (P) becomes the valid parity page; otherwise, the other parity page (P′) remains the valid parity page and its contents are used to recover the data page that was modified by the failed transaction. Figures 2.2 and 2.3 show the data striping organization and the parity striping organization when the twin page scheme is used for the parity. Twin parity pages are denoted Px and Px' in the data striping case and Pxy and Pzy', with $z = (x + 1) \mathrm{mod}(N + 2)$, in the parity striping case. Figure 2.4 shows the contents of a parity group including the twin parity pages. To recover the old version of a data page after a transaction abort, it is sufficient to XOR the contents of both parity pages and the new data page: $\mathrm{D_{old}} = (\mathrm{P} \oplus \mathrm{P'}) \oplus \mathrm{D_{new}}$. When a parity group is dirty because one of its data pages D_i has been stolen from the buffer and another page D_j has to be

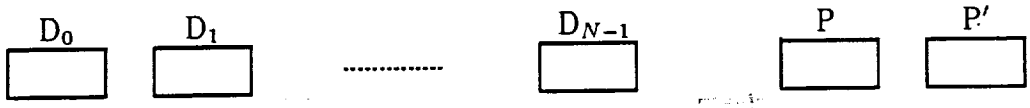


Figure 2.4 The contents of a page parity group.

written to disk, UNDO logging must be performed for D_j[3] then both parity pages P and P′ have to be updated, since when the group is dirty, it is necessary to maintain a current parity page reflecting the actual parity of the data on disk and an "old" parity page that would be used to recover the uncommitted data page D_i in case of a transaction abort. In all cases, when writing a data page to disk the corresponding parity page(s) must be updated first.

2.3.2 Twin page management

The twin parity pages are stored on different disks. This is required for the purpose of enabling the system to recover from transaction aborts following a disk failure. To identify which of the twin parity pages contains the valid parity information, a timestamp is stored in the page header. The page with the highest timestamp contains the valid parity information. When an update is undone after a transaction or system failure, the timestamp of the current parity page is reset to 0. Algorithm **Current_Parity** shown in Figure 2.5 selects the current parity page. When a data page is updated, both parity pages are read and one of them is selected for modification. Then the parity is computed and the modified parity page is written back to disk. To avoid reading both parity pages, a bit map can be maintained in main memory indicating the current parity page for each

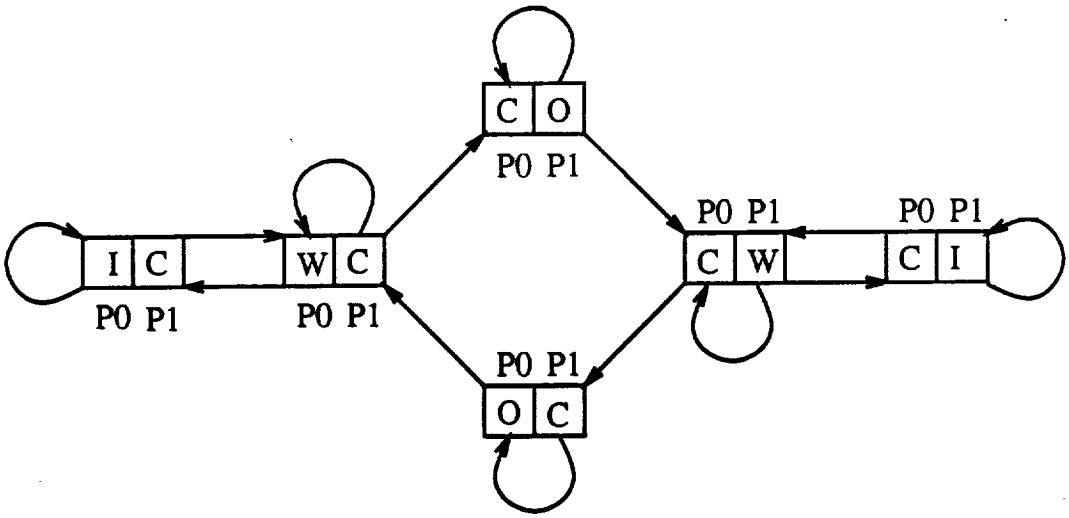
[3]The before-image of the page in the case of page logging or of the modified record(s) in the case of record logging must be written to a log file.

```
Current_Parity(pg)
begin
        Read twin parity pages in parity group pg;
        if Timestamp(P) > Timestamp(P') then
                Current_Parity ← P;
        else
                Current_Parity ← P';
end
```

Figure 2.5 Algorithm **Current_Parity** determines the current parity page.

of the parity groups in the database. However, such a bit map may not survive a system crash. Hence following a crash that destroys the map, algorithm **Current_Parity** will have to be used to identify the current parity page and to reconstruct the bit map. In this case, two bits would have to be used in the bit map for each parity group to code the three possible states: parity page P is the current parity page, parity page P′ is the current parity page or the information is not available and algorithm **Current_Parity** has to be used. Following a system crash, a background process that runs during idle periods of the system can be initiated to reconstruct the bit map.

Each of the twin parity pages can be in one of four states: *committed, obsolete, working* or *invalid* [14]. A parity page is committed when it contains the last committed parity update. It is obsolete when it contains old committed parity information. It is in the working state when it has been updated by an active transaction, and it is in the invalid state if the last transaction updating it has aborted. Figure 2.6 shows the state transition diagram of the twin parity pages.



C: committed; O: obsolete; I: invalid; W: working

Figure 2.6 State transition diagram of the twin parity pages.

2.3.3 Recovery from system failure

Following a system crash, we have to identify the transactions which have to be backed out and the pages which have been modified *on disk* by those transactions. A Begin-Of-Transaction (BOT) record has to be written to a log file after the transaction begins and before it writes back any modified pages to disk, and an EOT record must be written to the log file when the transaction commits. Modified database pages for which UNDO logging has been performed can be recovered by reading their before-images from the log. Modified database pages for which UNDO logging has not been performed can be recovered using the parity pages. However, information on these pages which have been written to the database without UNDO logging has to be saved in permanent storage. To solve this problem, a technique similar to the one used in TWIST [15] can be employed. In TWIST, a twin page scheme is used to store all database pages, no

before-image logging is performed and the same problem of identifying which pages to undo after a crash is encountered. The solution makes use of a log chain which consists of pointers stored in the page headers that link together pages modified by the same active transaction. In our case, only modified pages written back to the database before EOT without UNDO logging will be part of the log chain. The head of the chain though has to be logged along with the transaction id. I/O operations to maintain the log chain can be hidden behind regular I/O requests and do not significantly affect system performance.

2.4 Performance Analysis

To evaluate the benefit of RDA-recovery, an analytical model is developed to evaluate transaction throughput for different algorithms. Since the cost of maintaining parity information in a system with redundant disk arrays is relatively high, we do not advocate the use of RDAs solely for the purpose of supporting transaction and crash recoveries. The benefit of using RDA recovery in a system that already needs RDAs for the purpose of rapid media recovery is examined. This is done by comparing the throughput of systems using traditional recovery algorithms and redundant disk arrays to systems with the same recovery algorithms in combination with RDA recovery. Both page and record logging are considered and, in each case, we examine two different recovery algorithms and evaluate the improvement achieved by adding RDA recovery to them. As far as storage is concerned, the extra cost involved in using RDA recovery is that of the twin page scheme for the parity which is $(100/N)\%$ of the initial data storage cost.

RDA recovery reduces the amount of UNDO logging and, hence, is appropriate for systems using update-in-place, which implies $\neg ATOMIC$ propagation and a $STEAL$ policy for page replacement. We therefore restrict ourselves to the analysis of such algorithms. Within this class of algorithms, we examine both the $FORCE$ and $\neg FORCE$ strategies for EOT-processing. For algorithms of the type $\neg ATOMIC$, $STEAL$, $FORCE$, only a TOC checkpointing policy can be used. For algorithms of the type $\neg ATOMIC$, $STEAL$, $\neg FORCE$, both ACC or TCC checkpoints could be used; however, algorithms using ACC checkpointing were shown to outperform those using the TCC type [16].[4] Hence, we only look at the former type of checkpointing.

We use the same basic model as the one introduced by Reuter in his evaluation of the performance of several database recovery techniques [16]. We assume that the system is I/O bound and therefore we look only at the number of I/O requests required to perform a given operation. We also assume that the system is running continuously with no periodic shutdown. This implies that all cleanup activities required by the algorithm are accounted for in the cost calculations instead of assuming that they are performed by some background process or during shutdown periods.

The workload considered consists of a set of P transactions executing concurrently in the system. Transactions are of two types: *update* or *retrieval*. The fraction of update transactions is f_u. Each transaction accesses s database pages. The fraction of accessed pages that are modified by an update transaction is p_u. To characterize the behavior

[4] Also TCC checkpointing contradicts our assumption of a continuously running system since it requires the establishment of a quiescent point where no update transactions are present in the system.

of the database buffer, we use the communality C which denotes the probability that a page requested by an incoming transaction is present in the buffer. The number of page frames in the buffer is denoted by B. It is assumed that the buffer is sufficiently large so that once a transaction has referenced a page, the page will remain in the buffer until it is no longer needed by the transaction.[5]

The cost of recovery after a system crash is denoted by c_s and is measured by the number of page transfers between main memory and the disk subsystem required to perform recovery. The cost of executing a transaction is denoted by c_t. The transaction throughput r_t is defined as the number of transactions processed during an availability interval. An availability interval T is the period between two system crashes. Since all cost measures are evaluated in terms of number of I/O operations, we assume that the availability interval is measured in units of page transfers.[6]

If checkpointing is used, then the length of a checkpointing interval is denoted by I and is also measured in units of page transfers. The cost of generating a checkpoint is denoted by c_c. Assuming that the crash occurs in the middle of a checkpointing interval, the number of page transfers available for processing transactions in an availability interval is $T - c_s - c_c((T - c_s - I/2)/I)$. Hence the throughput is given by

$$r_t = ((T - c_s)(1 - c_c/I) + c_c/2)/c_t.$$

[5]The page could still be replaced before the transaction commits if a *STEAL* policy is used; however, if it is replaced it will not be rereferenced by the transaction.

[6]Mathematically, T can be defined as follows: $T = \frac{\text{length of availability interval in seconds}}{\text{time to transfer a page to/from disk in seconds}}$.

We assume that c_c is independent of I. Hence the optimal checkpointing interval can be easily derived from the following equation [16]:

$$\frac{dr_t}{dI} = (1/c_t)\left(-\frac{dc_s}{dI}(1 - c_c/I) + (T - c_s)(c_c/I^2)\right) = 0. \tag{2.1}$$

Let c_r denote the cost of updating a retrieval transaction and c_u that of an update transaction. Then c_t can be expressed as follows:

$$c_t = (1 - f_u)c_r + f_u c_u,$$

where c_r itself includes two components: the cost of reading pages that are not found in the database buffer and the cost of writing back the replaced pages if they have been modified. Hence,

$$c_r = s(1 - C) + \alpha s(1 - C)p_m, \tag{2.2}$$

where p_m denotes the probability that the replaced page was modified and α denotes the number of page transfers necessary to perform one write to the disk array. α is equal to 3 or 4 depending on whether or not the old data page is in the buffer at the time of writing the new data. For c_u, we have two additional components which represent the cost of logging the transaction (c_l) and the cost of backing out the transaction (c_b) in the case where an abort occurs. Hence,

$$c_u = s(1 - C) + \alpha s(1 - C)p_m + c_l + p_b c_b, \tag{2.3}$$

where p_b denotes the probability of an abort.

2.4.1 Evaluation of the probability of logging

We consider a set of K pages that have been modified by active transactions and compute the expected value of the size of the subset of pages that can be written back to the database without UNDO logging. N is the number of data pages in a parity group and S is the total number of data pages in the database. The K pages are assumed to be randomly chosen from the S pages in the database. Note that by using data striping (RAID) with a large striping unit or parity striping, any sequentiality in database accesses will act in favor of our scheme by distributing the pages accessed over distinct parity groups.

The parity groups in the database are numbered from 1 to S/N. Let X_i, $1 \leq i \leq S/N$, be the random variable whose value is 1 if one of the K pages is a member of parity group i, and 0, otherwise. Let X be the random variable denoting the number of parity groups that contain all K pages. X is also the number of pages that can be directly written back to the database since one page per parity group can be written back. We have

$$X = \sum_{i=1}^{S/N} X_i.$$

Since the K pages are assumed to be randomly chosen, each parity group has the same probability of being accessed by those K page references. Hence the X_i's are identically distributed. Therefore, the expected value of X is $\mathrm{E}[X] = \sum_{i=1}^{S/N} \mathrm{E}[X_i] = \frac{S}{N}\mathrm{E}[X_1]$. Since X_1 is a Bernoulli random variable, $\mathrm{E}[X_1] = \Pr(X_1 = 1)$ and $\mathrm{E}[X] = \frac{S}{N}(1 - \Pr(X_1 = 0))$, which can be written: $\mathrm{E}[X] = \frac{S}{N}\left(1 - \frac{\binom{S-N}{K}}{\binom{S}{K}}\right)$. Hence if K modified, "uncommitted"

pages are to be written to the database, the probability of having to log one of those pages is given by

$$p_l = 1 - \mathrm{E}[X]/K = 1 - \frac{S}{KN}\left(1 - \frac{\binom{S-N}{K}}{\binom{S}{K}}\right). \tag{2.4}$$

2.4.2 Page logging

2.4.2.1 Algorithm of the type ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*

With the *FORCE* discipline, the checkpoint is taken at the end of each transaction. The cost of checkpointing is therefore accounted for in the cost of logging. In the model, we set $c_c = 0$. Given our assumption that pages are not rereferenced by the calling transaction after they have been replaced in the buffer, the cost of writing and logging a page will be the same whether the page is stolen from the buffer before transaction commit or whether it stays in the buffer until EOT and is then logged and written to the database. Hence we will account for all of the costs involved in logging the pages and writing them back to the database as part of the cost of logging. \This allows us to set $p_m = 0$ in the expressions for c_r and c_u. The expression for c_l is

$$c_l = 3 \times sp_u + 4 \times (2sp_u) + 4 \times 4.$$

The first term is the cost of writing the pages back to the database. Each write to the disk array costs three I/O operations since, with the FORCE discipline, the old data are kept in the buffer until EOT for the purpose of UNDO logging. The second term is the

cost of writing to the UNDO and REDO log files. REDO information is needed only in the case where an operator error or a system software error damages more than one disk in the disk array. The log files are stored separately which makes reading the log to backout aborted transactions less costly. The last term in the expression of c_l is the cost of writing BOT and EOT records to each of the log files.

The probability of having to log a page with RDA recovery is dependent on the number K of pages written back to the database by incomplete transactions. We assume that when a transaction writes back a page to the database before committing, the other concurrent transactions are halfway through writing their own modified pages. Therefore, K is equal to half the total number of pages modified by concurrent update transactions. Hence the probability of logging is given by Equation (2.4) in which K is replaced with[7] $Psf_u p_u/2$. With RDA recovery, the formula for the cost of logging becomes

$$c_l' = (3 + 2p_l)sp_u + 4(sp_u + sp_u p_l + 4) + 4(p_l - p_l^{sp_u}).$$

The major difference with c_l is that UNDO logging has to be performed only when the parity group is dirty, i.e., with probability p_l. The term $2p_l$ is added to 3 to account for the fact that when writing to a dirty parity group both parity pages have to be updated.[8] The last term in the expression of c_l' denotes the cost of writing the log chain header to the log. The header is normally written along with the BOT record in the same page except when the first page written by the transaction to the database has to be logged and not all pages updated by the transaction have to be logged.

[7]Page logging implies the use of page locking; hence, the sets of pages modified by concurrent update transactions are disjoint.

[8]We assume that log file pages and data pages are not mixed in the same parity groups.

To evaluate c_b we assume that a transaction aborts in the middle of processing its pages and that the other concurrent update transactions have also logged half their modified pages. The UNDO log has to be read up to the BOT record of the aborting transaction.

$$c_b = (p_u s/2)(Pf_u) + Pf_u + 4(p_u s/2) + 4.$$

The first term is the number of before-images that have to be read from the log. The second term is the number of BOT/EOT records to be read. The third term is the number of page transfers to and from the database to undo the modifications performed by the aborting transaction and the last term accounts for the writing of a rollback record. With RDA recovery the above formula becomes

$$c_b' = (p_u p_l s/2)Pf_u + (p_l - p_l^{sp_u})Pf_u + Pf_u + (p_u s/2)(6p_l + 5(1 - p_l)) + 4.$$

In the first term the number of logged before-images to be read is now multiplied by p_l. The second term is the expected number of log chain headers to be read from the log. The other major difference is in the fourth term. It is due to the fact that, when recovering a page that has been logged, up to six I/O operations might be necessary since its parity group may still be dirty.[9] On the other hand, if the page has been written to the database without being logged, it is necessary to read both parity pages in its parity group and the "new" data page and then overwrite the database page with the old data and modify the state of the parity page from *working* to *invalid* by resetting the timestamp in its header. Hence five I/O operations will be necessary in the latter case.

[9]In this instance and in other instances in the evaluation, we use an upper bound for the costs involved in RDA recovery in order to keep things simple. This will lead to a conservative estimate of the benefit of our method.

After a system crash, only UNDO recovery has to be performed. Hence the formula for c_s contains the cost of reading the UNDO log file up to the BOT record of the oldest transaction alive at the time of the crash and then overwriting the modifications. The work of the oldest transaction alive overlapped with the work of some committed transactions; therefore, the log records for half of the work of about $2Pf_u$ transactions have to be read. Hence, the expressions for c_s and c'_s are

$$c_s = Pf_u(sp_u + 2) + 4(Pf_u p_u s/2)$$

$$c'_s = Pf_u(sp_u p_l + 2(p_l - p_l^{sp_u}) + 2) + Pf_u(p_u s/2)(4p_l + 5(1 - p_l)) + S/N.$$

The term S/N is an upper bound for the cost of reconstructing the bit map for the current parity page.

We evaluate the algorithms in two different environments depending on the frequency of update transactions. Figure 2.7 shows the throughput[10] as a function of the communality C in a system with high update frequency and in a system with high retrieval frequency. As expected, the improvement in throughput using RDA recovery is much more significant in the high update frequency environment. For the latter environment and for $C = 0.9$, the increase in throughput is about 47%. For the different cost measures, the relative change in cost is as follows: for the cost of transaction processing (c_t) -30%, for the cost of transaction backout (c_b) -34%, and for the cost of recovery from system crash (c_s) -16%. The reduction in the cost of transaction processing is mainly due to the fact that almost all updated pages can be written back to the database without

[10]For clarity, we plot $r_t/1000$.

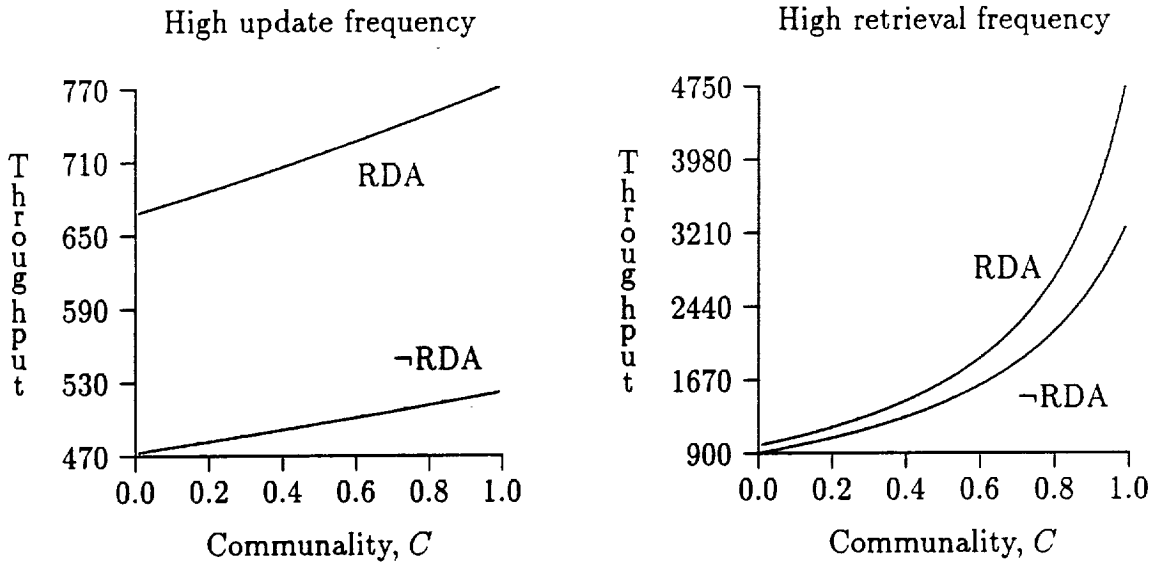


Figure 2.7 Results for ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*.

logging. The reduction in the cost of transaction backout is due to the fact that fewer pages have to be read from the log to get to the BOT record of the aborted transaction. The reduction in the cost of recovery from system crash is also due to the fact that less pages have to be read from the log to recover the aborted transactions. Most of the values for the different parameters of the model were taken from [16]. These values are $B = 5000$, $S = 5 \times 10^6$, $N = 10$, $P = 100$, $p_b = 0.01$ and $T = 5.10^7$. For the high update frequency environment, $s = 10$, $f_u = 0.8$ and $p_u = 0.9$, while for the high retrieval frequency environment, $s = 40$, $f_u = 0.1$ and $p_u = 0.3$.

2.4.2.2 Algorithm of the type ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*

In this case, at EOT, before- and after-images of modified pages are written to the log but the modified pages are not written back to the database. They remain in the

buffer until they are replaced. REDO recovery has to be performed after a system crash and ACC checkpointing is used to reduce the amount of REDO during crash recovery.

First, we have to evaluate p_m. To do so, we have to compute the number of transactions that successively reference a page during its life in the database buffer. If we look at the stream of references to a page by successive transactions, we can see that with probability C the page is referenced when it is in the buffer, and with probability $1 - C$, it is referenced when it is not in the buffer. Hence, the number of references to the page during its life in the buffer follows a geometric distribution with parameter C which implies that the average number of references to the page while it is in the buffer is $1/(1-C)$. Since the probability of a page being modified by a transaction that references it is $f_u p_u$, the probability of a replaced page being modified during its life in the buffer is[11]

$$p_m = 1 - (1 - f_u p_u)^{1/(1-C)}.$$

The cost of logging is simply the cost of writing before- and after-images of modified pages and the BOT/EOT records to the log:

$$c_l = 4(2sp_u + 2).$$

With RDA recovery, pages that have been stolen from the buffer before EOT do not have to have their before-images logged. Therefore, we have to evaluate the probability p_s for a page being stolen. The number of references that could cause a given page to be stolen is $(1 - C)s(P - 1)$, and the probability that any one of those references causes the

[11]The same equation for p_m was derived in [16] using a slightly different argument.

replacement of the page is $1/(B - Cs)$. Hence the formula for p_s is

$$p_s = 1 - \left(1 - \frac{1}{B - Cs}\right)^{(1-C)s(P-1)}.$$

In the formula for p_l, the value of K is $Psf_up_up_s/2$. The before-image of a modified page will not be logged with probability $p_s(1 - p_l)$. Hence the cost of logging with RDA recovery is

$$c_l' = 4(sp_u + sp_u(1 - p_s(1 - p_l)) + 2) + 4(p_l - p_l^{\lceil sp_up_s \rceil}).$$

For the cost of backing out a transaction, one difference with the *FORCE* scheme is that the log file contains both before- and after-images which will be read until the BOT record of the aborting transaction is found. Another difference is that, with probability C, the modified pages to be undone are still in the buffer. Hence,

$$c_b = 2 \times (p_us/2)(Pf_u) + Pf_u + 4p_u(s/2)(1 - C) + 4.$$

With RDA recovery, the cost of transaction backout becomes

$$\begin{aligned} c_b' &= 2 \times (p_us/2)(Pf_u) + Pf_u + Pf_u(p_l - p_l^{\lceil sp_up_s \rceil}) + p_u(s/2)((4 + 2p_l)(1 - C)(1 - p_s) \\ &\quad +6p_sp_l + 5p_s(1 - p_l)) + 4. \end{aligned}$$

The costs of performing a checkpoint for ¬RDA and for RDA are given by

$$c_c = 4(Bp_m + 2),$$

$$c_c' = (4 + 2p_l)(Bp_m + 2).$$

To evaluate the cost of recovery after a crash, we assume that a crash occurs in the middle of a checkpoint interval. All transactions executed since the last checkpoint have

to be redone. Let r_c denote the number of transactions executed during a checkpoint interval, r_c is given by $r_c = I/c_t$ and the expression for c_s is

$$c_s = (r_c/2)f_u(c_l/4 + 4sp_u) + Pf_u(c_l/4 + 4(s/2)p_u - 1).$$

The -1 term corresponds to the EOT record which is accounted for in $c_l/4$ but is not read. The cost of recovery from a crash with the RDA recovery technique is

$$c'_s = (r'_c/2)f_u(c'_l/4 + 4sp_u) + Pf_u(c'_l/4 + (s/2)p_u(4(1-p_s) + 4p_sp_l + 5p_s(1-p_l)) - 1) + S/N.$$

The value of the optimal checkpointing interval I is obtained by plugging the expression for c_s in Equation (2.1), which yields

$$I = (2c_tc_c(T - Pf_u(c_l + 4(s/2)p_u) - Pf_u)/(f_u(c_l + 4sp_u)))^{1/2}.$$

The formula for I in the case of RDA recovery is derived in a similar fashion. The value of α in the expressions of c_r and c_u is 4 for $\neg$RDA and $4 + 2p_l$ for RDA because with the $\neg FORCE$ discipline, when replacement takes place, the old version of the data is not available any more in the buffer.

Figure 2.8 shows the results for both environments. It can be seen that the improvement is not significant in this case. However, the interesting result is that while without RDA recovery, the $\neg FORCE$, ACC type algorithm outperforms the $FORCE$, TOC scheme; when RDA recovery is used, the situation is reversed and the latter algorithm outperforms the former by a significant margin.

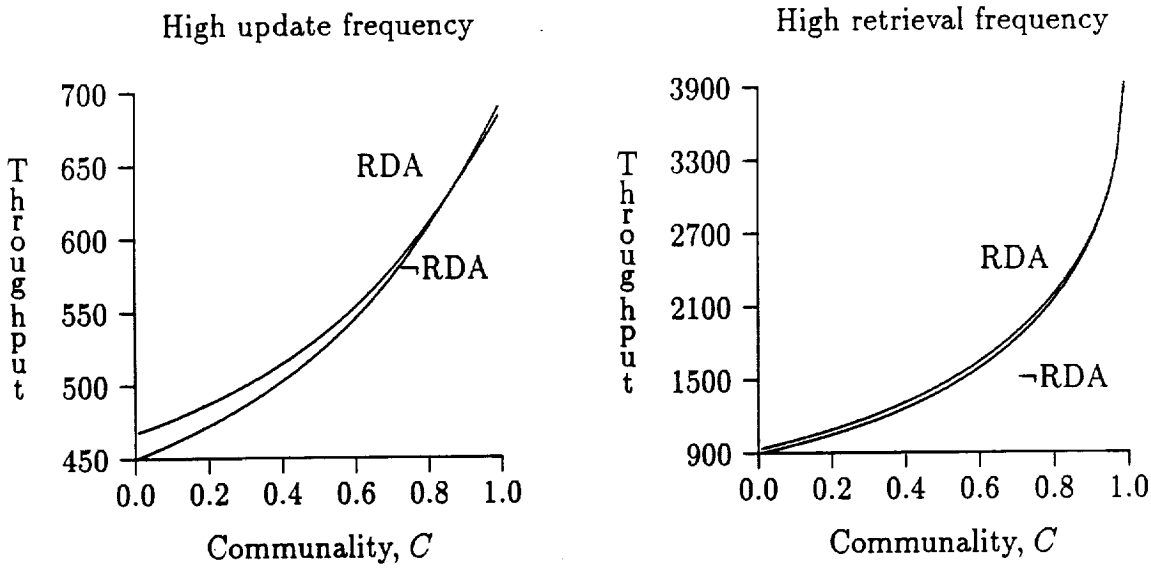


Figure 2.8 Results for ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*.

2.4.3 Record logging

In this section we look at recovery algorithms in which only modified records are logged. The unit of transfer between main memory and secondary storage is still a page; however, when logging is performed, logged records are encapsulated into pages and then written to the log file. Some additional parameters of the system have to be introduced for the analysis of record logging: d denotes the number of update statements per transaction; r denotes the average length (in bytes) of a long log entry such as a data record; e denotes the average length of a short log entry such as a table entry; l_{bc} denotes the length of the BOT and EOT records; l_p denotes the length of a physical page; and l_h denotes the length of a log chain header. The values for the first five parameters are taken from [16]. These values are $d = 3$ for high update frequency environments and $d = 8$ for low update frequency environments, $r = 100$, $e = 10$, $l_{bc} = 16$ and $l_p = 2020$. The value for l_h was set to 4. Assuming that each update statement causes one long log

entry and that $s > d$, the average length of a log entry can be derived [16]:

$$L = (dr + (s - d)e)/s.$$

2.4.3.1 Algorithm of the type ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*

With record logging, the locking granule can be less than a page. We assume that record locking is used in order to enhance concurrency. This implies that the total number of pages modified by a given set of P concurrent transactions is not the same as for the above algorithms for which page locking was assumed. We will denote this number by s_u. An expression for s_u is derived in the following.

Let $S^{(k)}$ denote the number of pages in the buffer updated by k update transactions. Since there are Pf_u update transactions executing concurrently in the system, we have $s_u = S^{(Pf_u)}$. If we number the Pf_u update transaction from 1 to Pf_u in the order of their entry in the system, then when the kth update transaction enters the system, it will find Csp_u of the sp_u pages it has to modify already in the buffer. We make the assumption that out of those pages, $Csp_u \times S^{(k-1)}/B$ belong to the $k-1$ update transaction already executing in the system.[12] Hence, we have the following recurrence equation:

$$S^{(k)} - S^{(k-1)} = sp_u(1 - CS^{(k-1)}/B).$$

Using $S^{(1)} = sp_u$, we obtain $s_u = S^{(Pf_u)} = \frac{B}{C}(1 - (1 - Csp_u/B)^{Pf_u})$.

The value of K in the expression of p_l is $s_u/2$. We assume that group commit is used so that log records from different transactions can be grouped in the same page

[12]Update transactions can share pages because record logging is used instead of page logging.

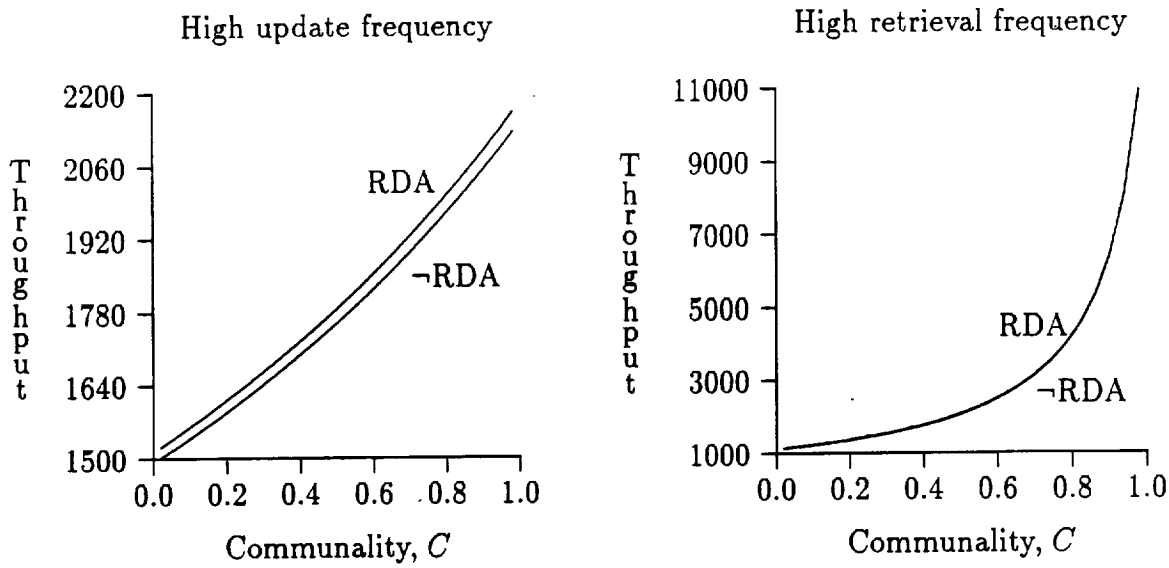


Figure 2.9 Results for ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*, in the case of record logging.

and written to the log. The derivations of the cost equations are similar to those in Section 2.4.2.1. We simply list the equations without detailed explanation.

$$
\begin{aligned}
c_l &= 3sp_u + 4 \times 2(2l_{bc} + sp_u(l_{bc} + L))/l_p \\
c_l' &= (3 + 2p_l)sp_u + 4(2l_{bc} + sp_u(l_{bc} + L))/l_p + 4(2l_{bc} + sp_u(l_{bc} + L)p_l + (l_{bc} + l_h)(p_l - \\
&\quad p_l^{sp_u}))/l_p \\
c_b &= Pf_u(l_{bc} + sp_u(l_{bc} + L)/2)/l_p + 4(p_u s/2) + 4 \\
c_b' &= Pf_u(l_{bc} + sp_u(l_{bc} + L)p_l/2 + (l_{bc} + l_h)(p_l - p_l^{sp_u}))/l_p + (p_u s/2)(6p_l + 5(1 - p_l)) + 4 \\
c_s &= Pf_u(2l_{bc} + sp_u(l_{bc} + L))/l_p + 4Pf_u(p_u s/2) \\
c_s' &= Pf_u(2l_{bc} + sp_u(l_{bc} + L)p_l + 2(l_{bc} + l_h)(p_l - p_l^{spu}))/l_p + (Pf_u p_u s/2)(4p_l + 5(1 - p_l))
\end{aligned}
$$

Figure 2.9 shows the throughput for the *FORCE*, *TOC* type of algorithms with and without RDA recovery as a function of the communality in the buffer for the case of record logging.

2.4.3.2 Algorithm of the type ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*

The cost equations for this case can be derived using the results of Sections 2.4.2.2 and 2.4.3.1. The value of K in the expression for p_l is $s_u p_s/2$.

$$
\begin{aligned}
c_l &= 4(2l_{bc} + sp_u(l_{bc} + 2L))/l_p \\
c'_l &= 4(2l_{bc} + sp_u(l_{bc} + L(2 - p_s(1 - p_l))) + (l_{bc} + l_h)(p_l - p_l^{\lceil sp_u p_s \rceil}))/l_p \\
c_b &= Pf_u(c_l/8) + 4p_u(s/2)(1 - C) + 4 \\
c'_b &= Pf_u(c'_l/8) + p_u(s/2)((4 + 2p_l)(1 - C)(1 - p_s) + 6p_s p_l + 5p_s(1 - p_l)) + 4 \\
c_s &= (r_c/2)f_u(c_l/4 + 4sp_u) + Pf_u(c_l/4 + 4p_u(s/2)) \\
c'_s &= (r_c/2)f_u(c'_l/4 + 4sp_u) + Pf_u(c'_l/4 + p_u(s/2)(5p_s(1 - p_l) + 4(1 - p_s(1 - p_l)))).
\end{aligned}
$$

The equations for c_c and c'_c are the same as in Section 2.4.2.2. The equations for c_r and c_u have to be modified to account for the extra cost involved in logging modified records in pages stolen from the buffer before EOT. The modified record of a stolen page has to be written to the log before the page can be replaced. Let p_i denote the proportion of replaced pages modified by uncommitted transactions. We have $p_i = s^*_u/(B - Cs)$, where s^*_u is the number of pages in the buffer modified by the concurrently executing transactions as seen by an incoming transaction. s^*_u is obtained by replacing P with $P - 1$ in the expression for s_u. This gives the following equations for c_r and c'_r; the equations for c_u and c'_u are obtained in a similar fashion:

$$
\begin{aligned}
c_r &= s(1 - C) + 4s(1 - C)(p_m + 2p_i) \\
c'_r &= s(1 - C) + 4s(1 - C)(p_m + 2p_i p_l).
\end{aligned}
$$

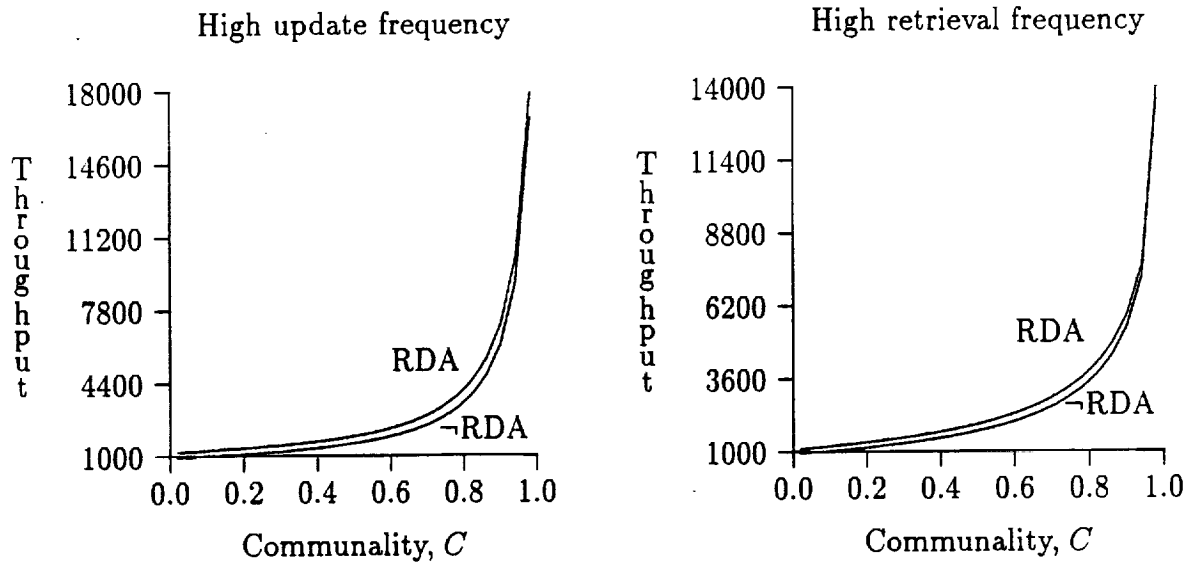


Figure 2.10 Results for ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*, in the case of record logging.

Figure 2.10 shows the throughput for the ¬*FORCE*, *ACC* type of algorithms with and without RDA recovery as a function of the communality in the buffer for both evaluation environments. Unlike the page logging case, ¬*FORCE*, *ACC* scheme performs much better than the *FORCE, TOC* scheme for the range of values of C encountered in typical applications [17]. Also, for the ¬*FORCE*, *ACC* algorithm, the increase in throughput achieved by using RDA recovery is higher than for the same algorithm with page logging. This is the case because, with record logging, the cost of logging the updates of a stolen page is high relatively to the cost of logging nonstolen pages and RDA recovery reduces that cost by eliminating the need for logging in most cases. For example, for the high update frequency environment and for $C = 0.9$, the increase in throughput is about 15%. The relative change in the various cost measures is as follows: for the cost of transaction processing (c_t) -14%, for the cost of transaction backout (c_b) +1%, and for the cost of

recovery from system crash (c_s) +7%. For the cost of checkpointing (c_c), the relative change is almost nil.

The decrease in the cost of transaction processing is mainly a result of not having to force the log block to disk when a page is stolen from the buffer. The increase in the cost of transaction backout is due to the fact that the savings resulting from reading fewer pages from the log to obtain to the BOT record become much smaller in the case of record logging, because log information is more dense while the extra overhead involved in reading the parity pages to recover unlogged pages remains the same as in the case of page logging. The overall effect is a net increase in the cost of transaction backout. The increase in the cost of recovery from system crash can also be attributed to the same reasons. The cost of checkpointing increases, because when writing a modified page to a dirty parity group, the old version must be logged. However, the probability of logging is very close to zero. Hence the increase in checkpointing cost is negligible.

The benefit of RDA recovery increases with the amount of work performed by each transaction. Figure 2.11 shows the percent increase in throughput achieved by RDA recovery as a function of the number of pages accessed by each transaction (s) for the high update frequency environment with $C = 0.9$.

2.5 Experimental Evaluation of *FORCE, TOC* Algorithms

We have conducted some experiments to corroborate the findings of our analytical model. We have used data from an operational OLTP system, namely a Tandem NonStop

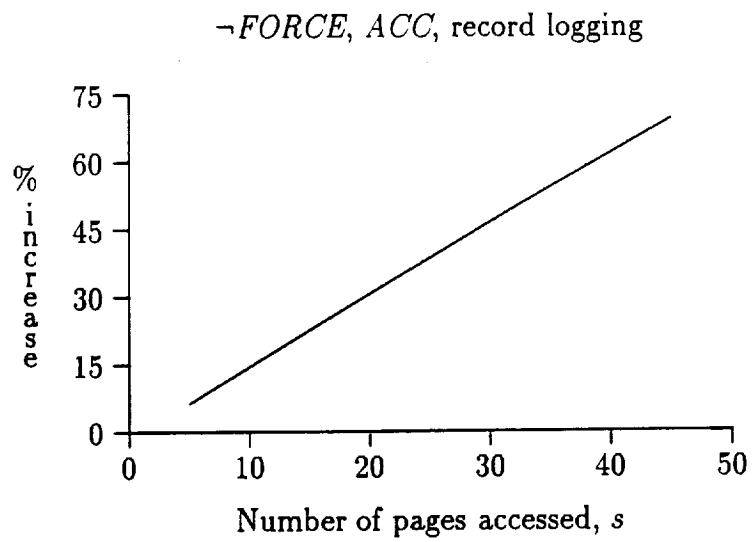


Figure 2.11 Benefit of RDA recovery as a function of the number of pages referenced by a transaction.

system. The data were extracted from log files generated by the Transaction Monitoring Facility (TMF)[18] during normal system operation. The log entries contained transaction status information, before and after images of modified data, names of accessed files and disks as well as timing information. Using the log entries, we constructed a trace of update accesses performed by each transaction before it commits or aborts.

Using these data, we simulated the behavior of the database buffer, the recovery algorithm and the I/O subsystem. As in the analytical model, we assumed that the system was I/O bound; hence, we ignored cpu processing times and accounted only for the cost of performing I/O. However, in the simulations, we did not simply count the number of I/O operations performed, but rather we simulated the execution of the I/O requests in the disk array. We have simulated a parity striping organization. Since the data did not contain any multiblock references, we expect the performance of a data striping organization to be similar to that of parity striping. The disk parameters used

Table 2.1 Disk parameters.

max. latency	16.6 ms
max. seek	47 ms
tracks per platter	1260
sectors per track	52
bytes per sector	512
number of platters	15

in the simulations are shown in Table 2.1. The database accessed in our trace resided on five disks. We assumed that the log file is stored on a separate disk array.

The data available to us did not include any read access information. Thus it was not possible to use it to evaluate $\neg$*FORCE, ACC* algorithms because, for these algorithms, the improvement afforded by RDA recovery is dependent on the frequency of replacement of modified pages and, hence, requires a detailed simulation of the buffer behavior. Without a trace of read accesses, we could not obtain reliable simulation results for those algorithms. However, for *FORCE, TOC* algorithms, page replacement does not affect the cost of recovery operations as much as in the $\neg$*FORCE, ACC* case. Therefore, we were able to use the available data to obtain simulation results for such algorithms. Since our analytical model did not show much promise for RDA recovery in the case of *FORCE, TOC* algorithms with record logging, we concentrated instead on page logging. We did not simulate recovery from system crash since none occurred in the interval during which the data were collected. Hence, the throughput r_t of the system was taken to be the reciprocal of the average cost per transaction c_t which was measured as the total cost (disk usage time) of executing the I/O requests in the disk array system divided by the

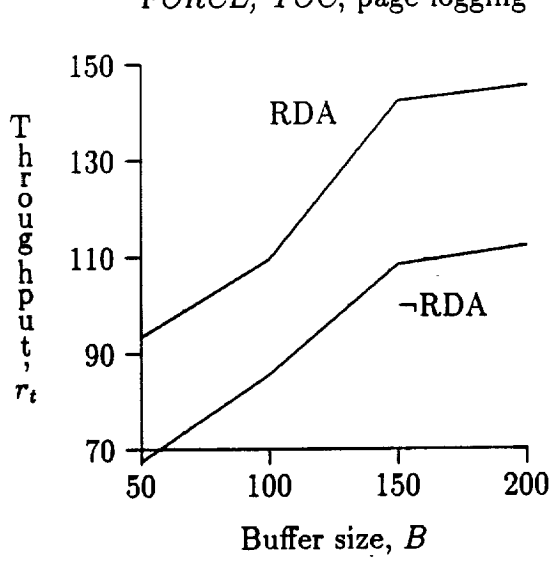


Figure 2.12 Empirical results for *FORCE, TOC* algorithms with page logging.

number of transactions. Figure 2.12 shows the measured throughput[13] of *FORCE, TOC* algorithms with and without RDA recovery in the case of page logging as a function of the number (B) of frames in the buffer.

An LRU buffer replacement policy was assumed. The hit ratio ranged from 77% for $B = 50$ to 97% for $B = 200$. The improvement in throughput decreases from 39% for $B = 50$ to 28% for $B = 100$ and then increases slowly as B increases to reach about 30% for $B = 200$. The reason for the decrease in the first part of the curve is that for small values of B, some pages in the buffer are replaced more than once , which increases the amount of logging in the $\neg RDA$ case. For $B > 100$, pages in the buffer are replaced once at most, hence the amount of logging remains about constant as B increases while the amount of I/O to the database continues to decrease. Hence, as B goes from 100 to

[13]For clarity, we multiplied the throughput values by a constant factor.

200, the savings due to RDA recovery become more significant relative to the overall I/O cost.

2.6 Conclusions

In this chapter, we have presented a scheme that uses redundant disk arrays to achieve rapid recovery from media failures in database systems and simultaneously provide support for recovery from transaction aborts and system crashes. The redundancy present in the array is exploited to allow a large fraction of pages modified by active transactions to be written to disk and updated in place without the need for undo logging thus reducing the number of recovery actions performed by the recovery component. The method uses a twin page scheme to store the parity information so that it can be efficiently used in transaction undo recovery. The extra storage used is about $(100/N)\%$ of the size of the database, N being the number of disks in the array.

We used a detailed analytical model to evaluate the benefit of our scheme in a system equipped with redundant disk arrays. We found that, in the case of page logging, a *FORCE, TOC* algorithm combined with RDA recovery significantly outperforms a *FORCE, TOC* algorithm without RDA recovery as well as $\neg$*FORCE, ACC* type of algorithms. In the case of record logging, we found that a $\neg$*FORCE, ACC* algorithm performs best and that the addition of RDA recovery to it improves significantly its performance especially for transactions with a large number of updated pages. We also performed

simulations using data from an operational OLTP system to validate some of the results of the analytical model.

CHAPTER 3

SITE PARTITIONING FOR DISTRIBUTED REDUNDANT DISK ARRAYS

3.1 Introduction

Redundant disk arrays can be used in a distributed setting to increase availability in the presence of temporary site failures, disk failures, or major disasters. Stonebraker and Schloss have proposed the Redundant Arrays of Distributed Disks (RADD) scheme [19] as an alternative to multicopy schemes which are much more costly in terms of storage requirements. Cabrera and Long [20] have proposed the use of redundant distributed disk striping in a high speed local area network to support such I/O intensive applications as scientific visualization, image processing, and recording and play-back of color video. The RADD concept can also be used in multicomputer I/O subsystems such as the one proposed by Reddy and Banerjee [21] for hypercubes. The IDA approach proposed by Rabin [22] provides another way to tolerate failures in distributed storage systems with limited extra storage cost. However, in that approach, when a file or table is dispersed over several sites and a portion of it is updated at a given site, the portions on the other sites have to be read in order to recompute the encoding before they are all written back.

In the case of RADD, when a block is updated, only one parity block has to be read and updated.

When RADDs are used, sites are grouped together to form a redundant array containing data and parity and capable of recovering from a single site failure. The size of each array is fixed and is determined by the tradeoff between the availability requirements of the system and the cost of the storage overhead. Hence, a large distributed data storage system may have to be divided among several arrays of fixed size. In this chapter we look at the problem of partitioning the distributed storage systems into fixed size arrays in such a way as to minimize the cost of remote accesses that have to be performed to update the parity information. This problem is somewhat related to the problem of file allocation and replica placement in a distributed system, which has been studied extensively in the literature [23, 24]. However the two problems are different in nature because, in the RADD case, there is one redundant item for N data items while in the file allocation problem each file is replicated several times. More importantly in the replica placement problem, there is no stringent constraint on the number of sites "sharing" a replica because when the replica becomes unavailable those sites can access the second nearest replica while in the RADD case there is a hard constraint on the number of sites in an array. Note that the assignment of sites to redundant arrays (parity groups) can occur after all decisions on placing the data have been made. Data placement decisions are governed by a different set of criteria and are more influenced by the read access patterns since reads are usually more frequent than updates. Decisions on site assignment to redundant arrays are based on the update rate at each site and the cost

of communication between sites and are independent of the read access rate. Changing the assignment of sites to redundant arrays does not change the placement of the data. The purpose of site assignment is to reduce the parity traffic and does not directly affect the data traffic.

In the following section, we describe the RADD organization. In Section 3.3, we present the model used to formulate the problem mathematically and we prove that the problem is NP-hard. In Section 3.4, heuristic algorithms for solving the problem are described and results from an experimental evaluation are presented. In Section 3.5, we develop heuristics with guaranteed bounds on the deviation from the optimal cost. In Section 3.6, we address the issue of hot spots and nonuniform site capacity and discuss the use of RADD for disaster recovery in OLTP systems. Finally, in Section 3.7, we discuss the issue of when and how often site reassignment should be initiated.

3.2 Distributed Redundant Disk Array Organization

The RADD organization is shown in Figure 3.1. The data at each site are partitioned into blocks. Data blocks from different sites are grouped into a block parity group. The bitwise parity of the data blocks in each parity group is computed and written at a different site. In Figure 3.1, D_{ij} denotes a data block, P_i denotes a parity block and S_i denotes a spare block, all at site i. The number under *block* in the first column of the figure denotes the physical block number on disk. Each row in the figure represents a parity group. The position of the parity block is rotated among the sites in order to avoid creating a bottleneck at the site where parity is stored. For every update to one of the

block	$Site_0$	$Site_1$	$Site_2$	$Site_3$	$Site_4$	$Site_5$
0	P_0	S_1	D_{20}	D_{30}	D_{40}	D_{50}
1	D_{00}	P_1	S_2	D_{31}	D_{41}	D_{51}
2	D_{01}	D_{10}	P_2	S_3	D_{42}	D_{52}
3	D_{02}	D_{11}	D_{21}	P_3	S_4	D_{53}
4	D_{03}	D_{12}	D_{22}	D_{32}	P_4	S_5
5	S_0	D_{13}	D_{23}	D_{33}	D_{43}	P_5

Figure 3.1 Organization of a distributed redundant disk array ($N = 6$).

data blocks in the parity group, the parity block has to be updated using the following formula:

$$P_{new} = (D_{old} \oplus D_{new}) \oplus P_{old}.$$

Spare blocks are provided in order to be able to reconstruct data blocks that become inaccessible due to a site failure. The failed data block is reconstructed by XORing all other data blocks and the parity block in its parity group. If K denotes the number of data blocks per parity group, then $N = K+2$ denotes the number of sites in a distributed disk array. The storage overhead for the parity *and* spare blocks required by RADDs is $(200/K)\%$ compared to a 100% overhead for the case of two copy schemes.

3.3 The Model

We model the distributed computing system by an undirected connected graph $G = (V, E)$ where V is the set of sites and each edge $e \in E$ represents a bidirectional communication link between two sites. For each $e \in E$, w_e denotes the cost of communication over link e. For $e = (u, v)$, w_e could be the actual distance between site u and site v. We assume that if n is the number of sites in V then $n = mN$ for some m. We will assume that the site capacity is uniform. In Section 3.6.2 we show how to deal with nonuniform

block	$Site_0$	$Site_1$	$Site_2$	$Site_3$	$Site_4$	$Site_5$
6	P_0	D_{14}	S_2	D_{34}	D_{44}	D_{54}
7	D_{04}	P_1	D_{24}	S_3	D_{45}	D_{55}
8	D_{05}	D_{15}	P_2	D_{35}	S_4	D_{56}
9	D_{06}	D_{16}	D_{25}	P_3	D_{46}	S_5
10	S_0	D_{17}	D_{26}	D_{36}	P_4	D_{57}
11	D_{07}	S_1	D_{27}	D_{37}	D_{47}	P_5

Figure 3.2 Alternative placement pattern for parity and spare blocks.

site capacity. In the pattern shown in Figure 3.1, the parity blocks of the $N-2$ data blocks of site i reside on sites $i+1$ mod N through $i+N-2$ mod N. Therefore there is no parity update traffic from site i to site $i-1$ mod N. To make the problem symmetrical and thus easier to tackle, we assume that for the next set of N blocks the pattern shown in Figure 3.2 is used. In all, there are $N-1$ such patterns obtained by changing the distance between the parity block and the spare block on a given row. These $N-1$ patterns should be alternated throughout the range of blocks so that update traffic from a given site is distributed over the remaining $N-1$ sites. This will also provide more load balancing for the parity update traffic in the array.

Let μ_v designate the rate of update accesses to data blocks at site v. Each update will cause communication between the site where the update took place and the site holding the parity for the given data block. At each site, the set of data blocks that have their corresponding parity blocks on the same site is called a data group. To simplify the model, we assume that the $N-1$ data groups share equally the update rate. This implies that the rate at which site v sends parity update information to each other site in its redundant array is $\lambda_v = \mu_v/(N-1)$. This assumption is supported by the fact that consecutive data blocks have their parity blocks on different sites which implies that

accesses to a heavily used file that is stored on consecutive disk blocks will be spread over different data groups. In Section 3.6, the above assumption will be removed. The problem of partitioning the sites into arrays of size N in such a way that parity update costs are minimized can be mathematically formulated as follows:

Problem 1 (SP) *Find a partition of V into m disjoint subsets V_1, V_2, ..., V_m of size N such that if $d(u,v)$ denotes the length of the shortest path between u and v then $\sum_{i=1}^{m} \sum_{u \in V_i} \lambda_u \sum_{v \in V_i - \{u\}} d(u,v)$ is minimum.*

Theorem 1 *Problem SP is NP-hard for any fixed $N \geq 3$.*

Proof: We prove that problem SP is NP-hard by showing that there is a polynomial time transformation from the problem of partitioning a graph into cliques of size N to problem SP. The Partition into Cliques of size N (PC) problem can be stated as follows:

Instance: A graph $G = (V, E)$, with $|V| = Nm$ for some positive integer m.

Problem: Is there a partition of V into m disjoint subsets V_1, V_2, ..., V_m such that the subgraph of G induced by V_i is a clique of size N (complete graph with N nodes)?

Problem PC is NP-complete for any fixed $N \geq 3$ (see Partition into Isomorphic Subgraphs [25]). To transform an instance of PC into an instance of SP, it is sufficient to set $\lambda_v = 1$ for all $v \in V$, and $w_e = 1$ for all $e \in E$. Then graph G can be partitioned into cliques of size N if and only if the cost of the optimal solution to the above instance of problem SP is $n(N-1)$. □

The cost function $\sum_{i=1}^{m} \sum_{u \in V_i} \lambda_u \sum_{v \in V_i - \{u\}} d(u,v)$ can be rewritten as

$$\sum_{i=1}^{m} \sum_{u,v \in V_i, u \neq v} (\lambda_u + \lambda_v) d(u,v) = \sum_{i=1}^{m} \sum_{u,v \in V_i, u \neq v} D(u,v),$$

where $D(u,v)$ is defined as $D(u,v) = (\lambda_u + \lambda_v)d(u,v)$. In this form the general problem is reduced to a uniform load problem with the distance D replacing d. However, D is not a true distance since it does not necessarily satisfy the triangular inequality.

3.4 Approximation Algorithms

3.4.1 Description of the heuristics

The first heuristic is based on a greedy strategy that consists of satisfying first the sites with the largest update rate. Let Λ be the list of update rates for all sites. When sites are grouped into clusters, their update rates are removed from Λ and replaced by a single update rate for the cluster. The cluster update rate is the average update rate of the sites in the cluster.

Algorithm 1:

Step 1. Select the largest value in Λ and let a be the corresponding site (or cluster). Find the site (or cluster) b such that merging a and b results in the smallest increase in the cost function. Merge the two sites (or clusters) if the resulting cluster has less than N sites and the total number of clusters does not exceed m. If the clusters cannot be merged, find the next best choice for b and repeat. Remove the update rates of the merged sites (or clusters) from Λ and replace them with the cluster update rate.

Step 2. Repeat Step 1 until m clusters having N sites each have been formed.

The computational cost of Algorithm 1 is $O(Nn^2)$. But it requires that the all-pair shortest path algorithm be performed first which requires $O(n^3)$ operations.

The second approach consists of two stages: in the first stage m sites are identified to be used as cluster seeds, and in the second stage, the remaining sites are allocated to the clusters to form m subsets of N sites each.

Algorithm 2:

Step 1. Select the two sites with the largest distance between each other and include them in the set S of cluster seeds.

Step 2. Select the site v with the largest average distance to the sites already in S and add it to S.

Step 3. Repeat Step 2 above until $|S| = m$. Each cluster initially contains one of the m seeds in S.

Step 4. For each of the m clusters, compute the average update rate of the sites in the cluster. In decreasing order of their average update rate, allocate to each cluster the site that is closest to it in terms of the distance metric D.

Step 5. Repeat Step 4 above until all sites have been allocated to the m clusters.

We use the distance metric D in Step 4 because it provides the actual increase in the cost function of a cluster when a node is added to it. The computational cost of the Algorithm 2 is $O(Nn^2)$. It also requires that the all-pair shortest path algorithm be performed first.

The third approach is based on the hierarchical clustering technique [26]. We use the distance matrix whose entries are $d(u, v)$ for all $u, v \in V$. Clusters are formed by merging together sites or smaller clusters that are close to each other. When two sites (or clusters) are grouped together, the distance matrix is modified by eliminating the

columns and rows corresponding to the merged sites (or clusters) and replacing them with a single column and a single row reflecting the average distance between the merged sites and other sites (or clusters). The procedure is as follows:

Algorithm 3:

Step 1. Find the smallest entry in the distance matrix and merge the two sites (or clusters) together if the resulting cluster has N sites or less and if the total number of clusters does not exceed m. If any of the latter conditions is not satisfied, select the next smallest entry and repeat. Once two sites (or clusters) have been merged, update the distance matrix and the number of clusters accordingly.

Step 2. Repeat Step 1 above until m clusters having N sites each have been formed.

The complexity of Algorithm 3 is $O(n^3)$.

After an initial partition has been found, the following procedure may be used to improve it.

Procedure Improve:

Step 1. Select the site u with the highest update rate. For each site v outside site u's partition, compute the change in cost $\Delta C(u,v)$ if u and v were swapped. Let v^* be the site corresponding to the minimum change in cost: $\Delta C(u,v^*) = \min_{v \notin V_u} \Delta C(u,v)$. If $\Delta C(u,v^*) < 0$, then swap u and v^*.

Step 2. Repeat Step 1 for all sites in V in decreasing order of their update rate.

The complexity of the above procedure is $O(n^3)$. The procedure may be repeated several times to improve the total cost. The procedure may also be repeated until a local minimum of the cost function is reached. However, it is not guaranteed that such a local

minimum will be reached in finite time. The procedure can also be employed as the basic move in meta-heuristics, such as simulated annealing [27] or tabu search [28] that avoid getting trapped in a local minimum.

3.4.2 Experimental evaluation

We have conducted experiments to evaluate the approximate solutions obtained using the heuristics and to compare the three proposed approaches for site assignment. In the experiments, we used randomly generated graphs. The distance on each edge in the graph was drawn from a uniform distribution over the interval [1,K_w]. The update rates at each site were drawn from a uniform distribution over the interval [1,K_λ].

In our experiments we found that Algorithm 2 performs better when the distance D is also used in the first stage of the algorithm. This can be explained by the fact that using D in the generation of the cluster seeds ensures that edges with large $D(u, v)$ will not be used within a cluster, i.e., sites that have large loads and that are far apart are not placed in the same cluster. The results shown here for Algorithm 2 were obtained using D instead of d.

In the first experiment, we compare the approximate solution provided by the heuristics to the optimal solution. The optimal solution was obtained using exhaustive search. N was taken to be equal to 5 and n equal to 15. Table 3.1 shows the results for three situations: one where the edge weights vary more widely than the site loads, one where both are picked from the same interval and one where the site loads vary more widely than the edge weights. Each entry represents the average over 100 randomly generated

Table 3.1 Comparison between approximate solutions and the optimal solution.

K_w, K_λ	Random	Algorithm 1	Algorithm 2	Algorithm 3	Exhaustive
1000, 10	68400	52439	53462	52649	47475
100, 100	66071	50012	51347	51237	45661
10, 1000	96757	76388	77362	77062	70004

graphs. The costs of the approximate solutions are within 10% of the cost of the optimal solution. In the first column of the table, we have listed the cost of a random solution.

Since, in the first experiment, an exhaustive search was used to find the optimal solution, the number of nodes n could not be very large. In a second experiment, we compared the performance of the three heuristics for larger values of n. Figure 3.3 shows the results for the second experiment. For clarity of the figure, we plotted the cost of the approximate solution divided by 1000. In the case $N = 10$, Algorithm 3 outperforms Algorithms 1 and 2 for all values of n except when $n = 20$, in which case, Algorithm 2 performs better. For the first and second environments Algorithm 1 outperforms Algorithm 2 for large values of n but for the last environment Algorithm 2 outperforms Algorithm 1. For $N = 5$, Algorithm 2 does not do very well except in the last environment in which the range of site loads is much larger than the range of edge weights. Algorithm 3 performs best in the first two environments. The main point that can be deduced from this experiment is that, in spite of the fact that Algorithm 3 does not use any information about site loads, it outperforms the other two algorithms when n and N are relatively large, and, in the other cases, its performance is always very close to that of the best algorithm. This means that, in a large system, it is more important to minimize

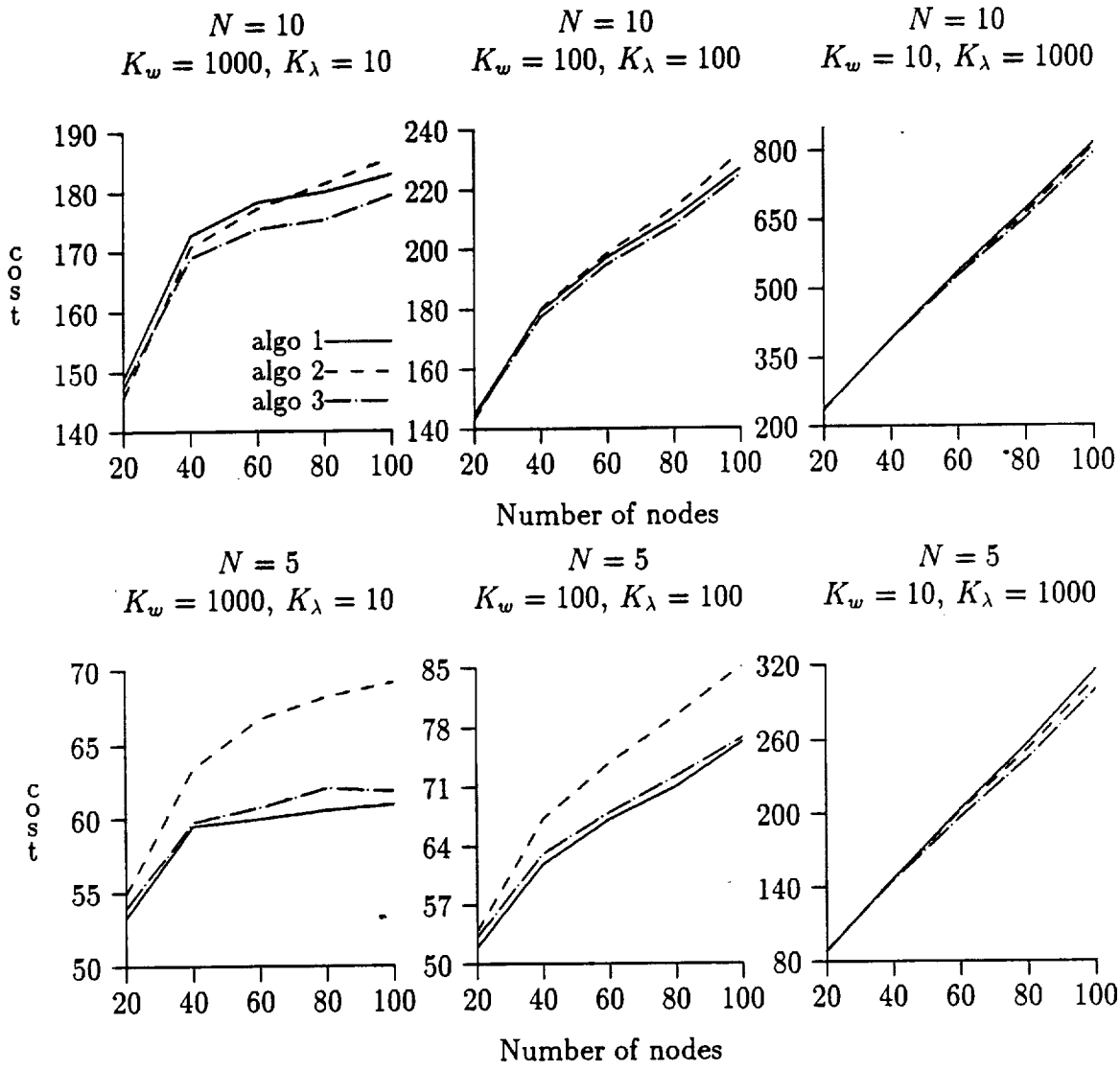


Figure 3.3 Comparison between the three heuristics.

the sum of the edge weights within each cluster than to use the greedy approach that attempts to assign to the sites with large loads their nearest neighbors.

3.5 Heuristics with Performance Guarantees

The heuristics described in Section 3.4 provide in general a good approximate solution. However, there is no guarantee that the approximate solution will not diverge significantly from the optimal one in certain cases. In this section, we seek to find a heuristic for which it is possible to establish a bound on the error between the approximate solution and the optimal one. We develop such a heuristic first for the case of a system with balanced load, $\lambda_v = \lambda$, for all $v \in V$, and uniform edge weights, then we look at the more general case of a balanced load system with arbitrary edge weights. Since a problem with arbitrary site loads can always be transformed into a problem with uniform site load as shown in Section 3.3, then the heuristic for the balanced load case with arbitrary edge weights will also provide performance guarantees for the arbitrary load case.

3.5.1 Balanced load and uniform edge weights

The heuristic requires the use of a spanning tree with many leaves. The problem of finding a spanning tree with a maximum number of leaves is NP-hard [25]; however, polynomial time algorithms exist for generating spanning trees with many leaves. Typically these methods guarantee that a certain fraction of the nodes will be leaves. The fraction

of leaves is a function of the minimum degree k of the graph. Kleitman and West proved the following result [29]:

Theorem 2 (Kleitman-West) *If k is sufficiently large, then there is an algorithm that constructs a spanning tree with at least $(1 - b \ln k/k)n$ leaves in any graph with minimum degree k, where b is any constant exceeding 2.5.*

It was also conjectured that a spanning tree can be constructed with a larger fraction of leaves. More specifically, Linial conjectured that the number of leaves could be at least $\frac{k-2}{k+1}n + c_k$. This stronger result was proved for $k = 3$ with $c_3 = 2$ and for $k = 4$ with $c_4 = 8/5$ [29].

Algorithm 4

Step 1. Find a spanning tree with many leaves.

Step 2. Partition the spanning tree into m clusters of N nodes each using procedure **Partition_Tree** described below.

The partition found for the tree will be used for the original graph. In the description of the procedure **Partition_Tree**, we assume that the tree is levelized starting from the root.

Procedure Partition_Tree:

The procedure partitions the tree from the bottom up. As the clusters are built, whenever the size of a cluster reaches N nodes, that cluster is removed from the tree. Starting from the deepest level in the tree, sibling leaves are placed together in a cluster. If all siblings have been used, then their parent is included in the cluster. At an internal

node v, all subtrees rooted at its siblings must be processed so that only less than N nodes are left in each subtree. Those subtrees are numbered from 1 to $d(v)-1$, $d(v)$ being the degree of v. Then the clusters are formed by adding to the nodes of subtree i enough nodes from subtree $i+1$ to make an N node cluster. If there are not enough nodes in subtree $i+1$ to form a complete cluster, the nodes of the two subtrees are placed together and the next subtree is used to complete the cluster. If all of the subtrees have been used, and an incomplete cluster remains, then the parent node is added to the remaining cluster and the procedure continues at the next level. When adding a portion of the nodes of a given subtree to the preceding subtree(s) to complete a cluster, the nodes at the deepest level in that subtree are used first so that removal of the newly completed cluster will not disconnect the tree.

Theorem 3 *The cost (HEU) of the approximate solution found using a spanning tree with many leaves and the cost (OPT) of the optimal solution satisfy the following relationship:*

$$\frac{\text{HEU}}{\text{OPT}} \leq 2\alpha + (1-\alpha)\frac{N^2}{N-1},$$

where α is the fraction of leaves in the spanning tree.

Proof We need to establish an upper bound on the cost of the approximate solution and a lower bound on that of the optimal one. The cost in the graph of the approximate solution is at most the cost of that solution in the tree. We evaluate the cost in the tree by adding the contributions of each edge in the spanning tree to the overall cost. If an edge connects a leaf node to the tree, it will be referred to as a leaf edge; otherwise, it

will be called an internal edge. A leaf edge will be used in only one cluster and only for communication between the leaf node and the other $(N-1)$ nodes in the cluster. Therefore, the contribution of a leaf edge to the overall cost is $2(N-1)$. An internal edge will be used in at most two clusters, and in each cluster, it will be used by i nodes to communicate with the other $N-i$ nodes in the cluster. If α designates the fraction of leaf nodes in the tree, we have

$$\begin{aligned} \text{HEU} &\leq \alpha n \times 2(N-1) + (n-1-\alpha n) \times 2 \times \max_{1 \leq i \leq N-1} 2i(N-i) \\ &\leq n(N-1)(2\alpha + (1-\alpha)N^2/(N-1)) \end{aligned}$$

For the cost of the optimal solution, an obvious lower bound is the cost in a complete graph which is $n(N-1)$. Hence $\text{HEU/OPT} \leq 2\alpha + (1-\alpha)N^2/(N-1)$. □

As stated in Theorem 2, for large k, α converges to 1 and the above bound approaches 2. Note that it is reasonable to assume that the minimum degree will be large in practice because the underlying network has to have sufficient connectivity to enable communication under node failures and hence requires a reasonably large minimum degree.

The complexity of the algorithms for generating trees with many leaves [29] is $O(|E|)$. The complexity of the **Partition_Tree** procedure is $O(n)$.

3.5.2 Balanced load and arbitrary edge weights

For arbitrary edge weights, the problem of finding a heuristic with guaranteed performance bounds is much harder. In the following, we describe a heuristic for which a worst-case performance bound can be established. The bound is more significant for

systems where link communication costs (edge weights) do not vary widely. The heuristic consists of finding a minimum spanning tree, partitioning the tree into clusters using procedure **Partition_Tree** and that partition as an approximate solution. The following result will be used to establish a lower bound on the cost of the optimal solution.

Lemma 1 *In a complete graph, the average weight of the edges in a minimum spanning tree is at most the average weight of all edges.*

Proof We use induction on the number of nodes n. The lemma is obviously true for $n = 2$ or $n = 3$. Suppose it is true for graphs with $n - 1$ nodes and consider an n-node graph. Select node v such that the average weight of edges incident on v is at least the average weight of all edges in the graph. Remove v from the graph and find a minimum spanning tree in the remaining $(n-1)$-node graph. Then add to this spanning tree the lightest edge e^* connecting v to the other nodes to form an n-node spanning tree. Let MST_{n-1} and MST_n be the *total* weights of the $(n-1)$-node and the n-node spanning trees, respectively. Let $\mathcal{E}(v)$ be the set of edges incident on v. Using the induction hypothesis, we have

$$\frac{\mathrm{MST}_{n-1}}{n-2} \leq \frac{\sum_{e \in E - \mathcal{E}(v)} w_e}{(n-1)(n-2)/2}.$$

Therefore,

$$\begin{aligned} \mathrm{MST}_n &\leq \mathrm{MST}_{n-1} + w_{e^*} \leq \frac{\sum_{e \in E - \mathcal{E}(v)} w_e}{(n-1)/2} + \frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} \\ &\leq \frac{\sum_{e \in E - \mathcal{E}(v)} w_e}{(n-1)/2} + \frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} + \underbrace{\frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} - \frac{\sum_{e \in E} w_e}{n(n-1)/2}}_{\geq 0} \end{aligned}$$

$$= \frac{\sum_{e \in E} w_e}{n/2}.$$

Hence the average weight of the edges in the minimum spanning tree is $\mathrm{MST}_n/(n-1) \leq \sum_{e \in E} w_e/(n(n-1)/2)$. □

To obtain a lower bound on the cost of the optimal solution, we consider the optimal partition and build a spanning tree by first finding a minimum spanning tree in each cluster and then replacing each cluster by a single node and connecting each pair of these nodes by the lightest edge linking the initial clusters. An intercluster minimum spanning tree is then found. The intracluster spanning trees along with the intercluster spanning trees form a spanning tree for the entire graph.

Lemma 2 *The list of edge weights of the intercluster minimum spanning tree (ICMST) is included in the list of edge weights of the global minimum spanning tree (GMST).*

Proof Let e be an edge in the ICMST that does not appear in the GMST. Let u and v be its endpoints in the original graph and w be its weight. The path in the GMST from u to v induces a path in the intercluster graph from the cluster of u to that of v. If the path is a single edge, then this edge must have weight w and could replace the edge e in the ICMST. If the induced path has more than one edge, then, since the ICMST cannot contain a cycle, some of the edges on the induced path must not appear in the ICMST and at least one of these induced edges that do not appear in the ICMST forms a cycle containing e when added to the ICMST. Let e' be such an edge, which must have weight at most w; otherwise, it could be replaced in the GMST by (u, v) to

obtain a spanning tree with a smaller cost. In addition, e' cannot have weight less than w because it would then be possible to replace e by e' in the ICMST and obtain a smaller intercluster spanning tree. Hence, the weight of e' is w and we could remove e and replace it with e' in the ICMST. This process can be repeated until all edges in the ICMST also appear in the GMST. Hence, the lemma is proved. □

Theorem 4 *The cost (HEU) of the approximate solution found using a minimum spanning tree and the cost (OPT) of the optimal solution satisfy the following relationship:*

$$\frac{\text{HEU}}{\text{OPT}} \leq N \frac{\text{MST}}{\text{MST} - (m-1)\overline{w}},$$

where MST is the total weight of the edges in the minimum spanning tree and $\overline{w}$ is the average weight of the $m-1$ heaviest edges in the minimum spanning tree.

Proof In evaluating an upper bound on the cost of the approximate solution, we follow the same procedure as in the proof of Theorem 3 but we will not distinguish between leaf edges and internal edges. Each edge e in the tree will be used by at most two clusters and the contribution of e to the overall cost is bounded by $2 \times w_e \times \max_{1 \leq i \leq N-1} 2i(N-i)$. Hence we have $\text{HEU} \leq N^2\text{MST}$.

Let MST_i be the weight of the minimum spanning tree of cluster i for $1 \leq i \leq m$ and MST_c be the weight of the intercluster tree. We have $\sum_{i=1}^{m} \text{MST}_i + \text{MST}_c \geq \text{MST}$. Using Lemma 2, we have $\sum_{i=1}^{m} \text{MST}_i + (m-1)\overline{w} \geq \text{MST}$. Let OPT_i be the contribution to the optimal cost by cluster i. Using Lemma 1 we have $\text{OPT}_i/N \geq \text{MST}_i$ therefore $\text{OPT} \geq N(\text{MST} - (m-1)\overline{w})$. □

Let r be the ratio of the largest edge weight to the smallest edge weight. A looser but simpler bound than the one established in Theorem 4 can be derived using the parameter r:

$$\text{HEU/OPT} \leq N\left(1 + \frac{m-1}{n-m}r\right) \leq N(1 + r/(N-1)).$$

3.6 Generalization of the Model

3.6.1 Nonuniform load within site

In our model, we assumed that each site sends parity updates to each other site in its partition at the same rate. This implies a uniform update rate to each of the $N-1$ data groups of a given site that have parity information on each of the $N-1$ other sites. If the update rate information for each data group at each site is available, then the model can be refined to account for the difference in the rate of parity update requests issued by a given site and destined to the other sites in the array. The refined model should yield better results in the presence of hot spots. The update rate λ_u of site u is replaced by $N-1$ update rates $\lambda_{u,1}, \ldots, \lambda_{u,N-1}$ corresponding to each of its data groups. In this case, an obvious optimization would be to have the parity of the i^{th} most frequently accessed data group of a given site placed on the i^{th} nearest site in its partition. Note that this can be implemented without having to reshuffle the data on disk by saving the permutation describing the remapping of the $N-1$ data groups for each site and using it to send parity update requests to the proper site. Given the above optimization, the algorithms of Section 3.4 with some minor modifications can still be used to partition the

sites. The site update rate used in Algorithms 1 and 2 is set to the sum of all $N-1$ data group update rates at that site. We have evaluated the three algorithms of Section 3.4 in the case of the refined model along with a new greedy strategy that looks at data groups instead of sites and tries to place the parity of the data groups with the largest update rates on the closest sites.

Algorithm Greedy

Let Λ be the list of update rates for all data groups at all sites.

Let p_v be the number of site v's partition. Initially $p_v = -1$ for all $v \in V$.

Let n_i be the number of sites in partition i. Initially, $n_i = 0$. Assume $n_{-1} = 1$ throughout.

Let k be the current number of partitions. Initially $k = 0$.

Let $\mathcal{N}(v) = V - v$, for all $v \in V$.

Let $l = 0$.

Step 1. Select the largest value λ in Λ and let u be the corresponding site. If $n_{p_u} = N$ go to Step 4.

Step 2. Find the site v in $\mathcal{N}(u)$ that is nearest to u and satisfies: p_u or $p_v \neq -1$ and $n_{p_u} + n_{p_v} \leq N$ or if $p_u = p_v = -1$ and $k < m$. If none exist go to Step 4.

Step 3. Remove v from $\mathcal{N}(u)$.

If $p_u = p_v = -1$ set $p_u = p_v = l$, $n_l = 2$, $l = l + 1$, and $k = k + 1$.

If $p_u = -1$ and $p_v \neq -1$ set $p_u = p_v$ and $n_{p_v} = n_{p_v} + 1$.

If $p_u \neq -1$ and $p_v = -1$ set $p_v = p_u$ and $n_{p_u} = n_{p_u} + 1$.

If $p_u \neq -1$ and $p_v \neq -1$, set the partition number for every site in v's current partition to p_u, set $n_{p_u} = n_{p_u} + n_{p_v}$, $n_{p_v} = 0$, and $k = k - 1$.

Step 4. Remove λ from Λ.

Step 5. If $\sum_i n_i < n$, go to Step 1, otherwise, stop.

Algorithm Greedy is similar to Algorithm 1 in that it tries to satisfy first the nodes with the highest data group update rates. The complexity of the algorithm is $O(Nn^2)$ but as in the case of Algorithm 1, it requires the all-pair shortest path algorithm.

Figure 3.4 shows the results of the comparison between the four algorithms. The individual data group update rates are chosen randomly from the interval $[1, K_\lambda]$ while the edge weights are chosen from $[1, K_w]$. We found that Algorithms 2 and 3 perform best for $N = 10$ with Algorithm 2 being the winner for lower values of n while Algorithm 3 is better for the high values of n. For $N = 5$, Algorithm 3 performs best in almost all situations. We also found that the parity assignment within a cluster is as important as the problem of partitioning the sites into clusters. The policy that consists of placing the parity of the ith most accessed data group on the ith closest site within the cluster reduces the cost by 15 to 20%.

3.6.2 Nonuniform site capacity

The case of nonuniform site capacity can be handled in the same fashion as proposed by Stonebraker and Schloss [19]. We assume that the total number of disks is Np for some p and that the number of disks at any given site is at most p.[1] The system could then be partitioned using the following procedure.

[1]This replaces the assumption that $|V| = mN$.

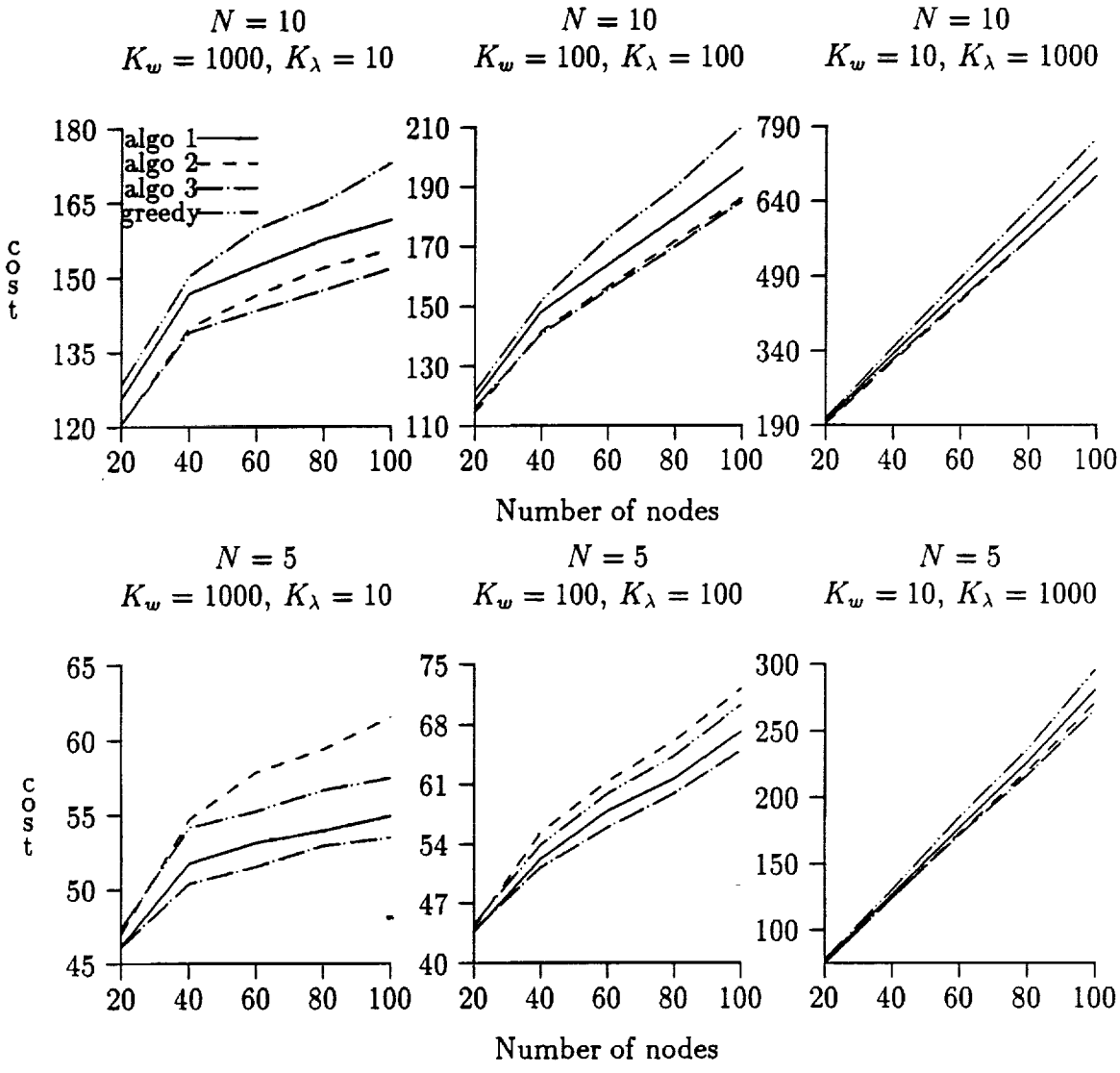


Figure 3.4 Evaluation of the heuristics for the refined model.

Step 1. Select the $N\lfloor |V|/N \rfloor$ sites with the largest number of disks and apply one of the partitioning algorithms described in the previous sections to assign one disk from each of the selected sites to an array.

Step 2. Remove the assigned disks and remove sites with no disks left.

Step 3. Repeat the above steps until all disks have been assigned.

Nonuniform disk capacity can be dealt with by using logical disks of size B blocks such that the site capacities are multiples of B [19].

3.6.3 Disaster recovery in OLTP systems

Disaster recovery is an important issue in On-Line Transaction Processing (OLTP) systems [30–32]. However, in such systems, updating the remote parity after each disk update may be too expensive especially since there are usually stringent requirements on transaction response time in those systems.

Typically, disaster recovery in OLTP systems is implemented by duplicating the data of a given site at a remote backup site and shipping Redo log information to the backup site where the updates are applied to the backup database. There are two approaches used in shipping the log [33]. In the first approach, the log records are shipped asynchronously to the backup site. Therefore, the transaction response time is not affected by the communication with the backup. However, some transactions may be lost in the case of a disaster. This configuration is called *1-safe*. In the second approach, log records are sent to the backup at commit time and the transaction waits for an acknowledgment

before it is allowed to commit. No transactions are lost in this case. This configuration is called *2-safe*.

Similar configurations can be implemented using RADD. In a *1-safe* implementation, parity updates (XORs of old and new data) can be accumulated at the originating site and shipped to the remote parity locations periodically. In a *2-safe* implementation, the parity updates originated by a transaction are grouped according to their destination site and shipped to that site while the transaction waits for an acknowledgment. If the updates performed by the transaction involve only one of the $N-1$ data groups, then only one remote message has to be sent by the committing transaction and the delay will be the same as in the traditional remote backup scheme. The advantage of RADD over the traditional schemes is that it uses much less storage space than full duplication.

Our model can still be used to solve the site assignment problem in both of the above implementations. However, instead of using the update rate at each site, the frequency of the periodic updates should be used in the *1-safe* case and the *update* transaction rate should be used in the *2-safe* case.

Another optimization that might be useful in OLTP environments consists of using the scheme proposed by Bhide and Dias in [34] to reduce the number of random I/O's performed in updating the parity at the remote site. The scheme consists of storing the parity updates in nonvolatile memory or sequentially on a dedicated disk and then periodically propagating them to their permanent locations. The scheme was originally proposed for use with a RAID level 4 organization [6] to reduce the load on the parity

disk. When the parity updates are stored sequentially on a dedicated disk, disk sorting is used to apply the parity updates to their permanent location.

3.7 Applying the Algorithms

Another important question is when and how often to apply the algorithm in order to obtain a lower cost site assignment. Clearly the algorithms can be used when the RADD scheme is first implemented as long as information on the site loads is available. As these loads change, the performance of the system degrades and the site assignment may have to be modified. Changing the site assignment is a costly operation. It involves reading large amounts of data to recompute the new parity and then updating the parity. This operation should be performed when the following two conditions are met: 1/ the difference between the cost of the current assignment and the cost of the best solution found by the algorithms should be large enough, and 2/ the parameters of the system (site loads) should be relatively stable so that the benefits of the new site assignment last long enough to offset the cost of performing the reassignment.

The cost of reassignment can be reduced if some clusters are kept unchanged. Hence one might be better off choosing a solution that is not the best possible but that preserves most of the current clustering. Procedure Improve described in Section 3.4 can be used to perform a limited number of swaps that decrease the cost of updating the parity without a full scale reassignment.

3.8 Summary

We looked at the problem of partitioning the sites of a distributed storage system into redundant disk arrays while minimizing the communication costs for updating the parity information. The problem was shown to be NP-hard in its general form. Several heuristic methods were investigated to obtain approximate solutions to the site partitioning problem. It was found that the heuristic that minimizes the sum of distances between sites within each cluster performs consistently well in all environments especially in large systems with a relatively large array size. In such systems, the above approach outperforms greedy methods that attempt to satisfy first the sites with the largest loads by placing their nearest neighbors in their partition. The solutions produced by this heuristic are also more robust because they provide good performance under different site loads. Guaranteed upper bounds were established on the deviation from the optimal cost for some of the heuristics. It was also found that modifying the parity assignment within each cluster to place the parity of the heavily accessed data groups on the nearest sites within the cluster can significantly decrease the parity update cost. Finally, we discussed implementations of the RADD scheme for disaster recovery in OLTP systems and described various optimizations that can be helpful in those environments.

CHAPTER 4

PERFORMANCE OF REDUNDANT DISK ARRAY ORGANIZATIONS IN TRANSACTION PROCESSING ENVIRONMENTS

4.1 Introduction

Disk arrays provide high data transfer rates by striping data over multiple disks. They also balance the load among disks in the same array. Redundant arrays use parity information to allow recovery from media failures in systems requiring high availability. In transaction processing environments, the high transfer rates of disk arrays are not fully exploited because I/O requests are typically small. However, redundant arrays are especially useful in such environments because they achieve media recovery at a significantly lower storage cost than mirrored disks.

In this chapter, the performance of RAID5 and parity striping is analyzed and compared to those of mirrored disk systems and systems using no striping and no redundancy. Trace data from large scale commercial transaction processing systems are used to evaluate the performance of the above organizations. Methods for reducing the write penalty in arrays using parity are investigated and their effect on performance is analyzed.

One such method uses a nonvolatile cache in the controller. Nonvolatile caches can provide significant improvements in the performance of disk arrays in transaction pro-

cessing environments. We also examine the benefits of using parity caching as a way to reduce the cost of individual writes, and we compare a RAID4 organization that caches both parity and data in the same nonvolatile cache to a RAID5 organization that caches only data.

Chen et al. [35] compared the performance of RAID0,[1] RAID1,[2] and RAID5[3] systems. They used a synthetic trace made of a distribution of small requests representing transaction processing workloads combined with a distribution of large requests representing scientific workloads and actual disk measurements on an Amdahl 5890. Gray et al. [8] proposed the parity striping organization and used analytical models to derive the minimum (zero load) response time and the throughput at 50% utilization for fixed size requests. Their results suggest that parity striping is more appropriate than RAID5 for database and transaction processing systems. Their model does not take into account the effect of skew in the distribution of accesses to disks, which turns out to be an important element in favor of RAID5. Chen and Towsley [36] developed queuing models for comparing the performance of RAID5 and parity striping. Menon and Mattson [37] analyzed the performance of RAID5 systems in the transaction processing environment using analytical models. They compared the performance of arrays made of different size building blocks and studied the effect of caching. Reddy [38] analyzed the effect of various parameters and policies used in the design of a nonvolatile I/O cache for systems where the cost of writes is higher than the cost of reads. He does not assume any particular

[1] Data striping without redundancy.
[2] Data striping with mirroring.
[3] Data striping with rotated parity.

array organization and the effect of the parity update traffic on read miss access time is not modeled. Ganger et al. [39] studied the benefits of data striping in reducing disk load imbalance and showed that it performs better than conventional data placement schemes.

In our evaluation of cached systems, we concentrate on comparing the behavior of the various array organizations when an I/O cache is used. Both read miss accesses and write (destage) accesses to data and parity are simulated. Bhide and Dias [34] have analyzed the RAID4 system with parity buffering in OLTP environments using analytical models. Their model suggests that a relatively large amount of nonvolatile memory is necessary. We show that this is not the case in the I/O traces examined in this study. They also propose an alternate scheme which writes parity updates sequentially on a log disk and then periodically writes them back to the parity disk. The log-based scheme uses up to four extra disks per array. The RAID 7 disk array system built and marketed by Storage Computer [40] uses the RAID4 disk organization with data and parity caching. Stodolsky et al. [41] proposed a parity logging scheme in which parity and log regions are distributed over the disks in the array.

Section 4.2 describes the trace data and the system model used in our simulations. In Section 4.3, we present the experiments conducted and discuss the results. Finally, Section 4.4 contains some conclusions.

4.2 Workload and System Model

To evaluate the different redundant disk array organizations, we have used data from operational transaction processing systems, from IBM customer sites. We use one very large trace containing over 3.3 million I/O requests accessing an active database residing on 130 data disks and a second trace from a smaller system containing about 70 thousand I/O's and accessing an active database consisting of 10 data disks. The traces were collected using a low overhead tracing facility at installations running the DB2 database management system. The trace entries contain the absolute address of the block accessed, the type of access (read, write) and the time since the previous request. The time field is set to zero when both accesses are part of the same multiblock request.

Using these data, we simulate the behavior of the I/O subsystem. We account for all channel and disk-related effects, but we ignore cpu and controller processing times. The disk parameters used in the simulations are shown in Table 4.1. The total capacity of each disk is about 0.9 GByte. To compute the seek time as a function of the seek distance, we use a nonlinear function of the form $a\sqrt{x-1}+b(x-1)+c$, x denoting the seek distance. Table 4.2 shows the characteristics of the traces used. We see that 98% of the accesses in Trace 1 and 95% of the accesses in Trace 2 are single-block accesses. The percentage of writes is 10% for Trace 1 and 28% for Trace 2.

We compare four different organizations: *Base*, *Mirror*, *RAID5*, and *Parity Striping*. In the Base organization, disks are accessed independently without any striping or redundancy. The disks are divided into arrays of equal capacity. Each array can hold

Table 4.1 Disk and channel parameters.

Rotation speed	5400 rpm
Average seek	11.2 ms
Maximal seek	28 ms
Tracks per platter	1260
Sectors per track	48
Bytes per sector	512
Number of platters	15
Channel transfer rate	10 MB/s

Table 4.2 Trace characteristics.

	Trace 1	Trace 2
Duration	3hr 3min.	1hr 40min
# of disks	130	10
# of I/O accesses	3,362,505	69,539
# of blocks transferred	4,467,719	143,105
# of single block reads	2,977,914	48,339
# of single block writes	312,961	17,557
# of multiblock reads	47,324	2,029
# of multiblock writes	24,306	2,098

the equivalent capacity of N independent data disks. In the Base organization, there are N disks per array. In the Mirror organization, an array consists of $2N$ disks. In the case of the RAID5 and parity striping organizations, data and parity in each array are spread over $N + 1$ disks. Each array has one controller and an independent channel connecting it to the host. When comparing the various organizations, we make equal capacity comparisons as opposed to equal cost comparisons. We basically assume that we have a given database that has to be stored and that, for each organization, only the minimum number of disks needed to store the data is used. Therefore, the Mirror organization uses twice as many disks as the Base organization while a RAID5 or parity striping organization with $N+1$ disks per array uses $N+1/N$ times the number of disks in the Base organization. For RAID5 and parity striping, the total number of disks used changes with N. For Trace 1 and $N = 5$, RAID5 and parity striping use 26 arrays containing 6 disks per array or a total of 156 disks while, for $N = 10$, 13 arrays containing 11 disks per array or a total of 143 disks are used.

We compare the organizations both with cached controllers and noncached controllers. In the case of cached controllers, we also consider a RAID4 organization that uses $N +$ 1 disks per array: N disks for data and one for parity. Table 4.3 shows the various organizations considered in our study. No spindle synchronization is assumed. Block size is 4 kBytes. For the RAID5 and parity striping organizations, we study the effect of various parameters such as the striping unit in RAID5, the placement of the parity areas in parity striping, and the policy used to synchronize the parity access and the corresponding data access(es) within an update request.

Table 4.3 Disk array organizations.

Non-cached organizations		Base
		Mirrored disks
		RAID5
		Parity striping
Cached organizations	Data caching	Base
		Mirrored disks
		RAID5
		Parity striping
	Data & parity caching	RAID4

For parity organizations (RAID5 and Parity striping), when updating a single block or a portion of a stripe (less than half a stripe), the old data and old parity have to be read to compute the new parity. The access to the parity disk consists of reading the old parity block waiting for a full rotation and then writing the new parity block at the same location as the old. However, the write to the parity disk cannot occur until the old data have been read and the new parity has been computed. If one or more of the data disks has not completed the read operation for the old data by the time the head of the parity disk comes back to the parity block location, then the parity cannot be written and another full rotation time will be spent before the parity write can be performed. This can occur more than once if one of the data disks is very busy. We compared five different strategies for handling the synchronization between the parity disk and the data disks. The first strategy is Simultaneous Issue (SI) in which the parity access is issued at the same time as the accesses to data, and if the old data are not available by the time the parity disk reads the old parity and accomplishes a complete rotation, then the parity

disk is held for the duration of some number of full rotations until the old data have been read. The second strategy is Read First (RF) and consists of waiting for the old data to be read before issuing the parity access. This strategy minimizes disk utilization but it might unnecessarily increase the response time of update requests. Another strategy that also minimizes disk utilization without unduly increasing update response time is the Read First with PRiority (RF/PR) method, which waits for the old data to be read before issuing the parity access, but gives the parity access higher priority than nonparity accesses queued at the same disk. The fourth strategy consists of waiting for the data access(es) to reach the head of the queue and acquire the corresponding disk(s) before issuing the parity request. This strategy is called Disk First (DF). This strategy reduces the response time of the update access compared with the RF policy but may increase disk utilization slightly since the parity access could finish reading the old parity block and perform a full rotation before the read of the old data is completed. A variation on the DF policy consists of giving parity requests priority over other requests. This is called Disk First with PRiority (DF/PR). An analytical model for the performance of the DF/PR policy was developed by Chen and Towsley [36].

In noncached organizations, we assume that a number of track buffers is used in the controller to reduce the effects of channel contention on performance. Write data are transferred on the channel to the buffers and when the disk head arrives at the appropriate location they are written to the disk surface. Similarly reads are transferred from the disk to the buffers and when the channel is available they are sent to the host. This avoids having to wait an extra rotation if the disk head is at the appropriate location

but the channel is busy. The buffers are also used to hold the old data and parity that are read from disk in order to compute the new parity. The number of track buffers in the controller is proportional to the number of disks in the array attached to the controller. In our simulations we use five track buffers per disk.

In cached organizations, nonvolatile memory should be used to protect against the loss of write data in the event of a power failure. If volatile memory is used, then the cache should be flushed frequently to reduce the extent of data loss when a failure occurs [42]. There is one cache per array. Read hits are satisfied from the cache. The response time for a read hit is equal to the response time (waiting time and transfer time) at the channel. On a read miss the block is fetched from disk. If the replaced block is dirty, it has to be written to disk. The cache replacement policy is LRU. On a write hit, the block is simply modified in the cache. In organizations using parity (RAID5 and Parity striping), when a block is modified, the old data are kept in the cache to save the extra rotation needed to read the old data when writing the block back to disk. The old parity still has to be read and an extra rotation is required at the disk holding the parity. On a write miss, the block is written to the cache and the block at the head of the LRU chain is replaced. A background *destage* process groups consecutive blocks and writes them back to disk in an asynchronous fashion. By using such a process, dirty blocks are destaged to disk before they reach the head of the LRU chain. Hence, write misses typically do not incur the cost of a disk access to write back a dirty replaced block. Only read misses have to wait for the block to be fetched from disk. The overall I/O response time is mainly determined by the read miss access time. The *destage* process

turns small random synchronous writes into large sequential asynchronous writes. In our simulations, the destage process is initiated at regular intervals. The time between two initiations of the process is the *destage period.* The write accesses issued by the destage process are scheduled progressively so that they will cause minimal interference with the read traffic.

For organizations using parity, the destage process accomplishes two purposes: it groups several dirty blocks together to perform a single multiblock I/O and frees up space in the cache by getting rid of blocks holding old data. Decreasing the destage period increases the write traffic seen by the disk. Increasing it reduces the hit ratio and increases the likelihood that the block at the head of the LRU chain is dirty which may cause a miss to wait for the replaced block to be written to disk.

One might wonder whether the destage policy used is better than the basic LRU policy in which dirty blocks are written back only when they get to the head of the LRU chain and a miss occurs. The question is even more relevant in the case of the Base and Mirror organizations in which old data blocks are not kept in the cache. We have compared the two policies for various cache sizes and found that the periodic destage policy always performs better for all organizations. In [38] a background process is used to write dirty blocks from the head of the LRU chain along with other dirty blocks in the cache that belong to the same track. In organizations using parity, there is a need for freeing old data blocks periodically even if the corresponding dirty block is not at the head of the LRU chain. It might be useful though to decouple the two issues by using the destage process that writes dirty blocks from the head of the LRU chain more frequently

while a flushing process is only initiated from time to time to scan the entire cache and free old data blocks.

We also examine the use of parity caching in combination with a RAID4 organization. In this case, when a write is performed, the parity is computed and written to the cache instead of writing it directly to the parity disk. The parity blocks are sorted by cylinder number and spooled to the parity disk using the SCAN policy. In the case of single block accesses, what is kept in the cache is not the actual parity but the xor of the old and new, data and when the block is to be updated on the parity disk, the old parity must be read to compute the new parity. In the case of full stripe writes, the actual parity is computed and held in the cache and then written to the parity disk without reading the old parity. For partial stripe writes, either case may occur depending on the size of the request. In the case where the parity disk queue becomes large enough to occupy the entire cache, reads and writes are serviced directly from disk and writes have to wait for a block to become free in the cache in order to store the parity update.

4.3 Experiments

Unless otherwise specified, the parameters shown in Table 4.4 are used by default in the following experiments.

Table 4.4 Default parameters.

Array size	$N = 10$
Block size	4KB
Synchronization method	Disk first
Striping unit for RAID5	1 block
Parity placement for ParStrip	Middle cylinders
For cached organizations:	
Cache size	16MB

4.3.1 Synchronization

Figure 4.1 shows the results for the various synchronization policies for both RAID5 and parity striping. We see that the naive strategy (SI) has significantly worse performance than the other policies and DF performs better than RF because it reduces the response time of update accesses without significantly affecting disk utilization. The variation that gives priority to the parity access achieves better performance with both the DF and RF policies. Hence, overall, DF/PR is the best synchronization strategy. For larger array sizes, the gap between the performance of the various strategies narrows because the amount of queuing is smaller for large arrays because of better load balancing in the case of RAID5 and because of the reduced correlation between increases in load in the case of parity striping.

4.3.2 Uncached arrays

In a first experiment, we looked at noncached organizations and measured the performance of all four organizations for different array sizes. Figure 4.2 shows the response

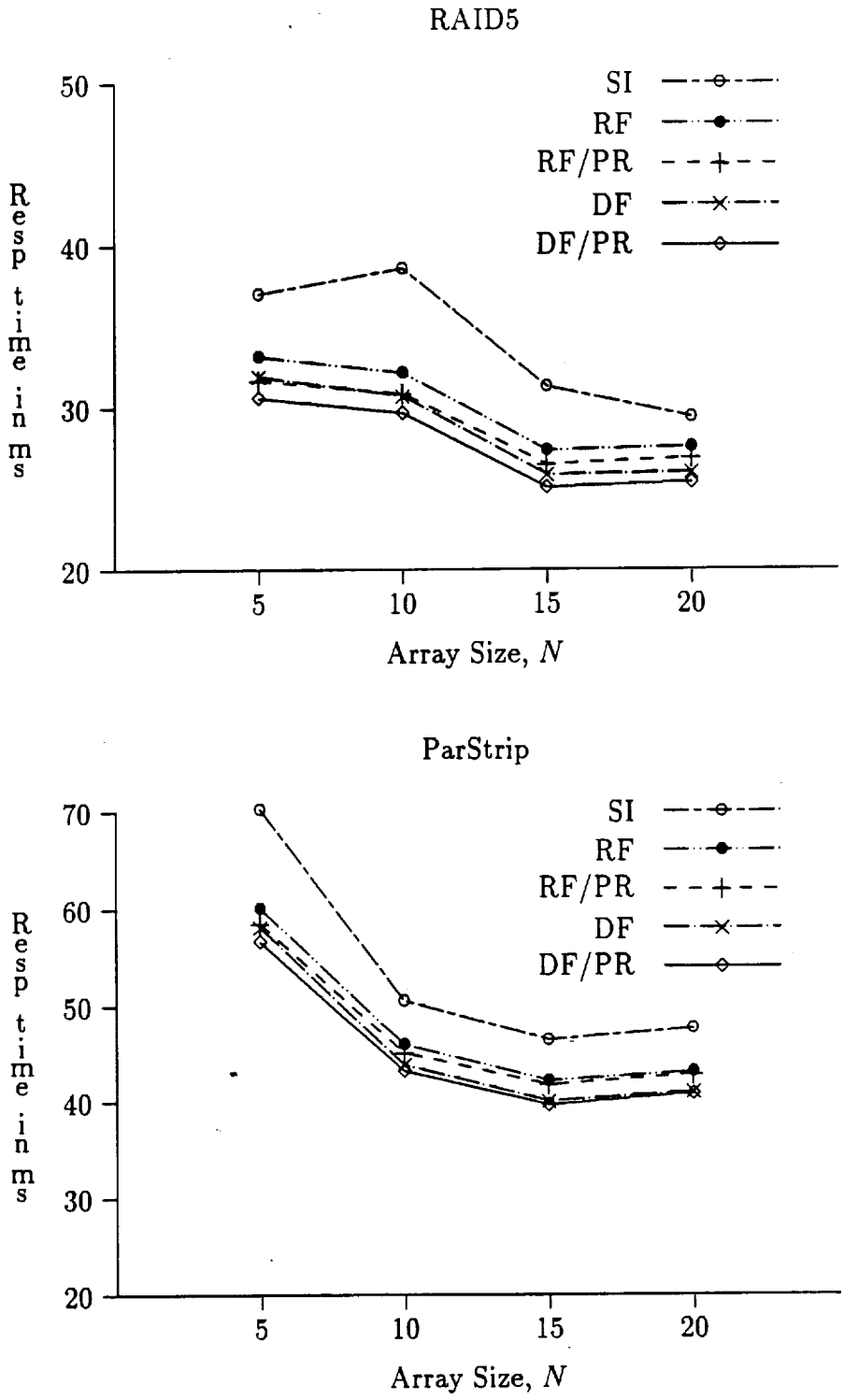


Figure 4.1 Response time for different synchronization methods vs. array size.

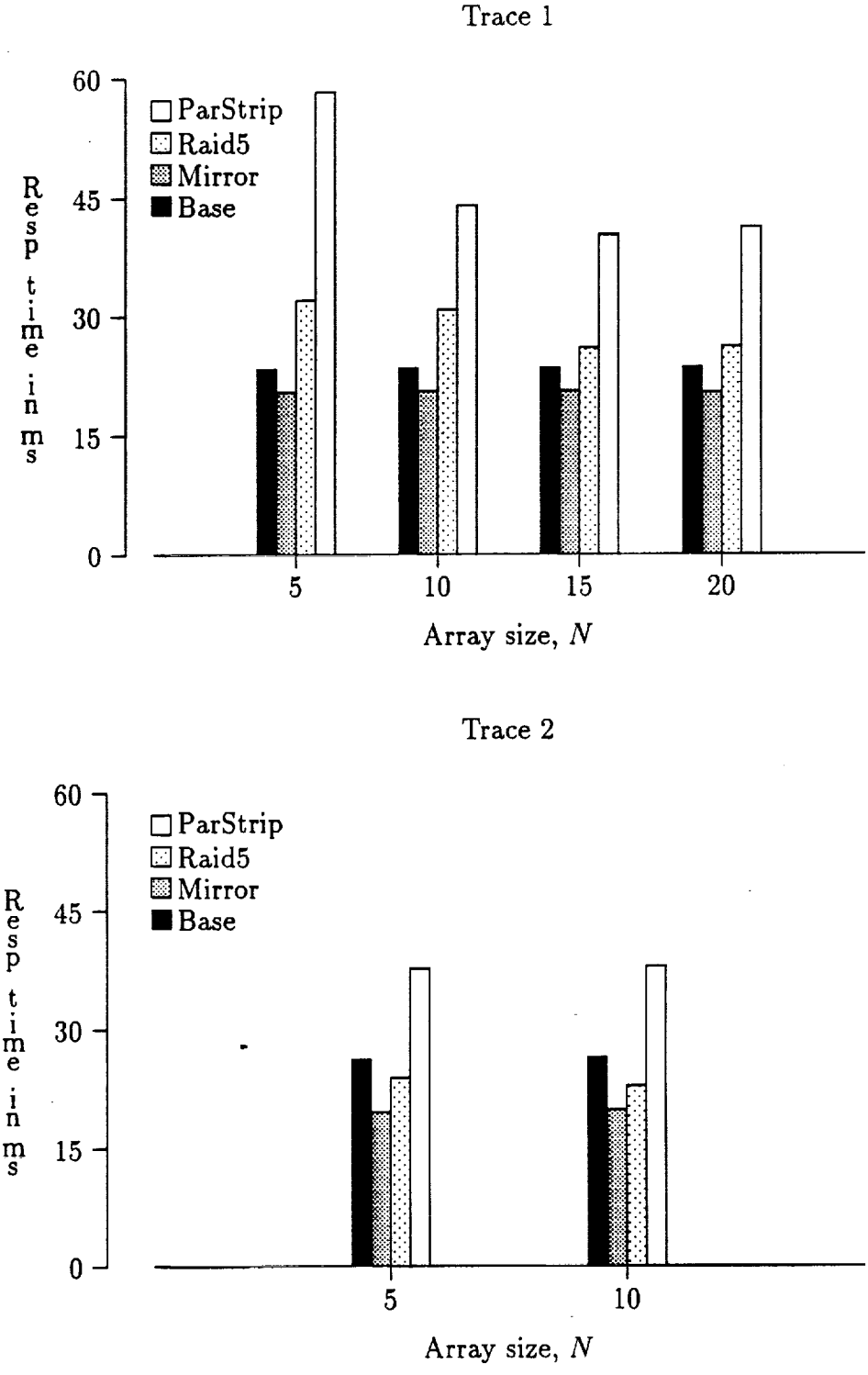


Figure 4.2 Response time vs. array size.

time in milliseconds for values of N from 5 to 20. In the parity striping organization, the parity areas were placed on the middle cylinders.

For mirrored disks, the response time for writes is the largest of the response times at the two disks in the mirrored pair. Reads, however, encounter less queuing since both disks of the pair can service reads in parallel. Moreover, the shortest seek optimization[4] is used to further reduce read response time. Since there are many more reads than writes, the overall performance of mirrors is better than the Base organization. For $N = 10$, the response time of mirrors is shorter than that of the Base organization by 12% for Trace 1 and 25% for Trace 2. The reason mirrors perform better for Trace 2 in spite of the higher write fraction is their ability to split the read load over two disks which reduces queuing.

Comparing RAID5 to the basic organization, we notice that for Trace 1, there is a significant decrease in performance associated with RAID5. Given that the fraction of large requests is small, the advantage of RAID5 in terms of high transfer rates cannot be fully exploited. There are two major effects that determine the performance of RAID5: one is the the cost of small write requests and the other is the load balancing issue. To service a single-block write request, the data and parity disks have to be read to get the old data/parity; then, the new data and parity blocks have to be written to the disk. Reading the old data/parity adds an extra rotation time to the response time of the request at each of the two disks involved. However, the response time of the parity update can be affected by queuing delays at the data disk since the parity write cannot be initiated until the old data have been read. The increased cost of write requests

[4]A read is directed to the disk that has its arm nearest to request's location.

also affects read requests since it increases queuing for the disks. RAID5 balances the load over the disks in the array which reduces queuing delays. Another parameter that affects both read and write requests is *seek affinity*, which is a measure of the spatial locality that may exist among disk accesses. The higher the seek affinity, the smaller the disk arm movements. Data striping decreases seek affinity and hence increases average seek distance and seek time. In the case of Trace 1, the write penalty issue is more important than the load balancing issue. For Trace 2, load balancing seems to have a more important effect on performance than the write penalty. This is due to the fact that there is a lot of skew in the accesses to the disks in Trace 2. Note that for $N = 10$, the results for Trace 1 represent the average over 13 different arrays while for Trace 2, there is only one array.

The difference in performance between parity striping and RAID5 is mainly a result of the ability of RAID5 to balance the load over all of the disks in the array. For single block accesses, the service time at the disk (seek + latency + transfer) is higher in the case of RAID5 because of decrease in seek affinity, but RAID5 more than makes up for it by reducing queuing delays. The main argument used in [8] against RAID5 is the increased disk utilization due to having many arms service a single request. This does not happen, however, if the striping unit is chosen appropriately. In transaction processing workloads, most requests are for single blocks. If the striping unit is a multiple of the block size, then most small requests are serviced by a single disk.

Note that tuning the placement of the data on disk can reduce the skew in disk accesses and, hence, reduce the gap in performance between RAID5 and parity striping.

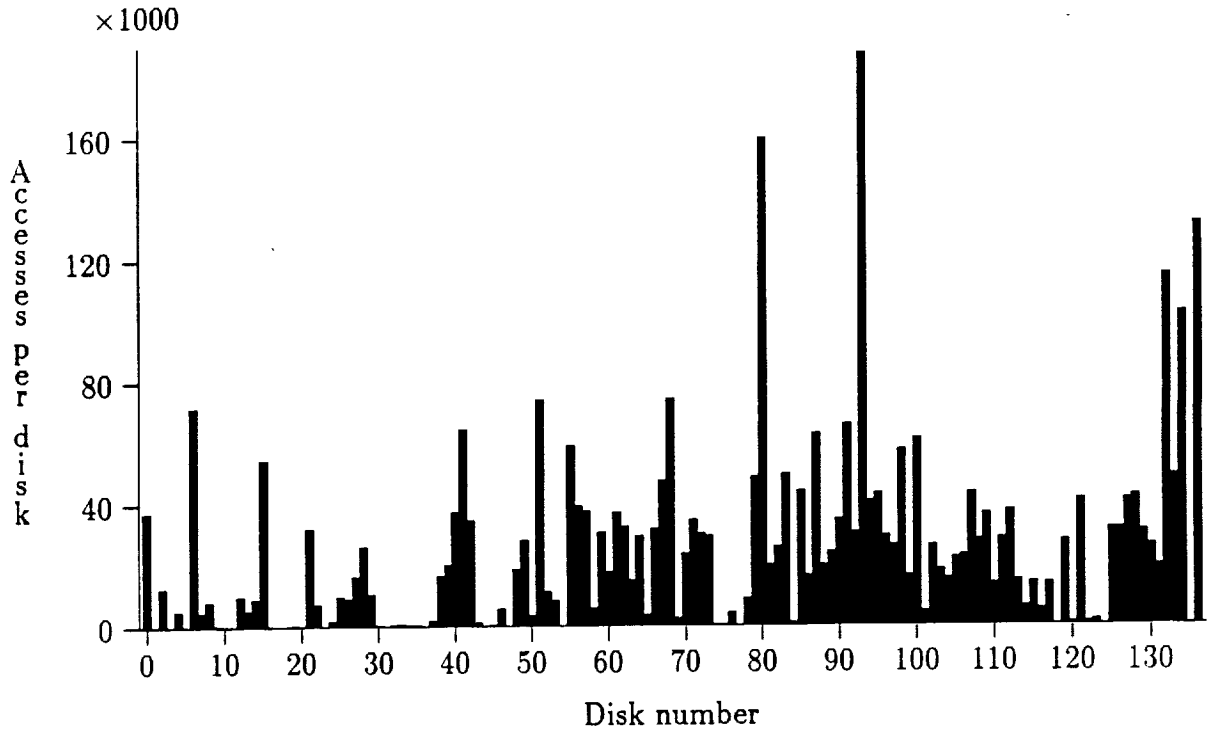


Figure 4.3 Distribution of accesses to disks in the Base organization (Trace 1).

However, RAID5 provides a way to balance the load automatically. Figures 4.3 and 4.4 illustrate this effect. In Figure 4.3, the total number of accesses to each disk is plotted for Trace 1 for the Base organization, while in Figure 4.4 the distribution is plotted in the case of the RAID5 organization with 4KB striping unit. Figure 4.3 shows that there is a significant amount of skew in the disk access rate. Most of the skew within the array is smoothed out in the RAID5 organization.

4.3.2.1 Array size

Changing the array size does not significantly affect the performance of the Base and Mirror organizations. There is only a very small increase in response time as the array size increases due to added channel contention since the same channel is servicing more

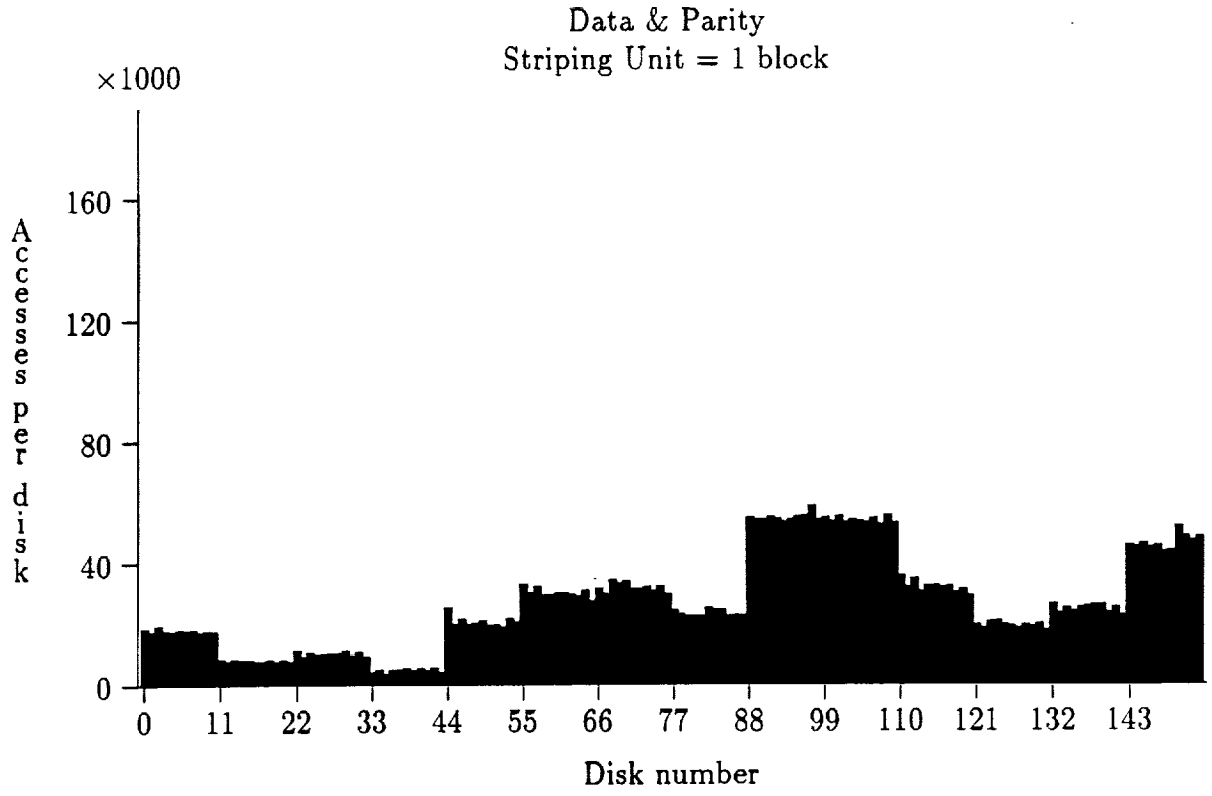


Figure 4.4 Distribution of accesses to disks in the RAID5 organization (Trace 1).

disks. In the case of RAID5, the performance is affected by the fact that smaller arrays use more disks (for $N = 5$, there is one extra disk for every five data disks, while for $N = 10$, there is one extra disk for every ten disks). The other effect is that for large arrays the load is balanced over more disks, which means that the risk of encountering large queuing delays is further reduced. For Trace 1, $N = 20$ has the lowest cost and very good performance. However, large arrays are less reliable and have worse performance during reconstruction following a disk failure.

Figure 4.2 shows that, for Trace 1, the parity striping performance deteriorates for small arrays. One cause of this behavior is the fact that the parity area becomes larger for small arrays which increases the seek distance of reads and data writes since the

C-2.

parity area is in the middle of the disk. This is more apparent in Trace 1 because it has a higher read fraction. The effect of the placement of the parity is analyzed in more detail in Section 4.3.2.3. Another cause of the performance degradation is that the parity striping organization aggravates the skew problem because when a hot spot appears on one disk and the disk becomes a bottleneck, the disk holding the corresponding parity area also experiences increased load and possibly long queues, which, in turn, affect the performance of other disks in the array. This phenomenon is more severe for small array sizes. One possible solution to this problem would be to use a finer grain in striping the parity so that the parity update load is more balanced over the disks. Such an organization would preserve the benefits of seek affinity in the case of read accesses and writes to the data. It also preserves other useful properties of parity striping, such as better fault containment than RAID5, control over the distribution of data over the disks and various software benefits.

4.3.2.2 Striping unit in RAID5 organizations

The striping unit for RAID5 was varied from 1 block to 64 blocks with $N = 10$. The tradeoff between small and large striping sizes is similar to the tradeoff between RAID5 and parity striping. Large striping sizes provide better seek affinity and reduce total disk utilization by avoiding situations in which multiple arms move to service a small multiblock request. However, they do not balance the load as well as with small striping units. Figure 4.5 shows the results. For Trace 1, the optimal striping unit is 8 blocks or 32 kBytes. There is little difference in performance, however, between values from 1 to

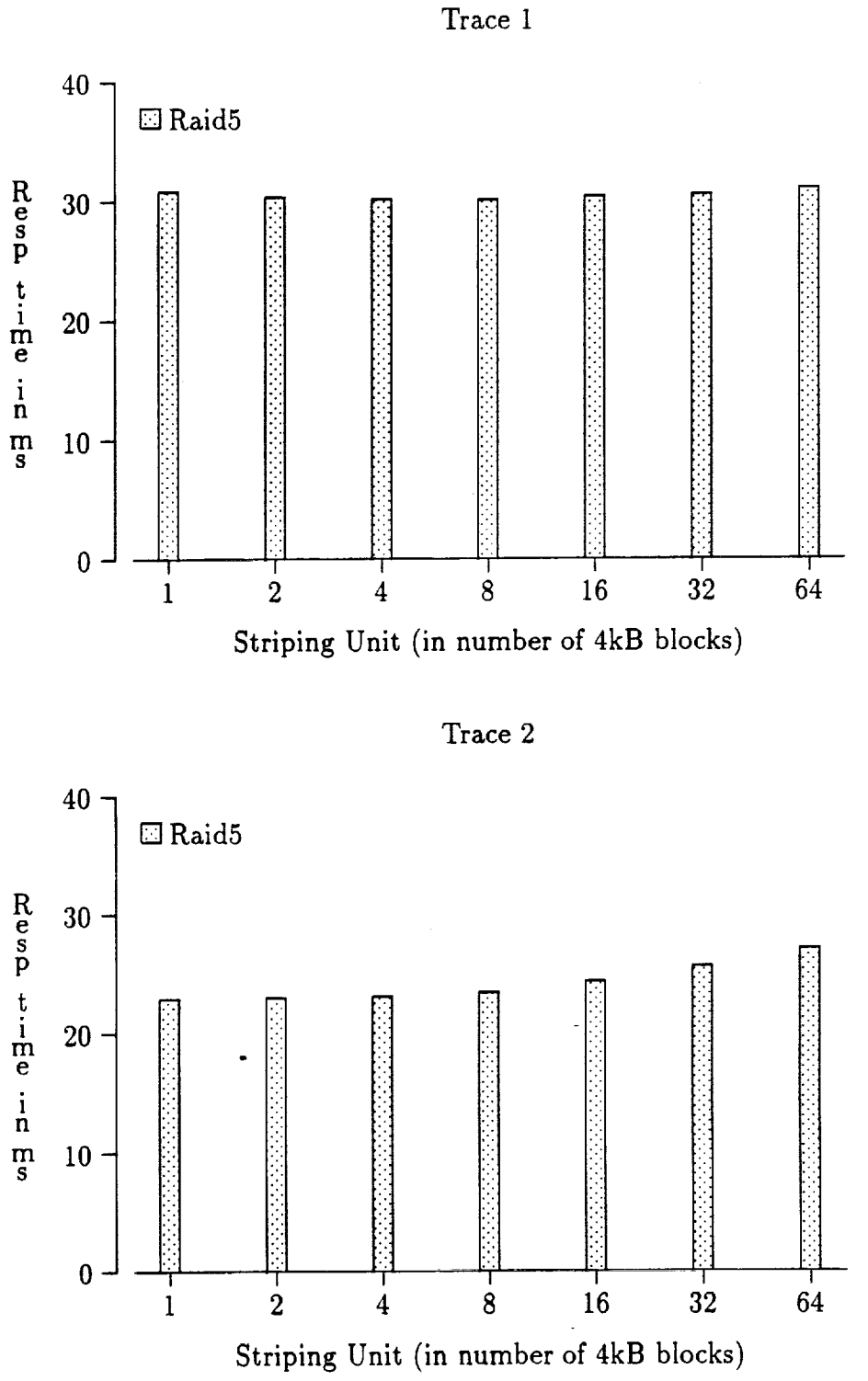


Figure 4.5 Response time vs. striping unit for RAID5.

16 blocks. For Trace 2, the optimal striping unit is 1 block which indicates that there is more need for load balancing in the case of Trace 2. For a striping size of 32 blocks or more, the performance starts degrading significantly and, for very large striping units, it approaches that of parity striping.

4.3.2.3 Parity placement in parity striping organizations

In [8], it was suggested that since the parity area is accessed frequently, it should be placed on the cylinders at the center of the disk. We found that this does not always improve performance especially for small arrays where the parity areas are quite big and when the workload has a high read-write ratio. A simple model can be used to explain this effect. Assuming that accesses are uniformly distributed over all disks in the array and that accesses to a given disk are uniformly distributed over all data areas on the disk, the access rate to any one of the N data areas on each disk is equal to $1/N^2$ times the total access rate to the array while the access rate to any given parity area is w/N times the total access rate to the array, where w is the fraction of accesses that are writes. Hence the parity areas are accessed more often than the data areas if and only if $w > 1/N$. In the workload of Trace 1, we have $w = 0.1$. Hence according to the model, for $N > 10$, the parity area should be placed in the middle of the disk while for $N < 10$, it should be placed at the end of the disk. In Figure 4.6 the results for the two placements are shown for various values of N. For Trace 1, we observe that the rule established by the above model is verified except that the cutoff point occurs somewhere between $N = 5$ and $N = 10$ (probably closer to 10 than to 5 given the large difference

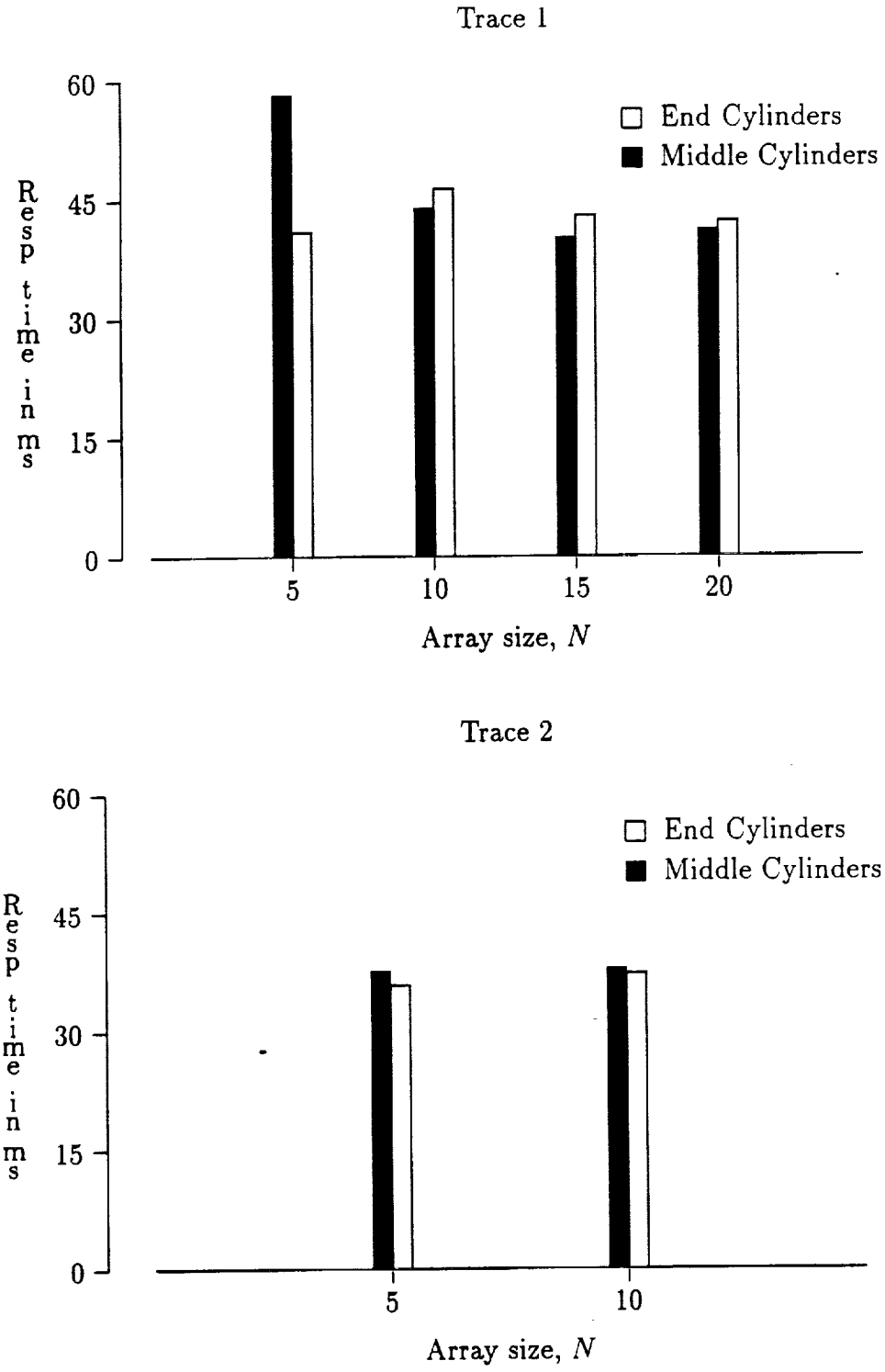


Figure 4.6 Comparison of parity placements for parity striping.

in performance seen for $N = 5$). For Trace 2, the rule does not seem to be satisfied which means that the uniform access assumptions break down in this case. However, we see that the middle cylinder placement is worse for small N, which confirms the trend suggested by the above model.

4.3.2.4 Modifying trace speed

To get an idea of the performance of the various organizations at higher or lighter loads, we have conducted an experiment in which the trace was speeded up by a factor of 2 and another one where the trace was slowed down by a factor of two. Note that the workloads obtained by speeding up the trace do not reflect the characteristics of any real system. Doubling the processor speed does not imply that I/O's will be issued twice as fast since transactions may have to wait for one I/O to finish before issuing another one. RAID5 response time degrades gracefully as the load increases. RAID5 does even better than mirrors when the load doubles. The response times for parity striping and to a lesser degree that of the Base organization degrade severely as trace speed is doubled. Figure 4.7 shows the results. For Trace 2 and for a trace speed of 0.5, the base organization performs better than RAID5 because at low trace speed there is little queuing and RAID5 loses its load balancing advantage.

4.3.3 Performance of cached organizations

The cache hit ratio is slightly lower for RAID5 and parity striping because of the space held by the old blocks in the cache. The read and write hit ratios are plotted

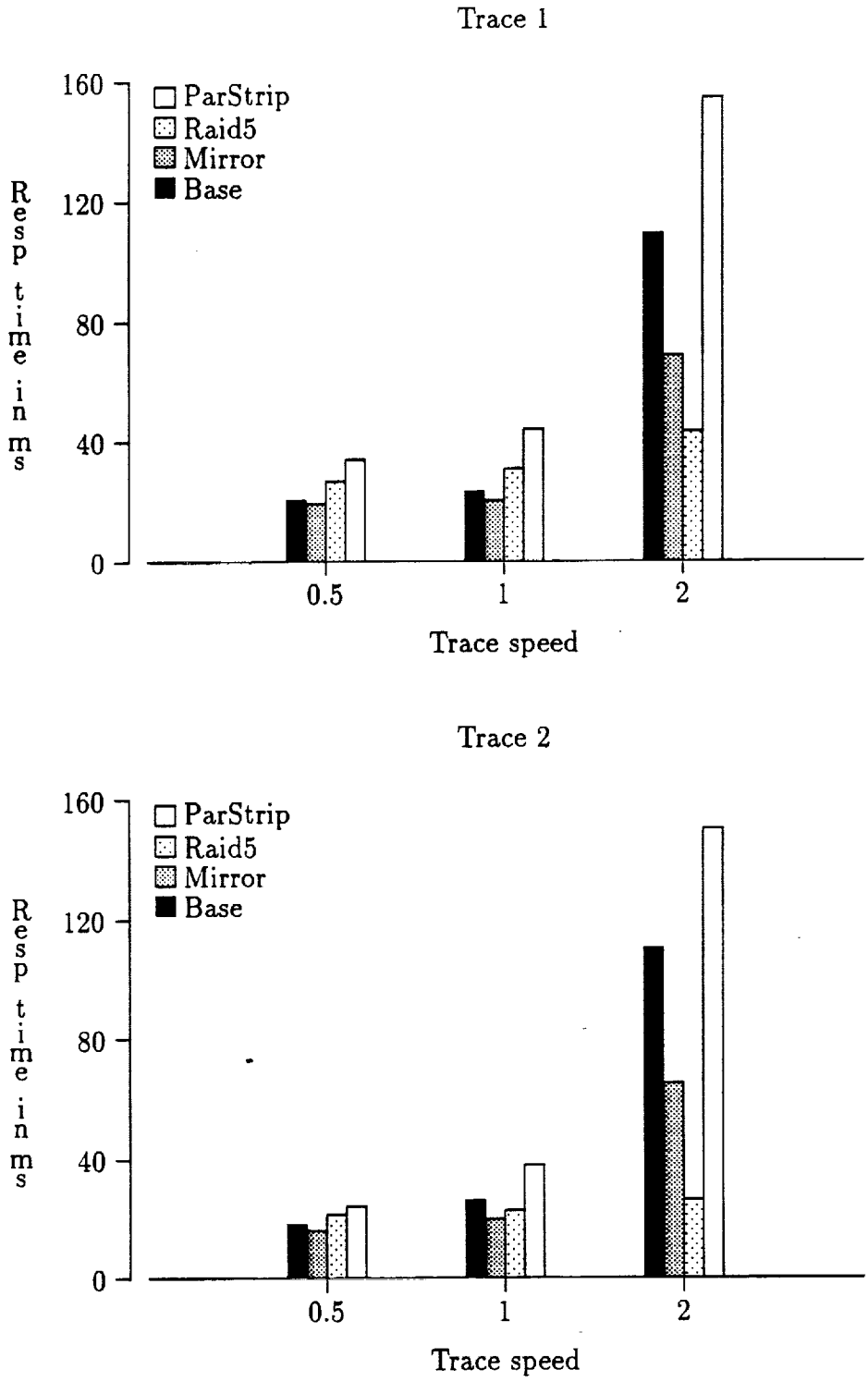


Figure 4.7 Response time vs. trace speed.

in Figure 4.8 for both traces and both for organizations using parity (RAID5, parity striping) and for those not using parity (Base, Mirror). Multiblock accesses are counted as hits only if all of the blocks requested are in the cache. For Trace 1, the write hit ratio is almost one for large caches because blocks are usually read by the transaction before being updated. For Trace 2, the write hit ratio starts at about 20% and increases to over 60%. The workload in Trace 2 seems to contain large working sets that require a relatively large I/O cache. The read hit ratio is relatively low for a small cache size ($\approx$ 9% for Trace 1 and $<$ 1% for Trace 2 for an 8 MB cache) but it increases to about 54% for Trace 1 and 40% for Trace 2 for a cache size of 256 MB. The effect on hit ratio of keeping the old blocks in the cache is minimal. The difference between the parity and the nonparity organizations is always less than 1% for writes. For reads, the difference in hit ratio is higher. For Trace 1, the hit ratio of the parity organizations is 6% lower for an 8 MB cache but the difference goes down to 2% for a 16 MB cache and keeps decreasing for higher cache sizes. For Trace 2, the relative difference is highest for the 32 MB case, where the hit ratio goes from 4.6% for Base to 3.5% for RAID5, but the gap narrows significantly for higher cache sizes.

4.3.3.1 Cache size

The response time results are shown in Figure 4.9. All organizations benefit from larger cache sizes. The performance of mirrors is still better than for the Base organization. Since each of the disks in the mirrored pair sees the same destage traffic as the corresponding disk in the base organization, the contribution of the destage traffic to read

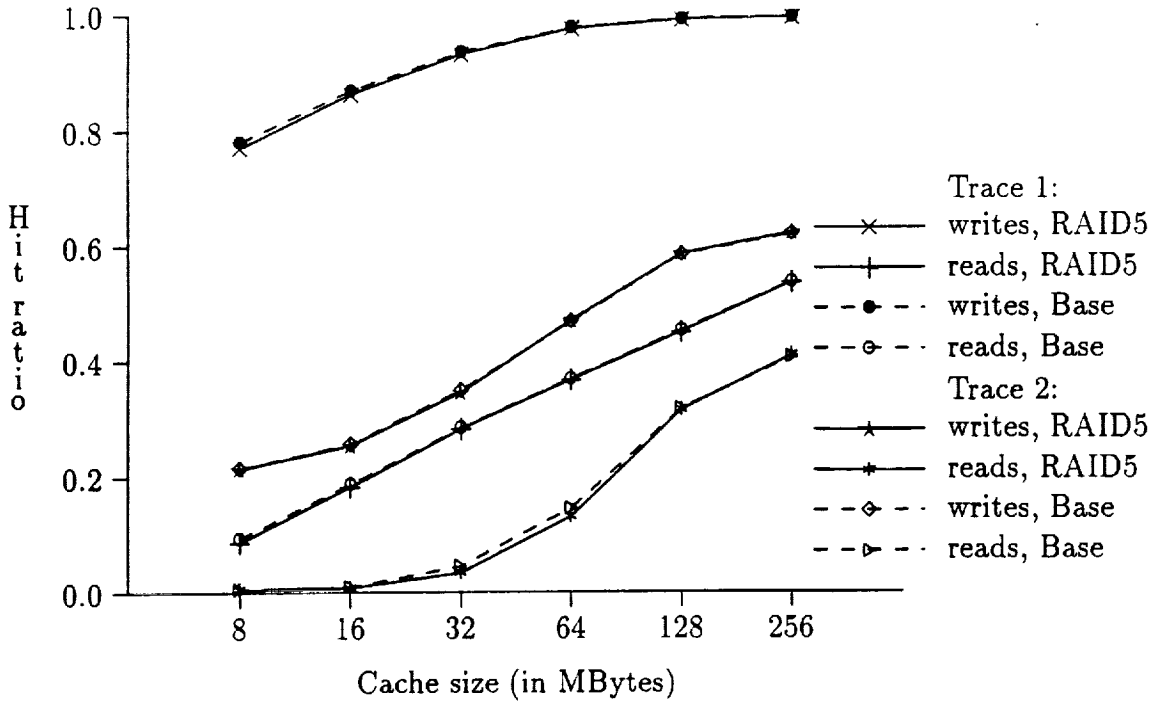


Figure 4.8 Hit ratio vs. cache size.

miss access time is the same in both organizations. In addition, mirrored disks service reads faster because the read load is shared between the two disks in the mirrored pair and because of the shorter seek optimization. For a 16 MB cache size, mirrors perform 22% better than the Base organization for both traces.

The gap in performance between RAID5 and the Base organization reduces considerably in the case of Trace 1 because the larger cost of writes in RAID5 does not affect the overall response time directly but only through its contribution to read miss waiting time. Write costs are also reduced by the fact that old blocks are kept in the cache and that the number of actual writes decreases because multiple updates to the same block while it is in the cache result in one actual write to disk. As cache size increases, the gap gets even smaller in relative terms because the difference in miss ratios becomes smaller;

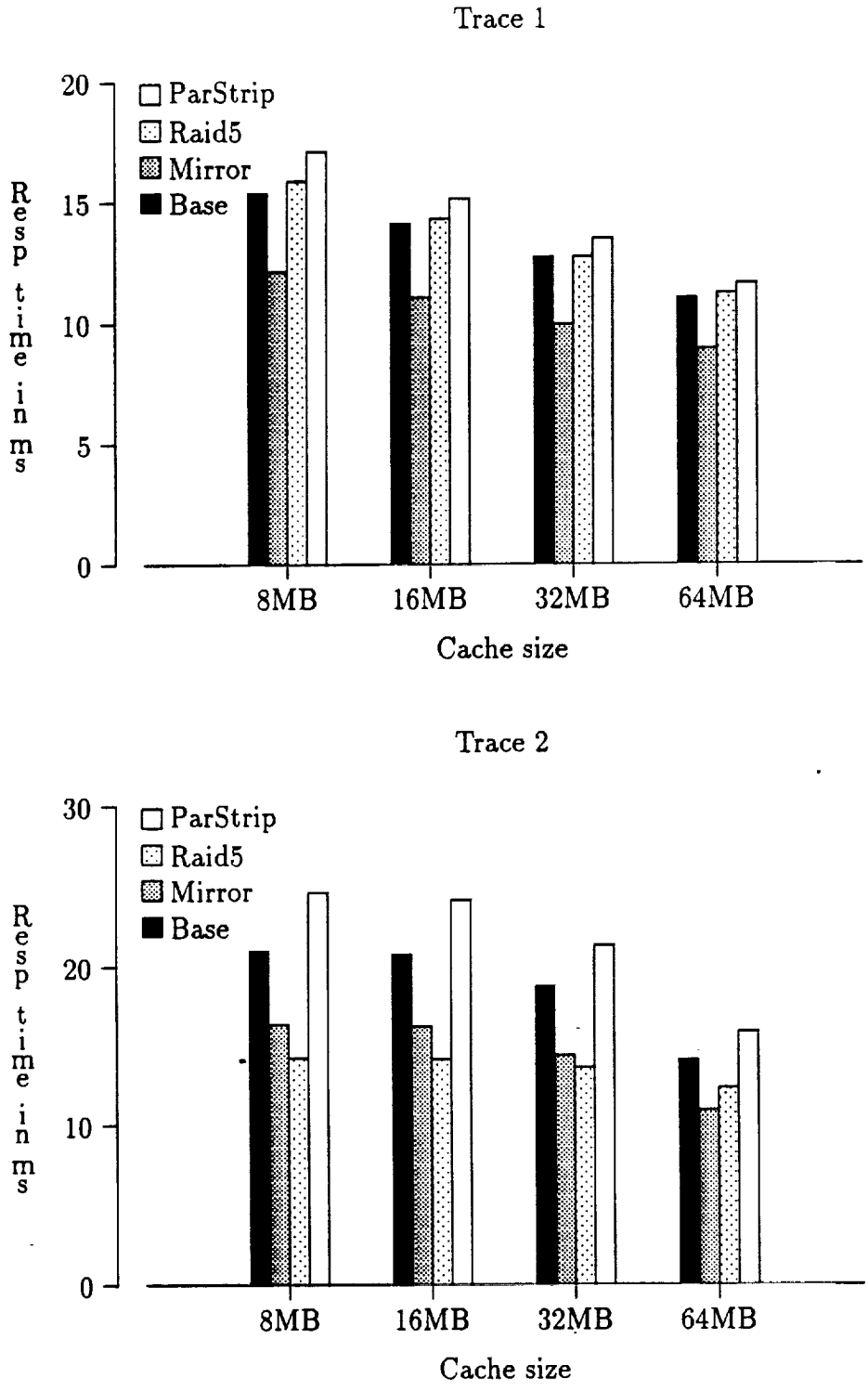


Figure 4.9 Response time vs. cache size.

the likelihood of having the old block in the cache when the write is destaged becomes higher and the probability (which should be already very small) of having to wait for a replaced page to be written to the disk on a cache miss becomes even smaller thus reducing further the contribution of the higher costs of write in RAID5 to the response time. For a cache size of 16MB, RAID5's performance is only about 1% worse than that of the Base organization. In the case of Trace 2, RAID5 does even better than in the noncached case especially for small cache sizes, since the write penalty is practically eliminated while the need for load balancing remains because of the low hit ratio. RAID5 even surpasses mirrored disks for cache sizes less than 64 MBytes. The gap between RAID5 and Base narrows as cache size increases because the need for load balancing decreases.

RAID5 still does better than parity striping. The gap between the two narrows for Trace 1 mainly because of the reduced load at the disks which makes RAID5's load balancing advantage less important in determining response time. For Trace 2, the difference is still significant because of the low hit ratio.

4.3.3.2 Array size

In Figure 4.10, we compare three organizations with different array sizes but with the same total cache size. For $N = 5$, the cache size in each array is 8 MB while for $N = 10$, the cache size is 16 MB and for $N = 15$, the cache size is 24 MB. For the Base and Mirror organizations, the performance improves slightly in the case of Trace 1 in the larger array in spite of the higher channel contention. This implies that a large shared cache for 10 disks is better than two partitioned caches for every five disks. For Trace 2,

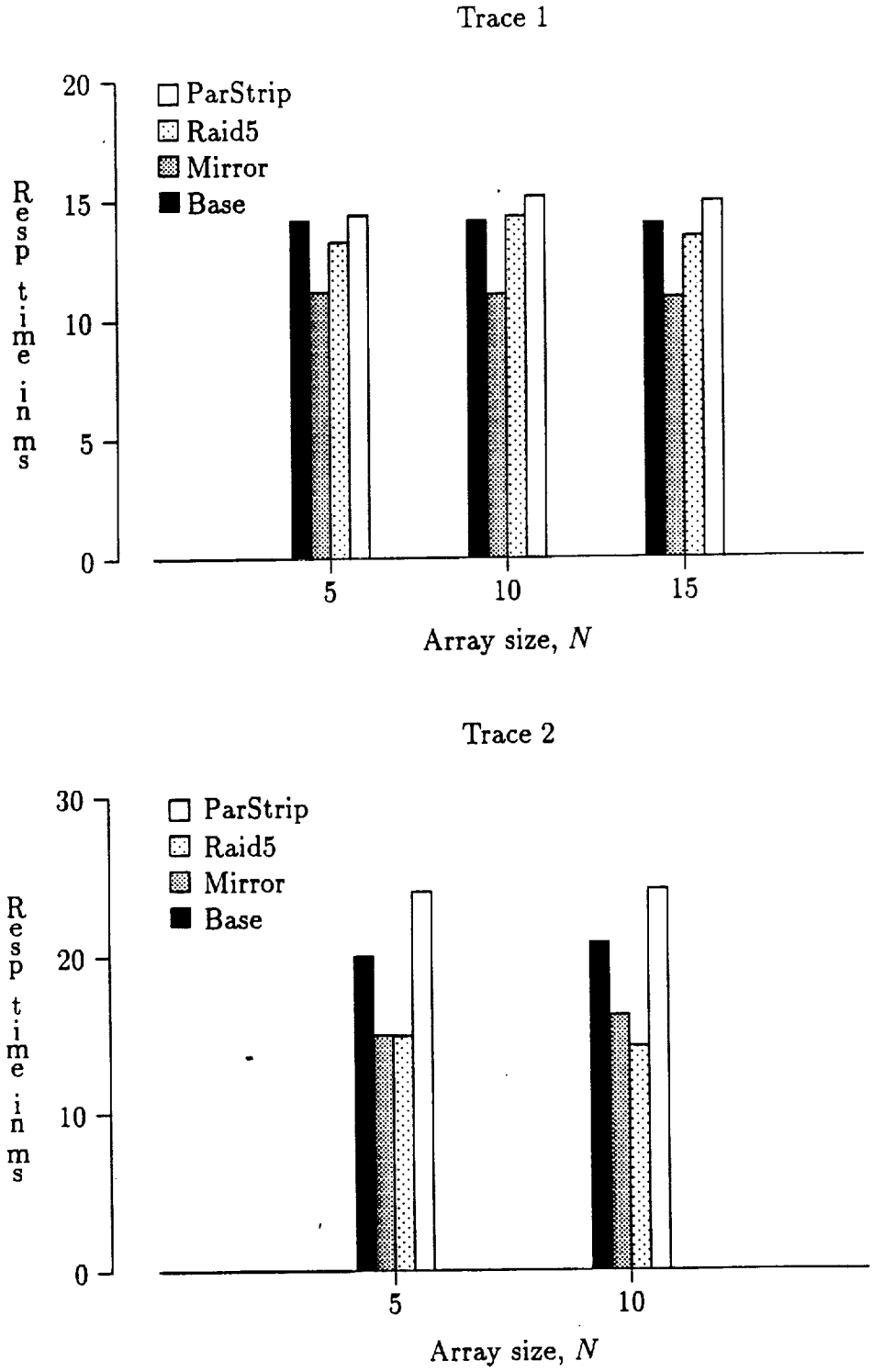


Figure 4.10 Response time vs. array size for cached organizations.

the effect of channel contention is more important than the increase of hit ratio due to the shared cache. In the case of RAID5 and parity striping, the number of disk arms and the load balancing issue have more effect on performance than the difference in hit ratio between a global and a partitioned cache or the channel contention.

4.3.3.3 Striping unit

The response time of the cached RAID5 organization is plotted in Figure 4.11 as a function of the striping unit. For Trace 1, the optimal striping unit in this case is 16 blocks or 64 kBytes compared with 8 blocks for the noncached organization. The reason for the difference is that the load on the array is lighter in the cached organization; therefore, the need for load balancing is not as high. This makes larger striping units more efficient because they can take better advantage of seek affinity and reduce disk utilization on multiblock accesses. For Trace 2, the optimal striping unit is still 1 block as in the noncached organization. This is the case because of the low hit ratio for this trace.

4.3.4 Parity caching

In this section, we examine a RAID4 organization in which the parity resides on a separate disk in the array. The parity updates are buffered in the controller cache before being written to the dedicated parity disk. We compare the performance of such an organization to the RAID5 organization in which only data blocks are cached. We look at the effect of various parameters such as cache size, array size, trace speed, and

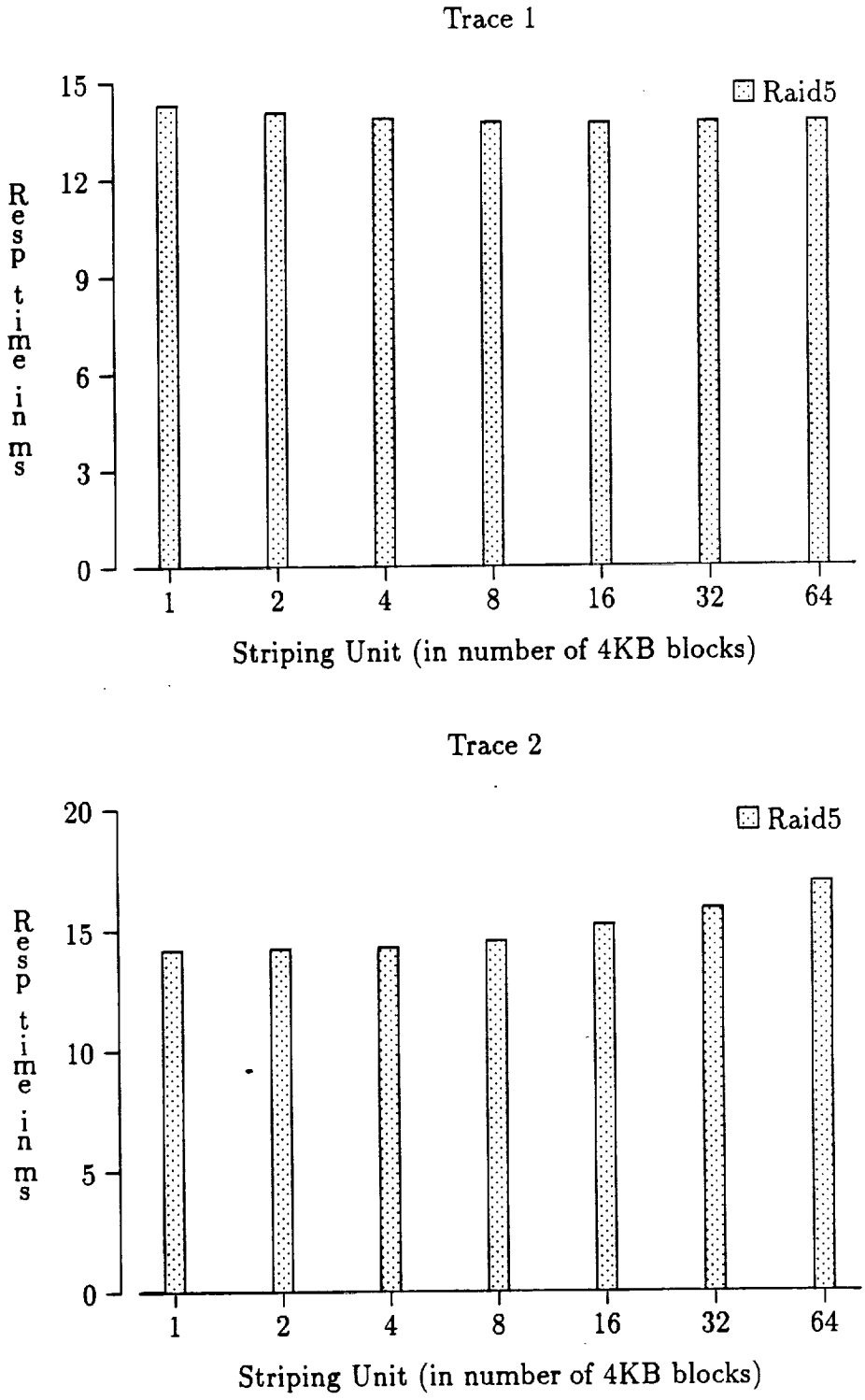


Figure 4.11 Response time vs. striping unit for RAID5 with cache.

striping unit. One benefit of using the RAID4 with parity caching organization is the fact that read miss accesses, will not have to wait behind parity accesses which include an extra rotation due to the need to read the old block before updating it. Another benefit is the reduced average seek distance because parity blocks are not placed in between data blocks. Disadvantages include the reduced hit ratio in the cache due to the space occupied by parity blocks and the fact that the number of disk arms available for servicing the synchronous read miss accesses decreases by one. Another issue is the fact that the parity disk may still become a bottleneck for the entire array if its queue grows and fill A necessary condition

4.3.4.1 Cache size

The read and write hit ratios are plotted in Figure 4.12 for RAID5 and for RAID4 with parity caching. The effect on hit ratio of buffering the parity blocks in the cache in the RAID4 organization is minimal for Trace 1. For Trace 2, the gap is wider; however, the relative difference is significant only in the region where the hit ratio is quite small and therefore has little effect on overall performance.

The response time results are shown in Figure 4.13. For Trace 1, the difference between the two organizations is small but the RAID4 organization always does better. The hit ratio is actually lower for RAID4 than for RAID5 but the fact that parity updates are performed on a separate disk and do not interfere with the read accesses seems to outweigh the effect of the lower hit ratio. As cache size increases, the gap between the two organizations becomes smaller in relative terms because the probability of a synchronous

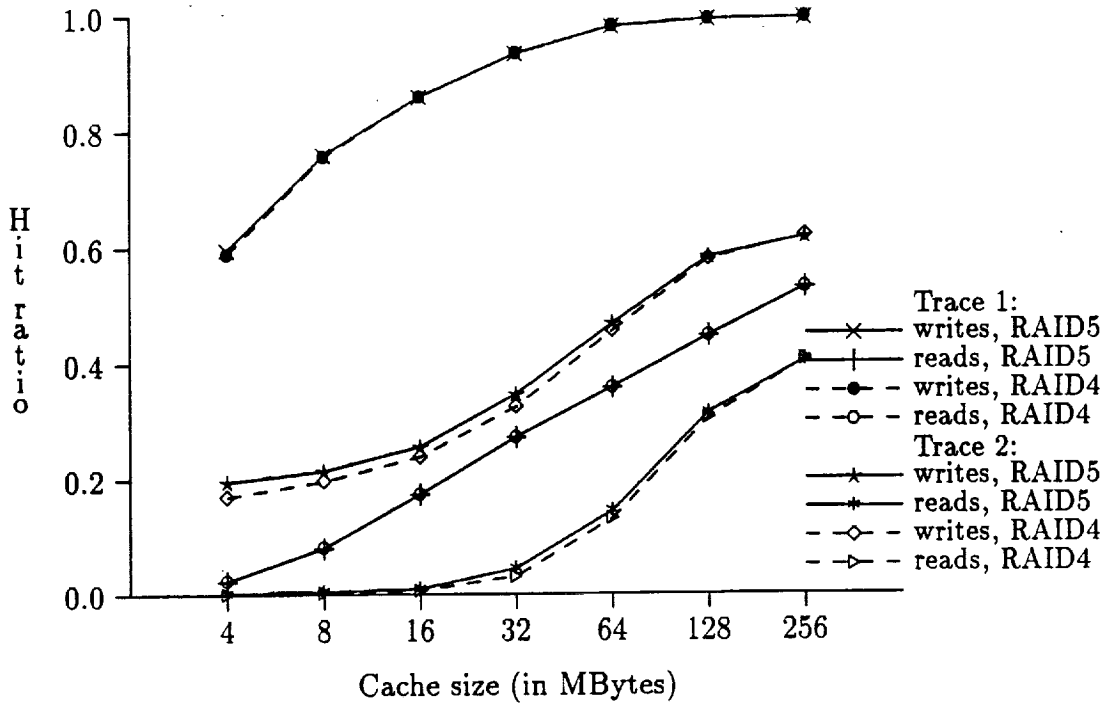


Figure 4.12 Hit ratio with parity caching vs. cache size.

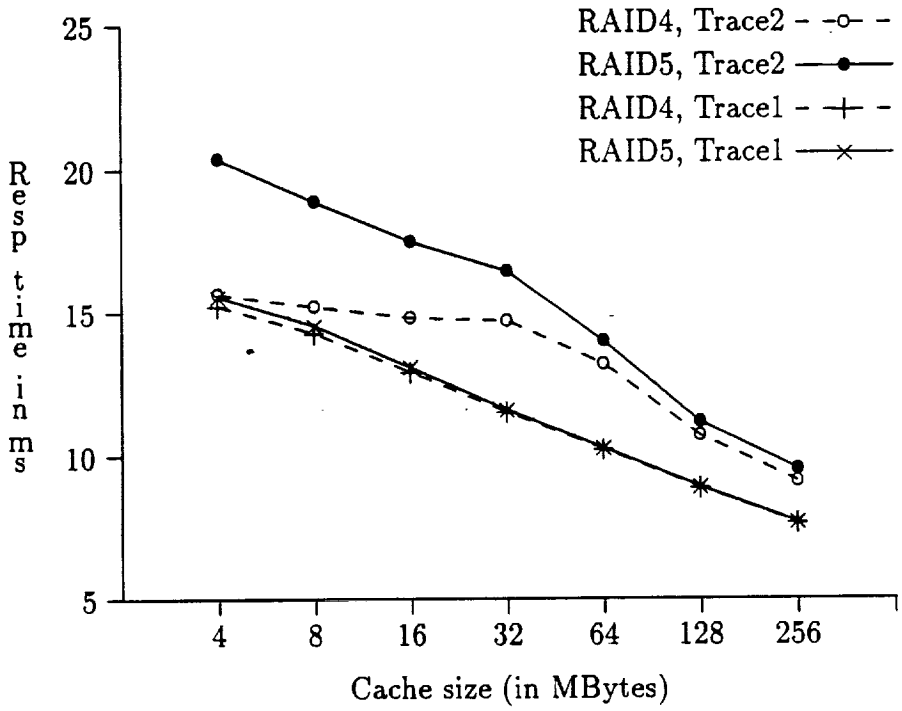


Figure 4.13 Response time vs. cache size.

I/O due to a read miss becoming smaller. The response time of RAID4 is 2% lower for a cache size of 8 MBytes and about 1% lower for a cache size of 16 MBytes.

In the case of Trace 2, RAID4 performs much better than RAID5, especially at small cache sizes. This is due to the higher percentage of writes in Trace 2 ($\approx$ 30%) compared to Trace 1 ($\approx$ 10%) and to the lower hit ratio in the case of Trace 2. The use of RAID4 and parity caching significantly reduces the cost of writes and their effect on read miss response time. For a 16 Mbyte cache the response time for RAID4 is 15% shorter than for RAID5. The gap between the two organizations narrows significantly as the cache size increases.

4.3.4.2 Array size

In Figure 4.14, we compare the two organizations for three different array sizes while maintaining the same total cache size. For $N = 5$, the cache size in each array is 8 MB while for $N = 10$, the cache size is 16 MB, and for $N = 20$, the cache size is 32 MB. As N increases, the number of disk arms decreases but the load is better balanced over the disks. There is also a border effect for Trace 1 and $N = 20$ since the last array is only half full. In this experiment, we are mostly interested in how the two organizations compare with each other at different values of N rather than comparing the same organization for different values of N.

For $N = 5$, RAID5 performs better than RAID4 for both traces because, with RAID4, fewer disks are available to service read requests (5 out 6 disks service read requests for $N = 5$ compared with 10 out of 11 for $N = 10$). This implies that dedicating one disk

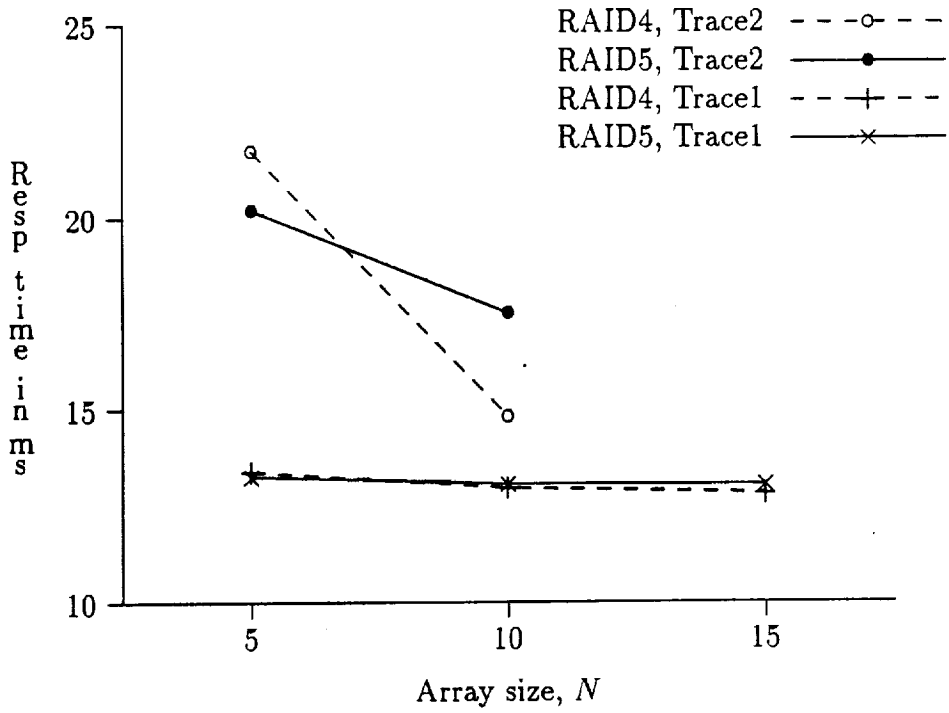


Figure 4.14 Response time vs. array size.

per array for parity updates does not pay off for small arrays. For Trace 1, we see that, going from $N = 10$ to $N = 20$, the gap between RAID4 and RAID5 widens. This is due to the fact that in RAID4, for larger N, a larger proportion of the disks can service read accesses (the ratio $N/N+1$ increases with N). The load on the parity disk increases as N increases but this does not seem to significantly affect the performance of RAID4.

4.3.4.3 Trace speed

The gap between the two organizations widens as the load increases. Figure 4.15 shows the results. In the case of Trace 2, RAID5's performance degrades significantly at high loads. The increasing load on the parity disk in the RAID4 organization did not seem to create a bottleneck. There are periods in the traces where the parity disk queue

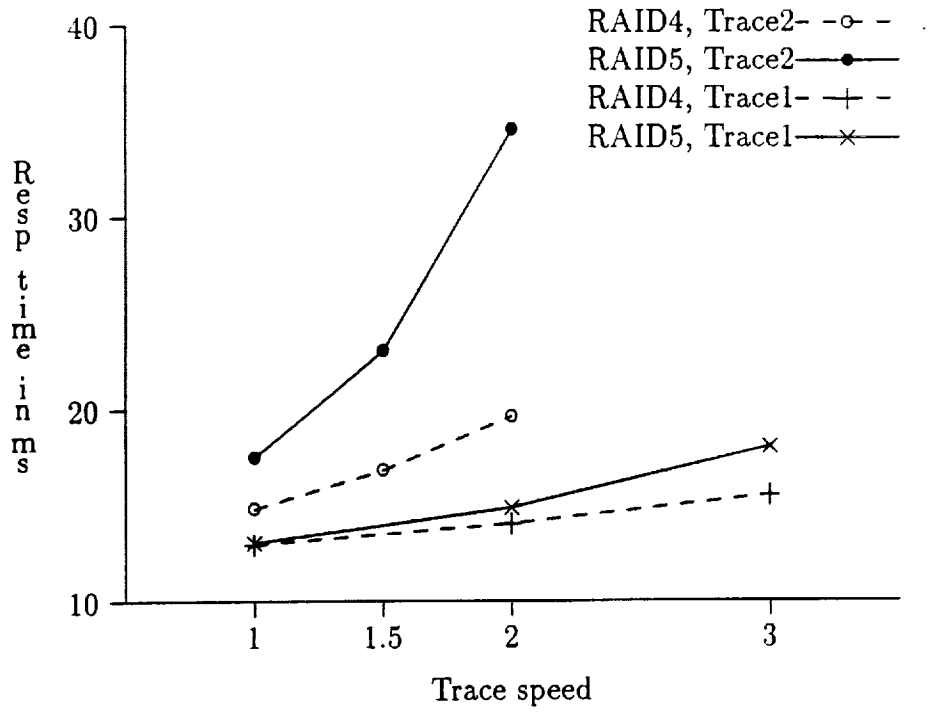


Figure 4.15 Response time vs. trace speed.

becomes large enough to fill most of the cache in which case writes have to wait for an empty slot to open in the cache for writing the parity. However these heavy load periods are rare and do not last very long; there are sufficient idle periods in the traces for the parity disk to catch up and empty its queue which is stored in the cache.

4.3.4.4 Striping unit

When disk utilization is high such as in the case of Trace 2, load balancing becomes an important issue, hence, a smaller striping unit is preferred. The response time of the two organizations is plotted in Figure 4.16 as a function of the striping unit. The shapes of the curves for Trace 1 and Trace 2 (RAID4 case) are predictable. The response time decreases at first as the striping unit increases because seek affinity is better exploited to

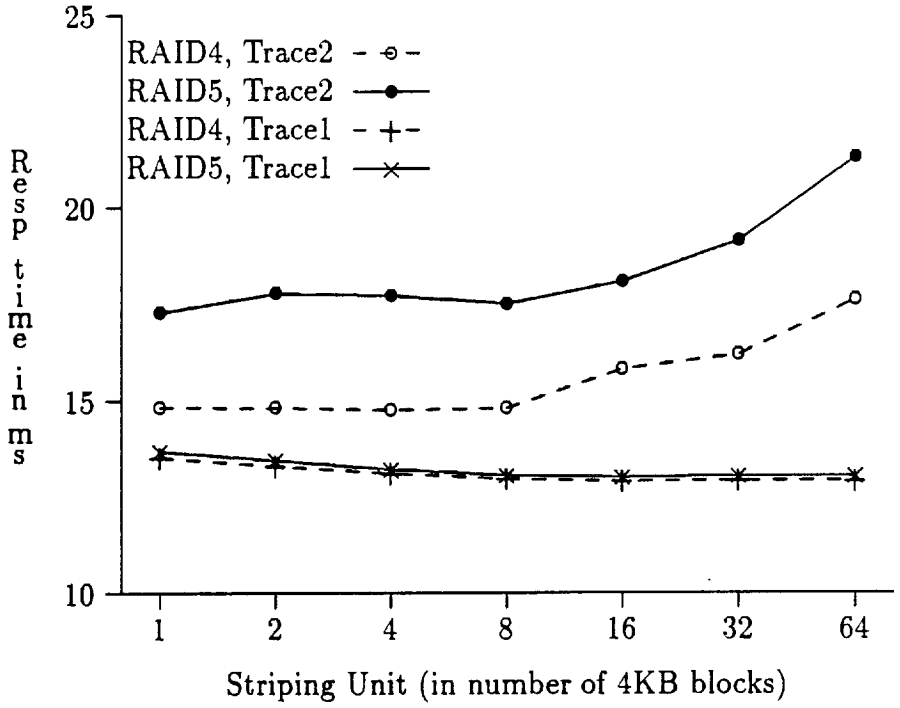


Figure 4.16 Response time vs. striping unit.

reduce average seek time. But as the striping unit becomes larger, the load becomes more unbalanced which causes long delays at some disks and increases the average response time. The optimal striping unit is lower for Trace 2 than for Trace 1 because of the higher disk utilization for Trace 2.

The shape of the first part (striping unit ≤ 4) of the curve for Trace 2 (RAID5 case) is not predictable and is probably due to the particular reference stream and block layout encountered in that trace which causes some heavily accessed blocks to land on the same disk(s) for the striping units 2 and 4.

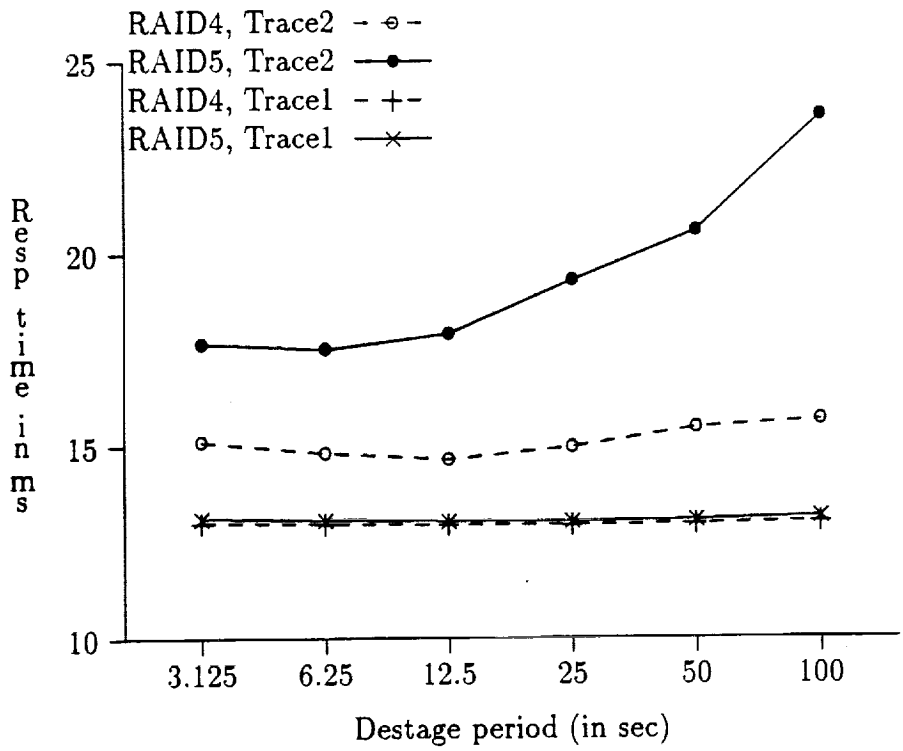


Figure 4.17 Response time vs. destage period.

4.3.4.5 Destage period

The longer a modified block stays in the cache the more likely it is for another update to the same block, to some consecutive block, or to some other block on the same track to occur. Hence a large destage period allows updates to the same track to accumulate and writes them in one single I/O. On the other hand, with a long destage period, modified blocks are more likely to reach the head of the LRU chain and cause a synchronous write due to replacement instead of being written asynchronously by the destage process. Figure 4.17 illustrates the effect of the destage period on the performance of RAID5. When the load is higher or the percentage of writes is higher, the cache has to be flushed more often. The optimal destage period is 25 sec for Trace 1, about 12 sec for Trace 2

(RAID4 case), and about 6 sec for Trace 2 (RAID5 case). The optimal destage period is shorter for Trace 2 than for Trace 1 because the percentage of writes is higher in Trace 2. It is shorter for RAID5 in the case of Trace 2, because the high cost of writes makes it even more advantageous to have more asynchronous destage I/O's than more synchronous write I/O's at replacement time.

4.4 Conclusions

We used traces from commercial transaction processing systems to evaluate the performance of two redundant disk array organizations and compare them to mirrored disks and to nonredundant nonarrayed systems (Base organization). The I/O workload is dominated by single block I/O's and contains a significant amount of skew in the distribution of accesses to disks. We evaluated both cached and noncached organizations. We found that RAID5 outperforms parity striping in all cases because of its load balancing capabilities.

In noncached organizations, RAID5 and parity striping may perform significantly worse than the Mirror and Base organizations because of the high cost of individual writes. For an organization with 10 data disks per array, RAID5's response time is 32% worse than for the Base organization for one of the traces. It was also found that, because of the large amount of skew in disk accesses found in the workload, large RAID5 arrays perform better than smaller arrays by balancing the load more evenly over the disks. By speeding up the trace, it was shown than RAID5 behaves better than the

other organizations under very high loads. For parity striping organizations, we found that placement of the parity area on the disk can affect performance significantly and we derived a simple rule for placing the parity in a way that minimizes seek times.

For cached organizations, we found that all organizations benefit from higher cache sizes. The write hit ratio is much higher than the read hit ratio and in one of the benchmarks is close to 1 for cache sizes over 32 MB. The read hit ratio on the other hand keeps increasing steadily as cache size increases. RAID5's performance is as good or better than for the Base organization's performance in cached organizations. A 16 MB cache practically eliminates the RAID5 write penalty. For one of the traces, RAID5 performance goes from 32% worse than the Base organization in the noncached case to only about 1% worse in the cached cache.

In our evaluation of cached organizations, we also studied a RAID4 organization that uses the controller cache to buffer parity updates before sending them to the parity disk. We found that RAID4 with parity caching generally performed better than RAID5 for array sizes of 10 or more. The improvement achieved is a function of the percentage of writes in the I/O workload and the load (arrival rate) at the disks. The load at the disks is a function of the cache size and the amount of locality in the workload. The improvement in response time varied from 1% for the first benchmark with a cache size of 16 MB per (10+1)-disk array to as much as 15% for the other benchmark with the same cache size. At smaller caches sizes, the number of I/O's going to the disks increases and so does the benefit of parity caching. We have studied the effect of array size on the performance of RAID4 with parity caching. We found that at higher array sizes,

RAID4's performance improves because a larger proportion of the disks can service read requests while the parity disk is able to keep up with the increased load. For a small array size ($N = 5$), RAID5 performs better than RAID4 with parity caching because it uses more disk arms to service reads. We have also experimented with speeding up the trace and found that parity caching can effectively remove the bottleneck on the parity disk in RAID4 even at high loads. For both benchmarks used, it was found that although the parity disk queue can grow at times to occupy most of the cache, this did not occur often and there were sufficient idle periods for the parity disk to catch up.

CHAPTER 5

CONCLUSIONS

Redundant disk arrays have the ability to use small form factor disks to replace single large expensive disks and provide better reliability. Transaction processing systems require high performance I/O subsystems and high levels of reliability. In addition, the I/O workload in transaction processing systems exhibits special characteristics. The workload typically consists of large numbers of essentially random small I/O requests. In this thesis, we investigated three issues related to the use of disk arrays in transaction processing systems: database recovery, distributed redundant disk arrays and performance of disk array organizations in transaction processing environments.

The first problem we addressed dealt with optimizing the recovery component in the database management system. We proposed a technique based on twin page storage for reducing the overhead of logging in transaction processing. The technique consists of implementing the twin page scheme for the parity data in the disk array so that the old parity pages can be used to perform UNDO recovery following a transaction abort or a system failure. To evaluate the performance of the proposed technique, we compared a transaction processing system using redundant disk arrays to a transaction processing system using redundant disk arrays and the twin page parity scheme. We analyzed the performance of both systems under two major transaction recovery paradigms: the

FORCE policy and the delayed write or $\neg$*FORCE* policy. We also examined both the case of page logging and record logging. We used analytical models as well as simulations using data from operational transaction processing systems. We found that the proposed twin page parity scheme typically improves transaction throughput by 10 to 40% depending on the recovery paradigm and the granularity of logging. The storage overhead required for the twin page parity scheme is on the order of 10% for systems using 10 disks per array.

In distributed transaction processing systems, recovery from site failures is a crucial issue. Site failures are of two types: temporary outages or permanent disasters. Disaster recovery is traditionally dealt with by maintaining a remote backup copy of the data of each site at another site or at some secure location. Temporary site failures can be dealt with by dispersing copies of essential files at multiple locations. Redundant arrays of distributed disks can be used to provide an efficient way to deal with temporary and permanent site failures as well as individual disk failures at a much lower storage cost than the remote backup scheme or the multicopy scheme. An important problem with using redundant arrays of distributed disks is optimizing the cost of remote accesses that are performed to update the parity information. We looked at methods for partitioning a large distributed storage system into redundant arrays of distributed disks in such a way that the cost of updating the remote parity is minimized. We developed a mathematical formulation for the site partitioning problem and showed that the problem is NP-hard. We proposed several heuristic algorithms for solving the site partitioning problem and

performed an experimental evaluation of the proposed heuristics. We also derived bounds on the deviation from the optimal solution for some of the heuristics.

Another issue addressed in this thesis is the performance of disk array organizations in transaction processing environments. We used I/O traces from operational transaction processing systems to compare the RAID5 and parity striping organizations to mirrored disks and nonredundant, nonstriped systems. We found that skew in the accesses to disks has a major effect on performance. Hence the RAID5 organization performed much better than parity striping. We also found that the write penalty can significantly affect performance even at high read-write ratios. Nonvolatile caches proved to be very effective at eliminating the write penalty. Larger caches were shown to be very beneficial because of the steady increase in hit ratio as cache size increases. Parity caching coupled with a RAID4 organization that dedicates one disk per array to parity was found to improve performance of large arrays especially for low read-write ratios and small cache sizes. In summary, cached disk arrays can provide equal or higher performance than nonarrayed systems while providing a high degree of availability at much lower storage costs than for mirrored disk solutions.

REFERENCES

[1] R. H. Katz, G. A. Gibson, and D. A. Patterson, "Disk system architectures for high performance computing," *Proceedings of the IEEE*, vol. 77, pp. 1842–1858, Dec. 1989.

[2] M. Y. Kim, "Synchronized disk interleaving," *IEEE Transactions on Computers*, vol. C-35, pp. 978–988, Nov. 1986.

[3] M. Livny, S. Khoshafian, and H. Boral, "Multi-disk management algorithms," in *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 69–77, May 1987.

[4] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceedings of the IEEE International Conference on Data Engineering*, pp. 336–342, Feb. 1986.

[5] A. Reddy and P. Banerjee, "An evaluation of multiple-disk I/O systems," *IEEE Transactions on Computers*, vol. 38, pp. 1680–1690, Dec. 1989.

[6] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, pp. 109–116, June 1988.

[7] D. Bitton and J. Gray, "Disk shadowing," in *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 331–338, Sept. 1988.

[8] J. Gray, B. Horst, and M. Walker, "Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput," in *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 148–161, Aug. 1990.

[9] P. M. Chen and D. A. Patterson, "Maximizing performance in a striped disk array," in *Proceedings of the International Symposium on Computer Architecture*, pp. 322–331, May 1990.

[10] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system R database manager," *ACM Computing Surveys*, vol. 13, no. 2, pp. 223–242, 1981.

[11] J. Kent and H. Garcia-Molina, "Optimizing shadow recovery algorithms," *IEEE Transactions on Software Engineering*, vol. 14, pp. 155–168, Feb. 1988.

[12] R. A. Lorie, "Physical integrity in a large segmented database," *ACM Transactions on Database Systems*, vol. 2, pp. 91–104, Mar. 1977.

[13] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, pp. 287–317, Dec. 1983.

[14] K.-L. Wu and W. K. Fuchs, "Rapid transaction-undo recovery using twin-page storage management," *IEEE Transactions on Software Engineering*, pp. 155–164, Feb. 1993.

[15] A. Reuter, "A fast transaction-oriented logging scheme for UNDO recovery," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 348–356, July 1980.

[16] A. Reuter, "Performance analysis of recovery techniques," *ACM Transactions on Database Systems*, vol. 9, pp. 526–559, Dec. 1984.

[17] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems*, vol. 9, pp. 560–595, Dec. 1984.

[18] Tandem Computers Inc., Cupertino, CA, *Introduction to the Transaction Monitoring Facility (TMF).* Part Number 15996.

[19] M. Stonebraker and G. A. Schloss, "Distributed RAID – a new multiple copy algorithm," in *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pp. 430–437, Feb. 1990.

[20] L.-F. Cabrera and D. D. E. Long, "Swift: Using distributed disk striping to provide high I/O data rates," *USENIX Computing Systems*, pp. 405–433, Fall 1991.

[21] A. L. N. Reddy and P. Banerjee, "Design, analysis, and simulation of I/O architectures for hypercube multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 140–151, Apr. 1990.

[22] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM*, vol. 36, pp. 335–348, Apr. 1989.

[23] L. W. Dowdy and D. V. Foster, "Comparative models of the file assignment problem," *ACM Computing Surveys*, vol. 14, pp. 287–313, June 1982.

[24] B. W. Wah, "File placement on distributed computer systems," *Computer*, pp. 23–32, Jan. 1984.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness.* New York: Freeman, 1979.

[26] M. R. Anderberg, *Cluster Analysis for Applications.* New York: Academic Press, 1973.

[27] S. Kirkpatrick, C. D. Gelatt, and M. P. Vechi, "Optimization by simulated annealing," *Science*, vol. 220, May 1983.

[28] F. Glover, "Tabu search methods in artificial intelligence and operations research," *ORSA Artificial Intelligence Newsletter*, vol. 1, 1987.

[29] D. J. Kleitman and D. B. West, "Spanning trees with many leaves," *SIAM Journal on Discrete Mathematics*, vol. 4, pp. 99–106, Feb. 1991.

[30] D. L. Burkes and R. K. Treiber, "Design approaches for real-time transaction processing remote site recovery," in *35th IEEE Compcon*, pp. 568–572, 1990.

[31] H. Garcia-Molina and C. A. Polyzois, "Issues in disaster recovery," in *35th IEEE Compcon*, pp. 573–577, 1990.

[32] J. Lyon, "Tandem's remote data facility," in *35th IEEE Compcon*, pp. 562–567, 1990.

[33] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* San Mateo, California: Morgan Kaufmann, 1993.

[34] A. Bhide and D. Dias, "RAID architectures for OLTP." IBM T.J. Watson Research Center Technical Report, 1992.

[35] P. M. Chen, G. A. Gibson, R. H. Katz, and D. A. Patterson, "An evaluation of redundant arrays of disks using an Amdahl 5890," in *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 74–85, May 1990.

[36] S. Chen and D. Towsley, "The design and evaluation of RAID 5 and parity striping disk array architectures," *Journal of Parallel and Distributed Computing*, pp. 58–74, Jan. 1993.

[37] J. M. Menon and R. L. Mattson, "Performance of disk arrays in transaction processing environments," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[38] A. L. N. Reddy, "A study of I/O system organizations," in *Proceedings of the International Symposium on Computer Architecture*, pp. 308–317, 1992.

[39] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt, "Disk subsystem load balancing: Disk striping vs. conventional data placement," in *Proceedings of the 1993 Hawaii International Conference on System Sciences*, pp. 40–49, Jan. 1993.

[40] J. O'Brien, "RAID 7 architecture features asynchronous data transfers," *Computer Technology Review*, Winter 1991.

[41] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging: Overcoming the small write problem in redundant disk arrays," in *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 64–75, May 1993.

[42] A. L. N. Reddy, "Reads and writes: When I/O's aren't quite the same," in *Proceedings of the Hawaii International Conference on System Science*, pp. 84–92, 1992.

VITA

Antoine Nagib Mourad was born in [redacted]. He received the Diplôme d'Ingénieur from Ecole Polytechnique, Paris, France, in 1986, the M.S. degree in Electrical and Computer Engineering from the University of Texas at Austin, in 1989, and the Diplôme d'Ingénieur from Ecole Nationale Supérieure des Télécommunications, Paris, France, in 1990. He is currently working towards the Ph.D. degree in Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His areas of interest include computer architecture, I/O subsystems, performance analysis, database systems and parallel processing.

Mr. Mourad is a member of the Association for Computing Machinery and the IEEE Computer Society. In 1989, he received the MCC Award for Excellence for research performed at the University of Texas.

Betsy Andrews

Sadun Anik

Jeffrey Baxter

David Bradley

Paul Chen & Melody Jeter

William Chen

Robert Dimpsey

Laura Drumm

Wuchun & Annette Feng

Kimberly Forsten

John Fu

David Gallagher

Kumar Goswami & Karyn Williamson

John Gyllenhaal

Grant Haab & Matt Hesson-McInnis

John Holm

Bob Janssens

Suzanne Kuo

Daniel Lavery

Scott Mahlke

Vicki McDaniel

Michael Peercy

Paul Ryan

Ann Saterbak

Dale Schouten

Jonathan Simonson

Krishna Subramanian

Tracy Tilton

Marta Verhoff

Joan Wilson