CONCURRENCY CONTROL AND RECOVERY IN MULTIPROCESSOR DATABASE MACHINES: DESIGN AND PERFORMANCE EVALUATION

by

Rakesh Agrawal

Computer Sciences Technical Report #510

September 1983

CONCURRENCY CONTROL AND RECOVERY IN MULTIPROCESSOR DATABASE MACHINES: DESIGN AND PERFORMANCE EVALUATION

by

RAKESH AGRAWAL

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1983

ABSTRACT

We present the design of mechanisms for concurrency control and recovery in multiprocessor database machines. We also analyze the relative performance of the various mechanisms and study their impact on database machine performance. While the multiprocessor-cache class of database machines has been the focus of our research, we also enumerate how our design can be adapted to the other classes of database machines.

We show that for concurrency control, a centralized 2-phase locking scheduler with deadlock detection is most appropriate. Amongst the recovery mechanisms, parallel logging is shown to have the overall best performance. With the architecture that we have proposed for logging, the recovery actions can be completely overlapped with the processing of data pages. Thus, the throughput of the database machine is not degraded by the recovery mechanism.

Some other important results emerged as a consequence of this research. We propose that the concurrency control and recovery are intimately related and describe the interaction between different concurrency control and recovery mechanisms. We also present the design of six integrated concurrency control and recovery mechanisms in the context of centralized database systems and the results of their performance evaluation.

We have extended the shadow and the differential-file recovery mechanisms for use in a multi-transaction environment. We also designed a linear deadlock detection algorithm. In addition, we have proposed a solution to the update

problem in hypothetical databases that permits the reinsertion of a previously deleted tuple while preserving the append-only nature of the differential relations.

We also present several parallel recovery mechanisms. Particularly interesting is the parallel logging algorithm that allows logging for a transaction to be performed asynchronously at more than one log disk, and yet does not require physical merging of distributed logs to recover from failures. We show how to take system checkpoints completely in parallel with the normal data processing and logging activities. Although designed in the context of database machines, these parallel recovery mechanisms may be easily adapted for use in any high performance database management system.

ACKNOWLEDGEMENTS

First and foremost, I wish to thank David DeWitt, my advisor; that without him this thesis never would have happened cannot be gainsaid. He provided more than generous support, arranged an excellent working environment, helped me formulate a number of my ideas, and translated this thesis into readable English. Randy Katz was very flexible and accommodative, whether it required sitting in my preliminary examination with jet-legs or taking my thesis to Berkeley for reading. Haran Boral squeezed in time to read this thesis in spite of his short visit from Technion. Jim Goodman made numerous insightful suggestions on the thesis. I also thank Andy Pleszkun, Doug Bates, and Dina Bitton Friedland for their comments. Tony Klug gave valuable advise in the early phases of this research. His unfortunate death in June was a personal loss.

The foundation for this research was laid in the initial part of my graduate study. I thank the excellent Computer Science faculty, in particular, Ray Bryant, Marvin Solomon, Charles Fischer, Raphael Finkel, Olvi Mangasarian, Bob Meyer, and Steve Robinson for all that I have learned from them.

Our stay in Madison was made pleasant and memorable by the company of many of our fine friends. The entire list is too long to include here, but I would like to mention Keith for his association in many projects, Kamesam for helping me prepare for the Math Programming qualifier, and Dinkar for all the music recordings.

Finally, I need to thank my family: my parents Dr. Ram Kumar and Mrs Urmila Agrawal who were the constant source of inspiration and encouragement;

my daughter Geetika who never quite understood why I spent so much time staring at the terminal and yet never insisted that I keep up with my last week's "next week positively" promises; and of course my wife Shalini who provided support, patience, and understanding.

This research was partially funded by the National Science Foundation under grant MCS82-01870.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. The Problem

A *database machine* is a collection of *specialized* hardware designed for supporting basic database management functions [Hsia79a]. The intent has been to use hardware to do *efficiently* various tasks of database management that are performed traditionally via software. During the past decade, a number of database machine designs have been proposed (see the surveys in Song81a, Bora81a, Hawt82a). However, most of these designs have been optimized only with respect to retrieval queries.

In reality, databases are continually *updated* by multiple transactions. In the presence of update operations, a major function of the database management software is to *synchronize* concurrent accesses to shared objects so that certain consistency assertions called *consistency constraints* are maintained and to *restore* the consistent state of such objects in case of failures. The first problem is referred to as the *concurrency control* problem and the second problem is called the *recovery* problem.

Database-machine designers have given no attention to the issues of concurrency control and recovery and their impact on the performance of the proposed machines. The study of recovery architectures for RAP-like associative processors in [Card81a] is the only work in this area.

Herein lies the *motivation* and the *objective* of this research: *to design and analyze the relative performance of concurrency control and recovery*

mechanisms for a multiprocessor database machine architecture and to study their impact on the performance of the database machine.

1.2. Multiprocessor-Cache Database Machine Architecture

The multiprocessor-cache database machine architecture (Figure 1.1) for which we have investigated concurrency control and recovery mechanisms consists of a set of general purpose processors, a multi-level memory hierarchy, and an interconnection device connecting the processors with the multi-level memory hierarchy.

Some of the processors, designated query processors (QPs), execute user queries and operate asynchronously with respect to each other. One of the processors, designated the back-end controller (BEC), acts as an interface to the host processor (the processor with which a user interacts) and coordinates the activities of the other processors. After a user submits a query for execution, the host compiles the query and sends it to the back-end controller for execution on the database machine.

We assume that the memory hierarchy consists of three components. The top level consists of the internal memories of the query processors. Each processor's local memory is assumed to be large enough to hold both a compiled query and several data pages. Mass storage devices (disks) make up the bottom level and the middle level is a disk cache that is addressable by pages.

The bottom two levels of the memory hierarchy are connected in a way that allows for data transfers between each mass storage device and any page frame in the disk cache. A processor, designated the I/O processor, is responsible for transferring pages between the mass storage devices and the disk cache. The I/O processor is logically a part of the back-end controller. The top two levels of the hierarchy are so connected that each processor can read or write a different

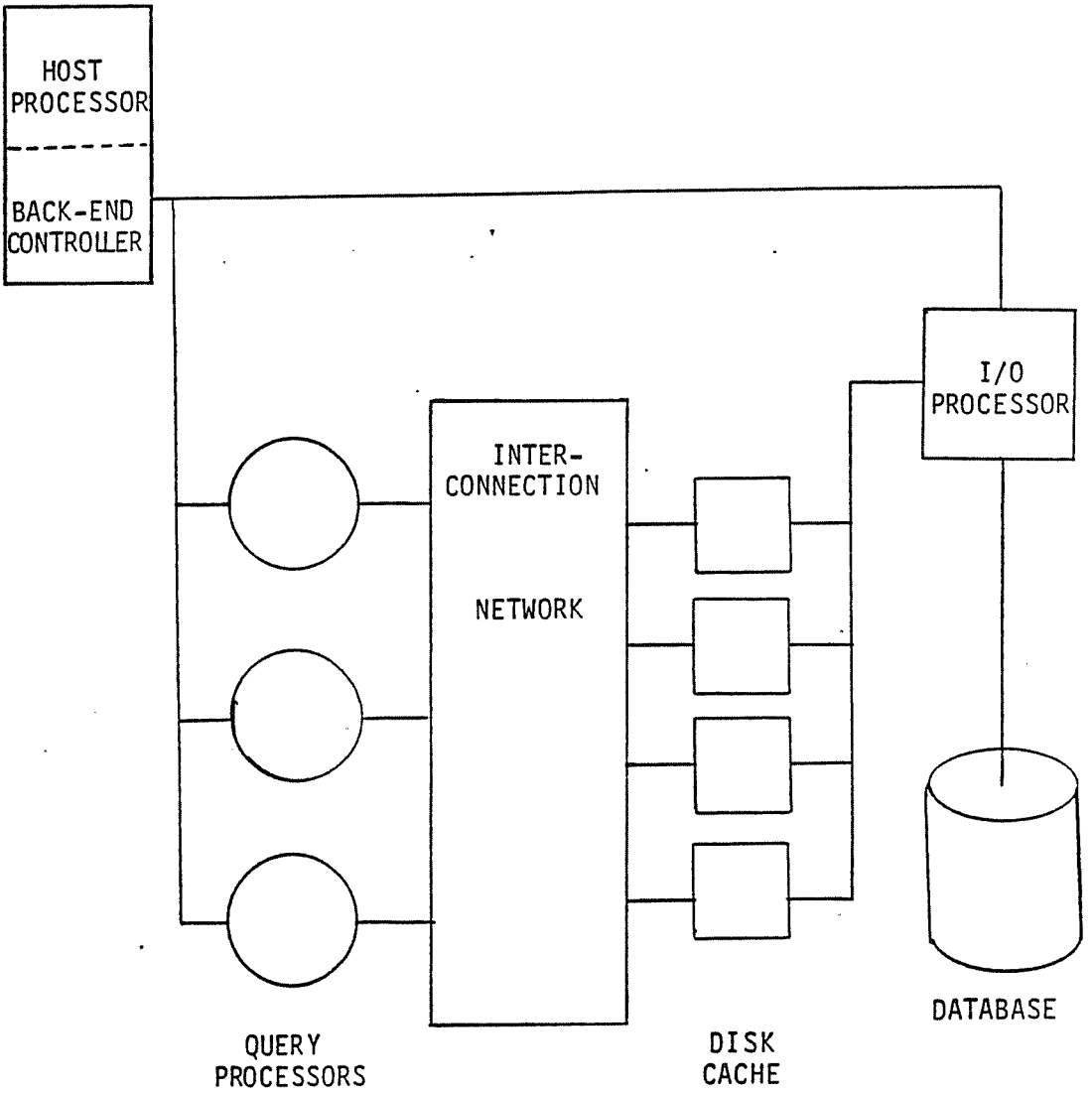

Figure 1.1 Multiprocessor Cache Architecture

page of the disk cache simultaneously and all processors can simultaneously read the same page of the cache.

This architecture captures the essence of the designs of the multiprocessor cache database machines like DIRECT [DeWi79a], INFOPLEX [Madn79a], RDBM [Hell81a] and the database machine project at Texas that utilizes the TRAC [Upch79a] processor. See [Bora80a] for query execution algorithms on such an architecture.

1.3. Other Architectures

DeWitt and Hawthorn [DeWi81a] have divided database machine architectures into five generic classes[1]:

CS - conventional systems

PPD - processor-per-disk systems

PPT - processor-per-track systems

PPH - processor-per-head systems

MPC - multiprocessor cache systems

Although all database machines may not directly fit into this classification, it has been argued in [DeWi81a] that most of the database machines may be represented as some combination of these architectures.

While the multiprocessor cache architecture of the database machines, as presented in Section 1.2, will be the focus of this dissertation, at the appropriate points we will describe how our design of concurrency control and recovery mechanisms may be adapted to the other architectures. In this section, we will present an overview of the other database machine architectures.

---

[1] Those architectures that are not feasible using presently available technology and those that may never become cost effective were not included in DeWitt-Hawthorn classification.

Conventional Systems (CS)

The first class of "database machines" is a database management system running on a single processor (Figure 1.2). The operating system on the processor is tuned to the needs of the database management software [Gray78a, Ston81a] and the CS uses sophisticated query execution strategies such as those employed in System R [Seli79a]. The IDM 500 from Britton-Lee Inc. [IDM83a] is an example of this class of database machines. The results that we will present in Chapter 2 will directly apply to this class of database machines.

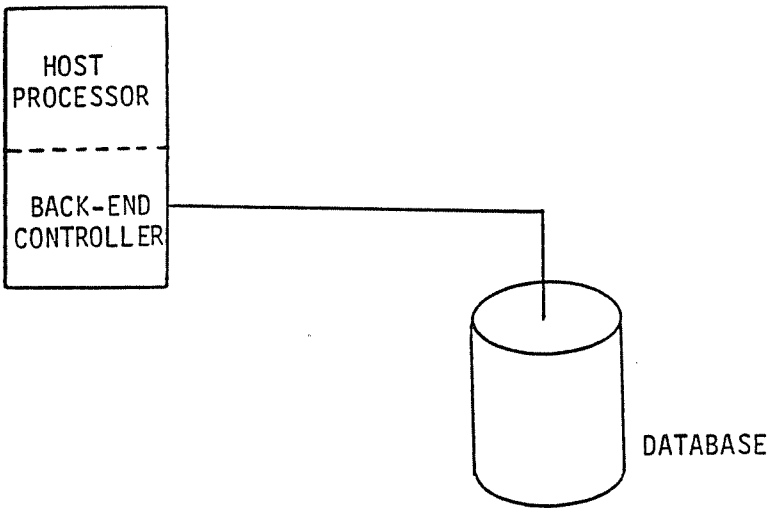

Figure 1.2 Conventional System

Processor-per-disk (PPD)

A search processor (or a set of processors [Leil78a]) is placed between the secondary storage devices and the primary memory (Figure 1.3) that filters out irrelevant data. Examples include CAFS [Babb79a] and the designs in [Lang77a], [Leil78a] and [Banc80a].

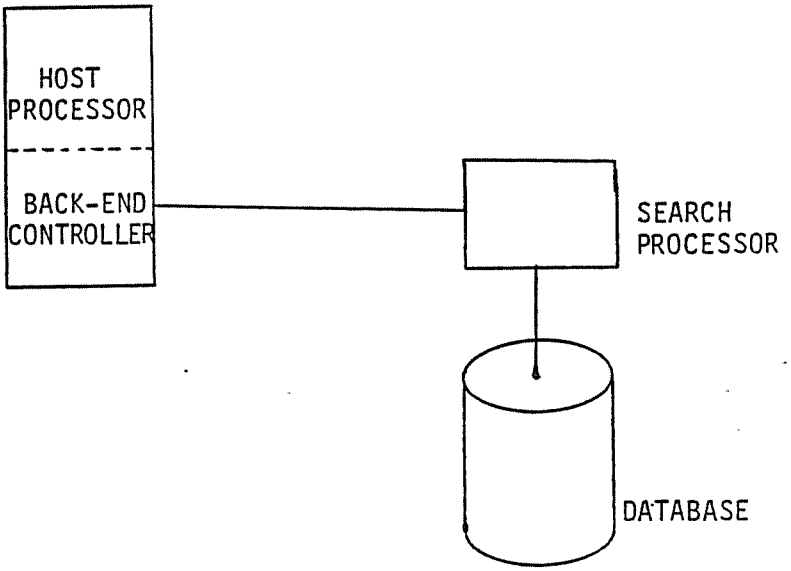

Figure 1.3 PPD System

Processor-per-track (PPT)

The mass storage device consists of cells. A cell can be a disk track, bubble memory, or charge-coupled device. A search processor is associated with each cell (Figure 1.4). Examples include RAP [Özka75a], CASSM [Su75a], RARES [Lin76a], and the early PPT designs in [Slot70a], [Park71a], [Mins72a], and [Parh72a].

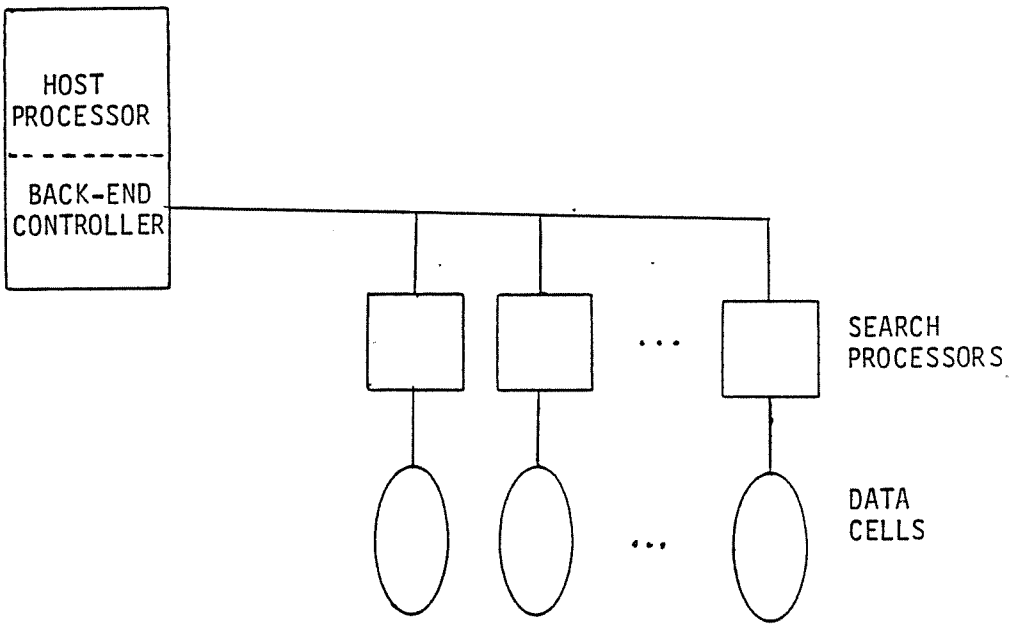

Figure 1.4 PPT System

Processor-per-head (PPH)

A PPH machine differs from PPT machine in that the mass storage cells are clustered into equal sized partitions and there is a search processor for each cell in a partition instead of a processor for every cell (Figure 1.5). The processors are dynamically shared by the partitions but at any time, all the processors are connected to cells in the same partition. An example of a PPH machine is DBC [Bane'78a] wherein each track of a disk is considered a cell and each cylinder, a partition.

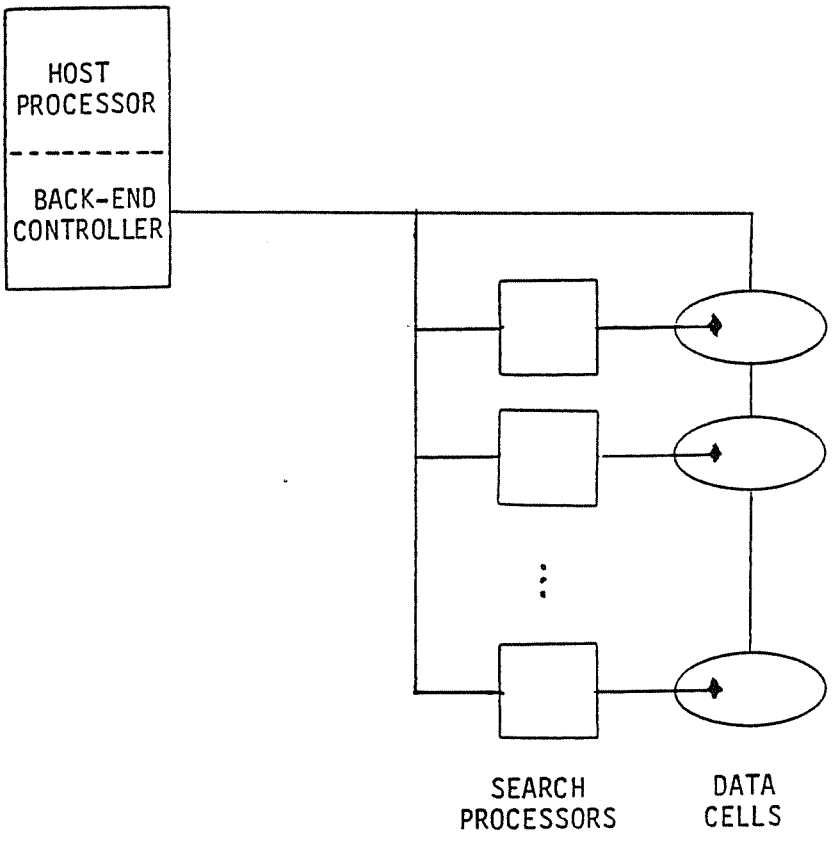

Figure 1.5 PPH System

1.4. Organization of the Dissertation

The organization of the rest of the dissertation is as follows.

As the first step in our research, we designed integrated concurrency control and recovery mechanisms for centralized database systems and evaluated their performance. The intent was that the results from this study would guide us in the design of concurrency control and recovery mechanisms for database machines. In spite of the numerous concurrency control and recovery mechanisms that have been proposed during the past decade, the behavior and the performance of various concurrency control and recovery mechanisms are still not well understood. In addition, although concurrency control and recovery mechanisms are intimately related, in the past they have been treated primarily as two independent problems and very little research has been devoted to explore the interaction between the two mechanisms. In our design, we took a unified view of the problems associated with concurrency control and recovery and developed integrated mechanisms. Our cost model for evaluating the performance of these mechanisms incorporates both the effect of the mechanism on the conflict rate between transactions and the overhead associated with each mechanism on the execution of the transaction. Our performance evaluation methodology isolates and quantifies the costs of various components of a mechanism. In Chapter 2, we present this design and the results of performance evaluation of integrated concurrency control and recovery mechanisms for centralized database systems.

We describe the design of concurrency control and recovery mechanisms for the multiprocessor database machines in Chapter 3. We show that for concurrency control, a centralized 2-phase locking scheduler with deadlock detection is most appropriate. We also present parallel recovery mechanisms.

We present a parallel logging mechanism that allows logging for a transaction to be performed asynchronously at more than one log disk, and yet does not require physical merging of distributed logs to recover from failures. We also show how to take system checkpoints completely in parallel with the normal data processing and logging activities. We also propose several approaches to parallelize the shadow mechanism. Finally, we present parallel algorithms for operations on differential files. We also describe in this chapter how our design can be adapted in other database machine architectures.

Chapter 4 contains the results of our experiments to evaluate the relative performance of the parallel recovery mechanisms and their impact on database machine performance. We first explain why we used simulation instead of queue theoretic modeling for our performance evaluation. We then describe the simulator for the bare (i.e., without recovery) database machine. The results of the performance analysis when the bare machine simulator is augmented with parallel logging, shadow, and differential file recovery mechanisms respectively are presented next. Finally, we present a comparison of the recovery mechanisms.

The conclusions and the directions for future research are presented in Chapter 5.

CHAPTER 2

CONCURRENCY CONTROL AND RECOVERY IN CENTRALIZED DATABASES

2.1. Introduction

During the past decade, alternative concurrency control and recovery mechanisms have been the subject of intensive research activity [Bern81a, Bern82a, Kohl81a, Verh78a]. However, in spite of the wide variety of mechanisms proposed, there remains a lack of experimental and/or analytical evidence regarding the behavior of various concurrency control and recovery mechanisms and their influence on database system performance. In addition, although concurrency control and recovery mechanisms are intimately related, they have been treated primarily as two independent problems and very little research has been devoted to explore the interaction between the two mechanisms.

In this thesis, we take a *unified* view of the problems associated with concurrency control and recovery for *centralized* database management systems, and present several integrated mechanisms. We then develop analytical models to study the behavior and compare the performance of these integrated mechanisms. The intent had been that the results from this study would guide us in the design of concurrency control and recovery mechanisms for database machines.

The organization of the rest of this chapter is as follows. In Section 2, we present a review of the related work. Summaries of the concurrency control and recovery mechanisms that we evaluated are contained in Section 3 and

Section 4 respectively. We present the interaction between concurrency control and recovery in Section 5. We describe the cost model that we use for performance evaluation in Section 6. Integrated recovery and concurrency control mechanisms are presented in Section 7 along with the cost equations for each. In Section 9, we present the results of our performance evaluation using the database, mass storage device, and processor characteristics specified in Section 8. Section 10 contains our conclusions from this study of concurrency control and recovery in centralized database systems.

2.2. Summary of Related Research

As far as concurrency control is concerned, some researchers have investigated the behavior of locking using both simulation and analytical models. Through the use of simulation, in [Spit76a] the difference in performance between a system in which locks are released as soon after the shrink point [Eswa76a] as possible and a system in which locks are held until the transaction completes was found to be insignificant. Alternate methods of choosing a victim for deadlock resolution were studied in [Munz77a]. The effects of locking granularity on database performance were examined in [Ries79a] and it was demonstrated that different settings for system and application parameters may favor different locking granularities. In [Lin82a] the probability that a lock request by a transaction either results in a deadlock or causes the transaction to wait (along with the expected average waiting time) are estimated.

Locking policies were analyzed in [Poti80a] using hierarchical analytical modeling and two queueing network models have been proposed in [Iran79a] to study the effect of locking granularity on database system performance. An analysis of the probability of waiting and deadlock has been presented

in [Gray81a].

In [Garc79a] the performance of two concurrency control algorithms [Alsb76a, Thom78a] for distributed database systems is compared using simulation. In [Lin81a] another two algorithms [Bern77a, Lin79a] for distributed database systems have been compared. Concurrent with this thesis, the performance of locking has been compared with the performance of basic timestamp ordering in [Gall82a], and with basic and multiversion timestamp ordering in [Lin83a]. Results of some experiments comparing locking to the optimistic method of concurrency control have appeared in [Robi82a]. Simulations to compare the performance of several concurrency control algorithms are in progress in [Care83a]. A criticism of these simulation studies is that they only generate one final number for comparison and do not help in isolating the costs of various components of a mechanism. With the approach that we have developed, in addition to saying that a particular mechanism is expensive, one can determine *why* the mechanism is expensive and *where* efforts should be concentrated to improve its performance.

Analytical models have been utilized to study log-based recovery systems in [Chan75a, Gele78a, Gele79a]. These models, however, only address the issue of selecting an optimum checkpoint interval. An assessment of shadows vis-a-vis logs for recovery has been given in [Gray81b].

The results that we will present in Section 9 are built upon several of these earlier efforts, in particular, those of Lin and Nolte [Lin82a]. Lin and Nolte have determined, through simulation, the probability of an access request by a transaction conflicting with another request as a function of a variety of parameters including the transaction size (the number of pages touched), the number of transactions running concurrently (i.e. the multiprogramming level),

the size of the database, and the access pattern of the transaction. They have also determined for two-phase locking the probability of a lock request resulting in deadlock and the average waiting time of a blocked request as a function of these same parameters. These results are used as inputs to our performance evaluation model. We also use the results in [Gray81a] to estimate the probability of waiting and probability of deadlock for small transactions since this range of transaction sizes is not covered by the results presented in [Lin82a].

2.3. Summary of Concurrency Control Mechanisms Evaluated

We have considered three basic approaches[1] that we feel form the basis of most concurrency control algorithms:

- Locking
- Timestamp ordering
- Validation

2.3.1. Locking

Locking synchronizes read and write operations by denying access to a certain portion of the database to a conflicting transaction. Before accessing an object, a transaction is required to own a *non-conflicting* lock on the object. Two requests for a lock on an object conflict if (a) one is for a read lock and the other for a write lock, or (b) both are for write locks.

Eswaran et al. [Eswa76a] have shown that for serializability[2], transactions should obtain locks in a *two-phase* manner. A transaction is said to be two-phase if it does not perform a lock action after the first unlock action. To avoid a

[1] The interested reader is encouraged to examine [Bern81a, Bern82a] for a complete exposition of all the approaches possible.

[2] serializability is the sufficient condition for consistency [Eswa76a, Bern79a, Papa79a].

cascade of backups if a transaction fails, it is required that the second phase be deferred to the transaction commit point [Gray80a].

2.3.1.1. Deadlock

Whenever a transaction waits for a lock request to be granted, it runs the risk of waiting forever in a deadlock. *Deadlock* has been shown to be equivalent to a *cycle* in a waits-for graph [Coff71a, Holt72a]. There are three approaches to deadlock resolution: prevention, detection and avoidance.

Deadlock *prevention* is a cautious scheme that does not let a transaction wait if it *may* get into a deadlock. Timestamp-based preemptive *wound-wait* and non-preemptive *wait-die* schemes proposed in [Rose78a] are examples of deadlock prevention. In deadlock *detection*, deadlocks are detected by explicitly building the waits-for graph and examining it for cycles. Deadlock *avoidance* is a conservative technique that avoids transaction restarts altogether using hierarchal allocation [Hans73a].

2.3.2. Timestamp Ordering

In locking, the ordering of transactions in a serialization order is dynamically determined while transactions are executing based on interleaving of their requests. With timestamp ordering, a serialization order is selected *a priori* and transaction execution is forced to obey this order. We will describe a *basic* implementation of timestamp ordering as presented in [Bern81a].

For each object X, the largest timestamp of any read(X) and the write(X) is recorded. Let these be R-ts(X) and W-ts(X) respectively. First consider rw-synchronization. A read(X) with timestamp TS is denied if TS < W-ts(X); otherwise, the read is permitted and R-ts(X) is set to max {R-ts(X),TS}. For a write(X) with timestamp TS, the request is rejected if TS < R-ts(X); otherwise,

the write proceeds and W-ts(X) is set to max{W-ts(X),TS}. For ww-synchronization, a write(X) with timestamp TS is rejected if TS < W-ts(X); otherwise, the write is allowed and W-ts(X) is set to TS. If a read or a write request of a transaction is denied, it is aborted and restarted with a new, and larger, timestamp.

Two variations of the basic algorithm: *multiversion* and *conservative* timestamp ordering have been described in [Bern81a]. Both attempt to reduce the number of restarts induced by the basic algorithm.

2.3.3. Validation

Unlike the locking or the timestamp ordering approach, algorithms based on validation allow a transaction to execute unhindered to its end. At the time of commit, the transaction is *validated* to determine whether or not to commit the transaction. The rationale for the validation approach is the *optimistic* assumption that only a few transactions conflict.

Kung and Robinson [Kung81a] have developed a timestamp-based approach to validation. As a transaction executes, information about the set of objects read, written and created by the transaction is collected and at the end, the transaction is validated using one of the three validation conditions. We will, henceforth, use the term "optimistic" instead of more general term "validation" to emphasize that we are specifically considering Kung and Robinson's optimistic method in the integrated mechanisms presented below.

2.3.4. Basic Timestamp Ordering versus Locking

For *centralized* databases and database systems, the basic timestamp ordering algorithm is very similar to locking in its behavior but has the disadvantage of inducing a larger number of restarts.

In basic timestamp ordering, the serialization order is decided a priori, whereas the serialization order is dynamically decided in locking. Because of this, when compared to locking, basic timestamp ordering is more prone to transaction restarts. Assume, for example, that ts(T2) > ts(T1) and the following sequence of operations:

T2 : read(X)
T2 : commit
T1 : write(X)

Basic timestamp mechanism will abort T1 but locking will permit both T1 and T2 to commit. Gray [Gray79a] has observed that the transaction restarts are very expensive.

We will now investigate the similarity in basic timestamp ordering and locking mechanisms. With basic timestamp ordering, a transaction's read(X) [write(X)] is translated into 3 actions: (i) checking that the timestamp associated with the access request is not less than W-ts(X) [R-ts(X)], (ii) updating R-ts(X) [W-ts(X)], and (iii) executing read(X) [write(X)]. It is necessary that these three actions are executed in an atomic fashion. Consider, for example, the consistency assertion that X=Y, assume R-ts(X) = W-ts(X) = R-ts(Y) = W-ts(Y) = 0, ts(T1) = 1, ts(T2) = 2, and the following sequence of execution:

1. T1 : read(X)
2. T1 : write(X:=X+1)
3. T2 : read(X)
4. T2 : write(X:=2*X)
5. T1 : read(Y)
6. T1 : write(Y:=Y+1)
7. T2 : read(Y)
8. T2 : write(Y:=2*Y)

Assume serial execution up to step 5. At step 6, ts(T1) is checked to be greater than R-ts(Y), write(Y) is accepted, and W-ts(Y) is set equal to 1. However, before Y is updated, processing of read(Y) at step 7 begins. Since ts(T2) > W-ts(Y), the read is accepted, R-ts(Y) is updated, and read(Y) is carried out. Subsequently, the pending write(Y) of step 6 is completed. After execution of step 8, we will have an inconsistent database. Therefore, as in the case of locking[3], while an object is being accessed, other conflicting (read and write) accesses to the object must be blocked[4].

Furthermore, if an updated object is allowed to be accessed before the transaction that updated it completes, the problem of *triggered aborts* will occur. Assume, for example, that ts(T2) > ts(T1) and the following sequence of execution:

T1 : write(X)
T2 : read(X)
T2 : commit
T1 : abort

When T1 is aborted, T2 will also have to be aborted and any updates of T2 will have to be undone. Consequently, once a transaction begins updating an object, access to that object must be blocked until the transaction either commits or

---

[3] However, once the transaction has finished reading an object, writes to that object may be allowed to proceed unlike the two-phase locking where the read locks must be kept until the shrink point.

[4] An alternative might be to recheck after executing read(X) [write(X)] that the timestamp associated with the request is still not less than W-ts(X) [R-ts(X)] and if the test fails, abort the transaction. This solution will further increase the number of restarts induced by the basic timestamp ordering.

aborts. This is equivalent to putting a write lock on the object and keeping that lock set until the end of the transaction (as in the case of two-phase locking).

Finally, with locking, the entries for a transaction in the lock table may be deleted as soon as the transaction completes. With basic timestamp ordering, however, timestamps corresponding to a transaction may have to be maintained even after the transaction has committed. Thus, the size of the timestamp table will be, in general, larger than the size of the lock table. Hence, granting an access request and adding and removing the timestamps with the basic timestamp ordering will not be less expensive than acquiring and releasing the locks.

To summarize, the only situation where timestamp ordering may offer additional concurrency over locking is the one in which a write request on an object is allowed to proceed once another transaction has *finished* reading the object. However, with timestamp ordering, a larger percentage of transactions will have to be aborted and rerun. The result is likely to be less net concurrency. In view of the above arguments, we will not consider basic timestamp ordering further.

2.4. Summary of Recovery Mechanisms Evaluated

We considered four basic recovery mechanisms for transaction oriented database systems[5]:

- Log
- Shadows
- Differential Files
- Versions

[5] For a different classification and some of the techniques that are not directly applicable to transaction-oriented database systems, see [Verh78a].

2.4.1. Recovery using Logs

The log-based approach [Gray78a] relies upon a *redundant* representation of the database on an append-only *log*. In addition to updating a data object, every update operation also creates a log record that includes information such as the transaction identifier, the object identifier, and "before" and "after" values. The log records of a transaction are threaded together. To limit the amount of work at the time of system restart, *system checkpoints* are taken periodically in an action-consistent state. At system checkpoint, buffers are flushed and a checkpoint record containing a list of all active transactions and pointers to their most recent log records is written to the log.

2.4.1.1. Commit Processing

Modification of the database follows the following *write-ahead-log* protocol:

(1) Before recording uncommitted updates of a transaction on stable storage, force its before-value log records to stable storage.
(2) Before committing updates of a transaction, force all its log records to stable storage.

2.4.1.2. Recovery Algorithm

The essential idea is to undo the effects of uncommitted transactions by reading log records for the transaction backwards and restoring the before-values. Similarly, the actions of committed transactions are redone by scanning log records for the transaction forward from the most recent checkpoint and reapplying the after-values.

2.4.2. Recovery using Shadows

The fundamental idea of shadows is not to do in-place updating but rather to keep two copies of the object being updated while the transaction is still

active: the modified copy and a copy of the object as it was before the transaction began. This latter is termed the *shadow* copy. When the transaction commits, the shadow copy is replaced by the updated copy. We will present a scheme based on the ideas in [Lori77a, Lamp79a].

For each relation, there is a shadow page-table, S-Map, that is maintained in stable storage. An *incremental* current page-table, C-Map, for each transaction is formed in the main memory as the transaction updates data pages. To update a page k, if k is already in C-Map then C-Map[k].PhysicalPage is used for updating. Otherwise, a free page j is obtained for the updated copy of k and an entry is added to C-map for k with C-Map[k].PhysicalPage = j.

2.4.2.1. Commit processing

At commit time, all the pages updated by a transaction are forced to the stable storage. Then, for all pages k that appear in a transaction's C-map, S-Map[k].PhysicalPage must be changed to C-Map[k].PhysicalPage. Since the system may fail when S-Map has been partially updated, S-Map is updated in two phases. First, C-map is written to a *commit list* on stable storage as transaction's *precommit* record. Once the precommit record of a transaction appears on the commit list, its effects cannot be undone. Next, S-Map is updated. Since system failure in the middle of writing of an S-Map block may garbage the block, the S-Map is updated *carefully*[6]. Finally, a *commit* record for the transaction is written to the commit list.

---

[6] As explained in [Lamp79a], careful updating requires two physical writes for each write operation.

2.4.2.2. Recovery Algorithm

Recovery from a transaction abort is straight-forward. First, the C-map associated with the transaction is discarded. Next, the updated data pages are reclaimed. To recover from a system crash, the commit list is examined to determine those transactions for which a precommit record appears in the list but not the commit record. For all such transactions, S-Map is updated using the precommit record.

2.4.3. Recovery using Differential Files

With the differential file scheme proposed in [Seve76a], all logical files comprise of two physical files: a read-only *base* file and a read-write *differential* file. The base file remains unchanged until reorganization. All updates are confined to the differential file.

2.4.3.1. Hypothetical Data Bases

In [Ston80a] the notion of Hypothetical Data Bases (HDB's) was introduced, and in [Ston81a] it was proposed that all databases (including the real ones) be treated as hypothetical. Each relation $R = (B \cup A) - D$ is considered a view [Ston75a] where B is the read-only base portion of R and A and D are append-only differential relations. Intuitively, additions to R go to A and deletions go to D. Operations on R are translated into operations on B, A and D.

2.4.3.2. Commit Processing

Assume that each transaction has been assigned a *unique* timestamp and that the tuples in the A and D files have been widened to have an extra field TS for such a timestamp. While a transaction is active, its updates go to its *local* A_l and D_l relations that are inaccessible to other transactions. When the transaction commits, A_l and D_l are appended to the *global* A_g and D_g relations

and are forced to stable storage. Finally, the timestamp of the committing transaction is written to a *CommitList* relation.

2.4.3.3. Recovery Algorithm

If a transaction aborts, its A_l and D_l are simply discarded and its timestamp is not appended to the CommitList relation. To recover from system crash, instead of R, start using the following view:

Range of (b,a,d,x) is (B_g,A_g,d_g,CommitList)

Define View R-Crash ([(b.all) ∪
(a.all) Where s.TS = x.TS] - [(d.all) Where d.TS = x.TS])

2.4.4. Recovery using Versions

In the version-oriented approach [Reed78a, Svob81a], an object is thought of as a sequence of unchangeable *versions* that are linked together through an *object header* to form a *history* of the object. Updating an object is considered as creating a new version, while reading an object is considered as selecting the proper version[7].

A *version* is a pair consisting of *value* and *time* attributes; the time attribute specifies its *range of validity*. The *start time* of a version is the time specified in the write request that created the version. The *end time* is initially the same as the start time, but it is extended by both read and write operations to the time specified in the request. When a new version gets created, the end time of the preceding version is frozen. To make versions immutable, only the start time is stored with versions. The end time of only the current version is kept in the associated object header which is mutable.

---

[7] By following the chain emanating from the object header.

2.4.4.1. Commit Processing

When an object is updated, first a tentative version called a *token* is created. All the tokens created by a transaction have embedded in them a reference to a *commit record* that contains the state of the transaction. Initially, the state is set to *unknown*. Eventually, it is changed to either *commit* or *abort* and it implicitly commits or discards all the tokens. Before committing a transaction, it is ensured that all the tokens created by the transaction have been forced to stable storage.

It is important to note that the object headers are updated in-place twice for each update of the object (in create-token and commit/discard token), and may have to be updated when the current version is read (to extend the end time). Updating object headers safely will require *careful writing* [Lamp79a] which is quite expensive as each write generates two physical writes. Therefore, object headers are designed to be only *hints:* they are not required to survive system crashes.

2.4.4.2. Recovery Algorithm

To recover from a transaction failure, the state field of the associated commit record is set to abort. Token pointers in the object header of all the objects updated by this transaction are also set to nil.

For recovery from system crash, the state of commit records is examined. If a commit record is found that has its state set to unknown, the state is changed to abort and the corresponding transaction is rerun. Since object headers are not assumed to survive system failure, a major part of recovery is the reconstruction of object headers. Object headers are reconstructed from the latest committed version of the object which is found by a sequential backward scan of version-storage.

2.4.5. Shadows vs. Versions

The version approach is, in a certain sense, a "super shadow" mechanism. No doubt, versions offer more functionality[8], and, when coupled with multiversion timestamp ordering, may have the potential of allowing more transactions to run concurrently[9]. Unfortunately, they have a severe performance penalty. The major problem is that simply reading the current version of an object may cause the corresponding object header to be updated. Thus, as compared to shadows, every read operation potentially requires one more disk access [Gray81c]. Because of their expected poor performance, we will not consider versions further.

2.5. Interaction of Concurrency Control and Recovery

With this background, we developed the following integrated concurrency control and recovery mechanisms: log+locking, log+optimistic, shadow+locking, shadow+optimistic, differential+locking, differential+optimistic. We will first present the interaction of the concurrency control and recovery mechanisms.

2.5.1. Sharing of Data Structures

With the optimistic method of concurrency control, while a transaction is active, all the updates are made to the local copies of each data object. Only after the transaction completes, will the updates be made globally available. Thus, redundant data structures are required for holding local copies during the active life of the transaction for concurrency control purposes. However, instead of creating separate data structures, those data structures required for

---

[8] It is possible to go back in time and answer questions such as "who did what when".

[9] Recently, in [Lin83a] it was found that the multiversion timestamp ordering performed only marginally better than the basic timestamp ordering.

recovery may be shared between concurrency control and recovery.

In the log+optimistic combination, the log records required for recovery may double as the local copies for concurrency control. In the shadow+optimistic combination, the incremental current page-table together with the new disk pages required for recovery may function as the local copies for concurrency control. In the differential+optimistic combination, local A and D pages that are private to a transaction may be used as the local copies for concurrency control.

2.5.2. Update of Data Pages

In any real system that has finite memory and finite I/O bandwidth, the way a mechanism handles updating of data pages on disk has crucial performance implication. If an updated page has to be paged out to disk while the transaction is still active, then there are two approaches.

1. *Immediate Updating:* Write the updated page to the disk block where it belongs. The advantage is that if the transaction completes, there is nothing more to be done as the updated page is where it should be. The disadvantage is that if the transaction aborts, then the image of the data page as it existed on disk before the transaction started will have to be restored.

2. *Deferred Updating:* Write the updated page in some scratch space and, after the transaction completes, move it to its original position. The advantage is that the transaction aborts can easily be handled by simply discarding the copy in the scratch area. The disadvantage is that successful transactions will incur two extra disk I/Os for each updated page: one to read the page from the scratch area and a second to write it in its proper location.

Amongst concurrency control mechanisms, locking can handle immediate updating but optimistic methods require deferred updating. Amongst recovery mechanisms, logging can support immediate updating but both shadow and differential file mechanisms require deferred updating. We will now investigate the integrated mechanisms from this point of view.

In log+locking, both the concurrency control and recovery mechanisms permit immediate updating and augment each other. An uncommitted update that migrates to disk cannot be seen by other transactions as it has locks put on it. If the transaction aborts, the uncommitted update can be undone by restoring the before value from the log.

In the log+optimistic combination, the recovery mechanism allows immediate updating but the concurrency control requires deferred updating. Thus, concurrency control and recovery interact adversely and this mechanism pays a large performance penalty for this adverse interaction. Recall that in the log+optimistic combination, log records double as local copies for concurrency control. Thus, at the time of making local copies global, all those log pages that have been flushed to disk due to buffer size constraints will have to be *reread*. Worse still, all those data pages that are to be updated and which could not be held in the memory due to buffer limitation will also have to be *reread*. Thus, this mechanism may involve considerable rereading. A positive byproduct of deferring updates to transaction completion, however, is that only an after-value log is required as a before-value log is used only for transaction undo.

Whereas in the log+optimistic combination the concurrency control inviolates the immediate updating feature of the recovery, in shadow+locking, it is the recovery mechanism that inviolates the immediate updating feature of the concurrency control mechanism. The result is that when the transaction

completes, those page-table pages that are to be updated and that are no long available in the memory will have to be *reread*. However, unlike the log+optimistic combination, this mechanism does not require rereading of data pages to be updated as the page-table is updated to point to new disk locations. In shadow+optimistic combination, both concurrency control and recovery require deferred updating. Therefore, like shadow+locking, *rereading* of page-table pages may be required.

The algorithms for concurrency control and recovery interact adversely in the differential+locking mechanism also. At the time of transaction completion, *rereading* of those local A and D pages that have migrated to disk will be required to append them to the global A and D files. In differential+optimistic combination, both concurrency control and recovery require deferred updating and *rereading* of local A and D pages may be required.

An observation that can be made from this discussion is that the behavior of shadow+locking vis-a-vis shadow+optimistic and the behavior of differential+locking vis-a-vis differential+optimistic, as far as the rereading of data is concerned, will be similar due to the manner in which the concurrency control and recovery mechanisms interact.

2.5.3. Commit Processing

When a transaction says that "I am done" (reaches D stage in Figure 2.1),

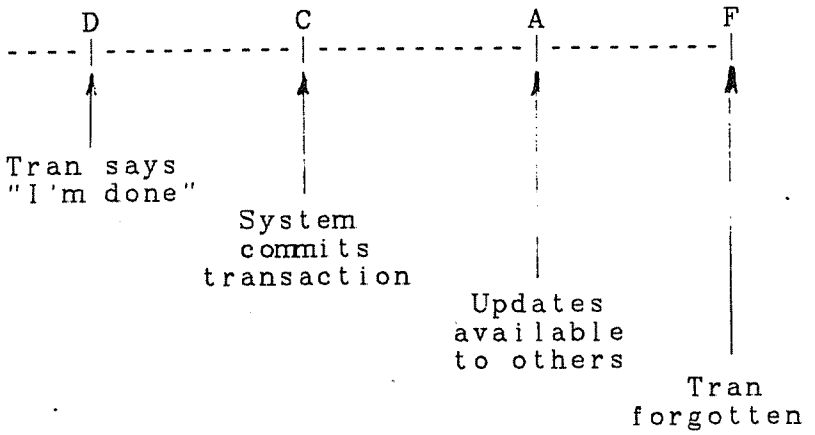

Figure 2.1. Commit Processing

the database system performs some actions (for example, writing the log and commit records to stable storage) before irrevocably committing the transaction (transaction reaches C stage). The difference between the D and the C stage is that if the system crashes during the DC interval then at the time of recovery, the transaction will be undone; whereas, the transaction will be redone if the system crashes after the D stage. Even after a transaction has been committed, there is a time gap before its updates are made available to other transactions (transaction reaches A stage). This time gap could be due, for example in the optimistic method, to the time required to make local copies global. The transaction is forgotten (reaches F stage) when the database system deletes all the control information that it is maintaining on the transaction. For example with the optimistic method, a transaction may be forgotten only after all those transactions that started before this transaction reached the A stage

have completed their validation. Various control sets for the transaction like read set, write set etc. will have to be retained till such time.

It is desirable that the length of the time-interval between the different stages during the commit processing be as short as possible. A long DC interval increases the chance of a transaction-abort if the system crashes. A long CA interval results in increased waiting for lock requests with locking and a higher number of transaction restarts with the optimistic method. A long AF interval increases the space overhead of the database system. We will now examine these time-intervals for the different integrated mechanisms.

With log+locking (Figure 2.2), to commit a transaction, the database system must flush the log records and the commit record to stable storage. Updates are made available to other transactions once the locks held by the transaction have been released. The locks may be released even before updated pages have been written to disk. Thus, if there is locality of reference, a page may be updated many times in the memory without being written to disk.

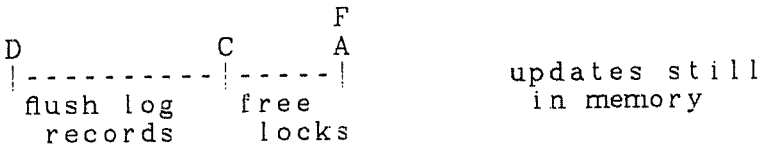

Figure 2.2. Log+Locking

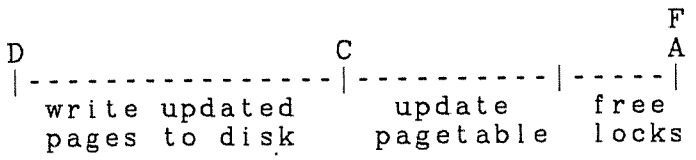

Figure 2.3. Shadow+Locking

With shadow+locking (Figure 2.3), on the other hand, updated pages have to be written to disk before a transaction can be committed. Thus, if there is locality of reference, the advantage of saving I/Os by not writing the updated data pages to disk is lost. Updates are made available to other transactions after the page-table entries have been updated and locks released. In general, locks will be held for a longer duration with the shadow+locking mechanism compared to the log+locking mechanism. Also the duration of the DC and CA intervals with shadow+locking will be longer than the corresponding durations with log+locking.

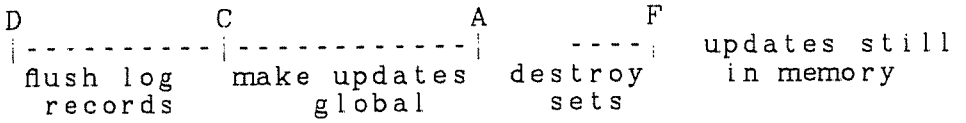

Figure 2.4. Log+Optimistic

With the log+optimistic combination (Figure 2.4), the database system has to flush the log and commit records before committing a transaction as in the

case of log+locking. Updates are made available to other transactions after the local copies have been made global. The time required for this operation could be substantial, particularly for long transactions, as this operation may require reading of data and log pages from disk.

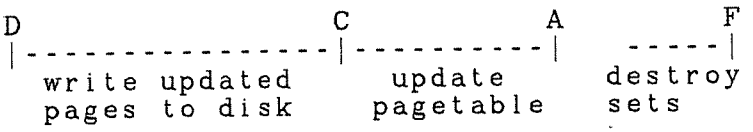

Figure 2.5. Shadow+Optimistic

The commit processing for shadow+optimistic (Figure 2.5) looks very much like shadow+locking. The transaction is committed once updated pages have been written to disk, and updates are made available after updating the page-table.

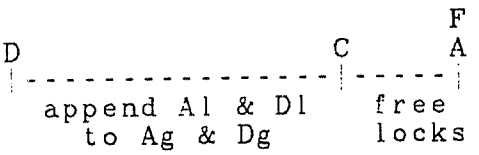

Figure 2.6. Differential+Lockig

With differential+locking (Figure 2.6), local A and D pages have to be appended to the global A and D files to commit a transaction. Updates are made available after the locks have been released. The duration of the DC interval

with differential+locking may be longer compared to log+locking because reading the local A and D pages from disk may be required.

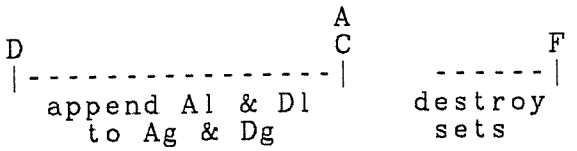

Figure 2.7. Differential+Optimistic

Commit processing for differential+optimistic (Figure 2.7) is similar to differential+locking. The transaction is committed and the updates are made available as soon as local A and D pages have been appended to the global A and D files.

The F stage is reached much later in all the optimistic-based mechanisms when compared with the corresponding locking-based mechanisms. The reason is that, with locking, a transaction can be forgotten as soon as the locks held by the transaction are released. With the optimistic method, however, the database system has to wait for all those transactions that started before the A stage of the transaction was reached to complete validation before forgetting the transaction.

2.6. The Cost Model

To evaluate the performance of various concurrency control and recovery algorithms, our cost model incorporates both the impact that the concurrency control mechanism has on the probability that the transaction will run to completion without conflicting with another transaction *and* the extra burden

imposed on the transaction by the algorithm. This burden is measured in terms of CPU and I/O resources consumed by the transaction (or by the system on behalf of the transaction) to execute the concurrency control and recovery algorithm.

When a transaction is started, there are three possible outcomes:

(1) the transaction runs to completion and commits (transaction *succeeds*),
(2) the transaction is aborted by the user or because of invalid input data (transaction *fails*),
(3) the transaction is aborted by the system and is restarted, perhaps many times, before it completes (transaction *succeeds after rerun(s)*).

In each of these three cases, the concurrency control and recovery mechanism adds *extra* but *varying* amount of burden on the transaction.

Let us examine the third case more closely. Assume that the transaction is restarted only once. The extra burden in this case consists of two parts:

[a] the burden from the time the transaction started to the time it was aborted by the system and its effects were undone,
[b] the burden during the final successful execution of the transaction from start to commit.

Note that the burden for case 3[b] is the same as for case 1. At first glance the burden for case 3[a] appears to be equal to the burden for case 2 (assuming that the transaction fails at the same point). However, the burden for case 3[a] must also include the execution cost of the transaction before it was aborted since this cost would not have been incurred if the transaction were run by itself. Another way of viewing this scenario is that transactions always succeed unless terminated by the user[10]. However, certain successful transactions get internally restarted before they succeed, creating extra burden.

[10] If a user restarts a transaction after aborting it, it is considered to be a new transaction.

The burden, BX, imposed on a transaction by the recovery and concurrency control algorithm utilized can be modeled as:

$$BX = B_{setup} + p_{fail} * B_{fail} + p_{succ} * B_{succ} + p_{rerun} * B_{rerun}$$

where,

B_{setup} is the initialization cost incurred irrespective of the ultimate fate of the transaction,

p_{fail} is the probability that the transaction fails, i.e. is aborted by the user,

B_{fail} is the cost incurred when a transaction fails,

p_{succ} is the probability that the transaction ultimately succeeds,

B_{succ} is the cost incurred when a transaction succeeds (e.g. for committing the transaction),

p_{rerun} is the probability that the transaction is rerun[11],

B_{rerun} is the cost incurred when a transaction is aborted by the system,

and,

$$p_{succ} + p_{fail} = 1.$$

We will develop cost equations for B_{setup}, B_{succ}, B_{fail} and B_{rerun} for various integrated concurrency control and recovery mechanisms in the following section. The value of p_{fail} will be based on Gray's estimates in [Gray81c]. Knowing p_{fail}, $p_{succ} = 1 - p_{fail}$.

When locking is used as the concurrency control mechanism, we assume that transactions that run into deadlock are rerun. We will take the value of p_{ddlk}, the probability that a lock request by a transaction will result in a deadlock, from Lin-Nolte's simulation study [Lin82a] and Gray's analysis of the probability of waiting and deadlock [Gray81a]. Knowing p_{ddlk}, the probability that a transaction will be restarted, p_{rerun}, is computed by assuming that all lock requests are independent and that a deadlock may be caused only at the time of a request for a write-lock[12].

---

[11] If a transaction is restarted more than once, it is modeled by suitably adjusting the value of p_{rerun}.

[12] The assumption that only write accesses may cause a transaction to be aborted underestimates the probability that a transaction will be restarted. In

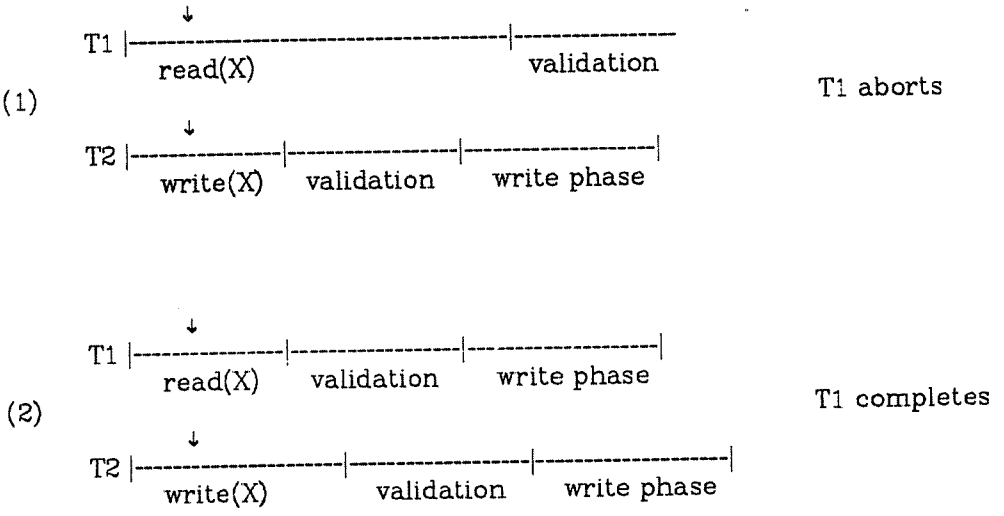

Figure 2.8. Transaction conflict with optimistic concurrency control

For the optimistic method of concurrency control, we will assume that if an access to an object by a transaction conflicts with the objects accessed by another transaction, then the probability that the transaction will be restarted is 0.5. To see this, consider conflicting accesses to an object X by transactions T1 and T2 and two scenarios as shown in Figure 2.8. In the first case, T1 is aborted, while in the second case, T1 runs to completion. Values for $p_{conflict}$ are based again on earlier simulation and analytical analyses [Lin82a, Gray81a]. We again assume accesses to be independent and compute p_{rerun} further

Lin-Nolte's simulation [Lin82a] and in Gray et al.'s analysis [Gray81a], it has been assumed that all locks are exclusive. Thus, in order to use their results we had to assume that for both the locking and optimistic mechanisms only a conflicting write access will cause a transaction abort. A sensitivity analysis that we performed showed that this assumption tends to favor the optimistic method.

assuming that only conflicting write accesses result in transaction aborts to be consistent with the assumptions made for locking.

The Transaction Model

A database system is characterized by a mix of read-only and the read/write transactions. We will model a transaction by the total number of database pages it touches, NP_t of which NP_u pages are updated (we assume that a transaction reads a page before updating it). Finally, for purposes of simplicity, we assume that each page is read by a transaction exactly once regardless of the number of records that must be accessed on the page[13].

2.7. Cost Equations

Before presenting integrated recovery and concurrency control mechanisms and their associated cost equations, we first specify the system parameters used in these cost equations and state our assumptions about the concurrency control mechanisms. Assumptions about the recovery mechanisms will be described along with the cost equations for the integrated mechanisms. Appendix 1 contains a glossary of the notation used in the cost equations.

2.7.1. System Parameters

Table 2.1 shows the system parameters used in the development of cost equations for various recovery and concurrency control algorithms. The actual values of these parameters will depend on the physical characteristics of the stable storage device (assumed to be a disk) and the processing unit. Some parameters also depend on the characteristics of the database for which

---

[13] Note that our model can be extended by including a parameter APPEND% to specify the fraction of update pages that were created by the transaction. These pages are not read before they are updated.

average values have been assumed. Before evaluating the performance of the alternative integrated concurrency control and recovery mechanisms in Section 9, values will be assigned to these parameters in Section 8.

$T_{l\text{-}io}$	time to read/write a disk page with disk seek
$T_{s\text{-}io}$	time to read/write a disk page without a seek
T_{page}	cpu time to process a page in memory
T_{rec}	cpu time to process a record in memory
DBSize	Size of the database
MPL	Level of multiprogramming

Table 2.1. System parameters

2.7.2. Assumptions About the Concurrency Control Mechanisms

2.7.2.1. Locking-Based Concurrency Control

1. The lock-acquisition discipline is "get only when needed".

2. The time to process a lock acquisition request is T_{al} and the time to process a lock release request is T_{rl}[14]. The probability that a lock request will conflict is $p_{conflict}$ and p_{wait} is the probability that a request will be queued. T_{wait} is the wait time for a blocked request.

3. The granularity of locking is a page[15].

---

[14] Gray [Gray82a] asserts that the lock table can always be maintained in the main memory, and that this is the case in IMS and System R. Lin and Nolte [Lin82a] in their simulation of two-phase locking assumed the lock processing to be instantaneous. If, however, the lock table must be maintained on secondary storage, it can be modeled by choosing appropriate higher values of T_{al} and T_{rl}.

[15] A page may not necessarily be the best level of granularity [Ries79a], but we will assume it to be so uniformly for all the algorithms.

4. Transaction abort (either system initiated or user initiated) occurs when the transaction has read $NP_t/2$ pages and has updated $NP_u/2$ pages.

5. Deadlocks are resolved by checking for cycles in the waits-for graph at each lock request that conflicts[16]. T_{ddlk} is the cpu time required for this test and p_{ddlk} is the probability that a cycle would be found. Thus, the probability that a lock request waits, $p_{wait} = p_{conflict} \cdot p_{ddlk}$.

2.7.2.2. Optimistic Concurrency Control

1. The granularity of the elements in the various control sets (readset, writeset etc.) is a page.

2. The cost of creating various control sets is a function of NP_t[17]. When $NP_t = 1$, the cpu time to create the control sets is assumed to be T_{as}. We assume that a background process is responsible for deleting various control sets and we will not model this cost.

3. Reads and writes on a page with optimistic concurrency control first check the write set to determine whether the local copy exists of the corresponding page. Since the write set can be maintained in the main memory, we will

---

[16] Deadlock prevention using one of the timestamp-based schemes proposed by Rosenkrantz et al. [Rose78a] can be modeled by assuming that half of the conflicting requests wait and the other half result in transaction aborts, that is, $p_{wait} = p_{rerun} = p_{conflict}/2$.

An alternative is to avoid the problem of deadlock altogether by assuming that all the locks needed for executing a transaction are requested at the initiation of the transaction as in [Ries79a, Poti80a]. If any lock cannot be granted, the transaction releases all the locks and tries again. This can be modeled by choosing a larger value for T_{al}.

[17] A much finer analysis is possible where the size of various sets is estimated and accordingly the cost of creating various sets is determined. However, because of the coarse granularity chosen for the elements of these sets, they can be maintained in the main memory and the creation of sets would not contribute significantly to the total cost.

assume the cost of this indirection to be negligible[18].

4. If a transaction is aborted by the user, it happens when the transaction has completed half of its read phase.

5. If a transaction fails to be validated, it is detected half way through the validation test.

6. The cost of validating a transaction is a function of the size of the transaction and the number of concurrently executing transactions, MPL. We will assume the cost of validation to be (MPL-1) * T_{valid}, where T_{valid} is the time to validate a transaction if only one other transaction executing concurrently with it.

2.7.3. Integrated Mechanisms

We will now sketch integrated recovery and concurrency control mechanisms and present cost equations for them.

2.7.3.1. Log+Locking

This is the well known scheme described in [Gray78a]. A transaction before accessing a data page acquires a lock on it and the database is updated using the "write-ahead log" protocol.

2.7.3.1.1. Assumptions

1. The number of log pages generated by a transaction is determined by the parameter Log% (Number of log pages = Log% * NP_u. NP_u is the number of pages updated by the transaction).

---

[18] If desired, the extra cost can be modeled by choosing a larger value of $T_{i/o}$ for reads and writes.

2. A transaction is assigned a fixed number of data buffers. We postulate a function DFlush(X) that, given the total number of data pages X updated by a transaction at some time t, returns the number of pages that have migrated to disk at time t and are not present in the main memory. Similarly, the function LFlush(Y), where Y is the number of log pages generated by a transaction at time t, returns the number of log pages that have been written to the disk and are no longer available in the main memory at time t[19].

For the write-ahead protocol, it *must* be the case that for all time t

$$LFlush(Y) \geq Log\% * DFlush(X).$$

3. We assume a separate log disk devoted to logging. Therefore, writing a log page does not require a disk seek except when a complete cylinder has been filled with the log pages. As specified in Section 8, we account for the cost of this seek by amortizing it across all write operations to the cylinder. Another situation that may necessitate a seek for writing a log page is after the disk heads have been moved to a random cylinder to read a log page for performing a transaction undo. As explained in Section 8, we account for this seek by multiplying the average seek time by the sum of p_{rerun} and p_{fail}.

4. Since we assume that on average a transaction gets into deadlock after reading and processing $NP_t/2$ pages and updating $NP_u/2$ pages, the execution

---

[19] Sometimes, the flushing of data and log buffers is delayed as much as possible until the transaction commits or aborts in order to reduce the cost of undo processing. On the other hand, data and log buffers may be flushed as soon as they are created to increase parallelism and minimize the commit time duration.

The first situation can be modeled by defining DFlush to be DFlush(X) = max {0,X-DBuff} where, DBuff is the number of data buffers allocated to the transaction. This assumes that the buffer manager first ejects a page that has only been read but not updated before replacing an updated page. The second situation can be modeled by defining DFlush as DFlush(X) = X. The function LFlush may be defined analogously.

cost of an aborted transaction = cost of reading $NP_t/2$ pages + cost of processing $NP_t/2$ pages + cost of updating $NP_u/2$ pages = $(T_{l\text{-}io}+T_{page})*NP_t/2$ + $DFlush(NP_u/2)*T_{l\text{-}io}$.

2.7.3.1.2. Cost Equations

B_{setup} = cost of writing the tran-begin log record { $T_{s\text{-}io}$ }
+ cost of writing the commit/abort log record { $T_{s\text{-}io}$ }

B_{succ} = cost of acquiring locks[20]
{ $NP_t*T_{al} + NP_t*p_{conflict}*T_{ddlk} + NP_t*p_{wait}*T_{wait}$ }
+ cpu cost of creating log pages { $ceil(Log\% * NP_u) * T_{page}$ }
+ i/o cost of writing log pages { $ceil(Log\% * NP_u) * T_{s\text{-}io}$ }
+ cost of releasing locks { $NP_t * T_{rl}$ }

B_{fail} = burden before the transaction abort
+ cost of undo processing

= cost of acquiring and releasing locks
{$(T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl})*NP_t/2$}
+ cpu cost of creating log pages { $ceil(Log\% * NP_u/2) * T_{page}$ }
+ i/o cost of writing log pages { $LFlush(ceil(Log\% * NP_u/2)) * T_{s\text{-}io}$ }
+ cost of reading log pages for undo { $LFlush(ceil(Log\%*NP_u/2)) * T_{l\text{-}io}$ }
+ cost of reading flushed data pages for undo { $DFlush(NP_u/2) * T_{l\text{-}io}$ }
+ cpu cost of undoing corrupted data pages { $DFlush(NP_u/2) * T_{page}$ }
+ i/o cost of writing undone pages { $DFlush(NP_u/2) * T_{l\text{-}io}$ }

B_{rerun} = B_{fail} + transaction execution cost before abort
= $B_{fail} + (T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$

---

[20] Cost of acquiring locks = Cost of (requesting locks + deadlock detection - waiting for locks)

2.7.3.1.3. Comments

Instead of incurring separate I/O's for writing the tran-begin and the commit/abort records, they can be written along with other log records for the transaction on the same page. In this case, we can assume that $B_{setup} = 0$.

2.7.3.2. Log+Optimistic

Transactions execute unhindered but instead of making separate local copies of updated objects during the read phase as required for concurrency control, the log records are used. Although it is possible to derive the writeset and the createset of a transaction by examining its log records, it is more efficient to create them separately in main memory. During the write phase of the transaction, log pages are used to make the updates global while observing the write-ahead-log protocol.

2.7.3.2.1. Assumptions

Assumptions 1-3 of the log+locking mechanism are again assumed to hold. In addition, we assume that a disk seek will be required to read a log page in order to make the local copies global as the log pages for one transaction may not be physically adjacent on the disk.

Observe that the decision to abort a non-serializable transaction is taken after the completion of its read phase, i.e. after reading and processing NP_t pages, but no updated pages are written during the read phase (log records double up as local copies during the read phase). Hence, the execution cost of an aborted transaction is $(T_{l\text{-}io}+T_{page})*NP_t$.

2.7.3.2.2. Cost Equations

B_{setup} = cost of writing the tran-begin log record { $T_{s\text{-}io}$ }
+ cost of writing the commit/abort log record { $T_{s\text{-}io}$ }

B_{succ} = cost of creating control sets { $NP_t * T_{as}$ }
+ cpu cost of creating log pages { $ceil(Log\%*NP_u) * T_{page}$ }
+ i/o cost of writing log pages { $ceil(Log\%*NP_u) * T_{s\text{-}io}$ }
+ cost of validation test { $(MPL-1) * T_{valid}$ }
+ cost of making local copies global[21]
{$LFlush(ceil(Log\%*NP_u))*T_{l\text{-}io}+DFlush(NP_u)*T_{l\text{-}io}$}

B_{fail} = burden before the transaction abort + cost of undo processing (= 0)[22]
= cost of creating the control sets { $NP_t/2 * T_{as}$ }
+ cpu cost of creating log pages { $ceil(Log\%*NP_u/2) * T_{page}$ }
+ i/o cost of writing log pages { $LFlush(ceil(Log\%*NP_u/2)) * T_{s\text{-}io}$ }
- cost of writing $DFlush(NP_u/2)$ data pages[23] { $DFlush(NP_u/2) * T_{l\text{-}io}$ }

B_{rerun} = cost of creating the control sets { $NP_t * T_{as}$ }
+ cpu cost of creating log pages { $ceil(Log\%*NP_u) * T_{page}$ }
+ i/o cost of writing log pages { $LFlush(ceil(Log\%*NP_u)) * T_{s\text{-}io}$ }

---

[21] The cost of making local copies global involves reading the log pages that have migrated to disk and the data pages to be updated that are no longer available in main memory. However, it would not include the cost of updating the data pages in the main memory and writing back the updated pages. These costs are not incurred during the read phase of the transaction and hence can be amortized during this phase.

[22] No undo processing is required as at this point all changes have been performed on the local copies.

[23] Since we are developing formulas that express the overhead (burden) incurred, we must model savings provided by a mechanism as well as costs. Thus, since no pages are actually updated in the log+optimistic approach until the transaction is validated, $DFlush(NP_u/2)$ write operations are avoided when compared with a system that provides no recovery mechanism and does in-place updating.

+ cost of the validation test $\{ (MPL-1) * T_{valid} / 2 \}$

+ transaction execution cost before abort $\{ (T_{l\text{-}io}+T_{page})*NP_t \}$

2.7.3.2.3. Comments

As in the case of the log+lock algorithm, the tran-begin and the commit/abort records for a transaction can be written together with the other log records for the transaction.

2.7.3.3. Shadow+Locking

Before accessing a data page, the transaction locks that page. However, no explicit locking is needed to access page-table (both S-Map and C-map) entries. The protocol required is that a transaction accesses a page-table entry to get the physical address of a data page only if it has been granted a lock for that page. Thus, it is not possible for a transaction to access a page-table entry while it is being updated. Once a transaction completes, its write-lock on a page is released only after the corresponding entry in the page-table has been updated.

2.7.3.3.1. Assumptions

1. The size of the page-table is PtSize pages. For relations of reasonable size, PtSize will be large. Thus, the S-Map cannot reside in the main memory and must be paged from the secondary storage [Gray81b]. Consequently a data page I/O may also cause a page-table I/O. However, in general, accessing X data pages will not result in access to X distinct pages of the S-Map since a number of page-table entries can be blocked into one S-Map page. The number of S-Map pages that will have to be accessed is determined by a function PtPages(X). For the random access of data pages, the number of S-Map pages required to be accessed is analogous to the number of pages accessed when randomly selecting

records from a blocked file. We will use the Cardenas' expression [Card75a] for this purpose[24], and define

$$PtPages(X) = PtSize(1 - (1 - 1/PtSize)^{X}).$$

For sequential access of data pages,

$$PtPages(X) = 1 + X / \text{blocking-factor}^{25}.$$

2. The tran-begin and the incremental C-Map can be written on the same page as the pre-commit record on the commit list.

3. The function SFlush(X), where X is the number of S-Map pages read by the transaction at time t, returns the number of S-Map pages that are no longer available in the memory. SFlush(X) = max {0,X-SBuff} where SBuff is the number of buffers available to the transaction for reading the S-Map pages. The function DFlush(Y) which returns the number of updated pages that have migrated to the disk is defined analogously.

4. A shadow-based algorithm generates extra NP_u allocate-page and free-page requests for data pages when compared to an in-place updating algorithm. The cost of processing an allocate-page or a free-page request will be assumed to be T_{rec}.

5. Writing to the commit list does not require a disk seek.

6. The cost of creating an entry in the C-Map is T_{rec}.

---

[24] It has been shown that the Cardenas' expression gives the lower bound for the expected number of pages accessed and more accurate expressions are available in literature (see [Yao77a]). However, for large blocking factors (> 10) such as would be present in the S-Map, the error in Cardenas' approximation is negligible.

[25] One has been added to account for the fact that the desired page-table entries may not start at the beginning of a page-table page.

7. As in the case of log+locking mechanism, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$.

2.7.3.3.2. Cost Equations

B_{setup} = cost of writing commit/abort record = $T_{s\text{-}io}$

B_{succ} = cost of acquiring locks $\{ NP_t * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}) \}$

+ i/o cost of reading S-Map pages for data reads $\{ PtPages(NP_t) * T_{l\text{-}io} \}$

+ cost of extra allocate-page requests $\{ NP_u * T_{rec} \}$

+ cpu cost of creating incremental C-Map $\{ NP_u * T_{rec} \}$

+ i/o cost of writing the pre-commit record $\{ T_{s\text{-}io} \}$

+ i/o cost of rereading flushed S-Map pages $\{SFlush(PtPages(NP_u)) * T_{l\text{-}io}\}$

+ cpu cost of updating S-Map entries $\{ NP_u * T_{rec} \}$

+ i/o cost of writing the updated S-Map pages $\{ PtPages(NP_u) * T_{l\text{-}io} \}$

+ cost of releasing locks $\{ NP_t * T_{rl} \}$

+ cost of extra free-page requests $\{ NP_u * T_{rec} \}$

B_{fail} = burden before the transaction abort + cost of undo processing

= cost of acquiring and releasing locks
$\{ (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl})*NP_t/2 \}$

+ i/o cost of reading S-Map pages for data reads $\{PtPages(NP_t/2) * T_{l\text{-}io}\}$

+ cost of extra allocate-page requests $\{ NP_u/2 * T_{rec} \}$

+ cpu cost of creating incremental C-Map $\{ NP_u/2 * T_{rec} \}$

+ cost of extra free-page requests $\{ NP_u/2 * T_{rec} \}$

B_{rerun} = B_{fail} + transaction execution cost before abort

= $B_{fail} + (T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$

2.7.3.3.3. Comments

1. The writing of commit/abort can be piggybacked with the pre-commit record of the next transaction at the expense of increasing somewhat the response time of the transaction.

2. It is possible to avoid writing the abort record when a transaction is aborted by the user. However, the disadvantage is that at the time of the recovery from system crash, it would not be possible to distinguish the user-aborted transactions from those that were active at the time of crash.

2.7.3.4. Shadow+Optimistic Algorithm

With shadow as the recovery mechanism, there are always two copies of each data page being updated by a transaction: the updated copy and the unmodified (shadow) copy on disk. When shadow is combined with the optimistic method of concurrency control, the updated copy of each data page being modified can also be used as the local copy for concurrency control purposes.

For purposes of concurrency control, as a transaction executes, it creates various control sets (readset, writeset etc.). There is, however, no need to create a C-Map as required by recovery mechanism since the writeset (which normally contains only the page numbers of the updated pages) can be augmented to include the disk addresses of the modified pages along with the page numbers. With this approach, the write phase in which local copies are made global requires simply updating the S-map entries using the writeset to point to new disk addresses.

Note that it is not required to keep S-Map or C-Map page numbers accessed by a transaction in its control sets. If the updates to S-Map by a transaction have been partially applied and meanwhile another transaction reads the not yet

updated S-Map entries, that transaction will not be validated.

2.7.3.4.1. Assumptions

We assume that assumptions 1-5 of the shadows+locking mechanism are valid for this mechanism also. However, since the decision to abort a non-serializable transaction is taken after the completion of its read phase, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t$ + DFlush(NP_u)*$T_{l\text{-}io}$

2.7.3.4.2. Cost Equations

B_{setup} = cost of writing commit/abort record = $T_{s\text{-}io}$

B_{succ} = cost of creating the control sets { $NP_t * T_{as}$ }

+ i/o cost of reading S-Map pages for data reads { PtPages(NP_t) * $T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $NP_u * T_{rec}$ }

+ cost of validation test { (MPL-1) * T_{valid} }

+ i/o cost of writing the pre-commit record { $T_{s\text{-}io}$ }

+ i/o cost of reading flushed out S-Map pages { SFlush(PtPages(NP_u)) * $T_{l\text{-}io}$ }

+ cpu cost of updating S-Map entries { $NP_u * T_{rec}$ }

+ i/o cost of writing the updated S-Map pages { PtPages(NP_u) * $T_{l\text{-}io}$ }

+ cost of extra free-page requests { $NP_u * T_{rec}$ }

B_{fail} = burden before the transaction abort + cost of undo processing

= cost of creating the control sets { $NP_t/2 * T_{as}$ }

+ i/o cost of reading S-Map pages for data reads { PtPages(NP_t/2) * $T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $NP_u/2 * T_{rec}$ }

+ cost of extra free-page requests { $NP_u/2 * T_{rec}$ }

B_{rerun} = burden before the transaction abort + cost of undo processing

+ transaction execution cost before abort

= cost of creating the control sets { $NP_t * T_{as}$ }

+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t) * T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $NP_u * T_{rec}$ }

+ cost of validation test { $(MPL-1) * T_{valid} / 2$ }

+ cost of extra free-page requests { $NP_u * T_{rec}$ }

+ transaction execution cost { $(T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$ }

2.7.3.4.3. Comments

As in the case of the shadow+lock algorithm, the commit/abort record may be piggybacked with the pre-commit record of the next transaction. Also, the writing of the abort record in the case of a user-initiated transaction abort may be avoided.

2.7.3.5. Differential File+Locking Algorithm

As the transaction executes, it locks the pages of the global base relation, B_g, and the differential relations, A_g and D_g[26], and creates the local differential relations A_l and D_l. Once the transaction commits, A_l and D_l are appended to A_g and D_g.

2.7.3.5.1. Assumptions

1. The sizes of the differential relations, A_g and D_g, are Size% of the size of the base relation, B_g. We will assume A_g and D_g to be of equal size.

2. Accessing NP_t pages of R needs Xtra% extra page accesses. Xtra% is a function of the size of A_g and D_g. We assume that the transaction is executed when half of A_g and half of D_g have been created. Thus, a transaction will read

[26] Recall that $R = (B \cup A) - D$

Size%/2 extra pages each from the A_g and D_g relations.

3. Processing of pages in the memory also incurs extra cpu overhead[27]. We will assume the extra cpu overhead to be CpuOH% of the total cpu time consumed if the transaction was run alone without any provision for recovery.

4. The number of A_l and D_l pages generated by a completed transaction is Comprs% of NP_u. Thus, a transaction writes ceil(Comprs%*NP_u) pages each to A_l and D_l. However, in-place updating would incur the cost of writing NP_u pages. Therefore, the net cost is the cost of writing (2*ceil(Comprs%*NP_u) - NP_u) pages.

5. DFlush(X) is the function that returns the number of A_l and D_l pages that migrate to disk[28].

6. Writing to the commit list does not require a disk seek.

7. The execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t/2$. Note that unlike an in-place updating mechanism like log+locking, with the differential file approach the transaction does not incur the cost of writing updated pages. The cost of writing A and D pages is considered to be recovery burden associated with the differential file approach.

2.7.3.5.2. Cost Equations

$B_{setup} = 0$[29]

---

[27] For example, since R = (B U A) - D, a retrieve on R will be translated into a retrieve on B, A, and D followed first by a union of tuples retrieved from B and A and then by a set-difference of the result and tuples retrieved from D.

[28] DFlush(X) = max {0,DBuff-X} where DBuff is the number of buffers available.

[29] Only the tran-id of committed transactions is written to commit list and this cost is included in B_{succ}.

B_{succ} = cost of acquiring and releasing locks
$\{(1+Size\%)*NP_t * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl})\}$

+ cost of extra data page reads $\{ Size\%*NP_t * T_{l\text{-}io} \}$

+ extra cpu cost of processing data pages $\{ CpuOH\% * (NP_t*T_{page}) \}$

+ cost of writing flushed A_l and D_l pages $\{2*DFlush(ceil(Comprs\%*NP_u))*T_{l\text{-}io}\}$

+ cost of rereading flushed A_l and D_l pages $\{2*DFlush(ceil(Comprs\%*NP_u))*T_{l\text{-}io}\}$

+ net cost of writing A_g and D_g pages[30] $\{(2*ceil(Comprs\%*NP_u) - NP_u)*T_{l\text{-}io}\}$

+ cost of extra allocate-page requests $\{ 2 * ceil(Comprs\%*NP_u) * T_{rec} \}$

+ cost of writing the tran-id to commit list $\{ T_{s\text{-}io} \}$

B_{fail} = burden before the transaction abort + cost of undo processing

= cost of acquiring and releasing locks
$\{ (1+Size\%)*NP_t/2 * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl}) \}$

+ cost of extra data page reads $\{ Size\%*NP_t/2 * T_{l\text{-}io} \}$

+ extra cpu cost of processing data pages $\{ CpuOH\% * (NP_t/2*T_{page}) \}$

+ net cost of writing flushed local A and D pages
$\{ (2 * DFlush(ceil(Comprs\%*NP_u/2)) - DFlush(NP_u/2)) * T_{l\text{-}io} \}$

B_{rerun}[31] = B_{fail} + cost of writing $DFlush(NP_u/2)$ pages $\{DFlush(NP_u/2) * T_{l\text{-}io}\}$

+ transaction execution cost before abort $\{ (T_{l\text{-}io}+T_{page})*NP_t/2 \}$

2.7.3.6. Differential File+Optimistic Algorithm

First observe that the A_l and D_l can also be used for the local copies of modified records for concurrency control purposes. As the transaction executes, it creates control sets and if it is validated, it appends A_l and D_l to global A_g and D_g.

---

[30] See assumption 4 above.

[31] The execution cost before abort does not incur the cost of writing $DFlush(NP_u/2)$ pages as is the case in an in-place updating algorithm. However, since the cost of writing $DFlush(NP_u/2)$ pages was subtracted from B_{fail}, this cost will be added here in order to make the formula correct.

2.7.3.6.1. Assumptions

The same assumptions stated for differential file+locking are assumed to hold. As in the case of differential file+locking mechanism, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t$. The difference is that the decision to abort the transaction is taken after reading and writing NP_t pages, instead of $NP_t/2$ pages as in the case of differential file+locking mechanism.

2.7.3.6.2. Cost Equations

$B_{setup} = 0$

B_{succ} = cost of creating the control sets { $(1+Size\%)*NP_t * T_{as}$ }
+ cost of extra data page reads { $Size\%*NP_t * T_{l\text{-}io}$ }
+ extra cpu cost of processing data pages { $CpuOH\% * (NP_t*T_{page})$ }
+ cost of writing flushed A_l and D_l pages {$2* DFlush(ceil(Comprs\%*NP_u)) * T_{l\text{-}io}$}
+ cost of validation test { $(MPL-1) * T_{valid}$ }
+ cost of rereading flushed A_l and D_l pages {$2*DFlush(ceil(Comprs\%*NP_u)) * T_{l\text{-}io}$}
+ net cost of writing A_g and D_g pages { $(2*ceil(Comprs\%*NP_u) - NP_u) * T_{l\text{-}io}$ }
+ cost of extra allocate-page requests { $2*ceil(Comprs\%*NP_u) * T_{rec}$ }
+ cost of writing the tran-id to commit list { $T_{s\text{-}io}$ }

B_{fail} = burden before the transaction abort + cost of undo processing (=0)
= cost of creating the control sets { $(1+Size\%)*NP_t/2 * T_{as}$ }
+ cost of extra data page reads { $Size\%*NP_t/2 * T_{l\text{-}io}$ }
+ extra cpu cost of processing data pages { $CpuOH\% * (NP_t/2*T_{page})$ }
+ net cost of writing flushed local A and D pages
{ $2 * DFlush(ceil(Comprs\%*NP_u/2)) - DFlush(NP_u/2)) * T_{l\text{-}io}$ }

B_{rerun} = burden before the transaction abort + cost of undo processing (=0)
+ transaction execution cost before abort

= cost of creating the control sets { $(1+Size\%)*NP_t * T_{as}$ }

+ cost of extra data page reads { $Size\%*NP_t * T_{l\text{-}io}$ }

+ extra cpu cost of processing data pages { $CpuOH\% * (NP_t*T_{page})$ }

+ cost of writing flushed local A and D pages
{ $2 * DFlush(ceil(Comprs\%*NP_u)) * T_{l\text{-}io}$ }

+ cost of validation test { $(MPL-1) * T_{valid} / 2$ }

+ transaction execution cost { $(T_{l\text{-}io}+T_{page})*NP_t/2$ }

2.8. Database, Mass Storage Device and Processor Specifications

In this section, we specify the characteristics of the database, the mass storage device, and the processor employed in our evaluation.

The mass storage device is modeled after the IBM 3350 disk drive [IBM77a] whose characteristics are shown in Table 2.2. Thus, the average time required to access a random block on the disk

$T_{l\text{-}io}$ = Average seek time + Latency + Transfer time = 37.525 ms.

When an I/O operation is performed on an append-on file such as a log, seek operations are only occasionally necessary. For algorithms that utilize shadows or differential files for recovery, a seek operation is only needed when the current cylinder has been completely filled. To simplify our cost expressions, we have amortized the cost of these occasional seek operations across every

Parameter	Value
No. of recording surfaces	30
No. of cylinders	555
No. of blocks per track	4
Block size	4096 bytes
Revolution time	16.7 ms.
Time to move head N cylinders	10 + 0.072*N ms.
Average seek time	25 ms.

Table 2.2. Disk drive Specifications

write operation on the cylinder. Thus,

$T_{s\text{-}io}$ = Latency + Transfer time

+ 1/120*(time to move heads to the adjacent cylinder)[32]

= 12.61 ms.

When the append-only file is used to hold a recovery log, the disk heads must be moved from the end of the log file to perform transaction undo. In this case,

$T_{s\text{-}io}$ = Latency + Transfer time

+ 1/120*(time to move heads to the adjacent cylinder)

+ $(p_{fail} + p_{rerun})$ * Average Seek Time

= 12.61 + $(p_{fail} + p_{rerun})$ * 25.0 ms.

We have assumed that a 1 MIP processor is used to execute transactions, that 500 instructions are required to process a record (T_{rec} = 0.5 ms), and that 5000 instructions are required to process a page of approximately 10 records[33] (T_{page} = 5.0 ms.).

The size of the database, DBSize, has been assumed to be 100 million bytes. We have evaluated the performance of the integrated concurrency and recovery algorithms under three different workloads: small (TS), medium (TM), and large (TL). Their sizes and some associated characteristics are shown in Table 2.3. The access pattern of the transactions has been assumed to be random.

Twait has been calculated using the formula:

$$T_{wait} = T_{wait}Fctr * (T_{l\text{-}io} + T_{page})$$

in which TwaitFctr=0.83 for the TS workload, 1.96 for TM, and 2.94 for TL. These numbers and the probability figures in Table 2.3 are based on the results presented in [Gray81a, Gray81c, Lin82a].

---

[32] There are 4 blocks per track and 30 tracks per cylinder

[33] A record can be, for example, either a database record or a log record.

Parameter	TS	TM	TL
NPt - number of pages touched	2	50	250
NPu - number of pages updated	1	15	50
MPL - multiprogramming level	15	10	7
Pfail - probability of transaction failure	0.05	0.05	0.05
Pconflict - probability of transaction conflict	0.0012	0.0065	0.01
Pddlk - probability of deadlock	1.92e-7	5.6e-6	3.0e-5
Prerun(locking) - probability of rerun	1.92e-7	8.4e-5	0.0015
Prerun(optimistic) - probability of rerun	0.0006	0.04766	0.22169
Twait - wait time for a blocked request	0.83 ms.	1.96 ms.	2.94 ms.
Tvalid - time to validate a transaction	0.1 ms.	0.5 ms.	2.0 ms.

Table 2.3. Transaction sizes and database characteristics

T_{valid} is based on the assumption that a transaction can be validated against a concurrent transaction in $O(NP_t+NP_u)$ time plus the time for a procedure call. T_{al} (the time to process a lock request), T_{rl} (time to release a lock request), and T_{as} (time to construct the control sets for the optimistic concurrency control algorithm) have been assumed to be 0.5 ms. A value of 0.5 ms. has also been used to represent the cost of determining whether granting a lock request will result in deadlock (T_{ddlk}) based on the results described in [Agra83a].

2.9. Evaluation

In this section we compare the performance of the different integrated concurrency control and recovery mechanisms by computing the *burden ratio* for each mechanism. The burden ratio is defined to be the ratio of BX (the extra burden imposed on the transaction by the concurrency control and recovery mechanism) to the execution time of the transaction if run without any concurrency control or recovery mechanism:

$$\text{Burden Ratio} = \frac{BX}{\text{Execution Time of the Transaction}}$$

$$= \frac{B_{setup} + P_{fail}*B_{fail} + P_{succ}*B_{succ} + P_{rerun}*B_{rerun}}{NP_t*T_{l-io} + NP_t*T_{page} + NP_u*T_{l-io}}$$

We first compare the relative performance of locking and optimistic concurrency control for each of the recovery mechanisms. Then the performance of the three finalists are compared.

2.9.1. Logging

The relative performance of the log+locking and log+optimistic mechanisms is shown in Figure 2.9 and Table 2.4. In Table 2.4, all costs have been expressed in milliseconds. We assumed that DBuff, the number of data buffers allocated to the transaction equals 10, LBuff, the number of buffers available to collect log records for the transaction, equals 1, and Log%, the fraction of each updated page that must be recorded in a log record, equals 0.1.

Based on Figure 2.9 and the data in Table 2.4, we make the following observations about the performance of these two mechanisms:

1. The operation of "making local copies global" in the optimistic concurrency control algorithm is very expensive since NP_u-DBuff data pages[34] that need to be

		Total Bur den	BSucc compo nent	BFail compo nent	BRerun compo nent	Locking			Set crea tion	Valid ation	Make local global
						total	wait	ddl'k			
TS	Lock	22.1	19.9	2.2	0.0	2.0	0.1	0.0			
	Opt	21.2	20.2	0.95	0.05				1.0	1.3	0.0
TM	Lock	113.4	109.2	4.1	0.1	75.3	26.4	0.2			
	Opt	385.5	280.0	1.7	103.8				25.6	4.4	343.6
TL	Lock	715.7	624.5	80.0	11.2	137.2	304.1	1.2			
	Opt	4200.9	1814.6	-22.3	2408.6				149.6	12.7	1568.6

Table 2.4. Log+locking & Log+optimistic

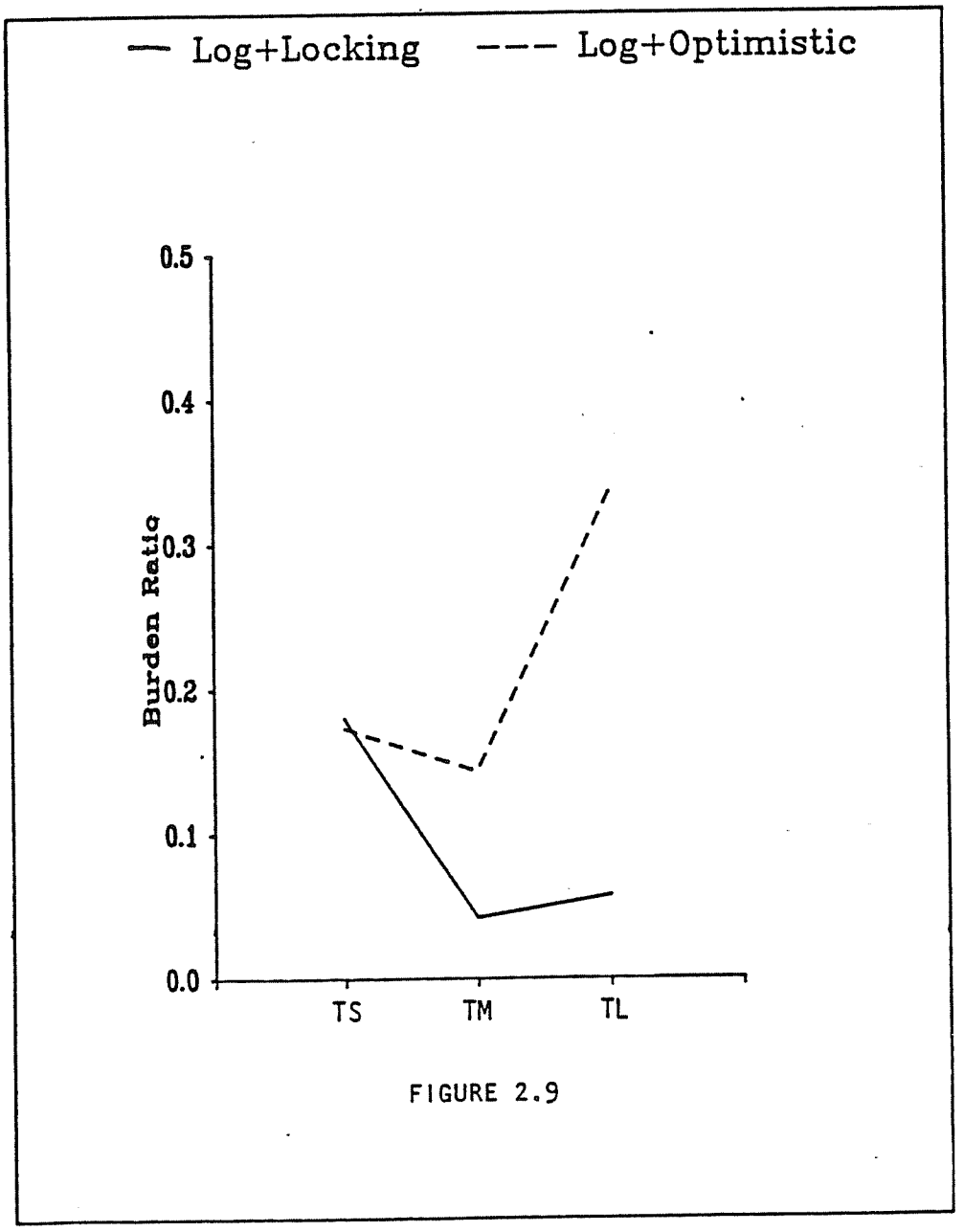
— Log+Locking
--- Log+Optimistic
Burden Ratio
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.9

updated will have migrated to disk before the write phase begins and will have to be reread during the write phase.

2. Backing up a user aborted transaction is more expensive with locking due to the cost of undo processing (reading back those updated pages that have migrated to the disk, undoing the changes and then rewriting them, and the cost of acquiring and releasing locks). In the case of the optimistic method, B_{fail} can actually have a negative value for large transactions as only LFlush(Log%*NP_u/2) data pages are written to the disk instead of DFlush(NP_u/2) pages[34] and no undo processing or validation cost is incurred.

3. As the average transaction size increases, the number of transaction restarts increases faster for the optimistic mechanism than for a lock-based mechanism that uses deadlock detection. Hence, the value of B_{rerun} increases faster for the optimistic method.

4. For a successful transaction, with an increase in the transaction size the cpu burden becomes a larger fraction of the total burden in the log-locking combination as the cost of transaction waits becomes very significant. For the optimistic method, the validation cost does not increase significantly with the transaction size but the I/O burden increases significantly due to the high cost of making the local copies global.

5. The dip in the burden ratio for TM in Figure 2.9 is due to the *blocking* effect while writing the log. TS, although it updates only 1 data page, writes 1 log page. On the other hand, TM, although it updates 15 pages, writes only 2 pages[35]. The

---

[34] Recall that for TL, NPu=50, DBuff=10, LFlush(Log%*NPu/2)=2 and DFlush(NPu/2)=15.

[35] ceil(0.1*1)=1 and ceil(0.1*15)=2.

increased cost of undo processing and waiting increases the burden ratio for TL in the case of locking. In the case of the optimistic method, the increased cost of making local copies global coupled with the high cost of transaction restarts result in a higher burden ratio.

6. Although the total cost of validation is less than the cost of lock management, the log+locking combination outperforms the log+optimistic combination because of the high cost of making local copies global and the the higher restart rate associated with the log+optimistic mechanism. Only in the case of small transactions does the performance of the log+optimistic combination becomes comparable to the performance of the log+locking combination. In this case the buffer space available to the transaction, DBuff, is large enough to hold all the pages updated by the transaction until the transaction is validated and hence the cost of making local copies global is not significant. In addition, for small transactions the value of B_{rerun} is quite low.

2.9.2. Differential Files

The performance of the differential file+locking and differential file+optimistic mechanisms is shown in Figure 2.10 and Table 2.5. We assumed that Size%, the relative size of the A and D files compared to the B file, equals

		Total Bur den	BSucc compo nent	BFail compo nent	BRerun compo nent	Locking			Set crea tion	Valid ation	extra size% I/O	extra CpuOh	I/O sav- ings
						total	wait	ddlk					
T	Lock	67.9	67.4	0.5	0.0	2.2	0.1	0.0			7.5	9.8	35.7
S	Opt	64.6	64.0	0.5	0.1				1.1	1.3	7.4	9.8	35.7
T	Lock	131.4	118.2	13.1	0.1	82.8	29.0	0.2			190.5	243.8	534.7
M	Opt	199.6	64.4	11.6	123.6				28.0	4.5	194.4	255.7	534.7
T	Lock	1311.7	1257.7	43.9	10.1	604.1	344.5	1.3			970.3	1219.7	1808.7
L	Opt	3713.6	808.0	31.9	2873.7				163.2	14.1	1137.6	1495.9	1808.7

Table 2.5. Differential+locking & Differential+optimistic

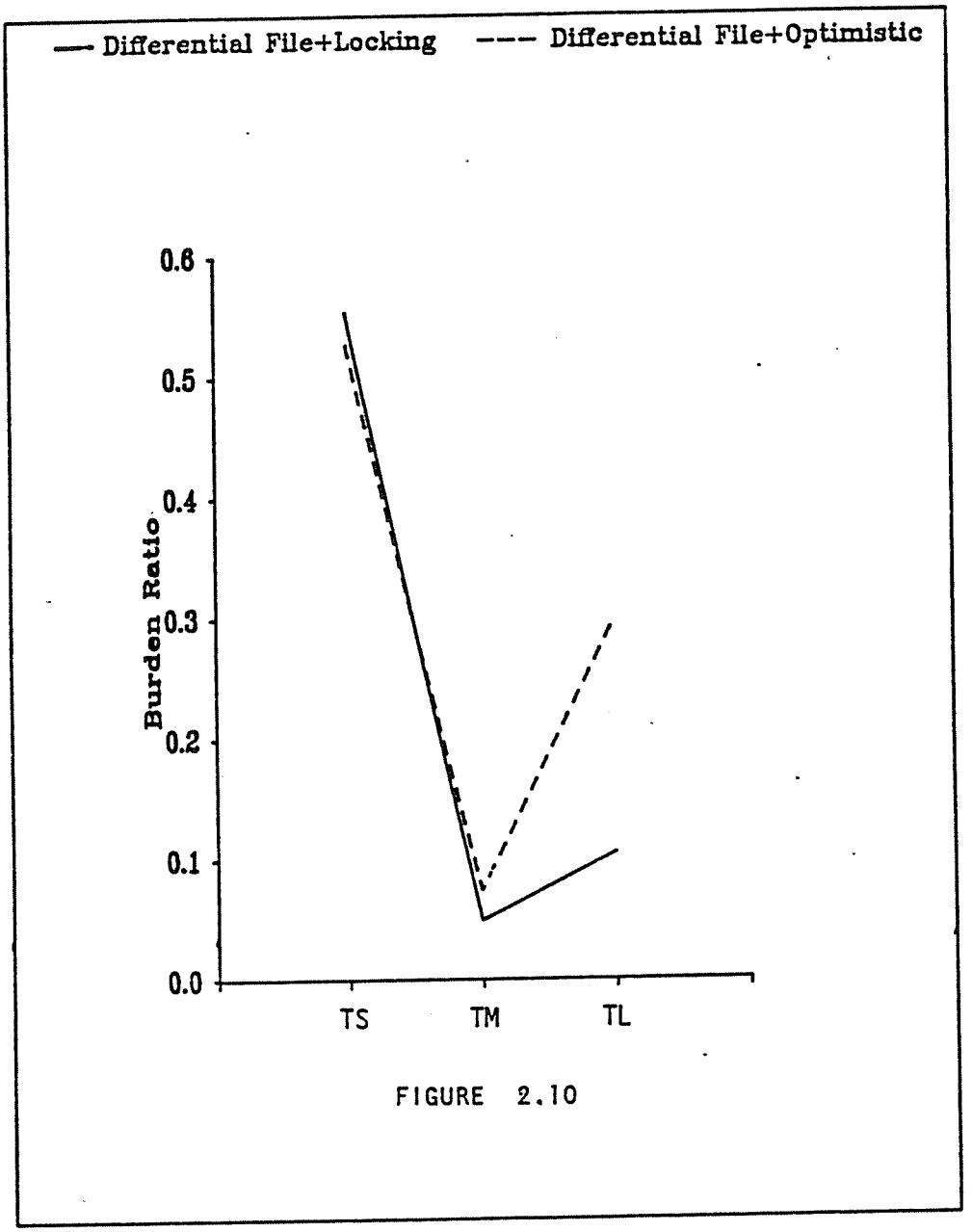
Differential File+Locking
Differential File+Optimistic
Burden Ratio
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.10

10%, that, CpuOH%, the extra cpu overhead equals 100%, (implying that, for example, T_{rec} is 1.0 ms. instead of 0.5 ms.), and that Comprs% equals 10% (i.e. a transaction writes ceil(0.1*NP_u) pages to both A_1 and D_1). One page-sized buffer was allocated for the base file and five each for the A and D files. We make the following observations based on Figure 2.10 and Table 2.5:

1. There is a considerable burden in accessing Size% extra data pages and extra CpuOH% processing. However, for the values assumed for Size% and CpuOH%, this burden is more than compensated by the savings that result from not writing the updated data pages as in an in-place updating algorithm.

2. Writing to the A and D files is akin to writing to the log and hence the performance characteristics of the differential file approach appears similar to that of the log approach. In particular, because of blocking effect while writing to the A and D files, TM performs better than TS. The burden ratio for TL becomes higher than for TM because of comparatively less savings in not writing the updated data pages[36]. In addition, the cost of waiting for the lock+differential file mechanism and the cost of transaction restarts in the case of optimistic+differential file mechanism increases considerably from the TM workload to the TL workload.

3. Overall, the differential+locking mechanism performs better than differential+optimistic mechanism for medium and large transactions due to the larger number of transaction restarts with optimistic method. Only for small transactions, where there are not many transaction restarts, does the performance of the differential+optimistic combination becomes comparable to the differential+locking combination.

---

[36] For the values assumed, TM updates 30% of the pages read while TL updates only 20% of the pages read.

2.9.3. Concurrency Control with Shadow for Recovery

The comparative performance of shadow+locking and shadow+optimistic mechanisms is shown in Figure 2.11 and Table 2.6. We assumed that DBuff=1 and SBuff=10. Each entry in the page table is assumed to take 4 bytes and thus the size of S-Map is 25 pages[37]. We make the following observations based on Figure 2.11 and Table 2.6:

1. The cost of reading and updating S-Map (the shadow page table map) constitutes the major portion of total burden.

2. The proportion of the cost of reading S-Map pages reduces with an increase in transaction size since more page-table entries can be found on the same S-Map page (see Table 2.7). The cost of updating S-Map increases for larger transactions because at the time of updating S-Map, PtPages(NP_u)-SBuff of S-Map pages are reread. However, the reduction in the cost of reading S-Map pages is much higher than the increase in the cost of updating the S-Map. This is the reason why in Figure 2.11, the burden ratio reduces with an increase in transaction size.

		Total Bur den	BSucc compo nent	BFail compo nent	BRerun compo nent	Locking			Set crea- tion	Valid ation	Smap read	Smap up- date
						total	wait	ddlk				
TS	Lock	119.8	117.8	2.0	0.0	2.0	0.1	0.0			71.8	36.1
	Opt	119.7	117.7	1.9	0.1				1.0	1.3	71.8	36.1
TM	Lock	1397.8	1365.1	32.5	0.2	75.3	26.4	0.2			805.5	486.5
	Opt	1511.0	1312.7	31.0	167.3				25.6	4.4	844.4	486.5
TL	Lock	2806.4	2732.6	62.6	11.2	549.2	304.1	1.2			939.2	1227.0
	Opt	5367.8	2304.3	51.0	3012.5				149.6	12.7	1145.8	1227.0

Table 2.6. Shadow+locking & Shadow+optimistic (random)

[37] DBSize = 100 million bytes ≃ 25000 pages and No. of S-Map entries per page ≃ 1000.

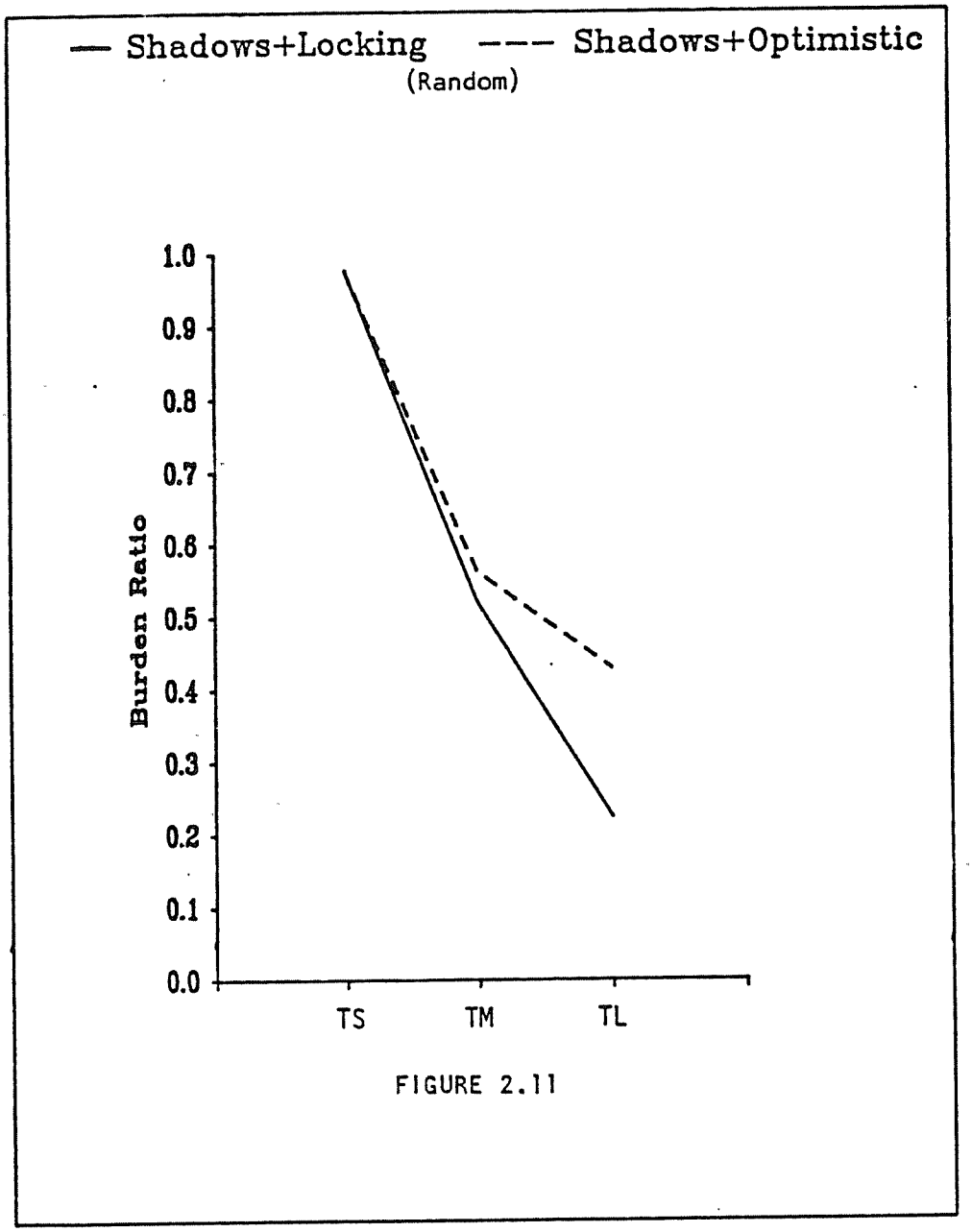
— Shadows+Locking
--- Shadows+Optimistic
(Random)
Burden Ratio
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.11

No. of Data Pages Accessed (N)	No. of Page Table Pages Accessed: PtPages(N)
1	1.0
2	1.96
15	14.48
50	21.75
250	25.00

Table 2.7. No. of accesses to the S-Map

3. Overall, the performance of shadow+locking and shadow+optimistic mechanisms are very similar since the cost of reading and updating S-Map (which is the dominant factor in the total burden) is independent of the concurrency control mechanism. For large transactions, the optimistic approach performs somewhat poorer because of the high cost of transaction restarts.

4. We also considered the case of sequential accesses to the database pages for TM and TL workloads. The performance results for this case are shown in Figure 2.12 and Table 2.8. The performance improves considerably because of large reduction in the cost of reading and updating the S-Map. The relative behavior of the locking and optimistic methods is similar to that of the random access case.

		Total Bur den	BSucc compo nent	BFail compo nent	BRerun compo nent	Locking			Set crea-tion	Valid ation	Smap read	Smap up-date
						total	wait	ddlk				
TS	Lock	119.8	117.8	2.0	0.0	2.0	0.1	0.0			71.8	36.1
	Opt	119.7	117.7	1.9	0.1				1.0	1.3	71.8	36.1
TM	Lock	259.3	252.9	6.3	0.1	75.3	26.4	0.2			75.1	78.4
	Opt	337.1	200.4	4.7	132.0				25.6	4.4	78.7	78.4
TL	Lock	810.4	780.8	19.7	9.9	549.2	304.1	1.2			75.2	95.1
	Opt	3181.8	352.5	8.1	2821.2				149.6	12.7	91.7	95.1

Table 2.8. Shadow+locking & Shadow+optimistic (sequential)

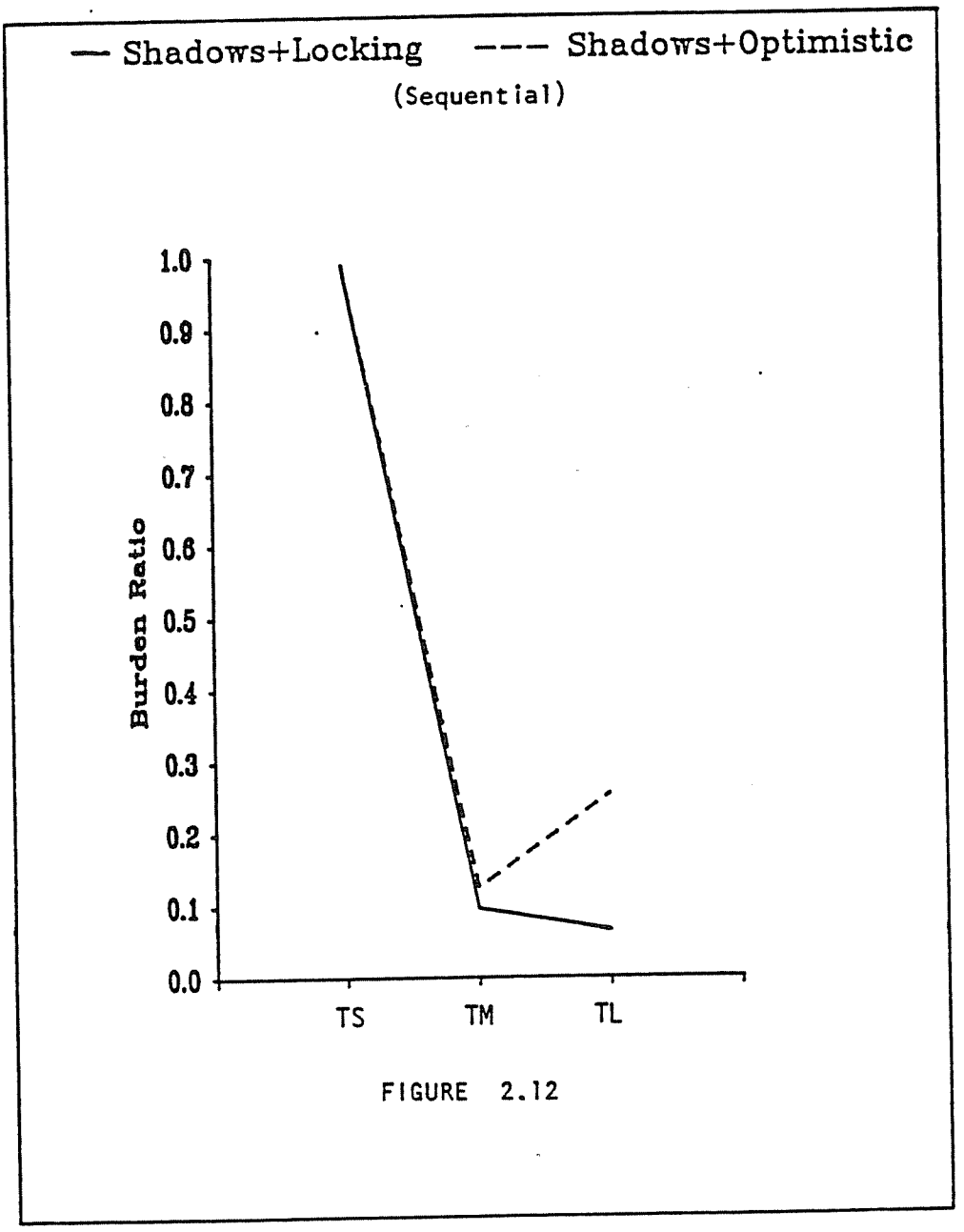
— Shadows+Locking
--- Shadows+Optimistic
(Sequential)
Burden Ratio
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.12

However, as pointed out in [Gray81b], a consequence of using shadows is that logically adjacent pages may not be physically adjacent. Thus, although accesses may be logically sequential, getting the next page may involve disk seek. [Lori77a] suggests a page allocation strategy that may maintain physical clustering of logically adjacent pages within a cylinder. We have assumed that the shadow mechanism employs such a scheme and we have not assigned any extra cost for potential disk seeks during sequential accesses.

2.9.4. Comparing the Finalists

The relative performance of the log+locking, differential+locking, and shadow+locking mechanisms is shown in Figure 2.13. We conclude with the following observations:

1. For small transactions, log+locking is the clear winner but for medium and large transactions, differential-file+locking also appears promising. As recovery mechanisms, the log and the differential file approach have many similarities. Both do not suffer from the one level of indirection found in the shadow mechanism. The A and D files in the differential file approach are in certain sense after-value and before-value logs. However, in the log approach a transaction, besides writing its log records, also writes to stable storage the updated data pages at the same time[38]. On the other hand, while the differential file approach must also write pages of the A and D files (that are like log pages) to stable storage, the actual updating of the data pages in the base relation (that is, merging of pages of the A and D files with the pages of B) can be deferred until a slack time.

---

[38] Writing of updated data pages may not be deferred to some slack time as the associated data buffers may have to be reallocated to another transaction.

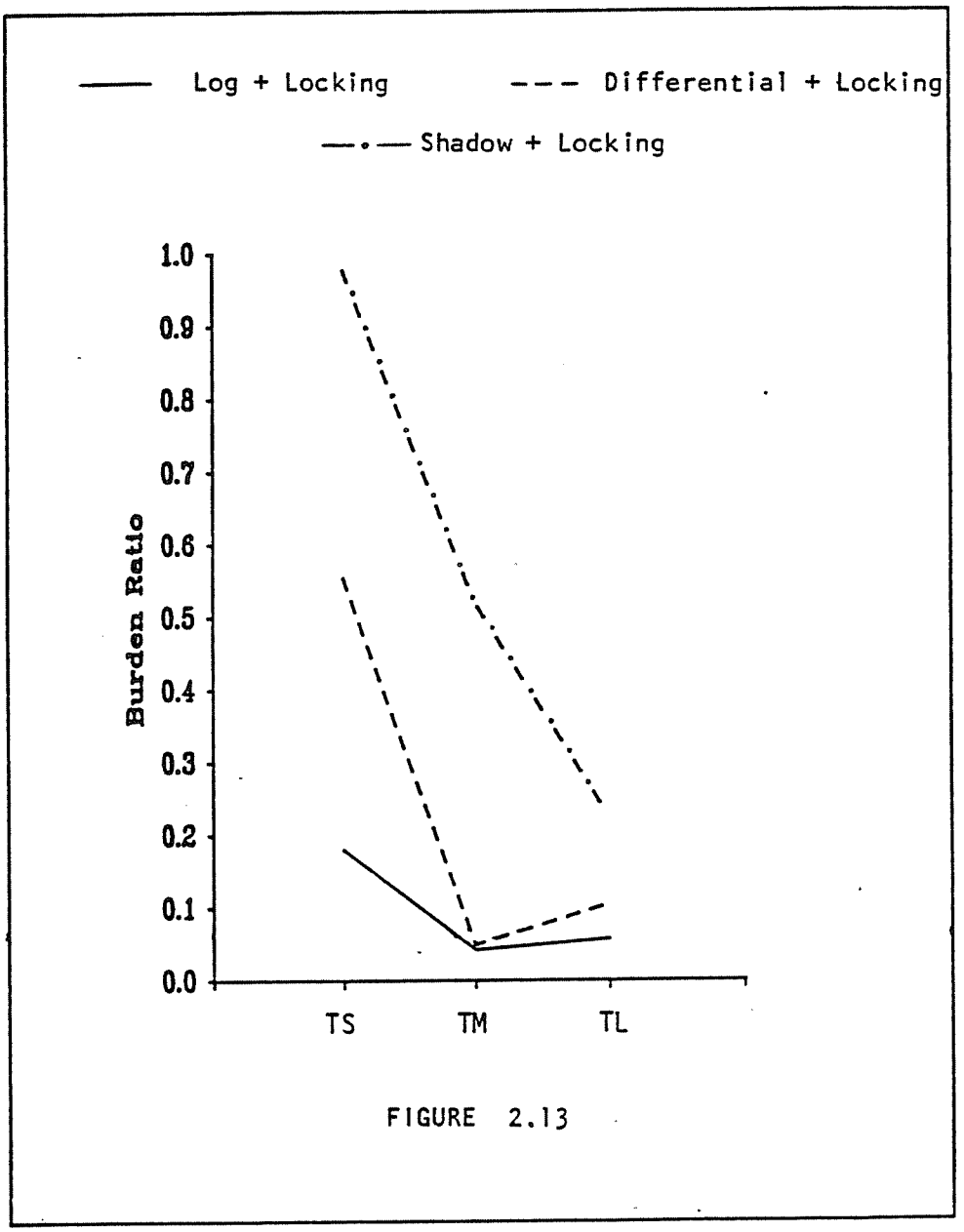
Log + Locking
Differential + Locking
Shadow + Locking
Burden Ratio
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.13

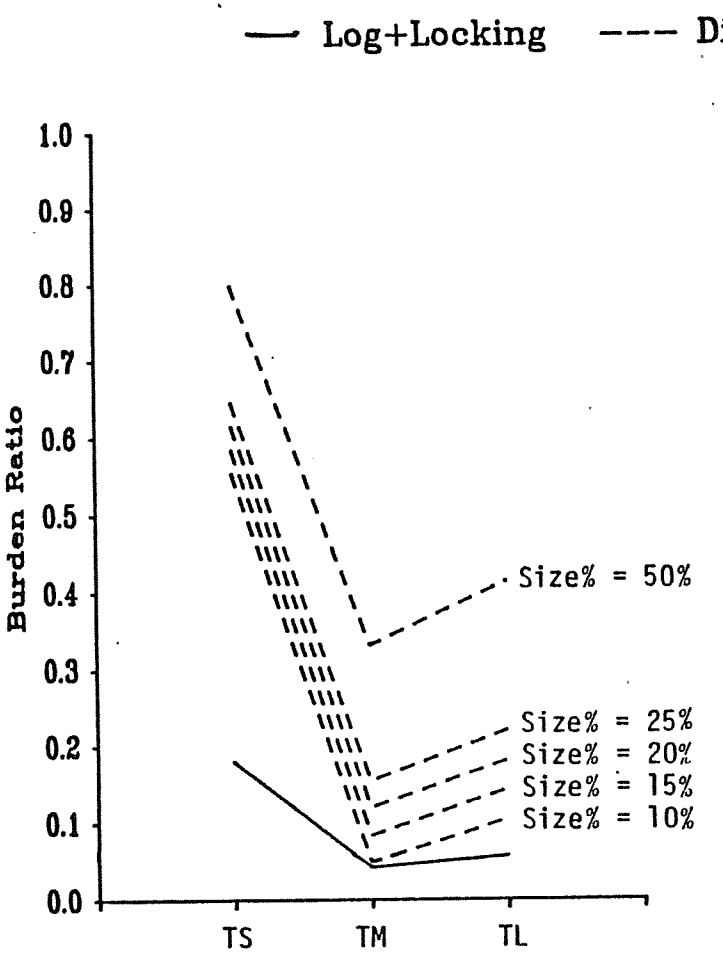

Figure 2.14 Effect of Size%

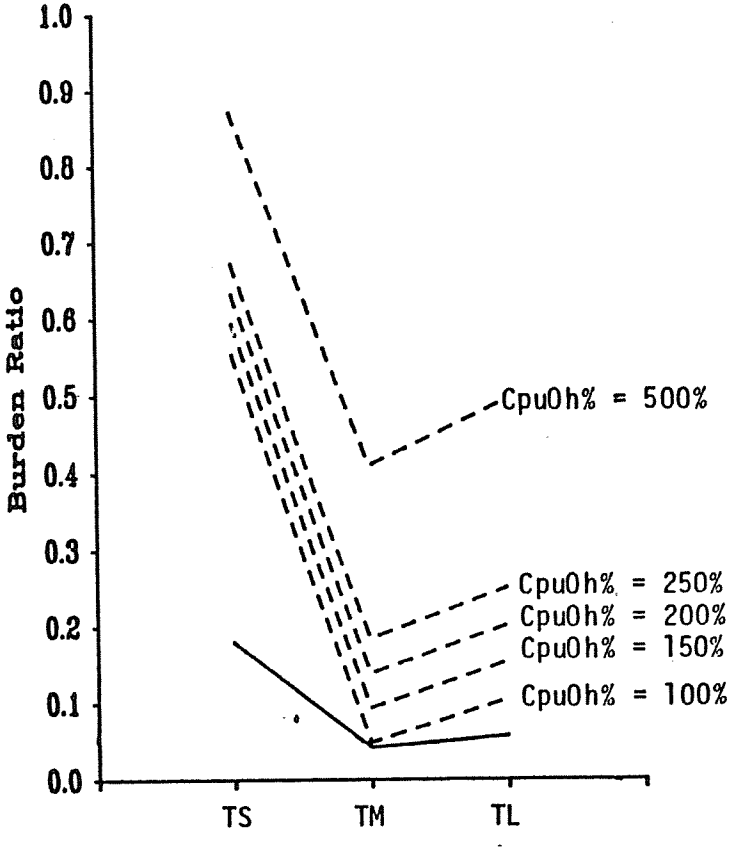

Figure 2.15 Effect of CpuOh%

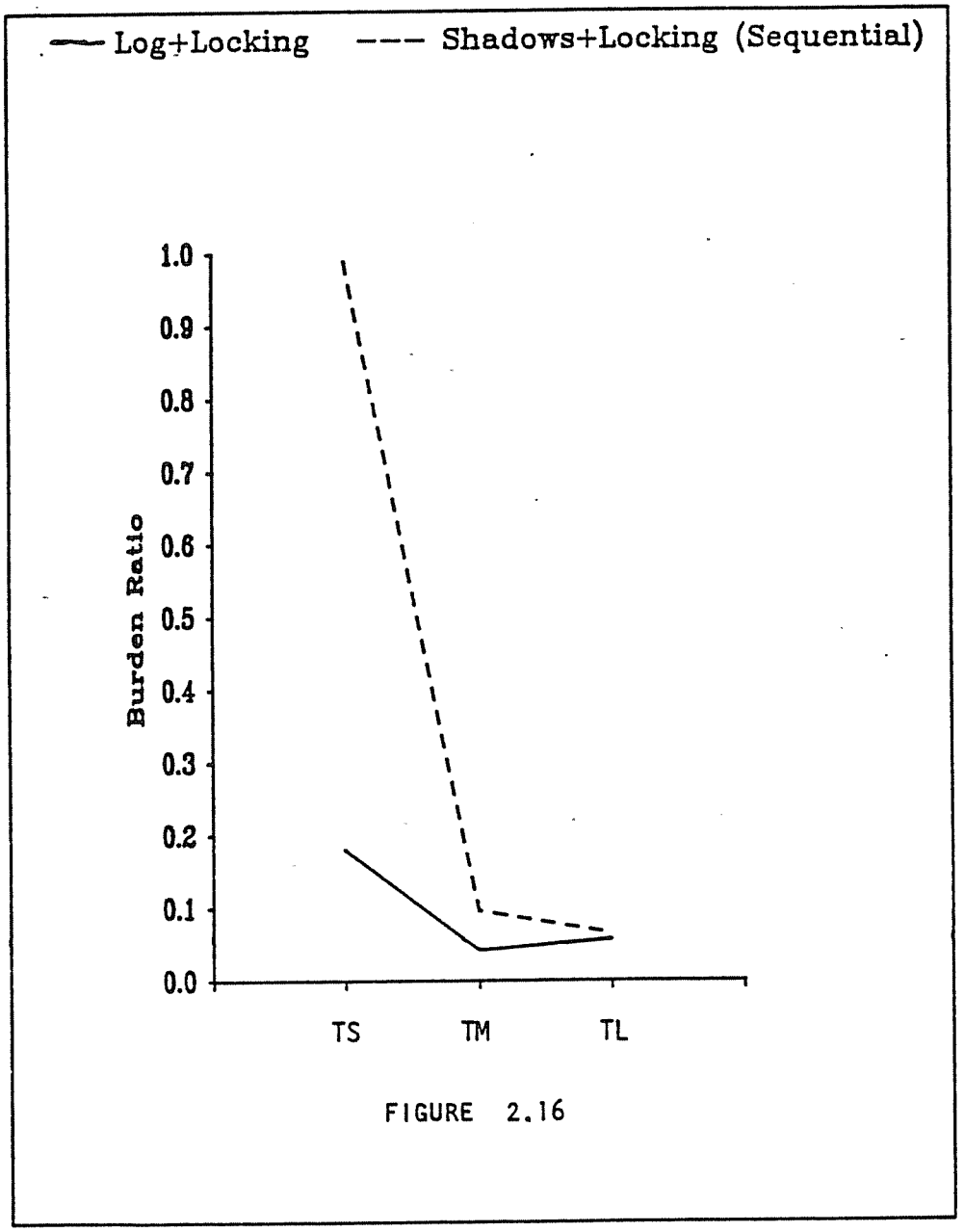
Log+Locking
Shadows+Locking (Sequential)
Burden Ratio
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TS
TM
TL

FIGURE 2.16

The disadvantage of the differential file approach is the cost of accessing Size%*N extra pages in order to access N data pages and the cpu processing overhead of CpuOH%. A sensitivity analysis we have performed indicates that its performance is very critically dependent upon the values of these two factors. Figure 2.14 and Figure 2.15 show the performance of the differential+locking mechanism for larger values of Size% and CpuOH% respectively. The performance degrades considerably for larger values of these parameters. In addition, the assumption that that the differential A and D files can be merged with the main file in slack time is crucial to the performance of this approach.

2. Figure 2.16 compares the performance of log+locking with shadow+locking when a sequential access pattern has been assumed for medium and large transactions. Only in case of large transactions, does their performance become comparable.

2.10. Conclusions

The choice of the "best" integrated concurrency control and recovery mechanism depends on the database environment. If there is a mix of transactions of varying sizes, log+locking emerges as the most appropriate mechanism. If there are only large transactions with only sequential access pattern, the shadow+locking mechanism is a possible alternative. In an environment of medium and large sized transactions, the differential+locking is a viable alternative to the log+locking mechanism.

The optimistic method of concurrency control induces a higher number of transaction restarts when compared to locking with deadlock detection. This is because locking resolves conflicts by blocking the conflicting transaction whereas the optimistic method resolves conflicts by aborting the conflicting

transaction. Furthermore, in the case of the optimistic method, nonserializability is detected after the transaction has run to completion, thus wasting total transaction processing. On the other hand, with locking, the deadlock detection is performed whenever a lock request conflicts. Therefore, if a transaction is to be aborted, it will be discovered relatively earlier. Thus, with the optimistic approach, not only are there a higher number of transaction restarts, but each restart is also more expensive. This factor is mainly responsible for the poorer performance of the optimistic combinations when compared to the locking combinations for medium and large transactions.

The optimistic method of concurrency control should only be considered in an environment where transactions are small with a very low probability of conflict. Even in a low conflict situation, if transactions are large and in-place updating is required, the cost of making local copies global will make the optimistic algorithm an expensive mechanism. The optimistic method can be attractive only in combination with a recovery mechanism that requires that all updates be collected in some scratch area and applied to the main copy only after a transaction has completed. Thus, recovery and concurrency control mechanisms may share the data structures and the cost of making local copies global.

Relatively, deadlock detection is so inexpensive that, in a locking scheme, it should be preferred over deadlock prevention which induces a larger number of transaction aborts.

Amongst recovery mechanisms, it is more expensive to do transaction undo with logging when compared to shadows or differential files. However, logging puts a smaller burden on a successful transaction. Since most of the transactions succeed rather than abort, logging emerges as the best

mechanism. The major disadvantage of shadows is the cost of indirection through the page table. This mechanism can become attractive only if the page table can always be maintained in the main memory or with an architecture that avoids this indirection. The disadvantage of the differential file approach is the overhead of reading differential file pages and extra cpu overhead to process a query. While the number of extra differential file pages that have to be processed depends on the frequency with which differential files are merged with the base file, parallel architectures may alleviate the cpu overhead problem.

CHAPTER 3

CONCURRENCY CONTROL AND RECOVERY MECHANISMS FOR DATABASE MACHINES

3.1. Introduction

In this chapter, we present our design of concurrency control and recovery mechanisms for the multiprocessor-cache database machine architecture described in Chapter 1. We discuss various design trade-offs and the choices we made. We also point out how our design can be adapted for other database machine architectures. The results contained in Chapter 2 from the study of concurrency control and recovery in centralized databases provided the basis for many of our design decisions.

The organization of the chapter is as follows. In Section 2, we present a summary of related research in this area. Section 3 and Section 4 contain our concurrency control and recovery designs respectively. Section 5 contains a summary of the chapter.

3.2. Review of Related Research

The only work in the area of concurrency control and recovery in database machines is a working paper by Cardenas, Alavian and Avizienis [Card81a] in which *recovery* in the processor-per-track database machine architecture is investigated. Recall that in the processor-per-track architecture, the mass storage consists of a number of data cells. A processor is associated with each cell and a back-end controller supervises the cell processors.

3.2.1. Recovery Mechanisms

Cardenas et al. analyzed three recovery mechanisms. The first two are special cases of the general logging mechanism. In the first mechanism, before committing a transaction, updated data records are modified in the data cells so that only a before-value log is required for transaction undo. In the second mechanism, updated data records are modified after the transaction has committed, so that only an after-value log is required for transaction redo. In both the mechanisms, there is *one* log for all the cell processors which is *centralized*. The log itself consists of a data cell and a cell processor, henceforth called the log processor.

In the third mechanism, before-value log records are interspread amongst data records. As we will see, the exposition of this mechanism is not complete, particularly in presence of system crashes. Each record is assigned one of the three following states: 1) present clean, 2) previous clean, or 3) present dirty. Initially, all data records are in the "present clean" state. When a data record is modified, a duplicate record with the state "present dirty" is created and the state of the original record is changed to "previous clean". All subsequent accesses are made to the record occurrence which is in the "present dirty" state. On transaction completion, the state of the updated records is changed from "present dirty" to "present clean" and then the records with the "previous clean" state are deleted. It is not clear from the paper when a transaction can be considered committed. There seems to be three alternatives: 1) after all the "present dirty" records have been inserted in the data cells, 2) after all the "present dirty" records have been changed to "present clean", or 3) after all the "previous clean" records have been deleted.

With the first alternative, to recover from a system crash, different recovery actions are required for records updated by committed and uncommitted transactions. For a committed transaction, "present dirty" records should be made "present clean" and the "previous clean" records should be deleted. "Previous clean" records of an uncommitted transaction should be made "present clean" and the "present dirty" records should be deleted. Thus, it is not sufficient to just record the state information with the records. It is required that the transaction identifier be encoded with the records and the back-end controller maintains sufficient information in stable storage to be able to do a "winner-loser" analysis [Gray81a]. For the second alternative, the recovery action is similar to the first alternative. The motivation for the third definition of commit is to avoid transaction redo. However, this definition is not safe. If the system crashes after some of the "previous clean" records for a transaction have been deleted, the transaction cannot be undone.

Distributed log records will be kept in the same cell as the corresponding data records to avoid excessive communication between the back-end controller and the cell processors. Thus, data cells will only be 50% utilized to handle the case when a transaction updates all records in a cell. Considering the criticism that the applicability of processor-per-track database machines is constrained to relatively small databases because of limitations on the size of data cells, this is a severe penalty.

3.2.2. Performance Analysis

The three recovery mechanisms were compared by computing the number of extra revolutions[1] of data cells required due to recovery. A *worst-case best-*

[1] A revolution is the traversal of entire data cell by the cell processor dedicated to it. For example, if a data cell consists of a disk-track, then a revolution of data cells will be the same as a revolution of the disk.

case approach has been adopted to compute extra revolutions. For example, in the first two mechanisms, collection of recovery data essentially consists of transferring log records from cell processors' memory to the log processor's memory and then inserting the records in the log. In the best case, all the records can be transferred in one revolution, and in the worst case, it will take min {#records,#data cells}[2] revolutions. Similarly, for inserting records in the log, one revolution is required in the best case and #records revolutions are required in the worst case.

Using this approach, the difference in the performance of the recovery schemes between the worst case and the best case was found to be very large, and the average case performance is difficult to ascertain. The impact of recovery on database machine performance has not been examined in the paper.

3.3. Concurrency Control Design

We will now present our design of concurrency control and recovery schemes. The concurrency control design has three basic components [Bern82a]:

(1) *location of the scheduler:* centralized or distributed,

(2) *type of algorithm:* locking, timestamp ordering or optimistic, and

(3) handling of *replicated data.*

3.3.1. Scheduler Location

There are two alternatives:

---

[2] Assuming that all the records in a cell can be transferred in one revolution.

(1) locate a scheduler with each query processor *(distributed)*, or

(2) locate the scheduler with the back-end controller *(centralized)*.

The major arguments in favor of distributed schedulers are:

(1) distributed schedulers are more robust (what if the node at which the centralized scheduler is located goes down?);

(2) a centralized scheduler may get saturated and become a bottleneck;

(3) communications overhead between the centralized-scheduler node and the processing node may become substantial, particularly if the network is geographically distributed *and* there is locality of reference.

In database machines, since there are a number of processing nodes, it is tempting to immediately conclude that the distributed scheduler is the right option. However, distributed schedulers presuppose that each processing node *owns* and *controls* the data it accesses. In the multiprocessor cache database machine architecture, there is no static binding of a processing node and the data it accesses; rather, pages are dynamically bound to processors at run time. Therefore, this architecture necessarily requires a centralized scheduler.

The scheduler may be located with the back-end controller or a separate processor may be entrusted with the task of concurrency control to whom the back-end controller may inquire before assigning a data page to a query processor. For example, the back-end controller in DIRECT has been implemented as three processes that may run on three separate physical processors [Bora82a]. One of these processes, designated MEM, is responsible for transferring data pages between the disk and the disk cache, managing data pages in the cache, and assigning them to the query processors. It would be logical to enhance MEM with the concurrency control scheduler.

In the processor-per-track, processor-per-head and processor-per-disk classes of database machines, the processors and the data pages accessed by each processor are statically bound. However, Garcia-Molina [Garc79a] and Robinson [Robi82a] have observed that unless the network connecting the processing nodes is geographically distributed, distributed concurrency control has no performance advantage over centralized concurrency control. On the other hand, distributed concurrency control has following disadvantages:

(1) the required process communication is much more expensive

(2) problems such as distributed deadlocks are difficult to handle efficiently

(3) system correctness is in general more often in doubt due to added complexity of the system.

Since the query processors and the back-end controller are closely-coupled in these architectures also, the concurrency control scheduler should be centralized with the back-end controller.

3.3.2. Concurrency Control Algorithm

Having settled on a centralized concurrency control scheduler, we draw upon the results of our evaluation of concurrency control mechanisms for centralized databases to decide on the concurrency control algorithm. As shown in Chapter 2, the behavior of basic timestamp ordering is very similar to that of locking as far as the effects of blocking are concerned. However, timestamp ordering induces a higher number of transaction aborts. The performance of the optimistic method and locking are comparable for small transactions, but for medium and large transactions, the optimistic method performs poorer than locking as there are a higher number of transaction restarts and each restart is more expensive. Database machines will be

primarily used in an environment where the transactions are large rather than small. For small transactions, the "Gray" argument (named after Jim Gray, a well-known proponent of this viewpoint) against the use of database machines applies. The argument states that a backend processor can only be effective when the work performed by the backend processor outweighs the cost of context switches and the cost of sending the request to the backend processor and receiving the response [Seli82a]. Therefore, we selected locking as the concurrency control algorithm.

Locking suffers from the problem of deadlocks. The folklore has it that the deadlock detection is expensive and thus timeouts or deadlock prevention techniques are used instead. With a distributed concurrency control scheduler, the deadlock detection could be expensive particularly if the cost of sending messages is high. However, for centralized schedulers, we have designed an efficient deadlock detection scheme [Agra83a] that has linear time and space complexity[3]. Using path compression [Aho75a], the linear check can be turned into an O(log N) check, where N is the number of blocked transactions [Skee83a]. We, therefore, propose using deadlock detection with locking as the savings that will result by avoiding unnecessary transaction restarts far outweigh the cost of deadlock detection.

3.3.3. Replicated Data

Replicated data is not an issue in our database machine architecture as data disks and query processors are not statically bound. A single image of data is kept on disks and data pages are paged from disk to the disk cache where they are assigned to the query processors.

---

[3] For completeness, the deadlock detection scheme has been presented in Appendix 2.

3.4. Recovery Design

Recovery has two aspects:

(1) collection of recovery data during the normal execution of the transactions,

(2) use of recovery data to perform necessary recovery actions in the event of a software or hardware failure.

All transactions, whether they complete or abort, incur the cost of collecting recovery data. However, only in case of a failure is the recovery data used. There is a trade-off involved here. A recovery mechanism may make collection of recovery data relatively less expensive at the price of making recovery from failures costly. On the other hand, a recovery mechanism may make recovery from failures cheaper but pay a large penalty during the collection of recovery data. Logging belongs to the first category whereas shadow and differential file techniques belong to the second category.

A simple but very important conclusion that may be drawn from the results in Chapter 2 is that from a performance viewpoint the focus should be on *making the normal case efficient*. Thus, although transaction undos are more expensive with logging, it outperformed the shadow or differential file mechanisms as it places a smaller burden on transactions that complete successfully, and in reality, a higher percentage of transactions complete than abort. Thus, in our design of recovery mechanisms, we have *optimized collection of recovery data even if it meant making recovery from a failure more expensive* while insuring that recovery can still be correctly performed.

3.4.1. Parallel Logging

The basic idea is to make the collection of recovery data efficient by allowing logging to occur in parallel at more than one log disk. We will show how

recovery from system crashes and transaction failures can still be performed correctly without physically merging the parallel logs into one log. We assume a page-level locking scheduler located with the back-end controller.

3.4.1.1. Architecture

We postulate N log processors, where $N \geq 1$. Each log processor has associated with it a log disk. We assume some interconnection between the query processors and the log processors that allows any query processor to send a log fragment to any log processor. We also assume a two way communication facility between the back-end controller and each of the log processors[4].

There may be a dedicated connection between the query and log processors, or the query processors may write the log fragments to the disk cache and then the log processors may read the fragments from there. We will investigate the effect of different interconnection strategies on the performance of the logging algorithm.

3.4.1.2. Data Structures

Recall that in our database machine architecture, the back-end controller is responsible for the movement of pages between the disk, the disk cache and the query processors. To manage this movement, the back-end controller needs some form of a page table. In addition to its normal contents, the page table contains status information for each page in the relation. This status information could be a dirty bit, for example. Our parallel logging algorithm requires that, for all updated pages, this status information be augmented with i) a bit, henceforth called the *log-flushed bit*, that indicates whether the

[4] The algorithm presented here can be trivially adapted to an architecture where there is only one log processor but more than one log disks are available to the log processor for logging.

corresponding log record has been written to stable storage, and ii) an identifier, henceforth called the *log processor identifier*, that indicates to which log processor the corresponding log record was sent for writing to stable storage.

3.4.1.3. Collection of Recovery Data

(1) To process a transaction, the back-end controller fetches the required data pages into the disk cache and assigns them to the query processors. We require that at the time of assigning a page, the back-end controller in addition communicate to the query processor the corresponding transaction number.

(2) When a query processor updates a page, it creates a log fragment for this page in its local memory. In addition to the before and after values, a log fragment contains the transaction number and the page number of the updated page. When the page has been completely processed, the query processor selects a log processor and sends it the log fragment. The *log processor selection* algorithms have been described below. The query processor then sends to the back-end controller the updated page and the identifier of the log processor to which the log fragment for this page has been sent. The back-end controller records the log processor identifier in its page-table entry for this page.

(3) The log processor assembles log fragments received from different query processors in a log page. When a log page is filled up, the log processor writes it to the log disk and sends the page numbers of all the updated pages that caused this log page to be created to the back-end controller. The back-end controller in turn sets the log-flushed bit for each of these pages.

(4) If the back-end controller is *forced* to flush an updated page when the cache gets completely filled up with updated pages, it first checks the log-flushed bit for the page to determine whether the log record for this page has been written to disk. If not, it sends a message to the corresponding log processor (using the log-processor id in the page table entry) to flush the log page. Only after the log processor acknowledges that the log page has been written to stable storage, will the data page be written to disk.

(5) At the time of committing a transaction, the back-end controller first ensures that all the log records of the transaction have been written to stable storage by checking the log-flushed bit of the updated pages. It then sends the commit record, augmented with the *commit-number*, to any one of the log processors. The commit-number gives the order in which the transactions have been serialized. The back-end controller can obtain the commit-number by simply appending the current time in its local clock to the current date. Another alternative would be to use a large sequential counter or append the counter to the current date. The locks held by the transaction are released only after the back-end controller receives acknowledgement from the log processor that the commit record has been written to the log disk.

Log Processor Selection

Query processors use one of the following algorithms for selecting the log processor to send the current log fragment:

(1) *Cyclic* selection: Each query processor cycles amongst all the log processors. That is, a query processor sends the first log fragment to the first log processor, the next fragment to the second log processor, and so on.

(2) *Random* selection: Each query processor uses a random number generator (with different seeds) to generate the log processor number every time it has to send a log fragment.

(3) *Query Processor Number mod Total Log Processors* selection: A query processor always sends its log fragment to the same log processor. The log processor is selected by taking the mod of its query-processor number with the total number of log processors.

(4) *Transaction Number mod Total Log Processors* selection: A query processor determines the log processor number by taking the mod of the transaction number with the the total number of log processors.

Except for the fourth algorithm, log fragments of a transaction will in general be distributed over more than one log processor. In Chapter 4, we will describe the results our evaluation of the performance of the different log-processor selection algorithms.

Comments

1. In our algorithm, a query processor sends a log fragment as soon as it has updated a page and the log processors are responsible for assembling log fragments into log pages. Another alternative would be that the query processors themselves assemble log fragments in their local memory and send only the full log page to the log processors. A simple analysis shows that this is not a good choice. Assume that there are 10 query processors and 1 log processor and updating a data page creates a log fragment that is 1/10 th of the size of the log page. If the second alternative were chosen then each query processor would have to update 10 data pages before sending the log page to the log processor. Thus, there may be 100 data pages in the cache waiting for the

corresponding log pages to be flushed. With the first alternative, there would be 10 data pages in the cache waiting for the log page to be flushed.

2. From the performance point of view, it is desirable that the back-end controller not force the log processors to flush out the log page. Therefore, the page-replacement algorithm for the disk cache may have to be enhanced so that it it ejects an updated page with the log-flushed bit on before it replaces an updated page for which the log page has not yet been written to disk.

3.4.1.4. System Checkpoint

We require the back-end controller and each log processor to independently maintain a current checkpoint number in their memory. When a log processor writes a log page, it appends its checkpoint number to the log page. Subsequently, when the log processor communicates to the back-end controller the page numbers whose log records were written, it also communicates the checkpoint number. The back-end controller in turn records the checkpoint number in the page table entries of these pages. We assume that the messages are not lost and are delivered in the order they are sent.

The checkpointing is coordinated by the back-end controller. Assume that the current check point number is N. The back-end controller first records transaction numbers of all the active transactions in its memory. It then broadcasts a checkpoint initiation message to all the log processors. At the same time, it starts writing to disk those updated pages whose log records have been written with the checkpoint number N.

On receiving the checkpoint initiation message, a log processor ensures that any log disk I/O in progress is completed and the corresponding page numbers of the updated pages are communicated to the back-end controller. It

then simply increments its current checkpoint number and sends an acknowledgement to the back-end controller. Incrementing the current checkpoint number signals checkpointing at that log processor. The log processor does not wait for the log page it is currently assembling to fill up or flush the partially filled log page before incrementing its checkpoint number.

After receiving acknowledgements from all the log processors, the back-end controller ensures that any updated page, whose checkpoint number in the page table is N, has been written to disk. It then sends a system checkpoint record to a predesignated log processor and increments its checkpoint number. The system checkpoint record contains transaction numbers of the active transactions that the back-end controller had earlier recorded and the current checkpoint number of the back-end controller. System checkpointing completes when the system checkpoint record is written to the log disk by the log processor.

This algorithm for checkpointing does not require a complete system quiescing and the checkpointing can be performed in parallel with the normal data processing and logging activities.

Establishing Checkpoint Locations

In our scheme, the system checkpoint record is written only on one log. In that log also, the location of the system checkpoint record need not correspond to the actual checkpoint location as the log processor may write log records while the back-end controller is flushing updated data pages for checkpointing. Thus, at the time of recovery from system crash, each log processor will have to establish the location of the checkpoint on its log.

To do so, the back-end controller obtains the most recent system checkpoint record and broadcasts it to all the log processors. Recall that the

system checkpoint record is written only at one predesignated log processor[5]. A log processor then finds its checkpoint location by scanning the log backwards from the end till it finds the the first log page that has the same checkpoint number as the number in the system checkpoint record.

To avoid scanning the log backwards, a log processor may, before incrementing its checkpoint number, save a pointer to the current end of the log in a fixed place on its log disk. However, the log processor will still have to compare the checkpoint number of the log page preceding the saved address with the checkpoint number of the system checkpoint record. This is necessary as the system may fail after a log processor has written the checkpoint location to its log disk but before the back-end controller completes its checkpointing operations. If the checkpoint number of this log page is greater than the checkpoint number of the system checkpoint record, a backwards scan to the earlier checkpoint location will be required.

3.4.1.5. Recovery from System Crash

Winner-Loser Analysis

(1) Each log processor makes its own list of winners and losers in parallel and sends it to the back-end controller. To do so, each log processor first establishes its checkpoint location as described in the previous section. It then initializes its loser list to all the active transactions whose number appear in the system check point record, and then scans its log in the forward direction starting from the checkpoint location. When the log

---

[5] The algorithm may easily be modified so that the back-end controller sends the system checkpoint record to any one or more than one log processors. In that case, the back-end controller will obtain from all the log processors their most recent system checkpoint records. The checkpoint record with the highest checkpoint number is the latest checkpoint record and the others may be discarded.

processor sees for the first time a log record for a transaction, it adds the transaction to the loser list, if the transaction is not already in the list. If the commit record of a transaction is found, then the transaction is moved to the winner list from the loser list.

(2) The back-end controller intersects the winner-loser lists received from different log processors and makes one final winner-loser list. Recall that for committing a transaction, the back-end controller sends the commit-record to only one log processor. Thus, a transaction which is a winner in any one of the lists sent by the log processors is the winner in the final list even if it is a loser in other lists.

Transaction Redo

The algorithm for the collection of recovery data physically splits the log in as many pieces as the number of log processors. One way of doing transaction redo would be to first merge these distributed log pieces to create one log in which the log records appear in the same order in which they would have appeared if only one log processor was used, and then perform the redo. This solution, besides being inefficient, would require that the timing information be associated with each log record, and the clocks of different query processors be kept synchronized.

Instead, we propose that the back-end controller take one log at a time and carry out redo. While this approach is very attractive as it does not require distributed logs to be merged into one physical log, it suffers from the following problem:

The Problem: Suppose that an object X is updated from x0 to x1 by the transaction T1 and to x2 by the transaction T2, and T1 commits before T2.

Further assume that the log record <x0,x1> is on the log L1 and the log record <x1,x2> is on the log L2. It has to be ensured that after redo processing, X = x2 and not x1 which would happen if the redo processing using log L1 is done after the redo processing using log L2.

The Solution: Define a data structure <X,t(X)> where t(X) is the commit-number associated with the commit record of the transaction that updated the object X. Observe that the commit-number information for the committed transactions can be collected at the same time as the winner-loser analysis. Furthermore, each log record has associated with it the transaction number and the page number that caused this log record to be created. Thus, a log record contains all the information necessary to access this data structure.

With this data structure, transaction redo is performed using the following algorithm:

Initialize:
for all X[7] do t(X):= 0

Redo Algorithm:
for all logs do {
scan the log forward
for each log record x_i with the corresponding commit-number t_i do
if t_i < t(X) then ignore this log record
else redo and t(X) := t_i }.

In the above example, assume that the commit-number of T1 is 1 and that of T2 is 2. Thus, if log L2 is processed first, then t(X) := 2 after processing the log record <x1,x2> and X = x2. Now when the log L1 is processed, the log record <x0,x1> will be ignored because the corresponding commit-number, that is 1, is

---

[6] Standard techniques like hashing [Knut73a] may be used for efficient access to this data structure.

[7] In an actual implementation, initialization will involve nulling only the hash table entries.

less than the current value of t(X) = 2.

Transaction Undo

For transaction undo, the back-end controller scans one log at a time backwards. At first, it seems that the transaction undo will have a problem similar to redo.

A Hypothetical Scenario: As before, assume that an object X is updated from x0 to x1 by the transaction T1 creating the log record on log L1. Then, X is updated to x2 by the transaction T2 creating the log record on log L2. Both T1 and T2 are found to be losers in the winner-loser analysis. It must be ensured that after undo X is restored to x0 and not x1 which would happen if undo processing using the Log L1 is done before undo processing using the log L2.

The Solution: This situation cannot arise with proper locking protocol. If T2 has updated X after T1 has updated it, then T1 must have released its lock on X. But a transaction does not release its locks before its commit record has been written to the log and T1 is a loser.

Therefore, the transaction undo can be performed simply by taking one log at a time, scanning it backwards, and restoring before values of the uncommitted updates.

3.4.1.6. Recovery from a Transaction Abort

The back-end controller determines the log processors where the log records corresponding to the pages updated by the aborted transaction exist by examining the page table entries of these pages. An updated page that has not yet been written to disk is ignored in this analysis as this page may be undone by simply discarding its updated version from the disk cache. The back-end

controller then obtains the log records for the transaction from each log processor and performs the undo. The order in which the back-end controller communicates with the different log processors is immaterial as the log record for a page updated by a transaction exists only on one log. In the next section, we will describe how the algorithm for transaction undo is modified if a page may be updated more than once by a transaction, and hence different log records for the same page may exist on more than one log.

3.4.1.7. An Embellishment

A transaction frequently consists of more than one database operations. For example, in System R, a transaction consists of one or more SQL statements bracketed with Begin_Transaction and Commit_Transaction commands [Gray81a]. We will call each of these operations a transaction-step. We assume that all the steps have been numbered in the increasing order. Suppose now that a transaction updates the object X from x0 to x1 by step 1 and to x2 by step 2. It is required that at the time of transaction redo, x2 is restored and not x1. Similarly, transaction undo should restore x0 and not x1.

To handle this situation, we require that the back-end controller, at the time of assigning a data page to a query processor, in addition to the transaction number, communicates the step number also to the query processor. The query processor appends the step number along with the transaction number to the log fragment before sending it to the log processor.

For *redo processing*, the only modification required is that in the data structure <X,t(X)>, t(X) is now defined to be the commit-number appended with the step number of the corresponding transaction. In the above example, assume that during transaction redo, X is first restored to x2 and t(X) := (1,2) where 1 is the commit-number of the transaction and 2 is the step number.

Subsequently, the log record created by step 1 of the transaction will be ignored as the step number appended to the transaction number for this log record, that is (1,1), is less than the current value of t(X).

Undo processing requires building a similar data structure and using an algorithm similar to that of the transaction redo. The only difference required in the algorithm is that the decision rule about when to ignore a log record is changed. For undo, the log is scanned backwards, and if the commit-number appended with the step number of the transaction that created the current log record of X is *greater than* the current value of t(X), then this log record is ignored; otherwise, the undo is performed and t(X) is updated. Thus, in the above example, if X is first undone to x0 and t(X) := (1,1), then subsequently the log record created by step 2 will be ignored as the transaction number appended with the step number for this log record, that is (1,2), is greater than the current value of t(X).

3.4.2. Shadow

In Chapter 2, our evaluation indicated that the major reason for the poor performance of the shadow mechanism was the cost of *indirection* through the page table to access data pages. We have identified two approaches to improve its performance:

(1) reduce the penalty of indirection,

(2) avoid indirection altogether.

3.4.2.1. Reducing the Penalty of Indirection

The performance penalty of indirection through the page table may be reduced by keeping page tables on more than one page-table disk (different from data disks), and using separate *page-table processors* for them. The page-

table processors will be under the control of the back-end controller. Thus, if the accesses are uniformly distributed over the page-table processors, then the processors may work in parallel reducing degradation in the performance due to indirection.

This solution requires the back-end controller to know the mapping between a relation's page table and the page-table processors. However, this information may be cached in the memory of the back-end controller. More significantly, this information will not be frequently modified and it need not survive system crash. Even if the information becomes corrupted while being updated, it may be reconstructed by making the data pages self-descriptive. The major problem with this solution is how to decide the distribution of page tables between the page-table processors.

Recall that in the shadow mechanism, for committing a transaction, a precommit record containing the disk addresses of the updated pages is first written to stable storage, and then the page table is updated. We propose that the precommit records of all the transactions be written on one page-table disk. It is possible to devise a scheme so that if a transaction updates page tables on more than one page-table disk, then a precommit record is written on each of the disks. However, this will require coordination similar to two-phase commit in writing the precommit records. In addition, for a random transaction, for all the accesses to the page-table disks to read or update page-table entries, only 1 access will be required to write the precommit record. Thus, maintaining only one precommit list would avoid considerable system implementation complexity without creating imbalance in the usage of the page-table disks.

3.4.2.2. Avoiding Indirection

We will describe two approaches that avoid indirection:

(1) *Version Selection:* Both the current and shadow copies are retrieved in response to a read request. A version selection algorithm is then applied to decide which is the current copy. It is expected that the cost of retrieving both the current and the shadow copies and doing version selection would be less than the cost of going through a page table to retrieve only the current copy.

(2) *Overwriting:* Separate shadow and current copies are kept only while the transaction is active. On transaction completion, the shadow copy is overwritten with the current copy.

3.4.2.2.1. Version Selection

We will present a scheme based on [Reut80a]. This scheme uses the basic disk property that because of the relatively long time required for head positioning and rotational delays, accessing an additional disk block on the same track requires only a small amount of additional time. For example, in IBM 3350 disk drive [IBM77a], accessing a random 4 kbyte page on average takes 37.525 ms. Accessing the subsequent block also will only require additional 4.175 ms. We assume a page-level locking scheduler located with the back-end controller.

Each pair of physically adjacent blocks on a disk track constitute a *slot*. A slot contains the physical representation of one logical page. At any time, one of the blocks contains the current version and the other contains the shadow.

For reading a page, both the blocks of the corresponding slot are brought into the disk cache and a version selection algorithm identifies the block that contains the current version. For writing an updated page, the block other than

the one that contains the committed current version is used. Thus, the use of a block in a slot alternates between holding the shadow and the current copy of a page.

Version Selection Algorithm

Store with every page updated by a transaction a timestamp composed by supplementing the transaction no. with the hardware timer at the time of writing the page. The current version is simply then the block with the higher timestamp.

Recovery from Transaction Aborts

Maintain for each transaction a list of pages updated by the transaction[8]. Augment this list with a one bit entry for every page to indicate which block of the corresponding slot contains the version written by this transaction. If a transaction aborts, the blocks containing versions created by this transaction are rewritten with timestamp = 0. As the version selection algorithm selects the version with the larger timestamp, the copies written by the aborted transaction will not be selected.

Recovery from System Crash

Recovery from system crash requires maintaining a list of all uncommitted transactions which should survive system crash. Recovery involves scanning all the disk blocks and if any block is found which has been written by an uncommitted transaction, it is rewritten with timestamp = 0. In [Reut80a], a scheme has been described for chaining together the pages updated by a uncommitted transaction to avoid scanning the whole database.

---

[8] Such a list would be required anyway for efficiently releasing the locks held by the transaction.

Comments

Note that this scheme requires reading of two adjacent disk blocks for every retrieval but only one block is written for an updated page. In the standard shadow mechanism, besides reading and writing the data page, a page-table page is potentially read for each read and updated for every write. The significant price that the new algorithm pays is in *doubling* the disk space requirement.

Instead of one extra disk block for each data page, a smaller number of blocks may be shared amongst data pages on a disk track by adding page numbers to the disk blocks and retrieving the whole track in order to select the current version. In an on-the-disk database machine architecture, in addition, retrieval may be restricted to the blocks containing the desired pages by adding the page number to the search condition which is applied by the processor. This approach has been used in [Room82a] for the design of a content-addressable page manager. The design provides extra blocks for shadowing on per cylinder basis. It uses parallel-search disks with capabilities similar to the processor-per-head architecture, and in one revolution, all the versions of a page are obtained and then the appropriate version is selected. A problem with these solutions is that overflow will occur if a transaction updates many adjacent pages and special procedures are required to handle overflow situations.

3.4.2.2.2. Overwriting

We will present two algorithms: one requires no redo and the other requires no undo at the time of recovery from a system crash[9]. Both algorithms require scratch space on disk which is managed as a ring buffer.

---

[9] The phrases no-redo and no-undo have been adopted from [Bern82b].

No-Redo Algorithm

Before updating a data page, write the original of the page (shadow) in the scratch space. Commit a transaction only after all its updates have been written to disk. A list of uncommitted transactions is also needed that should survive system crash. Recovery essentially requires restoring of shadows from the scratch space.

A recovery scheme similar to this has been implemented in the Wisconsin Storage System [Chou83a].

No-Undo Algorithm

Unlike the No-Redo algorithm, the shadow pages are kept in their original location while a transaction is active. All the updated pages are first written to the scratch space and then the transaction is considered committed. Locks are, however, released only after the updated pages have replaced the shadow pages. A list of committed transactions whose updates have not yet overwritten the shadows is required to survive system crash. Recovery actions are obvious.

Comment

The above algorithms may be extracted as special cases from the general logging mechanism by equating the before value to the shadow copy and the after value to the current version and by assuming page-level physical logging instead of logical logging.

3.4.3. Differential File

Recall that in the differential file approach, a relation R is considered a view $(B \cup A) - D$, where B is the base relation, A and D are differential relations, additions to R are appended to the A relation, and deletions to R are appended

to the D relation. In Chapter 2, it was found that the major cost of the differential file approach consisted of two components:

(1) I/O cost of reading extra pages from A and D relations,

(2) Extra cpu processing cost as a simple retrieval gets converted into a set-union and a set-difference operation.

The number of extra A and D pages read to process a query is a function of the size of A and D relations and depends on the frequency of update operations and the frequency with which A and D relations are merged with B relations. However, parallelism inherent in our database machine architecture may be exploited to efficiently process the set-union and the set-difference operation. We will present parallel algorithms for operations on hypothetical relations.

3.4.3.1. A Problem and A Solution

Before presenting the algorithms, let us first point out a major limitation of the differential-file approach. In hypothetical relations, it is impossible to insert a tuple that has previously been deleted, because the effect of a tuple in D can never be undone as it applies to the result of (B ∪ A). The same problem makes it impossible to update a tuple to have a previous value.

We have proposed a simple solution to this problem in [Agra83b]. The solution requires that each transaction is assigned a unique timestamp and that the tuples in A and D relations are widened to have an extra field TS. All appends to A and D relations put the timestamp of the updating transaction in this field[10].

The semantics of the set-difference operator is modified as follows. Let tuple $t_a \in A$ and tuple $t_d \in D$. Normally, if t_a.all = t_d.all, then t_a does not belong

[10] This change incurs no extra burden because the mechanism for recovery from system crash imposes these requirements anyway (see [Ston81a]).

to (A - D). In our approach, we will allow $t_a \in (A - D)$, although t_a.all = t_d.all, if t_a.TS > t_d.TS. With this modification, a previously deleted tuple cannot eliminate a newly inserted tuple as the timestamp of the deleted tuple would be less than the timestamp of the new tuple.

3.4.3.2. Parallel Algorithms

We will now present parallel algorithms for database operations on hypothetical relations. The database operations have been presented in QUEL-like [Ston76a] syntax. In the following discussion, assume that the range of (r,b,a,d,t) are (R,B,A,D,Temp) respectively.

RETRIEVE

The operation

retrieve r where Q(r)

is translated into

(retrieve b where Q(b) ∪ retrieve a where Q(a))
- retrieve d where Q(d)

Algorithm:

1. Give a different page of B (henceforth called, the outer page) to each query processor (When B is exhausted, give different pages of A).
2. Each query processor with an outer page initializes an array of booleans, Eliminated[No of tuples on the page], to False.
3. While all pages of D (henceforth called, the inner pages) have not been exhausted do{

   Broadcast an inner page to all the processors;

   Each processor executes subalgorithm 1 or 2 in parallel}.
4. Each processor does the following in parallel:

for each tuple, o-tup, in the outer page

if (Q(o-tup) is True) And (Eliminated[o-tup] is False) then output **o-tup.**

Subalgorithm 1: tuples within a page are sorted[11]

```
o-tup := beginning of the outer page;
i-tup := beginning of the inner page;

while the outer page is not completely scanned do {
      while (Key(i-tup) < Key(o-tup)) do i-tup := Next(i-tup);
      save-i-tup := i-tup;
      while ((Key(i-tup) = Key(o-tup) do{
            if (i-tup.TS > o-tup.TS) then Eliminated[o-tup] := True;
            i-tup := Next(i-tup);}
      i-tup := save-i-tup;
      o-tup := Next(o-tup);}
```

Subalgorithm 2: tuples within a page are not sorted

```
o-tup := beginning of the outer page;
i-tup := beginning of the inner page;

while the outer page is not completely scanned do {
      if ( Q(o-tup) is True) then {
            while the inner page is not completely scanned do {
                  if ((Key(i-tup) = Key(o-tup) And (i-tup.TS > o-tup.TS))
                        then Eliminated[o-tup] := True;
                  i-tup := Next(i-tup);}
      o-tup := Next(o-tup);}
```

Comments

1. Complexity of the retrieval algorithm is $O[(|D|*(|B|+|A|)/N) * T]$, where X is the size of the relation X in pages, N is the number of processors and T is the time for one execution of the subalgorithm. Subalgorithm 1 would generally require only one pass over each page of B (or A), and one pass over D page. Subalgorithm 2, on the other hand, requires one pass over each page of D for every tuple of B (or A).

2. Certain optimizations are possible in the basic algorithm described above.

[11] See [Bora80a] on how tuples in a page may be kept sorted.

For example, D may be first restricted and the resultant tuples may be projected on the Key and TS fields. The resultant relation, which is likely to be quite small compared to D, may then be used for the set-difference. Similarly, B & A may also be restricted before the above algorithm is applied.

DELETE

The operation

delete r where Q(r)

is translated into

append to D(b.all) where Q(b)
append to D(a.all) where Q(a)

Algorithm:

1. Query processors perform selection in parallel on pages of B and A.
2. The back-end controller appends the result tuples to D.

Comments

1. The complexity of the algorithm is $O[(|B|+|A|)/N]$.

APPEND (Unqualified)

The operation

append to R ({col-i = value-i})

is translated into

append to A ({col-i = value-i}).

Algorithm:

1. The back-end controller can handle this operation alone with out any assistance from any query processor.

APPEND (Qualified)

The operation

append to R ({col-i = Hi(r)}) where Q(r)

may be translated in two different ways:

1. [a] append to D ({col-i = Hi(d)}) where Q(d)
   [b] append to A ({col-i = Hi(a)}) where Q(a)
   [c] append to A ({col-i = Hi(b)}) where Q(b)

or,

2. [a] retrieve into Temp(r.all) where Q(r)
   [b] append to A({col-i = Hi(t)})

Algorithm:

Translation 1:

1. Query processors perform selections of A and D (steps 1[a] and 1[b]) in parallel. The back-end controller appends the transformed resultant tuples to A and D respectively.
2. Query processors do the selection of B (step 1[c]) and the back-end controller appends the resultant transformed tuples to A.

Translation 2:

1. Query processors perform retrieval using the retrieve algorithm presented before.
2. The back-end controller appends the transformed result tuples to A.

Comments

1. The query processors may do the necessary transformation on tuples resulting from selection (Translation 1) or set-difference (Translation 2), before they send them to the back-end controller for the append step. The tuples may be transformed while being written to the output page in the processor's local

memory.

2. The time complexity of the algorithm with first translation is $O[(|D|+|A|)/N + |B|/N)$. This time complexity is much better than the complexity with the second translation. However, the first translation may result in a larger increase in the size of the differential relations.

REPLACE

The replace operation

replace r({col-i=Hi(r)}) where Q(r)

also may be translated in two different ways:

1. [a] append to D (d.other,{col-i = Hi(d)}) where Q(d)
   [b] append to D (a.all) where Q(a)
   [c] append to A (a.other,{col-i = Hi(a)}) where Q(a)
   [d] append to D (b.all) where Q(b)
   [e] append to A (b.other,{col-i = Hi(b)}) where Q(b)

or,

2. [a] retrieve into Temp(r.all) where Q(r)
   [b] append to D(t.all)
   [c] append to A(t.other,{col-i = Hi(t)})

Algorithm:

Translation 1:

1. Query processors restrict the D relation and the back-end controller appends the transformed result tuples to D.
2. Query processors restrict the A and B relations in parallel. The back-end controller appends the result tuples to D and the transformed result tuples to A.

Translation 2:

1. Query processors do retrieval using the retrieve algorithm presented before.

2. The back-end controller appends the result tuples to D and the transformed result tuples to A.

Comments

1. The algorithm with the first translation requires only one liner scan of each of A,B and D. For steps 1[b] and 1[c], A is restricted only once and the query processors output both the selected tuples as well as the transformed tuples for appending to A. Similarly, 1[d] and 1[e] are done together. In addition, (1[b],1[c]) and (1[d],1[e]) are executed in parallel. Therefore, the time complexity of the replace algorithm with the first translation is $O[|D|/N + (|A|+|B|)/N]$. Thus, the first translation is much more time efficient than the second translation. However, the first translation causes a larger number of tuples to be appended to the differential relations.

3.5. Summary

In this chapter, we presented our design of concurrency control and recovery mechanisms for database machines. We showed that in the database machines environment, a centralized scheduler located with the back-end controller using locking with deadlock detection is the appropriate choice for concurrency control. We presented a scheme for parallel logging that allows recovery data to be logged in parallel on more than one log disks. We showed how recovery can be correctly performed using distributed logs without merging them into one physical log. We also described how the performance of the shadow recovery mechanism may be improved for use in a database machine. We described a scheme that reduces the penalty of indirection through the page table and schemes that avoid indirection altogether. Finally, we presented parallel algorithms for operations on hypothetical relations to alleviate the cpu-

overhead problem associated with using differential files. In the next chapter, we will investigate the performance of these different recovery options.

CHAPTER 4

PERFORMANCE EVALUATION

4.1. Introduction

In this Chapter, we present the results of our simulation experiments to study the performance of the parallel recovery mechanisms described in Chapter 3 and their impact on the performance of a database machine.

We begin by explaining in Section 2 why we used simulation instead of queue theoretic modeling for analyzing the alternative mechanisms. We then present the structure of the simulation model for the bare (i.e. without recovery) database machine in Section 3. The stability experiment we performed to decide when to stop the simulation and some results from the simulation of the bare database machine are also described in this section. Sections 4 through 6 contain the results of our performance experiments with the logging, shadow, and differential file mechanisms respectively. In Section 7, we present our conclusions.

4.2. Performance Evaluation Methodology

We used simulation to experiment with the different recovery mechanisms and investigate their effect on database machine performance. We did not use queue theoretic modeling because that would have required analysis of queueing systems with the following characteristics:

Multiple Resource Holding: In our database machine architecture, to fetch a data page from disk, a cache frame is first reserved for the page and then the page is read. Thus, during reading of a data page, the page simultaneously

possesses a disk and a cache frame. Another instance of multiple resource holding is during the time when a page is read from cache to a processor's local memory.

Blocking: A query processor may be blocked if it has to output a page and no cache frame is currently free.

Parallelism: A transaction "splits" itself into several data pages and these pages are processed in parallel by different query processors. In logging, updated data pages, in addition, spawn off log fragments and an updated data page waits in the cache for the corresponding log fragment to be written to stable storage. The transaction in turn waits for all the log fragments created by it to be written to stable storage before it writes the commit-record and terminates.

Nonproduct-form Disciplines and Distributions: Servers in the model of a database machine use a variety of service disciplines that are not product form[1]. For example, allocation of a cache frame to a query processor to output a page has priority over allocation for reading a page from disk. Priority service disciplines are not product form. Furthermore in database machines, service time distributions of some servers depend on the state of another server. For example, at a log processor, the time between receiving a log fragment and its assembly into a log page depends on factors such as how many query processors were updating some data page at that time, size of the log fragments created by the query processors, and how full was the log page when the log fragment arrived at the log processor.

---

[1] Two most common product-form service disciplines [Chan78a] are 1) *processor sharing* (i.e. when there are n customers in the service center each receiving service at the rate of 1/n sec/sec), and 2) *infinite server* (i.e. when the number of customers in a service center never exceeds the number of servers).

As observed in [Chan78a], obtaining exact solutions for queueing systems with the above characteristics is computationally *intractable* as these systems do not satisfy local balance [Bask75a] and therefore are not amenable to efficient exact solution techniques like convolution [Denn78a] or mean value analysis [Reis78a]. Approximate methods have been devised to solve queueing systems that have some of the above features (see the survey in [Chan78a]), but it is not clear how to use these techniques to analyze a queueing system that embodies all the above characteristics together. In addition, analytic error measures for approximate techniques are extremely difficult to obtain [Jaco82a] and approximations may introduce an *unknown* amount of error [Saue81a].

4.3. The bare Machine Simulator

We adopted a modular approach to simulation. As a first step, a simulator for the *bare* (i.e., without any recovery mechanism) database machine was built. Subsequently, modules for the different recovery mechanisms were "appended" to the bare machine simulator.

4.3.1. Model Description

The structure of the bare machine simulator is shown in Figure 4.1. When a transaction arrives for processing, its size in terms of number of pages and the reference string is generated. For each page accessed by the transaction, it is determined whether the page will be updated, whether the page will be found in the cache (e.g., a page of an intermediate result) or will have to be read from a disk, and how much query-processor time will be required to process this page[2].

---

[2] These parameters are determined for each page at the time of the arrival of a transaction so that a page has identical routing and processing time across different simulation experiments.

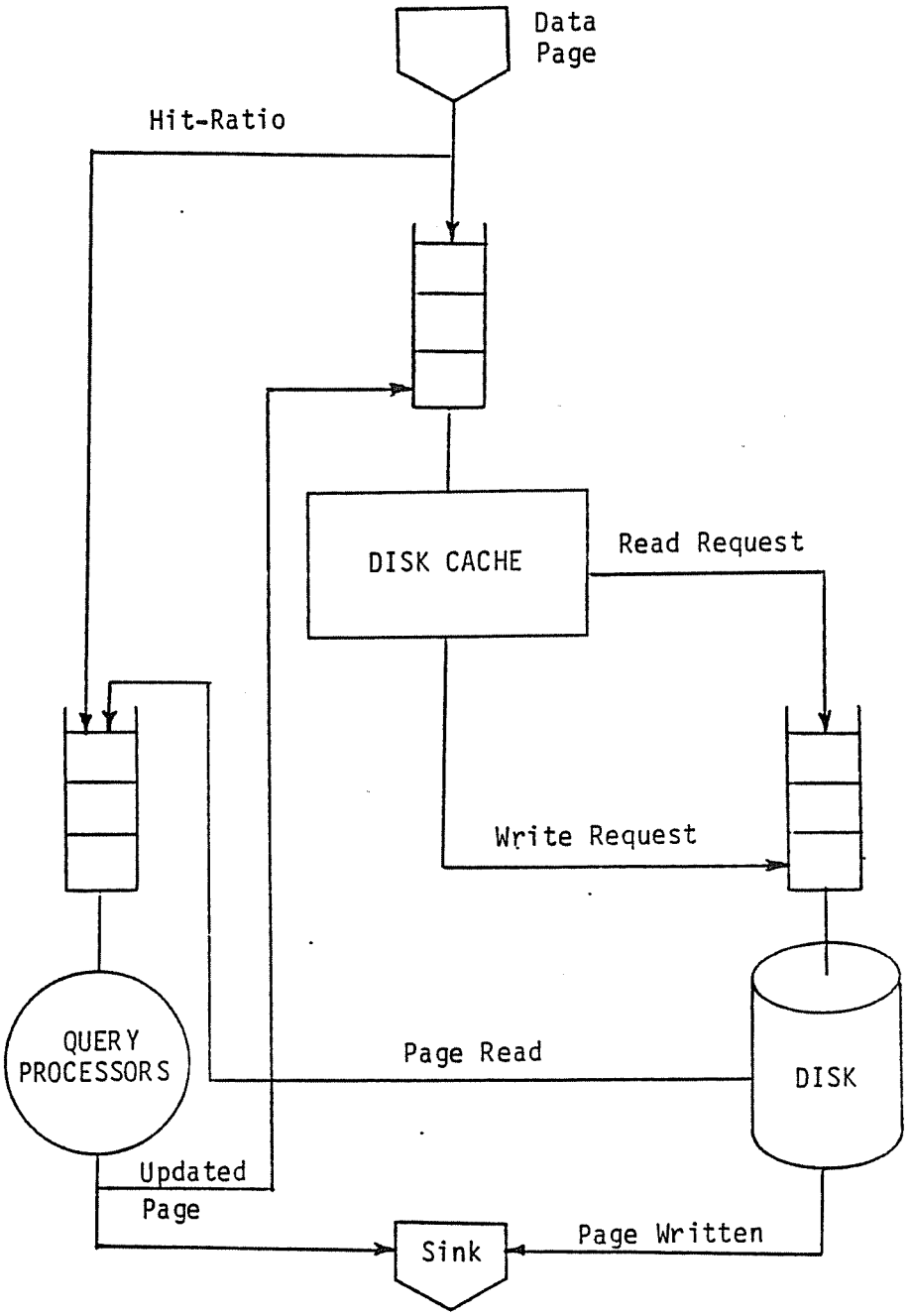

Figure 4.1 Bare Machine Simulator

Pages accessed by the transaction are put in the queue for the disk cache. When a page is allocated a cache frame, it is put in the disk queue. After the page has been read from disk, the page can be assigned to a free query processor and it is put in the query-processor queue[3]. If a page is determined to be already in the cache, then it is immediately put in the query-processor queue.

When a processor becomes free, it is assigned the page which is first in the processor queue. The processor reads this page into its local memory and releases the cache frame that was holding this page. If the page is read-only, the processor becomes free after processing the page. If the page is updated, the query processor requests a cache frame to output the updated page. The cache request for sending an updated page from a processor's local memory has priority over the request for prepaging data pages from disk. When a cache frame is allocated to the updated page, the processor writes the page to the cache. The processor is now available to process the next page.

The updated page is put in the disk queue for writing it to disk. In the disk queue, updated pages have priority over the pages to be read. After all the pages of a transaction have been processed and the updated pages have been written to disk, the transaction terminates.

Observe that the query execution in our simulation model is page driven which is similar to the data-flow approach proposed in [Bora81a]. We also model anticipatory paging or prepaging [DeWi79a] of data pages.

---

[3] There is one queue for all the processors as all the query processors are identical and a page can be processed by any one of them.

4.3.2. Model Characteristics

Transactions

A transaction is modeled by the number of pages it accesses. The size of a transaction is a uniform discrete random variable in the range 1 to 250. The size of a data page is 4096 bytes.

The reference string of a transaction can either be random or sequential. To generate the random reference string, the page number and the disk on which it resides are randomly selected for a page. The page number is between 1 and the maximum number of pages on the disk. For a sequential reference string, the first page is randomly determined. Subsequent pages are determined by incrementing the page number of the first page. All pages accessed by a sequential transaction reside on one disk which is determined by taking the mod of the transaction number with the total number of disks. In one simulation experiment, all the transactions are assumed to be either random or sequential to isolate the effect of access pattern on performance.

In database machines, all the pages processed in response to a query are not necessarily read from disk. Some of the pages may be found in cache. For example, if a query consists of restricting a relation followed by a join of the result relation with another relation, then it is likely that at the time of join some fraction of the pages produced by the selection operation will be present in the disk cache. The size of a transaction is the total number of pages processed by the transaction. To estimate the percentage of pages of a transaction that will be found in cache, we instrumented the simulation of the database machine DIRECT [Bora81a] and found the average cache-hit ratio to be 20%. We used this number to determine whether a page will have to be read from disk or it will be found in cache.

The write set of a transaction is a random subset of its read set. In most of our experiments, the percentage of pages updated by a transaction was taken to be 20% of the total number of pages accessed by the transaction.

It was ensured by using different random number streams for generating different parameters that the *size* and the *reference string* of a transaction were *same across different simulations*.

Query Processors

Query execution in a database machine is characterized by a mix of simple operations like restrict and complex operations like join. To estimate the average time to process a page, we again instrumented the simulation of DIRECT [Bora81a] and determined the average times for the restrict and the join operations on a page. The average processing time for a page was then estimated by weighting these timings with the corresponding number of pages. Thus, 36.65 ms. was determined to be the average time to process a page. In a recent benchmark of the commercial version of INGRES running on a VAX 11/750 [Bitt83a], the average cpu time per page for a selection operation was measured to be 27 ms. and the time per page for a join operation was found to be 46 ms. Thus, 36.65 ms. seems to be a reasonable estimate of the average time required to process a page. Time taken by a query processor to process a particular page is assumed to be normally distributed with the standard deviation equal to 1/3 of the average processing time.

The transfer rate between a processor's memory and disk cache was assumed to be 1/2 megabytes per second [Bora82a].

For most of our experiments, we assumed a database machine configuration consisting of 25 query processors.

Mass Storage Devices

The mass storage devices were modeled after the IBM 3350 disk [IBM77a]. This disk has 30 recording surfaces, 555 cylinders and 4 blocks of 4096 bytes each on every track. The revolution time is 16.7 ms. and it takes 10 + 0.0772*N ms to move the head by N cylinders.

To determine the access time for a disk page, the cylinder number on which the page resides is first computed. The access time is then computed by accounting for the seek from the current head position to this cylinder plus latency and the transfer time.

We have also modeled a parallel-access disk. On such a disk, pages on different tracks of the same cylinder may be read and written in parallel. Disk requests are serviced by determining the cylinder number of the page which is first in the disk queue and then examining the whole queue to find if there is another page belonging to the same cylinder. All such pages are then accessed in parallel.

We assumed 2 disk drives for all our experiments.

Disk Cache

The disk cache is addressable at page boundaries of 4096 bytes pages. The unit of access is a full page. For most of the experiments, a disk cache consisting of 100 page frames was assumed. We assume that the transfer rate between the cache and the processors, and between the cache and the disk drives is limited only by the bandwidth of the processor's bus and the disk transfer rates respectively. Thus, the disk cache has been modeled as a *passive* resource [Saue81a]. The disk cache does not have any service time associated with it; only the number of pages that may be active in the database machine

are limited by the size of the disk cache.

4.3.3. Stability Experiment

A principal problem in simulation is determining *when to stop* a simulation run. If a simulation is run for an insufficient period, it may lead to erroneous results. A general rule is to stop when the measurements of interest are within a predetermined confidence interval. However, successive observations in a simulation run are often positively correlated. Therefore, the classical statistical methods for determining confidence interval that are based on the assumption of independent and identically distributed measurements do not apply. Three methods have been advanced to circumvent this problem (see Sarg76a).

1. *Replication:* Make k independent simulation runs (replications) by using a different stream of random numbers and the same initial conditions.
2. *Batch Means:* Make one simulation run and divide the observations from this run into k batches.
3. *Regenerative Method:* Divide a simulation run into a sequence of independent and identically distributed blocks at the regenerative points. A regenerative point of a system is a state to which the system returns at intervals of finite length.

The regenerative method is mathematically most rigorous [Chan78a]. However, the regenerative method is not always practical because a large number of simulations do not have regenerative points [Sarg76a], particularly, if there are saturated servers [Schw76a]. The method of batch means is considered less rigorous than replication [Saue81a]. We, therefore, chose the method of replication to determine when our simulator can be considered to have run "long" enough.

We made 5 replications each for the following configurations:

1. *Conventional-Random:* conventional disks and random transactions,
2. *Parallel-Random:* parallel disks and random transactions,
3. *Conventional-Sequential:* conventional disks and sequential transactions,
4. *Parallel-Sequential:* parallel disks and sequential transactions.

There were 25 query processors, 100 disk cache frames and 2 disk drives for each configuration. Each replication was run for 500 transactions.

Table 4.1 shows the average execution times per page in milliseconds for the five replications. The average *execution time per page* is defined to be the time taken by the database machine to execute a given transaction load divided by the total number of pages processed by the machine, and is a measure of the *throughput* of the machine. Total pages processed is given by $\Sigma|T_i|$ where $|T_i|$ is the size of the transaction T_i in number of pages.

It can be seen from the execution times in Table 4.1 that our simulator was in the steady state on completion of 500 transactions. In all our future experiments, therefore, a simulation run was stopped after executing 500 transactions. The future simulations were run with the seeds corresponding to the first replication.

Configuration	Replication				
	# 1	# 2	#3	# 4	# 5
Conventional-Random	18.00	17.75	17.82	17.82	17.89
Parallel-Random	16.62	16.44	16.50	16.55	16.56
Conventional-Sequential	11.01	9.53	10.10	9.92	9.97
Parallel-Sequential	1.92	1.96	1.91	1.91	1.92

Table 4.1. Average Execution Time per Page

4.3.4. Conclusions

The major conclusion that we made from the simulation of the bare database machine is that its performance is severely limited by the I/O bandwidth.

Tables 4.2 and 4.3 contain various performance statistics related to the disk drives and the query processors respectively corresponding to the first replication. Unless otherwise stated, *the unit of time in this chapter is milliseconds*. In Table 4.2, Q-length is the average number of disk requests pending to be serviced and Q-time is the average waiting time before a disk

Configuration	Disk 1					Disk 2				
	Q length	Q time	Utili-zation	Total I/Os	Access time	Q length	Q time	Utili-zation	Total I/Os	Access time
Conventional-Random	69.48	2470.21	1.00	29905	35.55	26.32	952.24	0.98	29265	35.51
Parallel-Random	53.39	1753.53	1.00	27671	35.49	42.11	1408.07	1.00	27565	35.49
Conventional-Sequential	48.80	1055.76	0.76	30048	16.48	46.18	1031.05	0.74	29122	16.55
Parallel-Sequential	14.19	53.67	0.92	7066	14.85	13.23	51.64	0.91	6925	15.01

Table 4.2. Disk Characteristics

Configuration	Q length	Q time	Max QPs Used	Max Utili-zation	Effec-tive QPs
Conventional-Random	0	0	9	0.15	0.51
Parallel-Random	0	0	10	0.15	0.56
Conventional-Sequential	0	0	17	0.16	0.82
Parallel-Sequential	54.47	104.77	25	0.91	22.38

Table 4.3. Processor Characteristics

request is taken up for servicing. Access time is the average time for transferring a page between the disk and the cache and consists of seek, latency and the transfer times. Total I/Os is the total number of disk requests serviced by the disk drive. In the case of a parallel-disk, if more than one page is transferred in parallel, it is considered to be one I/O.

In Table 4.3, Q-length is the average number of data pages resident in the disk cache waiting to be assigned to a query processor and Q-time is the average time that a page stays in cache before it is assigned to a free processor. Max Qps Used is the number of maximum query processors in use at any time. Max Utilization is the maximum utilization of any one of the query processors. For computing the utilization of a query processor, in addition to the time a query processor is performing an operation on a page of data, the time during which a query processor reads a page into its local memory or writes a page to cache, the processor is also considered to be in use. Effective Qps is the sum of the utilizations of all the query processors.

Based on the results in Tables 4.1 through 4.3, the following observations can be made:

1. The average execution time is highly dependent on the type of the disk drive and the access pattern. A parallel disk may fetch more than one data pages in one disk access. Thus, for random accesses, the execution time using parallel disks is less than the time required when using conventional disks as a total of 55,236 disk I/Os are made with the parallel disks compared to 59,170 I/Os required with the conventional disks. In the conventional-sequential configuration, the same number of I/Os as in the conventional-random configuration are required but the execution time improves as the average access time is reduced by more than one half.

The execution time improves dramatically when parallel disks are used for sequential accesses. The improvement is the result of both the reduced access time and significantly fewer total I/Os (only 13,991 I/Os are required in the parallel-sequential configuration). We conjecture that a disk scheduling algorithm such as the shortest-seek-time first, instead of first-come first-serve, will further improve the execution time.

2. Parallelism in the database machine is severely limited by the I/O bandwidth. With conventional disk drives, although there were 25 processors, 16 of them were never used when the access pattern was random. Out of the 9 processors that were used, none of them were in use for more than 15% of the time and the sum of the utilization of these processors was only 51% of a single processor. When the accesses were sequential, 8 processors were never used, none of the processors were used more than 16% of the time, and the sum of the utilization of the 17 processors was only 81% of a single processor. Even the parallel disk did not alleviate the problem when the accesses were random. 15 processors remain unused, maximum utilization of any one of these processors was 15% and the sum of the utilization of the 10 processors was 56%. Only with parallel disks and the sequential access pattern, could all the processors be effectively used.

In contrast to processor utilization, the utilization of both the conventional as well as the parallel disks was nearly 100% for random accesses, and there were long disk queues with large waiting times. For sequential transactions also, the disk utilizations were very high.

The lower numbers for disk utilization in the case of sequential access compared to the random access is explained by the following. We assumed that a sequential transaction accesses all the pages from the same disk and the cache frames are allocated to the transactions on first-come first-serve basis.

Thus, if there is a sequence of large transactions that all access, say, Disk 1 interspread with small transactions that access Disk 2, then a situation will arise where Disk 2 becomes idle whereas there is a queue at Disk 1.

3. Parallel disks do not necessarily solve the I/O bandwidth problem. If the accesses are random so that at a time there are not many pages belonging to the same cylinder in the disk queue, the parallel-accessing capability of a parallel disk becomes redundant. However, if the accesses have sequential access pattern, parallel disks could be extremely useful.

4.4. Parallel Logging Simulator

We augmented the bare database machine simulator with the parallel logging of recovery data. We did not model transaction aborts in the simulation as our performance evaluation study in Chapter 2 showed that the major overhead of logging was in the collection of recovery data. Compared to successful transactions, transaction aborts are relatively too infrequent to have any significant effect on the performance results.

4.4.1. Specifications of the logging module

For most of our experiments, we assume *logical* logging, that is, the before and after values of only the updated records on a page are logged. In *physical* logging, the complete images of the page before and after the update are saved. With logical logging, we assume that the size of a log fragment is normally distributed with average size equal to 1/10 of the size of the data page and standard deviation equal to 1/3 of the average size. Extra query processor time required to create the log fragment is also assumed to be normally distributed with average time equal to 1/10 of the time to process a data page and standard deviation equal to 1/3 of the average time.

We have modeled two types of communication between the query processors and the log processors:

1. *Through an interconnection network:* There is an interconnection network between the query processors and the log processors (distinct from the interconnection network between the query processors and the cache), devoted to the task of transmitting log fragments. This network could be a shared bus, for example, or there could be a dedicated connection between each query processor and each of the log processors. We have modeled the interconnection network as a delay device. The delay between sending a log fragment and its receipt at a log processor depends on the size of the fragment and the *effective* communication bandwidth of the interconnection. The effective bandwidth takes into account the loss of bandwidth because of the contention. By choosing appropriate values for the communication bandwidth, different interconnections may be simulated. For most of our experiments, we assume an effective communication bandwidth of 1 megabytes per second.

2. *Through the disk cache:* A query processor, on creating a log fragment, requests a cache frame for it, and after a frame is allocated, writes the fragment to cache. A log processor reads the fragment from cache to its memory where it assembles log fragments into log pages before writing them to disk. A query processor may not be reading or processing data pages while it is writing a log fragment to the cache. Similarly, a log processor may not be assembling log fragments while it is reading a log fragment from cache. The time to transfer a log fragment between a processor and the cache is a function of the size of the log fragment and the processor-cache bandwidth which is constrained by the processor bandwidth of 1/2 megabytes per second.

We assume that the log processor takes 2 ms. to copy a log fragment received from a query processor to the log page it is assembling in its log buffer. This time is based on the average size of the log fragment and the instruction execution times for VAX 11/750. Time to send/receive a message between a log processor and the back-end controller is 2 ms. We further assume that while a page is being written to the log disk, the log processor may receive/send messages or assemble log fragments into log pages. The log disk specifications are based on IBM 3350 disk drive.

4.4.2. Experiments

We performed a number of simulation experiments to determine the characteristics of our parallel logging algorithm and its impact on database machine performance. The questions that we attempted to answer from these experiments include,

* Effect of logging on the average execution times of the database machine?
* When is it worthwhile having more than one log processor?
* Performance of various log processor selection algorithms (cyclic, random, query processor number mod total log processors, and transaction number mod total log processors)?
* Effect of the communication medium between the query processors and log processors?
* Effect of routing the log fragments through the disk cache?
* Effect of smaller block size for the log disk?
* Effect of the log fragment size?
* Effect of the percentage of pages updated by a transaction?

Effect on Database machine Performance

Tables 4.4 through 4.7 show the effect of logging on different performance parameters of the database machine. These experiments were performed with 25 query processors, 100 disk cache frames and 2 data disks. There was 1 log processor and the effective bandwidth of the interconnection network between the query processors and the log processor was taken to be 1 megabyte/second. In Table 4.4, transaction completion time is defined to be the time from the allocation of the first cache frame to a transaction to the writing of the last page

Configuration	Execution Time per Page		Transaction Completion Time	
	Without Log	With Log	Without Log	With Log
Conventional-Random	18.00	17.86	7398.41	7543.20
Parallel-Random	16.62	16.50	6476.04	6649.90
Conventional-Sequential	11.01	11.39	4016.46	4333.46
Parallel-Sequential	1.92	2.05	758.06	862.24

Table 4.4. Performance of Parallel Logging

Configuration	Disk 1						Disk 2					
	Without Log			With Log			Without Log			With Log		
	Utilization	Total I/Os	Access time	Utilization	Total I/Os	Access time	Utilization	Total I/Os	Access time	Utilization	Total I/Os	Access time
Conventional-Random	1.00	29905	35.55	1.00	29905	35.29	0.98	29265	35.51	0.98	29265	35.28
Parallel-Random	1.00	27671	35.49	1.00	27641	35.27	1.00	27565	35.49	1.00	27576	35.18
Conventional-Sequential	0.76	30048	16.48	0.76	30048	16.91	0.74	29122	16.55	0.73	29122	16.95
Parallel-Sequential	0.92	7066	14.85	0.92	6893	16.07	0.91	6925	15.01	0.91	6868	15.97

Table 4.5. Data Disk Characteristics
(one log disk)

Configuration	Without Log		With Log	
	Max QPs Used	Effective QPs	Max QPs Used	Effective QPs
Conventional-Random	9	0.51	10	0.55
Parallel-Random	10	0.56	13	0.59
Conventional-Sequential	17	0.82	18	0.84
Parallel-Sequential	25	22.38	25	23.87

Table 4.6. Query Processors Utilization
(one log disk)

Configuration	WAL		Log Processor		
	Q length	Q time	Disk Utilization	Total I/Os	Fragment Wait Time
Conventional-Random	3.92	367.06	0.02	1258	374.60
Parallel-Random	3.92	342.00	0.02	1257	349.88
Conventional-Sequential	4.31	251.52	0.02	1254	254.22
Parallel-Sequential	3.16	47.39	0.13	1255	49.86

Table 4.7. Log Characteristics
(one log disk)

updated by the transaction to disk[4]. In Table 4.7, WAL Q-length is the average number of updated pages in the disk cache that are waiting for the corresponding log records to be written and WAL Q-time is the average waiting time. Fragment wait time is the time between the receipt of a log fragment at a

[4] As discussed in Chapter 2, a transaction can be considered to have completed as soon as its commit record is written to stable storage. However, to make a comparative study of the different recovery mechanisms and their impact on the database machine performance on a *uniform* basis, we will consider a transaction to be completed only after all the changes in the database state because of the transaction have become permanent (written to disk).

log processor and the initiation of I/O to write the fragment to the log disk. We make following observations based on Tables 4.4 through 4.7.

1. Although logging causes the average transaction completion time to increase, the throughput of the database machine in terms of execution time per page is not significantly degraded. With the recovery architecture that we have proposed, assembly of log fragments into log pages and writing them to the log disk is completely *overlapped* with the processing of data pages, and therefore, does not affect the execution time. The effect of logging manifests in two ways:

i) Some updated pages are blocked in the disk cache for the corresponding log records to be written. This is the main reason for the increase in transaction completion times. The execution times are, however, not affected because the blocking of updated pages does not cause the disks or the processors to become idle. The blocked pages may hinder anticipatory reading of other data pages as they keep the corresponding cache frames occupied. This will happen only if cache frames are scarce and the blocked pages are large in number. In our experiments, more cache frames were allocated for anticipatory paging than the disks could feed, and on average, there were less than 5 pages in the WAL queue.

ii) Extra query processor time is required to create log fragments. However, except for the parallel-sequential configuration, the query processors were very poorly utilized in the bare database machine and the extra processing required to construct the log fragments did not increase processors' utilization significantly.

Thus, the logging did not significantly affect the execution times as the extra work required for logging could be done either in parallel with the processing of data pages or it used up the slack capacity of the database

machine.

2. The execution time per page continues to be limited by the bandwidth between the cache and the disk, and it is very sensitive to the average time required for accessing a data page. In Table 4.4, the execution times with logging for the random transactions, both with the conventional and parallel disks, are somewhat less than the corresponding times for the bare machine. So also are the corresponding times for accessing a disk page in Table 4.5. Both the average execution time and the disk access time with logging are somewhat higher when compared to the times for the bare machine for sequential transactions. To see how logging may affect the disk access time, consider the following example.

With a first-come first-serve service discipline, the average disk access time depends on the order in which pages appear in the disk queue. Suppose that the page #1 is being read and the page #250 is in the queue for reading. While page #1 is being read, the updated page #5000 is put in the disk queue. Since in the bare data base machine, a write has priority over a read, the pages will be accessed in the order (#1, #5000, #250). However, if the page #5000 is blocked temporarily for the corresponding log record to be written to disk and consequently the pages are read in the order (#1, #250, #5000), then the average access time will decrease as the total seek time for the second ordering is less than the seek time for the first ordering. Similar examples may be constructed to show an increase in the average access time due to logging.

When parallel disks are used. the number of disk accesses reduces with logging. In the bare machine, an updated page is written as soon as the page becomes available in the cache and the disk becomes free. Thus, if both pages #1 and #2 were updated by a transaction, two I/Os may be required to write

them to disk, even though they belong to the same cylinder, as I/O for page #1 may have started before page #2 became available in the cache. With logging, page #1 may first get blocked and then put in the disk queue together with page #2 if their corresponding log records are contained on the same log page. Thus, the number of disk writes may be reduced to one. Furthermore, if more than one page is written at a time, a corresponding number of cache frames will be freed simultaneously for reading data pages, and this may increase the probability that more than one disk page will be read in parallel. These observations suggest that the design of scheduling policies for parallel disks is an open area for research.

With a decrease in the number of disk accesses, however, the value of the average access time may increase. In the above example, when a separate access was made to write page #2 immediately after writing page #1, no seek would have been required as the head was positioned on the right cylinder, thus, decreasing the value of the average access time.

3. The fragment wait time reduces from the conventional-random to the parallel-sequential configuration as the rate at which log fragments are received at the log processor is a function of the rate at which the query processors update data pages.

One would expect that there would be more pages in the WAL queue in the parallel-sequential configuration compared to the conventional-random configuration as pages are processed at a higher rate in the parallel-sequential configuration. At the same time, however, the fragment wait time reduces in the parallel-sequential configuration and therefore the number and the queueing time of pages in the WAL queue decreases.

4. The most striking result from these experiments is the poor utilization of even one log disk. The rate at which the query processors update pages and hence create log fragments is just not fast enough to keep the log disk busy. As pointed out in Section 4.3.4, the I/O bandwidth between the data disks and the disk cache severely limits the rate at which the query processors update data pages. In the next subsection, we will characterize when is it worthwhile to have more than one log disk.

Number of Log Disks and Log Processor Selection

Tables 4.8 through 4.11 show the performance statistics when 2 log processors are used for logging. The results are shown for the four alternative

Log Processor Selection Algorithm	Execution time per page	Transaction Completion time	WAL		Log Processor 1			Log Processor 2		
			Q length	Q time	Disk Utilization	Total I/Os	Fragment Wait Time	Disk Utilization	Total I/Os	Fragment Wait Time
Cyclic	17.89	7927.06	8.50	797.19	0.01	633	771.91	0.01	625	793.11
Random	17.84	7916.48	8.58	801.37	0.01	632	787.97	0.01	625	787.19
QpNo mod TotLp	17.88	7952.61	8.52	797.44	0.01	559	898.52	0.01	698	695.66
TranNo mod TotLp	17.87	8306.33	8.53	799.79	0.01	617	806.01	0.01	641	766.22

Table 4.8. Effect of using 2 log processors
(Conventional-Random Configuration)

Log Processor Selection Algorithm	Execution time per page	Transaction Completion time	WAL		Log Processor 1			Log Processor 2		
			Q length	Q time	Disk Utilization	Total I/Os	Fragment Wait Time	Disk Utilization	Total I/Os	Fragment Wait Time
Cyclic	16.56	7062.74	8.52	740.81	0.01	627	726.27	0.01	628	732.26
Random	16.55	7097.28	8.54	742.22	0.01	630	729.13	0.01	625	735.00
QpNo mod TotLp	16.59	7126.93	8.50	738.69	0.01	553	827.87	0.01	705	651.48
TranNo mod TotLp	16.57	7556.79	8.48	737.91	0.01	617	725.42	0.01	642	728.81

Table 4.9. Effect of using 2 log processors
(Parallel-Random Configuration)

Log Processor Selection Algorithm	Execution time per page	Transaction Completion time	WAL		Log Processor 1			Log Processor 2		
			Q length	Q time	Disk Utilization	Total I/Os	Fragment Wait Time	Disk Utilization	Total I/Os	Fragment Wait Time
Cyclic	11.54	4705.06	9.02	532.95	0.01	631	519.43	0.01	628	537.84
Random	11.51	4684.67	8.95	529.37	0.01	634	523.19	0.01	624	523.44
QpNo mod TotLp	11.50	4725.13	8.96	527.19	0.01	573	579.77	0.01	688	476.80
TranNo mod TotLp	11.46	4973.15	8.47	495.44	0.01	616	501.42	0.01	641	491.81

Table 4.10. Effect of using 2 log processors (Conventional-Sequential Configuration)

Log Processor Selection Algorithm	Execution time per page	Transaction Completion time	WAL		Log Processor 1			Log Processor 2		
			Q length	Q time	Disk Utilization	Total I/Os	Fragment Wait Time	Disk Utilization	Total I/Os	Fragment Wait Time
Cyclic	2.07	941.07	7.28	95.93	0.06	628	102.11	0.06	629	100.24
Random	2.07	940.97	7.21	95.34	0.07	635	99.38	0.06	624	101.27
QpNo mod TotLp	2.06	933.86	7.19	95.30	0.06	615	100.89	0.07	643	97.18
TranNo mod TotLp	2.06	988.79	7.39	100.74	0.06	616	100.96	0.07	642	97.26

Table 4.11. Effect of using 2 log processors (Parallel-Sequential Configuration)

log processor selection algorithms described in Chapter 3, viz., the cyclic selection, the random selection, the query processor number mod total log processors selection, and the transaction number mod total log processors selection.

It can be seen that using 2 log processors did not improve the average execution time per pag. On the contrary, the transaction completion times became worse. The fragment wait time, WAL Q-length and WAL Q-time almost doubled and the utilization of the log disk reduced by half. This is not very surprising considering that with only one log processor, the utilization of the log disk was already very poor. When 2 log processors are used, the log fragments created by the query processors are distributed over two log disks. Since the

rate at which query processors update data pages does not increase by using an additional log processor, the effective rate at which the log fragments arrived at a log processor became half of what was before. Thus, the utilization of the log disks decreases and a log fragment waits for longer time before it can be written to the log disk. This increases the WAL Q-time for the updated pages that are waiting for the corresponding log fragments to be written to disk. Since there are two unassembled log pages now, the number of pages in the WAL queue increases.

We will present a simple analysis to characterize when is it worthwhile to have more than one log disks. Assume that in a given period of time, the database machine processes a total of N pages, where $N = \Sigma |T_i|$ and $|T_i|$ is the size of the transaction T_i. The average execution time per page for the bare database machine is E and a transaction updates u% of the data pages it accesses. The average size of a log fragment is f% of the size of a data page, and the average time to write a log page to the log disk is t. Finally, assume that the log pages may be written to the log disk in parallel with the processing of data pages by the database machine, and the execution time per page is not affected by logging. Thus, with only one log disk,

Total database machine time to process N pages = N * E
Number of log fragments = u% * N
Number of log pages = f% * (u% * N)
Time required to write the log pages to the log disk = (f% * u% * N) * t

Therefore,

$$\text{Log disk utilization} = \frac{f\% * u\% * t}{E}$$

In our experiments, f% = 10%, u% = 20%, and t = 12.61 ms. It may be verified, by substituting different values of E from Table 4.1, that the above equation quite accurately estimates the numbers in Table 4.7 for the log disk utilization. Thus, as long as the execution time of the database machine is

limited by the I/O bandwidth, more than 1 log processor will be necessary only if the database machine has a very high degree of update activity (updates all the pages, for example) or the size of the log fragments is large (physical logging instead of logical logging). However, a higher number of updated pages will require additional I/Os to write the updated pages and this may increase the value of the execution time per page. Later we will present the results of our experiments to determine whether the log disk may become a bottleneck for high values of u% while maintaining a constant value of f%.

One cannot make very meaningful conclusions about the relative performance of the various log processor selection algorithms from this set of experiments as the log disks were so underutilized. To compare the performance of different log processor algorithms and to test the usefulness of parallel logging, we designed another experiment.

In the simulation of the bare database machine, the utilization of the query processors was quite high when the parallel disks were used to process sequential transactions (Table 4.3). We, therefore, simulated the data base machine with 75 query processors and 150 disk cache frames, instead of 25 query processors and 100 disk cache frames, in the parallel-sequential configuration. We still assumed that there were 2 data disks and that each transaction updated 20% of pages that it accessed. However, instead of logical logging, the physical logging was modeled. In physical logging, for each updated page, two log pages are written; one contains the before image and the other contains the after image of the updated page. The effective bandwidth of the interconnection between the query processors and the log processors was assumed, as before, to be 1 megabyte per second.

The results of the experiment are summarized in Tables 4.12 and 4.13. In Table 4.13, the utilization, Q-length and Q-time are the averages of the mean values for the different units of the corresponding devices. Total I/Os is the sum of the disk accesses made on the two data disks. For computing the queue length at a log disk, the before and after value log pages corresponding to a data page have been counted as one.

The average execution time per page and the average transaction completion time degrade considerably in presence of physical logging, and using more than one log disks in parallel significantly improves the performance. The main reason for the degradation, when one log disk is used, is that the log disk

No. of Log Disks	Execution Time per Page				Transaction Completion Time			
	cyclic	random	QpNo mod TotLp	TranNo mod TotLp	cyclic	random	QpNo mod TotLp	TranNo mod TotLp
0	0.91	0.91	0.91	0.91	430.56	430.56	430.56	430.56
1	5.06	5.06	5.06	5.06	4518.07	4518.07	4518.07	4518.07
2	2.53	2.55	2.56	2.69	1999.51	2104.28	2231.98	2165.45
3	1.74	1.80	1.80	2.11	1078.94	1137.18	1135.72	1381.76
4	1.47	1.51	1.49	1.97	830.71	854.61	837.75	1137.50
5	1.33	1.35	1.32	1.96	716.28	741.73	714.12	1128.37

Table 4.12. Performance of the log processor selection algorithms

No. of Log Disks	Log Processor Selection Algorithm	Query Processors			Data Disks			Log Disks			WAL	
		Utilization	Q length	Q time	Utilization	Total I/Os	Q length	Utilization	Q length	Q time	Q length	Q time
0		0.205	9.56	8.68	0.95	5849	37.38					
1		0.061	0.02	0.10	0.86	25993	3.45	1.00	256.82	3246.72	128.87	3250.84
5	Cyclic	0.134	4.54	6.02	0.97	8348	41.10	0.76	3.79	67.95	9.55	78.63
	Random	0.132	4.29	5.79	0.97	8550	41.48	0.75	4.00	72.63	9.97	84.43
	QpNo mod TotLp	0.137	4.61	6.07	0.96	8067	40.85	0.77	3.96	70.36	9.92	80.80
	TranNo mod TotLp	0.080	1.79	3.50	0.95	11293	41.62	0.52	10.73	269.31	27.53	284.91

Table 4.13. Effect of log processor selection algorithms on device characteristics

becomes the bottleneck. Consequently, the log pages wait for a long time in the log-disk queue before they are written. This in turn increases the number of updated pages waiting in the cache for the corresponding log pages to be written. In our experiment with one log disk, out of 150 cache frames, on average 129 frames were occupied by the updated pages blocked in the WAL queue, and thus only 21 frames were available for reading new data pages from data disks. Availability of fewer cache frames for reading new pages severely affects the performance of the parallel disks. As compared to the no logging case, when 5849 disk accesses were made, a total of 25993 disk accesses are required with logging. Furthermore, with logical logging, when a log page is written, all the corresponding updated data pages are moved from the WAL queue to the disk queue at the same time and if they belong to the same cylinder, they may be written to disk in one I/O. With physical logging, only page at a time is transferred from the WAL queue to the disk queue.

Amongst the log processor selection algorithms, performances of the cyclic, random, and query processor number mod total log processors selection are comparable, whereas the transaction number mod total log processors selection turns out to be a loser. A log processor selection algorithm should avoid congestion at some log processor while the other log processors are idle. Table 4.14 shows the standard deviation in the mean queue length, queue time, utilization and total I/Os at five log disks for each algorithm. For the transaction number mod total log processors selection, the deviation in the average values at five log disks for all the four parameters is largest indicating that the log pages were not evenly distributed by this algorithm. For each algorithm, we have also tabulated the averages of the standard deviations in the queue lengths and the queue times at five log disks. The deviations in the queue length and the queue time at a log disk are largest for the transaction number

Log Processor Selection Algorithm	Standard Deviation of the Averages				Average of the Standard Deviations	
	Q length	Q time	Utili-zation	Total I/Os	Q length	Q time
Cyclic	0.09	1.52	0.0	2.45	4.56	60.70
Random	0.40	6.07	0.01	29.67	5.07	67.99
QpNo mod TotLp	0.16	2.45	0.005	15.34	4.77	63.29
TranNo mod TotLp	1.71	27.22	0.03	141.00	16.21	220.13

Table 4.14. Variances in the log processors' characteristics (5 log processors)

mod total log processors selection indicating that this algorithm not only unevenly distributed the log pages but also the pattern of arrival of log pages at a log disk was irregular. There were too many log pages sometimes and too few the other times. The uneven usage of the log disks is the main reason for the higher values of the average queue length and time for the WAL queue and the log disk queues with the transaction number mod total log processors selection.

An example will illustrate why the transaction number mod total log processors selection results in an uneven and irregular distribution of log pages. Suppose that the transactions T1, T2, and T3 arrive in that order. Their sizes are 250, 1, and 250 pages respectively. For simplicity, assume only 2 log disks. After T2 has been processed, if all the query processors are processing pages of either T1 or T3 then they will all send the log fragments to the first log processor while the second log processor is idling.

The good performances of the cyclic and query processor number mod total log processors algorithms relative to the random selection was rather surprising. One would suspect that, in the cyclic selection, the log processors may get lock-stepped so that all of them send their log fragments to the first log processor and then to the second and so on. The reason for the good

performances of the cyclic and the query processor number mod total log processors selections is that all the query processors have equal probability of updating a page and there is a random time interval between the two updates by a query processor. The random selection only further randomizes this already random phenomenon. Although over a long period of time, the random selection of log processors will result in an equal number of log fragments at all the log processors, yet during some short interval, the number of log fragments sent to the different log processors may differ. The advantage of cyclic method is that since each query processor sends equal number of log fragments to each log processor, each log processor will get equal number of log fragments. With the query processor number mod total log processors selection, if the query processors update equal number of data pages, then each log processor will get equal number of log fragments.

The query processor number mod total log processors selection may also simplify the parallel logging algorithm. Recall that in the parallel logging algorithm given in Chapter 3, a query processor, after sending a log fragment to a log processor, sends the log processor identifier to the back-end controller. The back-end controller will no longer need this information as it can itself compute the log processor number from the query processor number. Furthermore, if the network connecting the query processors and the log processors is a dedicated interconnection, the query processor number mod total log processors algorithm will simplify this interconnection.

Connection Between the Query and the Log Processors

To explore the effect of the medium connecting the log processors to the query processors on the database machine performance, we performed 3 sets of experiments. In the first two sets of experiments the effective bandwidth of the

interconnection was taken to be 0.1 and 0.01 megabytes/second respectively. In the third set of experiments, the log fragments were routed through the disk cache. The results of the experiment are summarized in Tables 4.15 through 4.18. We assumed 25 query processors, 100 disk cache frames, 2 data disks, and 1 log processor. We assumed logical logging with the average size of the log fragment set equal to 1/10 of the size of the data page.

The performance of the database machine is quite insensitive to the communication medium. The reduced bandwidth of the interconnection slightly increases the average fragment waiting time which in turn causes the average WAL queue length and time to marginally increase. The fragment waiting time

Band-width (Mbyte/sec)	Execution time	Transaction time	Query Processors		Data Disks			Log Disk Utilization	Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length			Q length	Q time
No Log	18.00	7398.41	0.0204	0	0.99	59170	95.80				
1.0	17.86	7543.20	0.0220	0	0.99	59170	91.50	0.02	374.60	3.92	367.06
0.1	17.83	7487.58	0.0224	0	0.99	59170	91.46	0.02	374.49	3.96	368.62
0.01	17.86	7548.53	0.0216	0	0.99	59170	90.92	0.01	385.30	4.49	400.91
Cache	17.86	7485.28	0.0224	0	0.99	59170	91.50	0.01	373.40	3.89	367.59

Table 4.15. Effect of the communication medium (Conventional-Random Configuration)

Band-width (Mbyte/sec)	Execution time	Transaction time	Query Processors		Data Disks			Log Disk Utilization	Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length			Q length	Q time
No Log	16.62	6476.04	0.0224	0	1.00	55236	95.50				
1.0	16.50	6649.90	0.0236	0	1.00	55217	91.11	0.02	349.88	3.92	342.00
0.1	16.51	6646.80	0.0240	0	1.00	55226	91.10	0.02	348.45	3.94	340.38
0.01	16.54	6686.15	0.0236	0	1.00	55227	90.49	0.02	362.43	4.54	375.24
Cache	16.51	6662.47	0.0240	0	1.00	55206	91.99	0.02	350.36	3.91	342.29

Table 4.16. Effect of the communication medium (Parallel-Random Configuration)

Band-width (Mbyte/sec)	Execution time	Transaction time	Query Processors		Data Disks			Log Disk Utilization	Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length			Q length	Q time
No Log	11.01	4016.46	0.0328	0	0.75	59170	94.98				
1.0	11.39	4333.46	0.0336	0	0.75	59170	90.37	0.02	254.22	4.31	251.52
0.1	11.42	4345.42	0.0332	0	0.74	59170	90.28	0.02	256.85	4.41	254.63
0.01	11.42	4384.94	0.0336	0	0.75	59170	89.47	0.02	266.33	5.22	297.47
Cache	11.39	4341.58	0.0352	0	0.75	59170	90.36	0.02	254.69	4.30	253.29

Table 4.17. Effect of the communication medium (Conventional-Sequential Configuration)

Band-width (Mbyte/sec)	Execution time	Transaction time	Query Processors		Data Disks			Log Disk Utilization	Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length			Q length	Q time
No Log	1.92	758.06	0.895	54.47	0.92	13991	27.42				
1.0	2.05	862.24	0.955	57.22	0.92	13761	19.97	0.13	49.86	3.16	47.39
0.1	2.05	865.94	0.950	55.71	0.91	13866	20.88	0.13	50.09	3.37	49.54
0.01	2.06	899.47	0.926	50.38	0.91	13693	21.42	0.13	50.58	6.25	77.45
Cache	2.06	866.78	0.947	56.60	0.91	13961	20.75	0.13	49.63	3.03	47.54

Table 4.18. Effect of the communication medium (Parallel-Sequential Configuration)

increases only marginally with a slower medium if there is a time gap between arrivals of fragments. As shown in Figure 4.2, after a log fragment arrives at a log processor, the delay in the arrival of subsequent log fragments is absorbed in the interarrival gap.

The performance of the database machine was not affected even when the log fragments were routed through the disk cache. Routing the fragments through the cache causes the usage of the query processors to increase and some cache frames get tied up to hold the in-transit fragments. However, the query processors or the number of cache frames are not the constraining factors for the performance of the database machine.

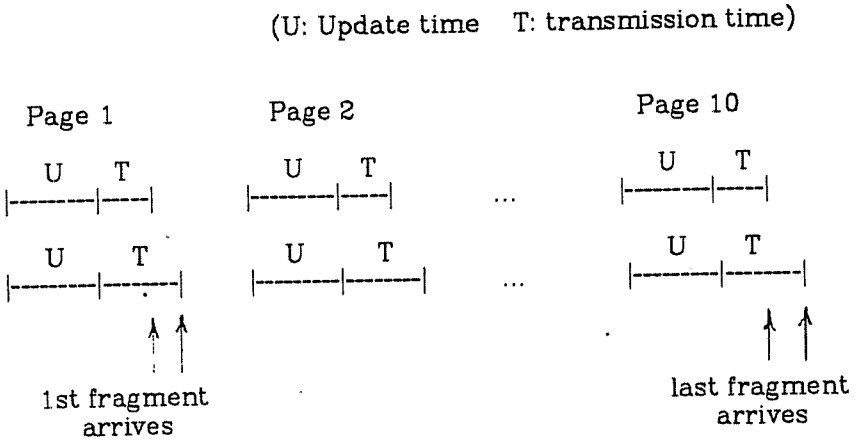

Figure 4.2. Effect of slower communication medium on fragment wait time

To examine if the communication medium will affect the database machine performance if the data pages were processed at a higher rate, we repeated the above set of experiments for 75 query processors and 150 disk cache frames in the parallel-sequential configuration of the database machine. We still assumed 2 data disks, 1 log processor, and logical logging with the average fragment size

Band-width (Mbyte/sec)	Execution time	Transaction time	Query Processors		Data Disks			Log Disk Utilization	Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length			Q length	Q time
No Log	0.91	430.56	0.205	9.56	0.95	5849	74.76				
1.0	0.95	499.17	0.220	10.47	0.96	5334	66.45	0.28	25.54	3.16	28.53
0.1	0.96	512.94	0.221	10.36	0.96	5410	66.59	0.28	25.50	3.51	28.95
0.01	1.02	557.88	0.201	8.17	0.96	5717	64.81	0.26	26.31	8.31	53.63
Cache	0.95	495.23	0.229	10.76	0.96	5303	66.13	0.28	23.17	2.74	26.51

Table 4.19. Effect of the communication medium
(75 Query Processors, 150 Cache Frames, Parallel-Sequential Configuration)

equal to 1/10 of the data page. The results of this experiment are presented in Table 4.19. It may be observed that at this higher data processing rate also, the communication medium does not significantly affect the performance of the database machine.

Size of the Log Pages

In all of the experiments so far, the block size of the log disk was assumed to be the same as the block size of the data disks (4096 bytes). We also experimented with the smaller block sizes of 2048, 1024, and 512 bytes for the log disk. The database machine consisted of 25 query processors, 100 disk cache frames, 2 data disks, and 1 log disk. In one experiment, we considered 75 query processors and 150 cache frames in the parallel-sequential configuration of the database machine. In all the experiments we assumed logical logging with an average log fragment size of 400 bytes. The results of this set of experiments have been summarized in Tables 4.20 through 4.24.

With 25 query processors, as the block size of the log disk decreases, the average fragment waiting time also decreases, as fewer log fragments are needed to fill a log page. Consequently, the average WAL queue length and time

Block Size (byte)	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	18.00	7398.41	0.02	0	0.99	59170	95.80					
4096	17.86	7543.20	0.02	0	0.99	59170	91.50	0.02	1258	374.60	3.92	367.06
2048	17.89	7426.91	0.02	0	0.99	59170	93.55	0.03	2676	193.59	1.93	196.43
1024	17.89	7403.96	0.02	0	0.99	59170	94.74	0.05	6070	85.28	0.79	102.81
512	17.93	7389.58	0.02	0	0.99	59170	95.44	0.10	11435	20.22	0.16	48.51

Table 4.20. Effect of the block size of the log disk
(Conventional-Random Configuration)

Block Size (byte)	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	16.62	6476.04	0.02	0	1.00	55236	95.50					
4096	16.50	6649.90	0.02	0	1.00	55217	91.11	0.02	1257	349.88	3.92	342.00
2048	16.53	6538.39	0.02	0	1.00	55241	93.22	0.03	2670	180.67	1.92	182.68
1024	16.55	6496.54	0.02	0	1.00	55299	94.45	0.06	6106	78.72	0.78	94.87
512	16.57	6468.28	0.02	0	1.00	55294	95.14	0.10	11442	18.73	0.15	45.05

Table 4.21. Effect of the block size of the log disk (Parallel-Random Configuration)

Block Size (byte)	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	11.01	4016.46	0.03	0	0.75	59170	94.98					
4096	11.39	4333.46	0.03	0	0.75	59170	90.37	0.02	1254	254.22	4.31	251.52
2048	11.29	4198.85	0.03	0	0.75	59170	92.61	0.04	2671	125.57	2.10	128.39
1024	11.15	4106.10	0.04	0	0.75	59170	93.83	0.09	6083	54.02	0.88	61.29
512	11.03	4031.30	0.04	0	0.75	59170	94.48	0.16	11410	14.18	0.21	20.82

Table 4.22. Effect of the block size of the log disk (Conventional-Sequential Configuration)

Block Size (byte)	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	1.92	758.06	0.90	54.47	0.92	13991	27.42					
4096	2.05	862.24	0.96	57.22	0.92	13761	19.97	0.13	1255	49.86	3.16	47.39
2048	2.05	853.18	0.93	57.89	0.92	14137	22.39	0.23	2671	24.64	1.33	24.38
1024	2.06	827.59	0.93	58.09	0.93	14515	23.48	0.47	6110	13.27	0.61	13.16
512	2.09	838.21	0.88	52.33	0.93	15128	27.45	0.82	11434	24.45	1.31	23.38

Table 4.23. Effect of the block size of the log disk (Parallel-Sequential Configuration)

Block Size (byte)	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	0.91	430.56	0.21	9.56	0.95	5849	74.76					
4096	0.95	499.17	0.22	10.47	0.96	5334	66.45	0.28	1256	25.54	3.16	28.53
2048	0.96	484.72	0.22	10.27	0.96	5630	70.05	0.49	2667	15.23	1.57	16.05
1024	1.11	569.97	0.17	6.57	0.96	7172	73.84	0.88	6081	67.48	7.28	56.53
512	1.72	1157.31	0.08	0.94	0.95	11611	49.98	1.00	11423	526.50	56.19	487.60

Table 4.24. Effect of the block size of the log disk
(75 Query Processors, 150 Cache Frames, Parallel-Sequential Configuration)

decrease, which in turn reduces the transaction completion times. The execution times do not improve as the logging actions were already completely overlapped with the processing of data pages.

A side effect of decreasing the block size is that the number of log pages increases. This is not a problem as long as the log disk does not become a bottleneck. In parallel-sequential configuration (Table 4.23), however, when the block size of the log disk is reduced to 512 bytes, the utilization of the log disk reaches 82%. Thus, compared to the block size of 1024 bytes, the fragment wait time increases as a queue begins to form at the log disk. The problem becomes still severe when 75 query processors and 150 disk cache frames are used in the parallel-sequential configuration. As the block size is decreased, the number of log pages steadily increases. With a block size of 512 bytes, the log disk becomes the bottleneck. The fragments wait for long time in the log disk queue before they are written to disk. Consequently, the number of updated pages in the WAL queue and the transaction completion time also increases. A large number of pages in the WAL queue results in fewer free cache frames for reading new data pages. This affects the performance of the parallel disk and the number of disk accesses increases considerably. Thus, the average execution time increases by almost a factor of two.

Size of the Log Fragments

Tables 4.25 through 4.29 summarize the results of our experiments to test the sensitivity of the performance of the database machine to the average log fragment size. We assumed 25 query processors, 100 disk cache frames, 2 data

Fragment Size	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	18.00	7398.41	0.02	0	0.99	59170	95.80					
10%	17.86	7543.20	0.02	0	0.99	59170	91.50	0.02	1258	374.60	3.92	367.06
20%	17.86	7405.14	0.02	0	0.99	59170	93.28	0.03	2671	215.80	2.20	210.15
50%	17.85	7306.39	0.02	0	0.99	59170	94.49	0.10	8270	104.22	1.03	105.14

Table 4.25. Effect of the size of the log fragments
(Conventional-Random Configuration)

Fragment Size	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	16.62	6476.04	0.02	0	1.00	55236	95.50					
10%	16.50	6649.90	0.02	0	1.00	55217	91.11	0.02	1257	349.88	3.92	342.00
20%	16.51	6517.88	0.02	0	1.00	55240	92.92	0.03	2677	200.65	2.19	195.05
50%	16.52	6463.19	0.02	0	1.00	55323	94.19	0.11	8292	97.07	1.02	97.61

Table 4.26. Effect of the size of the log fragments
(Parallel-Random Configuration)

Fragment Size	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	11.01	4016.46	0.03	0	0.75	59170	94.98					
10%	11.39	4333.46	0.03	0	0.75	59170	90.37	0.02	1254	254.22	4.31	251.52
20%	11.30	4209.16	0.03	0	0.75	59170	92.33	0.05	2666	139.08	2.37	139.34
50%	11.19	4113.16	0.03	0	0.75	59170	93.54	0.16	8317	66.51	1.17	68.29

Table 4.27. Effect of the size of the log fragments
(Conventional-Sequential Configuration)

Fragment Size	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	1.92	758.06	0.90	54.47	0.92	13991	27.42					
10%	2.05	862.24	0.96	57.22	0.92	13761	19.97	0.13	1255	49.86	3.16	47.39
20%	2.06	839.29	0.94	57.81	0.92	14031	22.03	0.28	2666	26.53	1.59	27.12
50%	2.09	860.85	0.87	50.29	0.93	14978	26.68	0.85	8290	42.81	2.83	41.74

Table 4.28. Effect of the size of the log fragments
(Parallel-Sequential Configuration)

Fragment Size	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
No Log	0.91	430.56	0.21	9.56	0.95	5849	74.76					
10%	0.95	499.17	0.22	10.47	0.96	5334	66.45	0.28	1256	25.54	3.16	28.53
20%	0.98	497.72	0.21	10.32	0.96	5855	70.26	0.58	2664	19.89	2.28	20.96
50%	1.78	1248.57	0.08	0.41	0.95	11516	41.24	1.00	8315	625.43	66.97	595.04

Table 4.29. Effect of the size of the log fragments
(75 Query Processors, 150 Cache Frames, Parallel-Sequential Configuration)

disks, and 1 log processor. The sensitivity experiment was also performed for the parallel-sequential configuration with 75 query processors and 150 cache frames.

Increasing the log fragment size is logically equivalent to decreasing the size of the log page as far as the impact on the fragment wait time and the log disk utilization are concerned. Therefore, the results of this set of experiments are similar to the results of decreasing the block size of the log disk. The difference is that a larger amount of time is required for transmitting a bigger log fragment between a query processor and a log processor. This is equivalent to using a communication medium with a lower bandwidth. However, we have seen that the bandwidth of the communication medium has marginal impact on the performance.

Percentage of Pages Updated

Tables 4.30 through 4.33 characterize the behavior of logging as the percentage of pages updated out of the total pages accessed by a transaction increases. The results in Tables 4.30 through 4.33 assume 25 query processors, 100 disk cache frames, 2 data disks, and 1 log disk.

% of Update Pages	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
20%	17.86	7543.20	0.02	0	0.99	59170	91.50	0.02	1258	374.60	3.92	367.06
50%	23.22	9716.01	0.04	0	0.99	76840	91.82	0.03	3136	207.58	3.47	173.42
80%	28.59	11916.27	0.06	0	0.99	94650	91.56	0.04	5028	182.95	3.34	130.31

Table 4.30. Effect of the percentage of the pages updated by a transaction (Conventional-Random Configuration)

% of Update Pages	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
20%	16.50	6649.90	0.02	0	1.00	55217	91.11	0.02	1257	349.88	3.92	342.00
50%	21.48	8486.24	0.05	0	1.00	71800	91.38	0.03	3135	195.66	3.47	162.14
80%	26.24	10203.90	0.06	0	1.00	87874	90.94	0.04	5028	172.25	3.34	122.33

Table 4.31. Effect of the percentage of the pages updated by a transaction (Parallel-Random Configuration)

% of Update Pages	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
20%	11.39	4333.46	0.03	0	0.75	59170	90.37	0.02	1254	254.22	4.31	251.52
50%	14.47	5325.37	0.07	0	0.75	76840	90.86	0.05	3139	133.66	4.01	122.04
80%	17.98	6544.54	0.09	0	0.74	94650	91.32	0.06	5025	104.42	3.58	91.08

Table 4.32. Effect of the percentage of the pages updated by a transaction (Conventional-Sequential Configuration)

% of Update Pages	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
20%	2.05	862.24	0.96	57.22	0.92	13761	19.97	0.13	1255	49.86	3.16	47.39
50%	2.16	828.89	0.97	50.63	0.92	13594	21.46	0.31	3142	22.28	3.37	21.33
80%	2.27	836.84	0.98	47.11	0.90	13522	20.85	0.47	5028	16.59	4.04	15.75

Table 4.33. Effect of the percentage of the pages updated by a transaction (Parallel-Sequential Configuration)

With an increase in the percentage of pages updated by a transaction, the fragment wait time decreases because of the higher arrival rate of the log fragments at the log processor. There is a corresponding decrease in the time for which updated pages wait in the cache in the WAL queue. However, the number of pages in the WAL queue does not change significantly as the reduction in the fragment wait time is offset by the higher rate of arrival of the updated pages to the cache. The transaction completion time and the execution time increase with the conventional disks and the parallel-random configuration as the number of disk accesses increases with the increased update activity.

A surprising phenomenon is that the number of disk accesses in the parallel-sequential configuration decreases with an increase in the percentage of updated pages (Table 4.33). In our parallel disk, there are 120 pages per cylinder that can be read in one disk access. The reference string of a sequential transaction consists of physically adjacent pages on a disk. Thus, if a transaction references 100 pages, potentially all the pages can be read in one disk access. However, at a time, only as many pages as the number of free frames can be read. An I/O begins as soon as the disk becomes free, if there is a page to be accessed and a free cache frame. The number of accesses required to read all the pages of a transaction, therefore, depends crucially on how many

cache frames become available at a time. When a log page is written, all the data pages waiting for the log fragments in this log page to be written become available for writing. When the update percentage is high, these pages are likely to belong to the same transaction and hence to the same cylinder and may be written in one disk access. Thus, a higher percentage of updated pages may not result in an increase in the number of disk accesses, but may cause several cache frames to become free at the same time. On the other hand, if the percentage of updated pages is low, the cache frames holding the pages that are read but not updated become free one at a time at random times. This discussion further confirms that 'first-come first-serve' and 'service as soon as can' are not appropriate for parallel disks and the scheduling strategies for parallel disks is an open area for research.

With 75 query processors and 150 cache frames in the parallel-sequential configuration (Table 4.34), however, the fragment wait time increases as the log disk becomes the bottleneck. This in turn increases the number of pages in the WAL queue resulting in fewer cache frames for reading data pages from disk. Consequently, the performance both in terms of the execution time and the transaction completion times degrades.

% of Update Pages	Execution time	Transaction time	QPs		Data Disks			Log Disk		Fragment Wait time	WAL	
			Utilization	Q length	Utilization	Total I/Os	Q length	Utilization	Total I/Os		Q length	Q time
20%	0.95	499.17	0.22	10.47	0.96	5334	66.45	0.28	1256	25.54	3.16	28.53
50%	1.17	554.20	0.33	4.19	0.95	7077	59.58	0.57	3137	51.27	15.18	46.65
80%	1.63	698.92	0.33	0.75	0.91	9105	36.08	0.66	5029	123.99	54.88	113.74

Table 4.34. Effect of the percentage of the pages updated by a transaction (75 Query Processors, 150 Cache Frames, Parallel-Sequential Configuration)

4.5. Shadow Mechanism Simulator

As in the case of parallel logging, in our simulation of the shadow mechanism, we have only modeled the recovery actions during the execution of a successful transaction. We will first describe our experiments to study the effect on the database machine performance when one or more page-table processors are used to obtain disk addresses of the data pages.

4.5.1. Specifications of the Shadow Mechanism Module

The page tables are maintained on disks separate from the data disks. The characteristics of IBM 3350 disk drives have been assumed for the page-table disks. Associated with each page-table disk is a processor whose function is to read and update the page tables. There is a page-addressable buffer common to the page-table processors for holding the page-table pages. Pages in the buffer are managed using the least recently used (LRU) replacement policy. This buffer is different from the disk cache for reading data pages. For most of our experiments, we assume a buffer of 10 pages of 4096 bytes each. Pages can be read from the page-table disk in blocks of 4096 bytes. We will present the sensitivity results of varying the the block size and the buffer size. We assume that a page-table entry is 4 bytes long.

In all our experiments, we assume 2 data disks, and either 1 or 2 page-table disks. The mapping problem when 2 page-table disks are used is solved by assuming one to one correspondence between the data disk number and the page-table disk number. We also assume that the back-end controller requests the disk-address of a data page from the page-table processor only if the page is not available in the disk cache.

For committing a transaction, all the pages updated by the transaction are written to the data disks and then a precommit record is written to a page-table

disk. As discussed in Section 3.4.2.1, all the precommit records are written to one page-table disk. We assume that after the precommit record of a transaction has been written, the page-table processors write the page-table pages updated by the transaction to disk. A transaction is considered complete only after all the page-table entries updated by the transaction have been written to disk. In an actual implementation, an updated page-table page would not be written to disk until the buffer replacement algorithm so requires. However, this assumption is consistent with the assumption made while modeling the logging mechanism that a transaction is considered complete only after all the changes effected in the database state by the transaction have become permanent.

4.5.2. Experiments

We performed a number of experiments with our simulator of the shadow mechanism to investigate the following issues:

* Effect of the shadow mechanism on the average execution times of the database machine?
* Is it worthwhile having more than one page-table processor?
* Effect of the size of the page-table buffer?
* Effect of smaller block size for the page-table disk?
* What if the logically adjacent pages are not physically adjacent on data disk as a result of the shadow mechanism?
* When should the version selection or the overwriting scheme described in Chapter 3 be used?

Effect on Database machine Performance

Tables 4.35 through 4.38 show the effect of the shadow mechanism on different performance parameters of the database machine when 1 and 2 page-table disks are used. These experiments were performed with 25 query processors, 100 cache frames and 2 data disks. In Table 4.36, Utilization is the average of the mean values of the utilization of the two data disks; whereas, Q-length and Total I/Os is the total of the queue length and the number of disk accesses respectively. In the case of 2 page-table disks in Table 4.38, Utilization, Q-length, and Total.I/Os are similarly defined, and Q-time is the average waiting before a page-table access request is accepted. We make following observations based on Tables 4.35 through 4.38.

1. With random transactions, when 1 page-table processor is used, the performance of the database machine degrades both in terms of the execution time per page and the transaction completion time. The degradation, however, is ameliorated because the page table accesses and the processing of data pages is *pipelined*. While a data page is being read from the disk and being processed by a query processor, the page-table processor fetches the disk-address of the next data page.

Configuration	Execution Time per Page			Transaction Completion Time		
	Bare Machine	1 PageTable Disk	2 PageTable Disks	Bare Machine	1 PageTable Disk	2 PageTable Disks
Conventional-Random	18.00	20.51	17.99	7398.41	8367.19	7758.92
Parallel-Random	16.62	20.49	16.69	6476.04	8352.91	6962.23
Conventional-Sequential	11.01	10.98	10.99	4016.46	4066.86	4061.19
Parallel-Sequential	1.92	1.94	1.93	758.06	829.34	816.29

Table 4.35. Performance of the Shadow Mechanism

Configuration	Bare Machine			1 PageTable Disk			2 PageTable Disks		
	Utili-zation	Q length	Total I/Os	Utili-zation	Q length	Total I/Os	Utili-zation	Q length	Total I/Os
Conventional-Random	0.99	94.98	59170	0.86	17.95	59170	0.99	91.07	59170
Parallel-Random	1.00	95.50	55236	0.85	15.53	58045	1.00	89.65	55455
Conventional-Sequential	0.75	95.80	59170	0.75	94.94	59170	0.75	94.94	59170
Parallel-Sequential	0.92	27.42	13991	0.90	28.28	13818	0.91	27.81	13812

Table 4.36. Data Disk Characteristics

Configuration	Bare Machine		1 PageTable Disk		2 PageTable Disks	
	Max QPs Used	Effective QPs	Max QPs Used	Effective QPs	Max QPs Used	Effective QPs
Conventional-Random	9	0.51	10	0.44	11	0.51
Parallel-Random	10	0.56	9	0.44	11	0.56
Conventional-Sequential	17	0.82	17	0.83	17	0.83
Parallel-Sequential	25	22.38	25	21.40	25	22.12

Table 4.37. Query Processors Utilization

Configuration	1 PageTable Disk					2 PageTable Disks				
	Utili-zation	Q length	Q time	Total I/Os	Access time	Utili-zation	Q length	Q time	Total I/Os	Access time
Conventional-Random	1.00	81.65	1796.95	61392	19.71	0.60	6.18	119.08	63845	19.81
Parallel-Random	1.00	84.10	1849.92	61342	19.71	0.64	7.43	132.86	63675	19.81
Conventional-Sequential	0.06	0.02	6.97	1559	25.68	0.03	0.01	5.08	1559	22.75
Parallel-Sequential	0.34	1.39	31.36	1559	24.96	0.16	0.57	13.88	1559	22.66

Table 4.38. Characteristics of the PageTable Disks

With 1 page-table disk, the page-table processor becomes the bottleneck as evidenced from the utilization of the page-table disk and the relative queue lengths at the page-table disk and the data disks. The average utilization of the data disks decreases from 100% to 85% utilization. Thus, there were times when a page was to be accessed from the data disk and the disk was free but the I/O could not be performed as the disk-address of the page had not yet been fetched from the page table. This forced idleness of the data disks is the main reason for the increase in the execution time per page.

The relative degradation is higher with the parallel data disks than with the conventional disks. The reason is that not only are the parallel disks under-utilized but the number of accesses also increases. With parallel disks, the probability that more than one data page will be accessed in one access depends on the number of pages in the disk queue. The indirection through the page table causes more pages to wait in the page-table disk queue and there are relatively less pages in the data disk queue.

When 2 page-table disks are used, the task of fetching and updating the page-table entries is shared between two page-table processors. The page-table processors are no longer the bottleneck and the performance of the database machine again becomes limited by the I/O bandwidth between the disk and the cache.

With 2 page-table processors, the buffer size to hold page-table pages was still assumed to be 10 pages. Thus, when compared to 1 log processor, a page could stay in the buffer for smaller duration. This is the reason for the increase in the total number of accesses to the page-table disks with 2 page-table processors.

2. Compared to the random transactions, very few accesses to the page-table disk are required when the transactions are sequential. With 4096 byte page-table pages, 1000 page-table entries are contained on one page. We have assumed that a transaction accesses at most 250 data pages. Thus, when the access pattern is sequential, at most 2 page-table pages will have to be accessed to get all the disk addresses. Therefore, the utilization of even 1 page-table disk is very low with the sequential transactions and the queue length is almost negligible. Thus, the execution time of the database machine is not affected at all by the shadow mechanism as the small amount of time spent in reading and updating of page-table entries is completely *overlapped* with the processing of data pages.

A consequence of using the shadow mechanism is that logically adjacent pages may not be physically adjacent. Thus, although accesses may be logically sequential, getting the next page may involve a disk seek. A crucial assumption in this set of experiments was that it is possible to maintain physical clustering of logically adjacent pages within a cylinder. Later on, we will present the results of the simulation of the impact of the shadow mechanism on the database machine performance if this assumption does not hold.

Size of the Page-Table Buffer

The number of accesses to the page-table disks depends on the size of the buffer available to hold the page-table pages. If the buffer size is large, a page-table page may stay in the buffer for a longer time and the probability of a page hit increases. On the other hand, if the buffer size is insufficient, then at the time of updating the page-table entries when a transaction commits, those pages that are to be updated and that are no longer available in the buffer will have to be reread.

Tables 4.39 and 4.40 show the reduction in the number of accesses to the page-table disk with an increase in the page-table buffer size for random and sequential transactions respectively. We assumed 25 query processors, 100 disk cache frames, 2 data disks, and 1 page-table disk. Data pages read from the data disks gives the number of data pages for which the back-end controller sought the disk-addresses from the page-table processor. The sum of the *distinct* page-table pages referenced by each transaction is given by the numbers in the page-table pages referenced column. Page-table pages read gives the *actual* number of pages that are read to satisfy all the disk-address requests. Page-table pages updated is the total number of pages updated, and page-table pages reread is the number of pages that had to be reread for updating because of the buffer size constraint. Observe that the page-table pages read in the case of sequential transactions is less than the pages

Buffer Pages	Data Pages Read from Data Disks	PageTable Pages					
		Referenced	Updated	Conventional		Parallel	
				Read	Reread	Read	Reread
10	47325	30639	10506	40613	9773	40596	9740
25	47325	30639	10506	32712	8550	31554	8576
50	47325	30639	10506	26377	6585	25541	6573

Table 4.39. Reduction in the number of page-table accesses (Random Configurations)

Buffer Pages	Data Pages Read from Data Disks	PageTable Pages					
		Referenced	Updated	Conventional		Parallel	
				Read	Reread	Read	Reread
10	47325	552	544	514	1	514	1
25	47325	552	544	459	0	459	0
50	47325	552	544	368	0	368	0

Table 4.40. Reduction in the number of page-table accesses (Sequential Configurations)

referenced as some transactions could use the page-table pages in the buffer brought in by the previous transactions. There would be page hits in the case of random transactions also. But a random transaction may request disk-addresses that belong to the same page-table page far apart in time, and therefore, the same page-table page may be read from disk more than once by a long random transaction.

With an increase in the buffer size, the number of page-table pages that are accessed to get the disk addresses decreases substantially. For random transactions, the number of pages reread also decreases. Tables 4.41 through 4.44 show the effect of the reduction in the number of page-table disk I/Os on various performance parameters of the database machine. It can be seen that

Buffer Pages	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	18.00	7398.41	0.02	0	0.99	59170	95.80	35.53				
10	20.51	8367.19	0.02	0	0.86	59170	17.95	35.32	1.00	61392	81.65	19.71
25	18.02	7863.42	0.02	0	0.98	59170	66.73	35.35	0.97	52268	32.86	19.69
50	18.01	7971.00	0.02	0	0.99	59170	85.04	35.46	0.82	43968	14.35	19.85

Table 4.41. Effect of the size of the page-table buffer (Conventional-Random Configuration)

Buffer Pages	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	16.62	6476.04	0.02	0	1.00	55236	95.50	35.49				
10	20.49	8352.91	0.02	0	0.85	58045	15.53	35.27	1.00	61342	84.10	19.71
25	17.18	7480.42	0.02	0	0.99	56320	54.36	35.36	0.99	51136	45.22	19.64
50	16.70	7330.73	0.02	0	1.00	55429	81.72	35.38	0.87	43120	17.67	19.83

Table 4.42. Effect of the size of the page-table buffer (Parallel-Random Configuration)

Buffer Pages	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	11.01	4016.46	0.03	0	0.75	59170	94.98	16.52				
10	10.98	4066.86	0.03	0	0.75	59170	94.94	16.48	0.06	1559	0.02	25.68
25	10.98	4069.14	0.03	0	0.75	59170	94.94	16.48	0.06	1503	0.02	25.86
50	10.99	4071.68	0.03	0	0.75	59170	94.95	16.49	0.06	1412	0.02	26.12

Table 4.43. Effect of the size of the page-table buffer (Conventional-Sequential Configuration)

Buffer Pages	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	1.92	758.06	0.90	54.47	0.92	13991	27.42	14.93				
10	1.94	829.34	0.86	52.02	0.90	13818	28.28	14.87	0.34	1559	1.39	24.96
25	1.94	825.55	0.87	52.48	0.90	13843	28.22	14.94	0.33	1503	1.04	25.20
50	1.93	829.81	0.87	53.17	0.91	13860	28.02	14.97	0.32	1412	0.65	25.43

Table 4.44. Effect of the size of the page-table buffer (Parallel-Sequential Configuration)

the degradation in the execution time with random transactions due to the shadow mechanism may be annulled by choosing a suitably large page-table buffer, even when only 1 page-table processor is used. For sequential transactions, the reduction in the number of page-table disk accesses with a larger buffer was not very beneficial as the utilization of the page-table disk was already very poor.

Size of the Page-Table Pages

Reducing the size of the page-table pages causes the average time to access a page-table page from the page-table disk to decrease as the transfer time decreases. The disadvantage, however, is that the number of page-table accesses to satisfy the disk-address requests may increase. Choice of an

appropriate block size for the page-table disk is a trade-off between these two conflicting requirements.

Tables 4.45 and 4.46 show the effect of reducing the size of the page-table pages on different performance parameters in the case of random transactions. The database machine consists of 25 query processors, 100 disk cache frames, 2 data disks, and 1 page-table disk. The page-table buffer is always assumed to be able to hold 10 page-table pages. Table 4.47 shows the effect on the number of accesses to the page-table disk with the reduced page size. As expected, the average access time decreases but the number of accesses increases. However, using a smaller block size for the page-table disks seems to be preferable as

Block Size (bytes)	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	18.00	7398.41	0.02	0	0.99	59170	95.80	35.53				
4096	20.51	8367.19	0.02	0	0.86	59170	17.95	35.32	1.00	61392	81.65	19.71
2048	20.05	8118.08	0.02	0	0.89	59170	20.65	35.42	1.00	66365	78.99	17.81
1024	19.72	7964.44	0.02	0	0.90	59170	22.95	35.46	1.00	68900	76.80	16.87
512	19.50	7871.68	0.02	0	0.91	59170	24.81	35.47	1.00	70152	74.93	16.39

Table 4.45. Effect of the block size of the page-table disk (Conventional-Random Configuration)

Block Size (bytes)	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	16.62	6476.04	0.02	0	1.00	55236	95.50	35.49				
4096	20.49	8352.91	0.02	0	0.85	58045	15.53	35.27	1.00	61342	84.10	19.71
2048	20.03	8073.36	0.02	0	0.87	58036	17.47	35.39	1.00	66362	82.24	17.81
1024	19.71	7921.62	0.02	0	0.89	57961	19.35	35.45	1.00	68914	80.37	16.87
512	19.49	7827.32	0.02	0	0.89	57914	20.84	35.47	1.00	70162	78.89	16.38

Table 4.46. Effect of the block size of the page-table disk (Parallel-Random Configuration)

Block Size (bytes)	Data Pages Read from Data Disks	PageTable Pages					
		Referenced	Updated	Conventional		Parallel	
				Read	Reread	Read	Reread
4096	47325	30639	10506	40613	9773	40596	9740
2048	47325	37511	11137	43967	10761	43961	10764
1024	47325	42033	11499	45615	11286	42033	11274
512	47325	44546	11665	46456	11531	46440	11557

Table 4.47. Increase in the number of page-table accesses (Random Configurations)

evidenced by somewhat reduced execution times and the transaction completion times. In the case of parallel disks, the queue length at the data disks increases somewhat as the disk-address requests are satisfied at a faster rate. This increase in queue-length causes total data I/Os to decrease as the probability of accessing more than one page in parallel increases with an increase in the queue length.

The reason for the improvement in the performance with the smaller block sizes is that the product of total page-table disk accesses with the average access time decreases with the smaller block sizes indicating that the gain in the reduced access times outweighs the loss due to a higher number of accesses. Another positive aspect of using smaller block size, not shown here, is that for the same size of the page-table buffer in bytes, a larger number of pages can fit in the buffer which would improve the page-hit ratio.

Tables 4.49 through 4.51 show the effect of reducing the size of the page-table pages in the case of sequential transactions. Again, with smaller block sizes, the number of accesses increases and the access time decreases. However, unlike random transactions, with sequential transactions, the product of the page-table disk accesses and the average access time decreases for the block size of 2048 bytes but then starts increasing for smaller sizes. With

Block Size (bytes)	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	11.01	4016.46	0.03	0	0.75	59170	94.98	16.52				
4028	10.98	4066.86	0.03	0	0.75	59170	94.94	16.48	0.06	1559	0.02	25.68
2048	10.99	4061.33	0.03	0	0.75	59170	94.94	16.49	0.06	1658	0.02	21.30
1024	10.99	4056.84	0.03	0	0.75	59170	94.93	16.48	0.06	1922	0.02	18.52
512	10.98	4059.05	0.03	0	0.75	59170	94.93	16.49	0.06	2428	0.04	16.53

Table 4.48. Effect of the block size of the page-table disk (Conventional-Sequential Configuration)

Block Size (bytes)	Execution time	Transaction time	QPs		Data Disks				PageTable Disk			
			Utilization	Q length	Utilization	Total I/Os	Q length	Access time	Utilization	Total I/Os	Q length	Access time
Bare	1.92	758.06	0.90	54.47	0.92	13991	27.42	14.93				
4096	1.94	829.34	0.86	52.02	0.90	13818	28.28	14.87	0.34	1559	1.39	24.96
2048	1.93	811.89	0.87	52.62	0.90	13809	28.14	14.90	0.31	1657	0.89	21.11
1024	1.94	809.71	0.86	52.51	0.90	13888	28.35	14.88	0.31	1921	0.94	18.28
512	1.93	809.94	0.88	52.40	0.90	13700	27.54	14.98	0.35	2403	1.62	16.57

Table 4.49. Effect of the block size of the page-table disk (Parallel-Sequential Configuration)

Block Size (bytes)	Data Pages Read from Data Disks	PageTable Pages					
		Referenced	Updated	Conventional		Parallel	
				Read	Reread	Read	Reread
4096	47325	552	544	514	1	514	1
2048	47325	594	585	570	3	570	2
1024	47325	719	704	709	9	709	8
512	47325	946	910	938	80	938	55

Table 4.50. Increase in the number of page-table accesses (Sequential Configurations)

random transactions, when a page-table page is read, only a few entries on the page are of interest. Therefore, smaller block sizes may actually cut down the redundant information read. We have assumed that a transaction may access

250 page-table entries and each page-table entry is 4 bytes long. Therefore, with the page sizes of 4096, 2048, 1024, and 512 bytes, a sequential transaction may potentially access 25%, 50%, 100% and 100% entries respectively on a page. Hence, the increase in the number of I/Os with smaller block sizes is relatively larger for sequential transactions.

The impact of the shadow mechanism on the sequential configurations of the database machine was very marginal, and the page-table disk was under-utilized. Thus, small variations in the usage-pattern of the page-table disk has almost negligible effect on the database machine performance.

Logically Adjacent Pages not Physically Clustered

In all our simulations of the shadow mechanism so far, we had assumed that the logically adjacent pages can be kept physically clustered. We will now examine the effect on the database machine performance if the disk-page allocator assigns the first free page to the request for a new disk block, thereby scrambling logically sequential pages all over the data disk.

The effect of scrambling of the logically adjacent pages on sequential transactions is shown in Tables 4.51 and 4.52. We assume 25 query processors, 100 cache frames, 2 data disks, 1 page-table disk, and a page-table buffer of 10

	Execu-tion time	Trans-action time	QPs		Data Disks				PageTable Disk		
			Utili-zation	Q length	Utili-zation	Total I/Os	Q length	Access time	Utili-zation	Total I/Os	Q length
Bare	11.01	4016.46	0.03	0	0.75	59170	94.98	16.52			
Clustered	10.98	4066.86	0.03	0	0.75	59170	94.94	16.48	0.06	1559	0.02
Scrambled	20.74	7488.51	0.02	0	0.85	59170	96.37	35.43	0.03	1559	0.01
Overwriting	24.08	10372.52	0.01	0	0.82	83360	96.72	27.83			

Table 4.51. Effect of logically adjacent pages not being physically clustered (Conventional-Sequential Configuration)

	Execu-tion time	Trans-action time	QPs		Data Disks				PageTable Disk		
			Utili-zation	Q length	Utili-zation	Total I/Os	Q length	Access time	Utili-zation	Total I/Os	Q length
Bare	1.92	758.06	0.90	54.47	0.92	13991	27.42	14.93			
Clustered	1.94	829.34	0.86	52.02	0.90	13818	28.28	14.87	0.34	1559	1.39
Scrambled	18.54	6873.45	0.02	0	0.88	54485	95.94	35.41	0.04	1559	0.01
Overwriting	2.31	913.89	0.50	24.66	0.94	12886	47.24	19.86			

Table 4.52. Effect of logically adjacent pages not being physically clustered (Parallel-Sequential Configuration)

pages. The performance of the database machine, both in terms of the execution time and the transaction completion time degrades significantly. The average time to access a data page increases by a factor of more than 2 as the logically adjacent pages are physically scattered over the whole disk. In addition, for parallel disks, this scattering causes the number of disk accesses to increase considerably as the logically sequential pages can no longer be fetched in one disk access. Since the performance of the database machine in these configurations is limited by the data disks, the adverse usage of the data disks results in a dramatic degradation of the performance.

For random transactions, accesses to the data disks are already scattered, and therefore, the scrambling due to the shadow mechanism has no adverse effect.

Overwriting Algorithm

When a shadow recovery mechanism is employed, physical clustering of logically sequential pages is not achievable unless one is willing to pay substantial disk storage penalty. If the physical clustering is not maintained, the performance is very severely affected for the sequential transactions. We, therefore, experimented with the *no-undo* version of the overwriting algorithm

described in Chapter 3. Recall that in the overwriting algorithm, a current copy separate from the original (shadow) is kept only while the transaction is active. Once the transaction commits, the shadow is overwritten with the current copy. Thus, the overwriting maintains the correspondence between the physical and logical sequentiality. This makes the page-table required with the standard shadow mechanism redundant. In the no-undo version, when committing a transaction, all the pages updated by the transaction are first written to a scratch area and then a precommit record is written to a commit list. This is followed by overwriting the original pages with the updated pages.

We made the following additions to the bare machine simulator in order to simulate the overwriting algorithm. We postulated that the commit list exists on the first data disk. Also, each disk has a scratch area for writing updated pages which is used in a circular fashion. On receipt of an updated page in the disk cache from a query processor, the page is put in the queue for the same data disk on which the shadow exists. The cache frame is released after the updated page has been written to the scratch area. When all the pages updated by a transaction have been written to disk, a precommit record is written for the transaction. Afterwards, cache-allocation requests are made for reading the updated pages from the scratch area. Once a frame is allocated, an updated page is read from the scratch area and the shadow version is overwritten with the current updated copy. The cache frame is then released. The allocation of cache for reading an updated page from the scratch area has priority over the allocation of cache for reading an updated page from a query processor's memory or for prepaging a data page for processing. We assumed 25 query processors, 100 cache frames, and 2 data disks.

The results of the performance of the overwriting algorithm for sequential transactions are presented in Tables 4.51 and 4.52 along with the effect of scrambling of logically sequential pages. The performance of the overwriting algorithm is very different for the conventional disks and for the parallel disks. With the conventional data disks, the overwriting performs much worse than the standard shadow mechanism even when the logically adjacent pages are scattered. The reason for the poor performance of the overwriting algorithm is the large increase in the number of disk I/Os. Also, the average access time increases because of the movement of the disk arm between the scratch area and the data area during the overwriting of the shadows.

With parallel disks, however, after a transaction completes, all the updated pages may be read from the scratch area potentially in one access. Similarly, all the shadows may also be overwritten in one or very few accesses. An additional advantage is that after the shadows have been overwritten, as many cache frames as the pages updated by a transaction will become free at the same time. This is very attractive for the parallel disks as the number of data pages fetched in parallel in the next access will increase. Thus, in Table 4.52, the number of disk accesses with the overwriting algorithm are less compared to the total disk accesses with the bare machine. However, the overwriting algorithm may result in a non-uniform usage of the query processors. While the shadows are being overwritten, the query processors may became idle as new data pages are not being read from disk during this time. Then, a large number of pages may be read simultaneously making all the query processors busy. The poorer utilization of the query processors and the higher average disk access time were responsible for the degradation in the execution time with the overwriting algorithm when compared to the bare machine in the parallel-sequential configuration. However, the overwriting algorithm performs much

better than the standard shadow mechanism when logical sequentiality can not be maintained on disk because of the smaller access time and less I/Os.

We experimented with the overwriting algorithm for the random transactions also. The results of the experiment are summarized in Tables 4.53 and 4.54. Compared to the standard shadow mechanism, the overwriting algorithm has poorer performance as the result of the higher number of disk I/Os. With the parallel disks, the updated pages can still be read from the scratch area in one access, as was the case with the sequential transactions. However, unlike the sequential transactions, where the shadows could also be overwritten in one access, the overwriting with the random transactions may require as many accesses as the number of updated pages.

	Execu-tion time	Trans-action time	Processors		Data Disks				PageTable Disk		
			Utili-zation	Q length	Utili-zation	Total I/Os	Q length	Access time	Utili-zation	Total I/Os	Q length
Bare	18.00	7398.41	0.02	0	0.99	59170	95.80	35.53			
Page Table	20.51	8367.19	0.02	0	0.86	59170	17.95	35.32	1.00	61392	81.65
Overwriting	26.94	12386.94	0.01	0	0.99	83360	96.36	37.57			

Table 4.53. Effect of the overwriting algorithm (Conventional-Random Configuration)

	Execu-tion time	Trans-action time	QPs		Data Disks				PageTable Disk		
			Utili-zation	Q length	Utili-zation	Total I/Os	Q length	Access time	Utili-zation	Total I/Os	Q length
Bare	16.62	6476.04	0.02	0	1.00	55236	95.50	35.49			
Page Table	20.49	8352.91	0.02	0	0.85	58045	15.53	35.27	1.00	61342	84.10
Overwriting	21.65	9341.52	0.02	0	1.00	71631	95.68	35.58			

Table 4.54. Effect of the overwriting algorithm (Parallel-Random Configuration)

Version Selection Approach

The version selection approach to avoid indirection through the page-table described in Chapter 3 is not appropriate from the performance view-point for the multiprocessor-cache class of database machines. We have shown that for random transactions, either by using large page-table buffer with 1 page-table processor or by using 2 page-table processors, the page-table accesses can be completely overlapped with the processing of data pages. The version selection approach requires that for reading a data page, all the versions of the page be fetched and then a version selection algorithm be applied. Thus, unless the disk heads are augmented with enough intelligence to perform "on-the-fly" version selection, the average time to access a data page will increase. Since the performance of our database machine architecture is limited by the I/O bandwidth, the version selection algorithm will have poor performance.

The standard shadow mechanism performs very poorly with the sequential transactions if the mechanism causes the logically adjacent pages to scatter. The logically adjacent pages can be kept physically clustered if one is prepared to pay the disk storage penalty, and in that case, the standard shadow mechanism performs very well. The version selection approach requires substantial redundant storage to hold versions, and hence for sequential transactions also, the standard shadow mechanism is preferable compared to the version selection approach.

4.6. Differential File Simulator

As in the case of the simulation of the parallel logging and the shadow mechanism, we have only modeled the recovery actions during the execution of a successful transaction in our simulation of the differential file mechanism.

4.6.1. Specifications of the Differential File Module

Recall that in the differential file approach, a relation R is expressed as (B ∪ A) - D, where B is the base relation, and A and D are the differential relations. Additions to R are appended to the A relation and the deletions are appended to the D relation. As compared to the no recovery case, where a transaction can be thought of as accessing only the B relation, with the differential file mechanism, the transaction will also have to access A and D relations to process a query. We assume that the number of A and D pages accessed by a transaction is a function of the size of A and D relations relative to the size of the base relation B. The size of A and D relations depend on the level of update activity and the frequency with which A and D relations are merged into the B relation. We assume that the A and D relations are equal in size and take this size to be 10% of the size of the B relation. We will present the results of the sensitivity analysis of the effect of the sizes of A and D relation on the performance of this recovery mechanism.

The base relation pages accessed by a transaction are considered to be same as the pages accessed by the transaction in the bare machine. When a transaction arrives, in addition to the pages of the B relation accessed by it, the A and D pages referenced by the transaction are also determined. It is ensured, by using separate random number streams, that the pages referenced by a transaction from the B relation are identical to the pages in the reference string of the transaction in the bare machine simulator.

Another consequence of the differential file mechanism is that a retrieval operation is translated into a set union and a set difference operation. We presented in Chapter 3 the algorithms that exploit the parallelism inherent in our database machine architecture to efficiently perform the operations on the

differential files. We will now describe how we model the update processing when using differential files in our simulator. All the D pages referenced by a transaction are first read in the cache. The B and A pages are then read and are assigned to the query processors when they become free. After reading a B (or A) page from the cache into its local memory, a query processor reads one D page at a time and takes the set difference of B page with this D page. In Chapter 3, we described two algorithms to perform the set-difference of the tuples in two pages depending upon whether the pages are sorted or not. If the pages are sorted, a linear scan over the B page and a linear scan over the D page will be required. If the pages are not sorted, for each tuple in B page, a scan over the D page is required. Therefore, the processing time required for this step can be approximated by the time for a selection operation on a page if the pages are sorted, and by the join time on a page if the pages are not sorted. Thus, we have assumed the time required to perform the set-difference between two pages to be 36.65 which is the same as the time required for an average operation on a page in the bare database machine (see Section 4.3.2).

Once a query processor has taken the set difference of the B page with all of the D pages that the transaction refers to, it can create the result tuples for this page. For the update operation, the result tuples consist of new values of the tuples that will be appended to the A relation and the old values of the tuples that will be appended to the D relation. We assume that if a page is updated, 10% of the tuples (standard deviation equal to 1/3 of the average) on the page are updated; otherwise, no result tuple is created. Recall that in the simulation of parallel logging, we had also assumed the size of the log fragment that is created when a page is updated to be 10% of the size of the data page. We will present the results of a sensitivity analysis of varying the size of the fraction of the output page that is created when a page is updated. The result tuples are

collected in the memory of the query processor. When a query processor has a page full of the resultant A or D tuples, it makes a cache-frame request, and upon allocation of the cache-frame, writes the A or D page to the cache. Subsequently, these A and D pages are written to disk.

We also modeled an optimization in this processing strategy. The only effect of the D relation is to eliminate some of the tuples that are otherwise in the result of a query. Therefore, before taking the set-difference of a page of B with each page of the D relation that the transaction references, a query processor may first make a scan over the B page. The set-difference with D pages is taken only if this scan yields a result tuple. To model this optimization, we take the set-difference for each B or A page that is to be updated[5]. However, the set difference is taken only for a few read-only B and A pages. A read-only page for which the set-difference will be taken is randomly determined with a probability which is equal to the fraction of the size of the D relation to the size of the B relation.

We have assumed that when a page is updated, only a fraction of an output page is created. This may lead to the problem of page-fragmentation if the data pages are not allocated to the processors judiciously. Assume that a transaction T is accessing a page P, and P is available in cache for assigning to a free query processor. Further assume that two query processors, QP1 and QP2, are free. We use the following query-processor allocation strategy algorithm to reduce fragmentation:

(1) If QP1 has a partially-filled output page belonging to T in its local memory whereas QP2 has a partially-filled output page belonging to some other

[5] As explained in Section 4.3, at the time of the transaction arrival itself, it is determined which of the pages referenced by the transaction will be updated by it.

transaction (or no output tuples), then P is assigned to QP1.

(2) If both QP1 and QP2 have partially-filled output pages belonging to T, then P is assigned to the processor whose output page is relatively less full.

(3) If all the free query processors have partially-filled output pages belonging to transactions other than T, then P is assigned to the processor whose output page is filled maximum. The processor first writes the output page to the cache and then reads P for processing.

While this strategy will minimize fragmentation, there will still be some fragmented pages. It is possible to use a compress operation [DeWi79b] to combine partially filled pages before writing them to disk. We have not modeled the compress operator in our simulation.

4.6.2. Experiments

We performed a number of experiments with our simulator of the differential file mechanism to explore the following issues:

* Effect of the differential file mechanism on the average execution times of the database machine?
* Effect of the fraction of the output page that is created when a data page is updated.
* Effect of the size of the differential relations

Effect on Database machine Performance

Tables 4.55 through 4.57 show the effect of the differential file mechanism on different performance parameters of the database machine. We have presented the results both for the *basic* query processing approach wherein a set-difference operation is performed on every page of the B and A relations, and

Configuration	Execution Time per Page			Transaction Completion Time		
	Bare Machine	Basic Approach	Optimal Approach	Bare Machine	Basic Approach	Optimal Approach
Conventional-Random	18.00	37.81	19.23	7398.41	11589.77	6634.34
Parallel-Random	16.62	37.67	17.99	6476.04	11565.10	6207.64
Conventional-Sequential	11.01	37.65	17.75	4016.46	11443.69	5795.54
Parallel-Sequential	1.92	37.55	13.90	758.06	11368.76	4573.48

Table 4.55. Performance of the Differential File Mechanism

Configuration	Bare Machine			Basic Approach			Optimal Approach		
	Utilization	Q length	Total I/Os	Utilization	Q length	Total I/Os	Utilization	Q length	Total I/Os
Conventional-Random	0.99	94.98	59170	0.51	1.93	62113	0.98	48.70	61548
Parallel-Random	1.00	95.50	55236	0.49	1.67	61902	0.99	46.33	59262
Conventional-Sequential	0.75	94.98	59170	0.27	2.36	62082	0.58	29.42	62386
Parallel-Sequential	0.92	27.42	13991	0.20	0.14	55234	0.40	0.72	45216

Table 4.56. Data Disk Characteristics

Configuration	Bare Machine			Basic Approach			Optimal Approach		
	Max QPs Used	Effective QPs	Q length	Max QPs Used	Effective QPs	Q length	Max QPs Used	Effective QPs	Q length
Conventional-Random	9	0.51	0	25	25.00	56.61	25	14.25	3.93
Parallel-Random	10	0.56	0	25	25.00	56.73	25	15.56	5.73
Conventional-Sequential	17	0.82	0	25	24.99	56.80	25	18.82	26.46
Parallel-Sequential	25	22.38	54.47	25	25.00	59.51	25	24.79	55.53

Table 4.57. Query Processors Utilization

the *optimal* approach wherein the set-difference is taken only on those pages of B and A relations that have at least one tuple that satisfies the qualification in

the query. These experiments were performed with 25 query processors, 100 cache frames and 2 data disks. In Table 4.56, Utilization is the average of the mean values of the utilization of the two data disks; whereas, Q-length and Total I/Os is the total of the queue length and the number of disk accesses respectively. We make the following observations based on Tables 4.55 through 4.57.

1. With the basic approach, the performance of the database machine both for the random and parallel transactions with the conventional as well as the parallel disks degrades very significantly. The execution time per page is almost the same for all the four configurations. The reason is that, unlike the bare machine, the performance is not limited by the I/O bandwidth between the disk drives and the cache, but rather by the query processors. Each of the 25 query processors are almost 100% utilized, whereas the utilization of the disk drives is very low. Hence, the access pattern is not a determining factor for the execution times. Since there is hardly any queue at the disk drives, with the parallel disks, only one or two data pages are accessed most of the times in each I/O operation. Thus, the total number of I/O operations when using the parallel disk drives is almost equal the number of I/O operations with the conventional disk drives.

2. Compared to the basic approach, the optimal approach reduces the degradation in performance as the set-difference operation is performed on significantly less pages. For random transactions, the query processors are no longer the bottleneck and the performance is again bound by the I/O bandwidth available from the disk drives, as evidenced by almost 100% utilization of the disk drives and about 60% utilization of each of the query processors. Compared to the bare machine, the execution times are somewhat poorer for random

transactions because of a slightly higher number of disk operations. The higher number of disk I/Os are caused by the extra accesses to the differential relations. In addition, in the parallel-disk case, the average queue length is smaller when compared with the bare machine and this decreases the average number of pages that are accessed in one I/O. The reason for the smaller queue lengths at the disks with the differential file approach is that the D pages keep the cache frames occupied until all the B and A pages for the transaction have been processed.

For sequential transactions, the average disk access time is less than half compared to the access time with the random transactions. Thus, if there are many B or A pages at one time in the cache that require set-difference operation, the query processors become the bottleneck. With the parallel disks, all the query processors are almost 100% utilized whereas each of the disks is busy only 50% of the time. With the conventional disks, the query processors are about 75% utilized and the disk drives are 58% utilized. This suggests that in the conventional-random configuration, there were instances when a disk drive was idle and query processors were busy as many set-difference operations were in progress and no cache frames were free, and at times, some query processors were forced to be idle as the data pages were not available from disk. On the other hand, in the parallel-sequential configuration, the query processors were not able to keep up with the rate at which the data pages were available from disk as the result of the extra processing required for performing the set-difference operation and the fact that, with the parallel disks, relatively fewer disk I/Os are required for accessing the same number of disk pages. However, both for conventional as well as parallel disks, the execution times for the sequential transactions are much poorer than the bare machine.

3. There is a substantial increase in the number of disk accesses due to extra differential file pages. However, this increase is somewhat ameliorated by the reduced number of updated pages. Recall that we have assumed that on average only 10% of the output page is created when a page is updated. Thus, if a transaction accesses N pages and updates u% of them, then with the differential file approach, potentially (N*u% - 0.1*N*u%) fewer updated pages will be written to disk. In practice, however, the decrease in the number of updated pages will be less than the number given by the above expression as the result of the page fragmentation. In the following subsections, we will present the results of the experiments to investigate the effect of the size of the output page created when a page is updated, and the effect of the size of the differential relations. These sensitivity experiments were performed assuming the optimal approach for performing the set-difference operation.

Fraction of the Output Page

Tables 4.58 through 4.61 show the effect of assuming that a larger fraction of the output page is created when a data page is updated. In these tables, the total number of output pages that are created is given by the Output pages, and Input pages is $\Sigma|T_i|$ where $|T_i|$ is the size of the transaction T_i. These experiments were performed assuming 25 query processors, 100 cache frames, and 2 data

Output fraction	Execution time	Transaction time	QPs		Data Disks			Total Pages	
			Utilization	Q length	Utilization	Q length	Total I/Os	Input	Output
No Recovery	18.00	7398.41	0.02	0	0.99	95.80	59170	59083	11845
10%	19.23	6634.34	0.57	3.93	0.98	48.70	61548	59083	4401
20%	19.25	6650.12	0.57	3.80	0.98	48.72	61793	59083	4646
50%	20.33	7026.50	0.52	2.05	0.98	50.95	65411	59083	8264

Table 4.58. Effect of the output fraction
(Conventional-Random Configuration)

Output fraction	Execution time	Transaction time	QPs		Data Disks			Total Pages	
			Utilization	Q length	Utilization	Q length	Total I/Os	Input	Output
No Recovery	16.62	6476.04	0.02	0	1.00	95.50	55236	59083	11845
10%	17.99	6207.64	0.62	5.73	0.99	46.33	59262	59083	4534
20%	17.99	6212.46	0.62	5.56	0.99	46.51	59376	59083	4731
50%	18.90	6531.07	0.57	3.39	0.98	48.94	62677	59083	8360

Table 4.59. Effect of the output fraction (Parallel-Random Configuration)

Output fraction	Execution time	Transaction time	Query Processors			Data Disks				Output Pages
			Utilization	Q length	Q time	Utilization	Q length	Q time	Total I/Os	
10%	16.71	5252.49	0.78	29.83	450.92	0.59	26.63	418.30	12066	1063
20%	16.86	5308.06	0.78	29.94	456.70	0.59	26.63	420.26	12115	1112
50%	16.76	5244.15	0.78	26.21	392.09	0.64	29.30	432.05	12945	1713

Table 4.60. Effect of the output fraction (Conventional-Sequential Configuration)

Output fraction	Execution time	Transaction time	Query Processors		Data Disks			Total Pages	
			Utilization	Q length	Utilization	Q length	Total I/Os	Input	Output
No Recovery	1.92	758.06	0.90	54.47	0.92	27.42	13991	59083	11845
10%	13.90	4573.48	0.99	55.53	0.40	0.72	45216	59083	5519
20%	13.90	4567.43	0.99	55.47	0.41	0.74	45330	59083	5694
50%	13.65	4503.98	1.00	54.28	0.46	0.88	47373	59083	9177

Table 4.61. Effect of the output fraction (Parallel-Sequential Configuration)

disks.

The number of output pages does not increase linearly with an increase in the output fraction. Particularly striking is the small increase in this number when the output fraction is increased from 10% to 20%. As alluded earlier, this small increase is explained by the page-fragmentation with the smaller output fractions. Also, relatively a larger number of output pages are created when the

transactions are sequential rather than random. This is because the processing of a sequential transaction is spread over a larger number of query processors compared to the case when the transactions are random, and therefore, for the same number of output tuples, there is a larger page-fragmentation.

In our query-processor allocation strategy, if a page P of transaction T is available and a query processor QP1 is free, then P is immediately assigned to QP1 even though QP1 may not have yet updated any page of T. It might be advantageous from the point of view of reducing the fragmentation to allow QP1 to idle for a while if there is another processor QP2 which is about to become free and QP2 has a partially filled output page corresponding to T. Investigation of the query-processor allocation strategies is beyond the scope of this thesis. However, since the performance of our database machine is bound by the I/O bandwidth provided by the disk drives, examining query-processor allocation strategies that minimize fragmentation appears to be an interesting research topic.

The higher number of I/O operations due to the increase in the number of output pages with an increase in the output fraction manifests itself in somewhat higher corresponding execution times for the random configurations, as in these configurations, the performance of the database machine is bound by the I/O bandwidth. In the sequential configurations, since the disk drives were under-utilized, the execution times are not affected significantly by a little higher number of disk I/Os.

Size of the Differential Relations

The effect of the increase in the size of the differential relations manifests in three ways:

(1) There are a higher number of disk I/Os as we have assumed that the number of the differential relation pages accessed by a transaction is a function of the size of the differential relations relative to the size of the base relation.

(2) The set-difference operation is performed on a higher number of B and A pages as we assume that the probability that the set difference will be performed on a read-only B or A page depends on the size of the differential relations.

(3) The time required to perform the set-difference on one page of B or A relation increases as the set-difference will have to be performed with a larger number of D pages.

The results of the simulation experiments for the higher values of the size of the differential relations expressed as percentage of the size of the base relation have been summarized in Tables 4.62 through 4.65. We assumed 25 query processors, 100 cache frames, and 2 disk drives.

The performance of the database machine, both in terms of the execution times and the transaction completion times, degrades nonlinearly in all the four configurations as the size of the differential relations increases. For higher values of the size of the differential relations, utilization of both the disk drives

Size	Execution time	Transaction time	Query Processors			Data Disks			
			Utilization	Q length	Q time	Utilization	Q length	Q time	Total I/Os
No Recovery	18.00	7398.41	0.02	0	0	0.99	95.80	1711.23	59170
10%	19.23	6634.34	0.57	3.93	68.51	0.98	48.70	898.18	61548
15%	24.81	7875.32	0.77	15.93	342.22	0.84	21.10	460.80	67084
20%	37.01	10871.46	0.77	15.37	472.81	0.61	8.27	251.47	71893

Table 4.62. Effect of the Size of the differential relations (Conventional-Random Configuration)

Size	Execution time	Transaction time	Query Processors			Data Disks			
			Utilization	Q length	Q time	Utilization	Q length	Q time	Total I/Os
No Recovery	16.62	6476.04	0.02	0	0	1.00	95.50	1580.80	55236
10%	17.99	6207.64	0.62	5.73	93.31	0.99	46.33	797.81	59262
15%	24.41	7699.32	0.80	18.42	389.46	0.81	18.38	394.83	65848
20%	37.01	10852.69	0.78	16.62	511.46	0.58	6.88	209.01	71202

Table 4.63. Effect of the Size of the differential relations (Parallel-Random Configuration)

Size	Execution time	Transaction time	Query Processors			Data Disks			
			Utilization	Q length	Q time	Utilization	Q length	Q time	Total I/Os
No Recovery	11.01	4016.46	0.03	0	0	0.75	94.98	1043.40	59170
10%	17.75	5795.54	0.75	26.46	425.35	0.58	29.42	494.48	62386
15%	25.79	7881.45	0.82	25.93	579.25	0.44	13.68	309.37	67351
20%	39.56	11337.05	0.77	20.24	665.40	0.33	5.93	192.41	72065

Table 4.64. Effect of the Size of the differential relations (Conventional-Sequential Configuration)

Size	Execution time	Transaction time	Query Processors			Data Disks			
			Utilization	Q length	Q time	Utilization	Q length	Q time	Total I/Os
No Recovery	1.92	758.06	0.90	54.47	104.77	0.92	27.42	52.65	13991
10%	13.90	4573.48	0.99	55.53	699.14	0.40	0.72	9.42	45216
15%	23.53	7194.68	0.96	39.49	805.22	0.28	0.40	8.22	51168
20%	36.44	10516.56	0.88	24.80	751.08	0.21	0.24	7.33	56041

Table 4.65. Effect of the Size of the differential relations (Parallel-Sequential Configuration)

and the query processors starts decreasing. This is because, during the time the D pages referenced by a transaction are being read from disk, the query processors become idle, and a disk drive becomes idle after reading a B or A page into cache while the query processors are performing the set-difference operations.

In order to keep the size of the differential relations small, the differential relations will have to be merged with the base relation frequently. In our simulation, we have not modeled the effect of merging of differential relations with the base relation. Designing optimum merging frequencies is an intersting open issue.

4.7. Comparison of the Recovery Mechanisms

Tables 4.66 shows the comparative performance of the three recovery mechanisms in terms of their impact on the average execution time per page of the database machine. The bare database machine consisted of 25 query processors, 100 cache frames, and 2 data disks. The logging results assume only one log disk and an effective bandwidth of 1 megabytes per second between the query processors and the log processor. The results for the shadow mechanism have been presented for i) one page-table disk and a page-table buffer capable of holding 10 page-table pages, ii) one page-table disk and a page-table buffer of 50 page-table pages, and iii) two page-table disks and a page-table buffer of 10 pages. It is assumed in the above three cases that the logically adjacent pages may be kept physically clustered. The numbers under the scrambled column

Configuration	Bare Machine	Logging	Shadow					Differential File
		1 log disk	1 PageTable Disk buffer=10	1 PageTable Disk buffer=50	2 PageTable Disks	Scrambled	Overwriting	
Conventional-Random	18.00	17.86	20.51	18.01	17.99	20.51	26.94	19.23
Parallel-Random	16.62	16.50	20.49	16.70	16.69	20.49	21.65	17.99
Conventional-Sequential	11.01	11.39	10.98	10.99	10.99	20.74	24.08	17.75
Parallel-Sequential	1.92	2.05	1.94	1.93	1.93	18.54	2.31	13.90

Table 4.66. Average Execution Time per Page

correspond to the case when this assumption is not satisfied, and the logically adjacent pages are scattered all over the disk. For this case also, we assume one page-table disk and a page-table buffer of 10 pages. The results of using the overwriting algorithm, wherein on transaction completion the shadows are overwritten with the current copies, are summarized under the overwriting column. For the differential file mechanism, the results have been presented assuming the size of the differential relations to be 10% of the size of the base relation.

The differential file mechanism degrades the throughput of our database machine architecture even when the size of the differential relations was assumed to be only 10% of the base relation. The degradation increases nonlinearly with an increase in the size of the differential relations. The degradation in the throughput is due to the extra disk I/Os to access the differential relation pages and the extra processing requirements for the set-difference operation. Since the I/O bandwidth available from the disk drives is the factor limiting the throughput of the database machine, the extra disk accesses have negative impact. The extra processing requirement would not be a problem so long as the query processors do not become the bottleneck. However, for the sizes larger than 10%, and in the case of sequential transactions, even for 10% size, the query processors become saturated and adversely affect the throughput of the database machine.

In the case of the shadow mechanism, for random transactions, the throughput degrades somewhat when 1 page-table processor is used with a page-table buffer of 10 pages. However by increasing the buffer size to 50 or by using 2 page-table processors, the reading and updating of page-table entries may be overlapped with the processing of data pages and there is virtually no

degradation in the performance due to the recovery mechanism. For sequential transactions, if it is assumed that the logically adjacent pages may be kept physically clustered, then the performance of the shadow mechanism is very good. In practice, this assumption is difficult to justify, and if the physical clustering of the logically sequential pages cannot be maintained, then the shadow mechanism performs very poorly for sequential transactions. The poor performance is due to the relatively large seek times. if the data pages are scattered on disk. The overwriting algorithm maintains the correspondence between the physical and logical sequentiality and avoids the need of indirection through a page table. However, the overwriting algorithm performs poorer than the standard shadow mechanism for random transactions and also in the case of sequential transactions when the data disks used are the conventional disk drives. The reason for the poorer performance is the extra accesses to the data disks that are required with the overwriting algorithm. Whereas accesses to the page-table disk in the standard shadow mechanism may be overlapped with the processing of data pages, the overwriting algorithm is not amenable to such overlapping. When parallel disks are used in the processing of sequential transactions, however, current copies may be read from the scratch area and the shadow copies may be overwritten in very few disk I/Os, and hence, the overwriting algorithm has quite good performance.

Overall, the parallel logging emerges as the best recovery mechanism for our database machine architecture as the recovery actions may be completely overlapped with the processing of data pages. We have seen that the communication medium between the query processors and the log processor have no significant effect on the performance of logging. In particular, the performance of the logging was not degraded when the log pages were routed through the disk cache, and hence, an interconnection dedicated to

communicating the log pages is not necessary between the query processors and the log processor. Therefore, the parallel logging can be implemented with no modification in the hardware architecture of our database machine by simply designating one or more query processors as the log processor and supplementing them with the log disks. It was shown in Section 4.3.2 that the rate of processing of data pages is not fast enough to warrant more than one log disk. However, if the data processing rate is improved in the future by solving the problem of I/O bandwidth available from the mass-storage devices, then logging can still be performed in parallel by using more than one log disk and our parallel logging algorithm.

CHAPTER 5

CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

In this dissertation, we have presented mechanisms for concurrency control and recovery in multiprocessor database machines. We have also analyzed the relative performance of various mechanisms and their impact on database machine performance. While the multiprocessor-cache class of database machines have been the focus of our research, we have also enumerated how our design can be adapted to the other classes of database machines.

We showed that for concurrency control, a centralized 2-phase locking scheduler with deadlock detection is most appropriate. The scheduler may be located with the back-end controller or a separate processor may be entrusted with the task of concurrency control to whom the back-end controller may inquire before assigning a data page to a query processor. Amongst recovery mechanisms, the parallel logging was found to have the overall best performance. With the architecture that we have proposed for logging, it is possible to completely overlap the recovery actions with the processing of data pages so that the throughput of the database machine is not degraded by the recovery mechanism.

Some other very useful results emerged as a consequence of this research. In Chapter 2, we proposed that the concurrency control and recovery are intimately related and described the interaction between different concurrency control and recovery mechanisms. We also designed six integrated concurrency control and recovery mechanisms in the context of centralized database

management systems and evaluated their performance. We developed an analytical model for this evaluation that helped us in isolating and quantifying the costs of the various components of a mechanism. Thus, in addition to saying that a particular mechanism is expensive, we were able to determine why it is expensive and where the efforts should be concentrated to improve its performance. This analysis provided us with the framework for designing the parallel recovery algorithms presented in Chapter 3.

We have extended the shadow and differential file mechanisms for use in a multi-transaction environment. We also designed a linear deadlock detection algorithm [Agra83a] which can be used by the concurrency control module of the back-end controller. In addition, we have proposed a solution to the update problem in hypothetical databases [Agra83b] that permits the reinsertion of a previously deleted tuple, while preserving the append-only nature of the differential and addition and deletion relations, which is necessary for using this approach as a recovery mechanism.

In Chapter 3, we presented parallel recovery mechanisms. Particularly interesting is the parallel logging algorithm that allows logging for a transaction to be performed asynchronously at more than one log disks, and yet does not require physical merging of distributed logs to recover from failures. We have also shown how to take system checkpoints in parallel with the normal data processing and logging activities. Although these mechanisms have been designed in the context of database machines, they may easily be adapted for use in any high performance database management system. For example, the main-frame computers with the main memory in the order of gigabytes are on the horizon. It will be possible to store the entire database in the main memory of such computers [Gray83a]. Our parallel logging algorithm can be gainfully

used to log updates in parallel in such a high performance environment.

Besides presenting the results of the comparative performance evaluation of the parallel recovery mechanisms and their impact on database machine performance, Chapter 4 contains an interesting result that the performance of the multiprocessor-cache class of database machines is severely limited by the I/O bandwidth of the disk drives. It has been suggested that in such machines several query processors (hundreds of them) may be used to process a database query in parallel. We demonstrated that if the current state-of-art disk drives, like IBM 3350 disks, are used for storing the database, then for 2 disk drives, not even 10 query processors are adequately utilized[1].

One way of increasing the I/O bandwidth is to use parallel-readout disk drives. For sequential transactions, when parallel disks were used the throughput of the database machine increased by a factor of more than 5 compared to the conventional disks. We determined that by using 75 query processors and 150 cache frames, instead of 25 query processors and 100 cache frames, throughput in the case of sequential transactions can be improved by a factor of more than 10. However, parallel disks do not necessarily solve the I/O bandwidth problem. If the accesses are random so that there are not many pages belonging to the same cylinder in the disk queue, the parallel-accessing capability of a parallel disk becomes redundant. Similar results are expected if the disk controller is augmented with a large internal cache so that it reads the whole cylinder at a time instead of reading one block at a time.

---

[1] For random transactions, the throughput of the database machine using 2 query processors was found to be the same as the throughput obtained when 25 query processors were used. For sequential transactions, the same throughput as 25 query processors could be achieved by using 8 query processors.

What then is the solution to the I/O bandwidth problem? May be, build a database machine so that the whole database or a very large part of it is always memory resident and use our parallel logging algorithm to log all changes. As many log disks as necessary may be used to avoid degradation in the throughput. This will, in addition, require efficient algorithms for incrementally saving the image of database on stable storage while the database is in operation so that the database may be reconstructed in acceptable time after a system crash. The feasibility and the details of the architecture, the incremental dumping and the database reconstruction algorithms, the query processing strategies in such an architecture, all appear to be very promising subjects for future research.

The conventional wisdom is that, for best results, if a device is free and there is a task to be performed, let the device start working on the task immediately. In our simulations, we observed many situations where the forced idleness was a better choice. For example, suppose that two adjacent pages P1 and P2 are to be read from the same cylinder of a parallel disk. Further, suppose that two cache frames become free at time t and $t+\Delta t$ respectively where Δt is very small compared to the disk access time, and disk is free at time t. If the disk begins accessing P1 at time t, then a separate access will be required to access P2. However, if the disk is kept idle for Δt time, then both P1 and P2 may be read in one disk access. Similarly, suppose that a page P referenced by transaction T has been read into the cache. For avoiding fragmentation of output pages, it may be desirable to let a free processor idle and assign P to another processor which is currently busy processing a page belonging to T. It will be interesting to explore further the scheduling of parallel disks and the query processor allocation from this point of view.

On conventional disks also, for servicing access requests, we defined various classes of customers with different class priorities, but in all our simulations, the service discipline within a class was assumed to be first-come first-serve. It will be worth investigating the effect of other disk-scheduling strategies on database machine performance.

For many recovery algorithms, it is necessary to maintain a list in stable storage that should survive system crash. For example, in the version-selection selection approach described in Chapter 3, a list of transactions active at the time of crash is required to recover from system crash. The transaction-id is added to this list when a transaction starts, and on transaction completion, the transaction-id is deleted. Maintaining such a list on a pseudo-random device like disk where the unit of access is a block is inefficient and clumsy. It will be very advantageous if the database machine is augmented with some truly random-access storage that survives power failures (for example, nonvolatile RAM[2]) for such functions. This stable random-access storage will have other interesting consequences for the recovery mechanisms in the conventional database management systems also.

We adopted a modular approach to building simulators for performance analysis. We first built modules for the different devices like disk, cache, and query processors and then these modules were glued together to realize the simulator for the bare database machine. The bare database machine simulator was in turn augmented with the modules for different recovery mechanisms. Besides cutting down on the development time, this approach ensured that we

---

[2] One kilobit nonvolatile RAMs that are hybrids of static RAM and EEPROM are currently available, and 4K units are in development at Xicor (Milipitas, California) and elsewhere. Recently, a cell combining a dynamic RAM and an EEPROM has been designed at United Technologies' Mostek subsidiary (Carrollton, Texas), and a 16K nonvolatile RAM may soon be readily realizable [Posa83a].

were not comparing apples with oranges. We used SIMPAS [Brya80a], a simulation language based on PASCAL [Jens74a], for constructing our simulators. While SIMPAS was of great help and saved many months of programming effort, we felt constrained on two counts. First of all, PASCAL does not provide sufficient information hiding, and secondly, PASCAL being a sequential language, some of the parallel processing constructs are difficult to code in it. At one level, it will be nice to have a simulation language based on a programming language like Ada [Ichb79a] that supports the notion of parameterized modules, provides separate compilation and library facility, and contains language constructs for expressing parallel processing. On another level, it will be very useful to build a library of modules using such a simulation language that a performance analyst may use off-the-shelf. Such tools will be very useful to a database-machine designer also to balance the different components of the design and examine its performance under various load conditions.

APPENDIX 1

NOTATION

BX	Total extra cost in running a transaction because of recovery & concurrency control
B_{fail}	Extra cost incurred when a transaction is aborted by the user
B_{rerun}	Extra cost incurred when a transaction is aborted by the system
B_{setup}	Fixed extra cost irrespective of the ultimate fate of the transaction
B_{succ}	Extra cost incurred when a transaction succeeds
DBSize	Size of the database
DBuff	Number of Data buffers allocated to a transaction
DFlush	The function that returns the number of updated data pages that have been flushed to the disk at some time, given the total number of updated pages
Comprs%	The number of differential file pages generated by a transaction is Comprs% of the data pages updated by it.
CpuOH%	With differential files, extra cpu time required to process a transaction is CpuOH% of the cpu time consumed if the transaction was run alone without any provision for recovery
LBuff	Number of buffers available to a transaction to collect log records
Log%	The number of log pages generated by a transaction is log% of the data pages updated by it.
LFlush	The function that returns the number of log pages that have been flushed to the disk at some time, given the total number of log pages
MPL	Level of multiprogramming
NP_t	Total number of pages accessed by a transaction
NP_u	Number of pages updated by a transaction

$P_{conflict}$	Probability that an access request of a transaction would conflict with that of another transaction
P_{ddlk}	Probability that a lock request of a transaction would result in a deadlock
P_{fail}	Probability that a transaction would be aborted by the user
P_{rerun}	Probability that a transaction would be aborted by the system
P_{succ}	Probability that a transaction would complete
P_{wait}	Probability that a lock request of a transaction would be blocked
PtPages	The function that determines the number of page-table pages that would be accessed to access certain number of data pages
SBuff	Number of buffers available to a transaction to get pagetable pages
SFlush	The function that returns the number of page-table pages that are no longer available in the memory, given the total number of page-table pages read by the transaction
Size%	The size of the differential files is Size% of the number of pages in the base file
T_{al}	Time to process a grant-lock request
T_{as}	Time to create control sets (read, write, active etc.) in the optimistic method of concurrency control, if $NP_t = 1$
$T_{l\text{-}io}$	Time to read/write a disk page with disk seek
T_{page}	Cpu time to process a page in memory
T_{rec}	Cpu time to process a record in memory
T_{rl}	Time to process a release-lock request
$T_{s\text{-}io}$	Time to read/write a disk page without a seek
T_{valid}	Time to validate a transaction in the optimistic method of concurrency control if there is only one concurrent transaction
T_{wait}	Wait time for a blocked lock request

APPENDIX 2

LINEAR DEADLOCK DETECTION

1. Introduction

Deadlocks arise in database systems in the context of concurrency control algorithms based on locking. Deadlocks are typically characterized in terms of a waits-for graph [Holt72a, King73a], a directed graph that represents which transactions are waiting for which other transactions. In this appendix, we will present some special properties of the waits-for-graph in the context of database systems, and present very efficient *linear* deadlock detection algorithms.

The organization of the appendix is as follows. In Section 2, we describe our assumptions regarding the locking protocol used for concurrency control. In particular, we assume all locks to be exclusive. We outline some important properties of a waits-for-graph in Section 3. Our *continuous* deadlock detection algorithm is presented in Section 4. The theoretical basis for the algorithm is presented in Annexure 1. In Section 5, we relax our assumption about the locks being exclusive to allow shared read-locks and present our modified deadlock detection algorithm. The proof of correctness of the modified algorithm is given in Annexure 2. In Section 6, we extend our algorithm to perform *periodic* deadlock detection.

2. Assumptions

We make the following assumptions about the locking protocol:

(1) The locking protocol is the *strict two-phase* protocol, that is, a transaction holds all its locks till its completion[1].

(2) A transaction requests one lock at a time and is blocked if a lock cannot be granted.

(3) All locks are exclusive.

3. Waits-for Graph in Database Systems

Deadlocks have been expressed in terms of waits-for graphs. It has been shown [Holt72a, King73a] that *there exists a deadlock if and only if there is a cycle in the waits-for graph.* A waits-for graph G is a directed graph whose vertices represent transactions and an edge $(T_i,T_j) \in G$ if the transaction T_i is waiting for a lock owned by T_j. We will say that T_i is **waiting* on T_j if there is a path from T_i to T_j in the waits-for graph.

Management of the Waits-for Graph

The waits-for graph is maintained by the lock manager. For each locked object, the lock manager keeps the transaction number of the owner of the lock and a queue of the transactions that are waiting for the object to become free. We will assume that the queue discipline is 'first in first out (FIFO)'[2]. Before allowing a transaction T_i to wait for a transaction T_j[3], the lock manager checks that the addition of the edge (T_i,T_j) to the waits-for graph will not result in a cycle in the graph. The edge (T_i,T_j) is added to the graph and T_i is blocked, only if this test succeeds. When a lock is released and a blocked transaction is

---

[1] Gray [Gray78a] has shown that to avoid a cascade of transaction aborts, a transaction must hold all the locks until it executes the commit action, and then release all the locks together.

[2] Other queue disciplines can be implemented with straight forward modifications to the algorithm presented in this paper.

[3] T_j is the transaction immediately preceding T_i in the FIFO queue.

activated, or when a transaction completes, the waits-for graph is appropriately modified.

Properties of a waits-for graph

(1) A cycle-free waits-for graph is a forest of trees (Theorem 2 in Annexure 1).

(2) If the transaction T_i waits for T_j, then deadlock can occur if and only if T_i is an ancestor of T_j, that is, T_j is *waiting for T_i. (Theorem 4 in Annexure 1).

(3) Only the transactions corresponding to the roots in a cycle-free waits-for graph are active. All descendants of each of the roots are blocked *waiting for the root (Theorem 3 in Annexure 1). Thus, a cycle is created only when the transaction corresponding to a root waits for one of its descendants.

(4) Any connected subgraph of a waits-for graph can have at most one cycle (Theorem 5 in Annexure 1).

4. Continuous Deadlock Detection Algorithm

The basic idea of the algorithm is that whenever a transaction T_i requests a lock owned by T_j, test if T_j is *waiting for T_i. This test is performed by taking a directed walk starting from T_j to the root of the tree. A deadlock occurs, only if the root corresponds to T_i.

Data Structures

Assume that each transaction is assigned a unique transaction number. Define the following data structure:

```
Tran : Array[0..N-1] of {
       Waiting-for : transaction#;
       SomeOne-waiting : boolean}.
```

N is a prime number that is used to map a transaction number (by taking mod)

to an array index[4]. If the transaction t_i is blocked for a lock held by t_j, then Tran[t_i].Waiting-for = t_j. Tran[t_i].SomeOne-waiting is true only if at least one transaction is waiting for t_i to complete.

Deadlock Management Module

```
Chk-cycle(t_i,t_j : transaction#) {  -- t_i requests a lock held by t_j

 if (Tran[t_i].SomeOne-waiting is false) then { -- deadlock not possible (Theorem 4)
       Add-edge(t_i,t_j);
       return(o.k.)}
    else  { -- take a directed walk from t_j
       ancestor := Tran[t_j].Waiting-for;
       loop {
          if (ancestor = t_i) then   -- t_j *waiting for t_i
             return(deadlock)
          else if (ancestor = Null) then { -- t_j not a descendant of t_i
             Add-edge(t_i,t_j);
             return(o.k.)};
          ancestor := Tran[ancestor].Waiting-for;
          } -- end loop
       }
    }


Activate(t_i : transaction#) {
    Tran[t_i].Waiting-for := Null;
    }
```

[4] Collisions may be handled using standard techniques [Knut73a].

```
Terminate(t_i : transaction#) {
    Tran[t_i].SomeOne-waiting := false;
    }

Add-edge(t_i,t_j : transaction#) {
    Tran[t_i].Waiting-for := t_j;
    Tran[t_j].SomeOne-waiting := true;
    }

Initialize {
    for i:=0 to N-1 do {
        Tran[i].Waiting-for := Null;
        Tran[i].SomeOne-waiting := false};
    }
```

Observations

Note that the loop in the function, Chk-cycle, always terminates because of Theorem 6 in Annexure 1. The loop is executed as many times as the path length, PL, from t_j to the root of the tree[5]. In the worst case, PL = number of blocked transactions in the connected subgraph of the waits-for graph that contains the vertex at which the function Chk-cycle begins the search.

An optimization has been built into the algorithm by keeping track for each transaction whether any transaction is waiting for it. Theorem 7 in Annexure 1 provides the basis for maintaining this information. The field, SomeOne-waiting,

[5]The tree that contains the vertex corresponding to t_j.

will avoid the execution of the loop in all the cases where there is no transaction waiting for t_i. Thus, the deadlock detection will be still more efficient.

The space complexity of the algorithm is O(N).

5. Shared Read Locks - An Embellishment

The deadlock detection scheme presented in the previous section is based on the assumption that all locks are exclusive. However, many real systems allow read-locks to be shared and only write-locks are required to be exclusive. In such an environment, the number of outgoing edges from a vertex in the waits-for graph is not bounded by one (Lemma 1 in Annexure 1) and the deadlock detection scheme described in the previous section is not directly applicable.

Modified Deadlock Detection Scheme

We will present a modification in the way the waits-for graph is managed that will guarantee that there is at most one outgoing edge from each of the vertices of the waits-for graph. With this modification, the deadlock detection scheme presented in the previous section can be used.

When a writer T_i wishes to wait on a read-lock and there are more than one readers, the lock manager selects *one* of the current readers, T_j, ensures that the addition of the edge T_i->T_j would not create a cycle, and adds T_i->T_j to the waits-for graph, Later, when T_j commits, the lock manager checks if there are still readers. If not, T_i is granted the lock and is allowed to proceed. If yes, T_i->T_j is changed to T_i->T_k for some ongoing reader T_k, if it does not introduce a cycle.

Observations

With this modification, the deadlock detection algorithm is still linear in time. However, we loose the *immediate* deadlock detection property, that is, a deadlock may not be detected as soon as it arises, although all deadlocks are eventually detected. To see this, suppose that T_1 and T_2 have read locks on X and T_1 is blocked for T_3. If T_3 now wants to update X then we have a deadlock situation. However, if the lock manager adds T_3->T_2 then the deadlock will not be detected until T_2 completes, at which time the edge T_3->T_1 is added and the cycle is found.

The proof of correctness is presented in Annexure 2.

6. Periodic Deadlock Detection

In this section, we present an outline of the extension to the continuous deadlock detection scheme that enables periodic deadlock detection in linear time. With periodic detection, instead of checking for a cycle before adding an edge to the waits-for graph, edges are added to the graph without any test and the graph is periodically examined for cycles.

Define a function Detect-cycle(v) analogous to the Chk-cycle function defined previously that causes a directed walk in the waits-for graph starting from the vertex v. The walk will either terminate at a root or will again reach v, in which case, a cycle has been detected. Detect-cycle marks every vertex that it touches in the process of searching for a cycle as *visited*. We will now present the periodic deadlock detection algorithm.

```
Periodic-detection {

    for index := 0 to N-1 do Tran[index].Visited := false;   -- initialize

    index := 0;
    while (index < N) {
        Detect-cycle(index);
        while ((Tran[index].Visited is True) And (index < N)) index := index +1;
        }
    }
```

The algorithm simply runs the function Detect-cycle on the first vertex, advances to the next unvisited vertex, runs Detect-cycle there, etc. In other words, it runs our linear deadlock detector at every connected subgraph of the waits-for graph. The time complexity of the algorithm is O(N), that is, it is linear in the total number of blocked transactions.

Annexure 1

Definitions

We will first introduce some graph-theoretic definitions adapted from [Deo74a, Hara72a].

A *directed graph* (or a *digraph* for short) G consists of a set of *vertices* $V = \{v_1, v_2, \ldots\}$, a set of *edges* $E = \{e_1, e_2, \ldots\}$, and a mapping that maps every edge onto some *ordered* pair of vertices (v_i, v_j). A vertex is represented by a point and an edge by a line segment between v_i and v_j with an arrow directed from v_i to v_j. The vertex v_i is called the *initial vertex* and v_j the *terminal vertex* of the edge.

The number of edges incident out of a vertex v_i is called the *out-degree* of v_i and is written $d^o(v_i)$. The number of edges incident into v_i is called the *in-degree* of v_i and is written $d^i(v_i)$. A *sink* is a vertex v_i with $d^o(v_i) = 0$.

A *(directed) walk* in a digraph is an alternating sequence of vertices and edges, $\{v_o, e_1, v_1, \ldots e_n, v_n\}$ in which each edge e_i is (v_{i-1}, v_i). A *closed walk* has $v_n = v_0$. A *path* is a walk in which all vertices are distinct; a *cycle* is a nontrivial closed walk with all vertices distinct (except the first and the last). An edge having the same vertex as both its initial and terminal vertices is called a *self-loop*. If there is a path from v_i to v_j, then v_j is said to be *reachable* from v_i. The *length* of a path is the number of vertices involved in the path.

Each walk is directed from the first vertex v_0 to the last vertex v_n. We need a concept that does not have this directional property. A *semiwalk* is again an alternating sequence $\{v_o, e_1, v_1, \ldots e_n, v_n\}$ of vertices and edges but each edge e_i may be either (v_{i-1}, v_i) or (v_i, v_{i-1}). A *semipath* and a *semicycle* is analogously defined.

A digraph is said to be *connected* if there is at least one semipath between every pair of its vertices; otherwise, it is *disconnected*. It is easy to see that a disconnected graph consists of two or more connected subgraphs. Each of these connected subgraphs is called a *component*.

An *in-tree* is a digraph G such that 1) G contains neither a cycle nor a semicycle, 2) G has precisely one sink. This sink is called the *root* of the in-tree.

Characteristics of a waits-for graph[6]

THEOREM 1. *A waits-for graph does not have any self-loop.*

Proof. A transaction does not wait for a lock that it owns itself.

LEMMA 1. *Assuming all locks to be exclusive, for all vertices v_i in a waits-for graph, $d^O(v_i) \leq 1$.*

Proof. A transaction cannot wait for more than one transaction at a time.

LEMMA 2. *G is a digraph with n vertices. If there is a unique semipath between every two vertices of G, then the number of edges in G = n-1.*

Proof. Theorem 4.1 in [Hara72a].

LEMMA 3. *In a digraph, the sum of the out-degrees of all vertices is equal to the number of edges in the digraph.*

Proof. Each edge contributes exactly one out-degree.

LEMMA 4. *Any component of a waits-for graph cannot have more than one sink.*

Proof. Suppose a component G has two sinks v_0 and v_n. Since G is connected, we can find a semipath between v_0 and v_n. Extract the subgraph G' that has only the vertices and the edges comprising this semipath. Let there be p

[6] We will assume through out that the graph is non-empty

vertices in G'. By Lemma 2, number of edges in G' = p-1 and by Lemma 3, the sum of out-degrees of all vertices in G' = p-1. However, since $d^o(v_0) = d^o(v_n) = 0$, there must be some vertex v in G', and hence in G, that has $d^o(v) > 1$. But, this contradicts Lemma 1.

LEMMA 5. *An acyclic digraph has at least one vertex of out-degree zero.*

Proof. Theorem 16.2 in [Hara72a].

LEMMA 6. *A connected digraph G is an in-tree if and only if exactly one vertex of G has out-degree 0 and* all others have out-degree 1.

Proof. Theorem 16.4' in [Hara72a].

THEOREM 2. *A cycle-free waits-for graph is a forest of in-trees.*

Proof. Follows from Lemma 1, Lemma 4, Lemma 5 and Lemma 6.

LEMMA 7. *In an in-tree, there is a unique path from every vertex to the root.*

Proof. Theorem 9.3 in [Deo74a]

THEOREM 3. *The root v of each of the in-trees in a cycle-free waits-for graph corresponds to an active transaction for which all other transactions in the tree are *waiting.*

Proof. If v corresponds to a waiting transaction, then $d^o(v) = 1$, a contradiction. The second part of the theorem follows from Lemma 7.

LEMMA 8. *Blocking of a transaction that has no other transaction waiting for it cannot create a cycle in the waits-for graph.*

Proof. The vertex corresponding to a transaction that has no transaction waiting for it has no incoming edge.

LEMMA 9. *Waiting for a lock owned by an active transaction cannot result in a*

cycle in the waits-for graph.

Proof. The vertex corresponding to an active transaction has no outgoing edge.

THEOREM 4. *Wait by a transaction T_i for a lock held by T_j will result in a cycle in the waits-for graph if and only if T_j is *waiting for T_i.*

Proof. First, if T_j is *waiting for T_i, then there is a path from T_j to T_i, and the addition of the edge (T_i,T_j) will create a cycle in the waits-for graph.

Suppose now that T_j is not *waiting for T_i and the addition of the edge (T_i,T_j) creates a cycle in the waits-for graph. We will show a contradiction. If T_j is active, waiting for T_j cannot create a cycle by Lemma 9. If T_j is *waiting for a transaction T ($\neq T_i$), let us traverse the cycle created by the addition of the edge (T_i,T_j) starting from T_i. Since, there is a unique path between T_j and T (Lemma 1), the cycle should have a path between T and T_i. But that would imply that T_j is *waiting for T_i.

LEMMA 10. *A vertex can be on at most one distinct cycle in a waits-for graph.*

Proof. Let a vertex v be on more than one distinct cycles. Starting from v and moving along the cycles, a vertex v' (v' may be v) will be reached such that there are two out-going edges from v'. But, then $d^{o}(v') > 1$, and that contradicts Lemma 1.

THEOREM 5. *Any component of a waits-for graph can have at most one cycle.*

Proof. The transaction T corresponding to the root of a cycle-free component of a waits-for graph is the only active transaction amongst the transactions involved in the component. Hence, by Theorem 4, only a wait by T can cause a cycle in the component, and by Lemma 10, T can cause at most one cycle.

THEOREM 6. *A directed walk from any vertex in a component of a waits-for graph would result either in detection of the cycle or termination at the root.*

Proof. Follows from Lemma 7 and Theorem 5.

THEOREM 7. *The in-degree of a vertex in the waits-for graph of a database system increases monotonically until the vertex is removed from the graph with the completion of the corresponding transaction.*

Proof. A transaction holds all the locks until its completion [Gray78a].

Annexure 2

Model

We postulate a waits-for graph, as before, whose vertices are transactions and edges (called *explicit* edges from here on) of the form T_i->T_j indicate that T_i is explicitly waiting on T_j. For proof purposes, there is also a collection of *implicit* edges of the form T_i->T_k, indicating that T_k is one of the *other* readers of some object X that T_i wishes to write. These implicit edges would be present in a waits-for graph, if we were not trying to bound the out-degree of vertices by one. Let the graph with just explicit edges be called the *E-graph*, and the graph with both explicit and implicit edges be called the *EI-graph*.

Proof of Correctness

Our algorithm will be deemed correct if it can be shown that no cycle (implicit or explicit) in the EI-graph can persist forever.

LEMMA 1. *If a vertex in the EI-graph has an outgoing implicit edge, there must be at least one outgoing explicit edge from it.*

Proof. A transaction T_i can implicitly wait for a transaction T_k if T_i requests a write-lock for some object X that has been read-locked by T_k and by at least one other transaction T_j ($j \neq k$), and T_i chooses T_j to explicitly wait for. The other scenario where T_i can implicitly wait for T_k is when T_i is already explicitly waiting for some transaction T_j that holds a read-lock on X, and T_k arrives later on and is also granted a read-lock on X. In both situations, the implicit edge T_i->T_k cannot be present without the explicit edge T_i->T_j being present as well.

Now, when transaction T_j commits, the explicit edge T_i->T_j is removed and an implicit edge T_i->T_m is made explicit. If k = m, the implicit edge T_i->T_k is now explicit; otherwise, the implicit edge T_i->T_k remains implicit, but still has a related outgoing explicit edge.

LEMMA 2. *Sinks in the E-graph are also sinks in the EI-graph.*

Proof. Suppose there is a vertex v that is a sink in the E-graph but not in the EI-graph. The vertex v cannot have an outgoing explicit edge in the EI-graph because, by definition of the E-graph, the same edge would be outgoing from v in the E-graph and v is a sink in the E-graph. If v has an outgoing implicit edge, then by Lemma 1, there must be an outgoing explicit edge from v as well, and that is not possible. Hence, v is a sink in the EI-graph as well.

LEMMA 3. *If our deadlock detection algorithm is applied to the E-graph, then, at any time, the E-graph will have at least one sink.*

Proof. By Theorem 2 in Appendix A, a cycle-free E-graph is a forest of in-trees and our deadlock detection algorithm never allows any cycle to be formed in the E-graph.

THEOREM 1. *If our deadlock detection algorithm is applied to the E-graph, we cannot reach a state in which no transaction can proceed.*

Proof. Either no transaction is waiting, or by Lemmas 2 and 3, at any time, there is at least one sink in the EI-graph that corresponds to a runnable transaction.

COROLLARY 1. *Cycles in the EI-graph cannot persist forever.*

Proof. Assume that we quiesce the system, in the sense that no new

transactions are allowed to enter. By Theorem 1, at any time, there is at least one runnable transaction in a non-empty system. Assuming that transactions are finite in length, all transactions will eventually either commit or abort. Either way, all vertices are eventually removed from the graph, and hence, all cycles eventually go away.

REFERENCES

[Agra83a]R. Agrawal, M. Carey, and D.J. DeWitt, "Deadlock Detection Is Cheap," *ACM-SIGMOD Record 13*, 2, pp. 19-34 (Jan. 1983) Also Electronics Research Lab. Mem. No. UCB/ERL M83/5, Univ. California, Berkeley, Jan. 1983.

[Agra83b]R. Agrawal and D.J. DeWitt, "Updating Hypothetical Databases," *Information Processing Letters 16*, 3, pp. 145-146 (April 1983).

[Aho75a]A.V. Aho, E. Hopcraft, and J.D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass. (1975).

[Alsb76a]P.A. Alsberg, G.G. Belford, J.D. Day, and E. Grapa, "Multi-Copy Resiliency Techniques," Center for Advanced Computation Doc. 202, Univ. Illinois, Urbana-Champaign, Illinois (May 1976).

[Babb79a]E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Trans. Database Syst. 4*, 1, pp. 1-29 (March 1979).

[Banc80a]F. Bancilhon and M. Scholl, "Design of a Backend Processor for a Data Base Machine," *Proc. ACM-SIGMOD 1980 Int'l Conf. on Management of Data*, pp. 93-99 (May 1980).

[Bane78a]J. Banerjee, R.I. Baum, and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," *ACM Trans. Database Syst. 3*, 4, pp. 347-384 (Dec. 1978).

[Bask75a]F. Baskett, K.M. Chandy, R.R. Muntz, and J. Palacios, "Open, Closed and Mixed Networks with Different Classes of Customers," *J. ACM 22*, 2, pp. 248-260 (April 1975).

[Bern77a]P.A. Bernstein, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)," Tech. Rep. CCA-77-09, Computer Corp. America, Cambridge, Mass. (Dec. 1977).

[Bern79a]P.A. Bernstein, D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Software Eng. SE-5*, 3, (May 1979).

[Bern81a]P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys 13*, 2, pp. 185-221 (June 1981).

[Bern82a]P.A. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control," *Proc. 8th Int'l Conf on Very Large Data Bases*, pp. 62-76 (Sept. 1982).

[Bern82b]P.A. Bernstein, Personal communication. (Oct. 1982).

[Bitt83a]D. Bitton, D.J. DeWitt, and C. Turbyfill, "Can Database Machines Do Better? A Comparative Performance Evaluation," *Proc. 9th Int'l Conf. on Very Large Data Bases*, (1983) to appear.

[Bora81a]H. Boral, "On the Use of Data-Flow Techniques in Database Machines," Computer Sciences Tech. Rep. #432, Univ. Wisconsin, Madison (May 1981) Ph.D. Dissertation.

[Bora81b]H. Boral and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," *ACM Trans. Database Syst. 6*, 2, pp. 227-254 (June 1981).

[Bora82a]H. Boral, D.J. DeWitt, D. Friedland, N.F. Jarrell, and W.K. Wilkinson, "Implementation of the Database Machine DIRECT," *IEEE Trans. Software Eng. SE-8*, 6, pp. 533-543 (Nov. 1982).

[Bora80a]H. Boral, D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," Computer Sciences Tech. Rep. #402, Univ. Wisconsin, Madison (Oct.1980).

[Brya80a]R.M. Bryant, "SIMPAS User Manual," Computer Sciences Tech. Rep. #391, Univ. Wisconsin, Madison (June 1980).

[Card75a]A.F. Cardenas, "Analysis and Performance of Inverted Database Structures," *Commun. ACM 18*, 5, pp. 253-263 (May 1975).

[Card81a]A.F. Cardenas, F. Alavian, and A. Avizienis, "Performance of Recovery Architectures in Parallel Associative Database Processors," *ACM Trans. Database Syst.*, (May 1981) submitted for publication.

[Care83a]M.J. Carey, "Modeling and Evaluation of Database Concurrency Control Algorithms," Computer Sciences Dept., Univ. California, Berkeley (1983) Ph.D. Dissertation (in preparation).

[Chan75a]K.M. Chandy, "A Survey of Analytic Models of Rollback and Recovery Strategies," *IEEE Computer 8*, 5, pp. 40-47 (May 1975).

[Chan78a]K.M. Chandy and C.H. Sauer, "Approximate Methods for Analyzing Queueing Network Models of Computing Systems," *ACM Computing Surveys 10*, 3, pp. 281-317 (Sept. 1978).

[Chou83a]H.T. Chou, D.J. DeWitt, R.H. Katz, and A. Klug, "The Design and Implementation of the Wisconsin Storage System," Univ. Wisconsin, Madison (1983) working paper.

[Coff71a]E.G. Coffman, M.J. Elphic, and A. Shoshani, "System deadlocks," *ACM Computing Surveys 3*, 2, pp. 67-78 (June 1971).

[DeWi79a]D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Trans. Computers C-28*, 6, pp. 395-406 (June 1979).

[DeWi79b]D.J. DeWitt, "Query Execution in DIRECT," *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, pp. 13-22 (May 1979).

[DeWi81a]D.J. DeWitt and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures," *Proc. 7th Int'l Conf. on Very Large Data Bases*, (Sept. 1981).

[Denn78a]P.J. Denning and J.P. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys 10*, 3, pp. 225-262 (Sept. 1978).

[Deo74a]N. Deo, "Graph Theory with Applications to Engineering and Computer Science," Prentice-Hall, Englewood Cliffs, N.J. (1974).

[Eswa76a]K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM 19*, 11, pp. 624-633 (Nov. 1976).

[Gall82a]B. Galler, "Concurrency Control Performance Issues," TR CSRG-147, Computer Systems Research Group, Univ. Toronto, Canada (Sept. 1982) Ph.D. Dissertation.

[Garc79a]H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database," Stan-CS-79-744, Computer Sciences Dept., Stanford Univ. (June 1979) Ph.D. Dissertation.

[Gele78a]E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems Under Intermittent Failures," *Commun. ACM 21*, 6, pp. 493-499 (June 1978).

[Gele79a]E. Gelenbe, "On the Optimum Checkpoint Interval," *J. ACM 26*, 2, pp. 259-270 (April 1979).

[Gray78a]J.N. Gray, "Notes on Database Operating Systems," in *Lecture Notes in Computer Science 60, Advanced Course on Operating Systems*, ed. G. Seegmuller,Springer Verlag, New York (1978).

[Gray79a]J.N. Gray, "A Discussion of Distributed Systems," Invited Lecture at the Congresso Annuale of Associazione Italiana per il Calcolo Automatico, Bari, Italy (Aug. 1979).

[Gray80a]J.N. Gray, "A Transaction Model," pp. 282-298 in *Lecture Notes in Computer Science 85, Automata, Languages and Programming*, ed. J. van Leeuwen,Springer Verlag, New York (1980).

[Gray81a]J.N. Gray, P. Homan, H. Korth, and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System," Rep. RJ3066, IBM Research Lab., San Jose, California (Feb. 1981).

[Gray81b]J.N. Gray, P.R. McJones, B.G. Lindsay, M.W. Blasgen, R.A. Lorie, T.G. Price, F Putzolu, and I.L. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys 13*, 2, pp. 223-242 (June 1981).

[Gray81c]J.N. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. 7th Int'l Conf. on Very Large Data Bases*, pp. 144-154 (Sept. 1981).

[Gray82a]J.N. Gray, Personal communication to D.J. DeWitt. (April 1982).

[Gray83a]J.N. Gray, "Practical Problems in Data Management," Invited Talk at ACM-SIGMOD 1983 Int'l Conf. on Management of Data, San Jose, California (May 1983).

[Hans73a]P.B. Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

[Hara72a]F. Harary, "Graph Theory," Addison-Wesley, Reading, Mass. (1972).

[Hawt82a]P. Hawthorn and D.J. DeWitt, "Performance Analysis of Alternative Database Machine Architectures," *IEEE Trans. Software Eng. SE-8*, 1, pp. 61-75 (Jan. 1982).

[Hell81a]W. Hell, "RDBM - A Relational Database Machine: Architecture and Hardware Design," *Proc. 6th Workshop on Computer Architecture for Non-Numeric Processing*, (June 1981).

[Holt72a]R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys 4*, 3, pp. 179-196 (Sept. 1972).

[Hsia79a]D.K. Hsiao, "Data Base Machines are Coming, Data Base Machines are Coming!," *IEEE Computer 12*, 3, pp. 7-9 (March 1979).

[IBM77a]IBM, "Reference Manual for IBM 3350 Direct Access Storage," GA26-1638-2, File No. S370-07, IBM General Products Division, San Jose, California (April 1977).

[IDM83a]IDM, "IDM 500 Reference Manual," Britton-Lee Inc., Los Gatos, California (Jan. 1983).

[Ichb79a]J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, and B.A. Wichmann, "Rationale for the Design of the Ada Programming Language," *ACM-SIGPLAN Notices 14*, 6, (June 1979).

[Iran79a]K.B. Irani and H.L. Lin, "Queueing Network Models for Concurrent Transaction Processing in a Database System," *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, pp. 134-142 (May 1979).

[Jaco82a]P.A. Jacobson and E.D. Lazowska, "Analyzing Queueing Networks With Simultaneous Resource Possession," *Commun. ACM 25*, 2, pp. 142-151 (Feb. 1982).

[Jens74a]K. Jensen and N. Wirth, "PASCAL User Manual and Report," 2nd edition, Springer-Verlag (1974).

[King73a]P.F. King and A.J. Collmeyer, "Database Sharing - An Efficient Method for Supporting Concurrent Processes," *Proc. AFIPS 1973 Natl. Computer Conf.*, pp. 271-275 (1973).

[Knut73a]Knuth, D.E., "The Art of Computer Programming: Fundamental Algorithms," Vol. 1, 2nd edition, Addison-Wesley, Reading, Mass. (1973).

[Kohl81a]W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys 13*, 2, pp. 149-183 (June 1981).

[Kung81a]H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst. 6*, 2, pp. 213-226 (June 1981).

[Lamp79a]B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Computer Science Lab., Xerox PARC (1979).

[Lang77a]T. Lang, E. Nahouraii, K. Kasuga, and E.B. Fernandez, "An Architectural Extension for a Large Database System Incorporating a Processor for Disk Search," *Proc. 3rd Int'l Conf. on Very Large Data Bases*, pp. 204-210 (1977).

[Leil78a]H.O. Leilich, G. Stiege, and H.Ch. Zeidler, "A Search Processor for Data Base Management Systems," *Proc. 4th Int'l Conf. on Very Large Data Bases*, pp. 280-287 (Sept. 1978).

[Lin76a]S.C. Lin, D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," *ACM Trans. Database Syst. 1*, 1, pp. 53-75 (March 1976).

[Lin79a]W.T.K. Lin, "Concurrency Control in a Multiple Copy Distributed Data Base System," *Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, (Aug. 1979).

[Lin81a]W.T.K. Lin, "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed DBMS," *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp. 84-92 (April 1981).

[Lin82a]W.T.K. Lin and J. Nolte, "Performance of Two Phase Locking," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 131-160 (Feb. 1982).

[Lin83a]W.T.K. Lin and J. Nolte, "Basic Timestamp, Multiple Version Timestamp, and Two Phase Locking," Computer Corp. America, Cambridge, Mass. (Jan. 1983).

[Lori77a]R.A. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Trans. Database Syst. 2*, 1, pp. 91-104 (March 1977).

[Madn79a]S.E. Madnick, "The INFOPLEX Database Computer: Concepts and Directions," *Proc. IEEE Computer Conf.*, (Feb. 1979).

[Mins72a]N. Minsky, "Rotating Storage Devices as Partially Associative Memories," *Proc. FJCC*, (1972).

[Munz77a]R. Munz and G. Krenz, "Concurrency in Database Systems - A Simulation Study," *Proc. ACM-SIGMOD 1977 Int'l Conf. on Management of Data*, pp. 111-120 (Aug. 1977).

[Ozka75a]E.A. Ozkarahan, S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," *Proc. AFIPS 1975 Natl. Computer Conf.*, pp. 379-388 (May 1975).

[Papa79a]C.H. Papadimitriou, "Serializability of Concurrent Updates," *J. ACM 26*, 4, pp. 631-653 (Oct. 1979).

[Parh72a]B. Parhami, "A Highly Parallel Computing System for Information Retrieval," *Proc. FJCC*, (1972).

[Park71a]J.L. Parker, "A Logic per Track retrieval System," *IFIP Congress*, (1971).

[Posa83a]J.G. Posa, "Memory Makers Try Crossbreeding," *high Technology 3*, 8, pp. 43-50 (Aug. 1983).

[Poti80a]D. Potier and Ph. Leblanc, "Analysis of Locking Policies in Data Base Management Systems," *Commun. ACM 23*, 10, pp. 584-593 (Oct. 1980).

[Reed78a]D.P. Reed, "Naming and Synchronization in a Decentralized Computer System," Lab. for Computer Science MIT/LCS/TR-205, Massachusetts Institute of Technology, Cambridge, Mass. (Sept. 1978) Ph.D. Dissertation.

[Reis78a]M. Reiser and S.S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," Rep. RC-7023, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (March 1978).

[Reut80a]A. Reuter, "A Fast Transaction Oriented Logging Scheme for Undo Recovery," *IEEE Trans. Software Eng. SE-6*, 4, pp. 348-356 (July 1980).

[Ries79a]D.R. Ries, "The Effects of Concurrency Control on Database Management System Performance," Computer Sciences Dept., Univ. California, Berkeley (April 1979) Ph.D. Dissertation.

[Robi82a]J.T. Robinson, "Design of Concurrency Controls for Transaction Processing Systems," CMU-CS-82-114, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania (April 1982) Ph.D. Dissertation.

[Room82a]W.D. Roome, "The Intelligent Store: A Content Addressable Page Manager," *Bell Systems Technical J.*, (Nov. 1982).

[Rose78a]D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Syst. 3*, 2, pp. 178-198 (June 1978).

[Sarg76a]R.G. Sargent, "Statistcal Analysis of Simulation Output Data," *Symp. on the Simulation of Computer Systems IV*, pp. 39-50 (Aug. 1976).

[Saue81a]C.H. Sauer and K.M. Chandy, "Computing Systems Performance Modeling," Prentice-Hall, Englewood Cliffs, N.J. (1981).

[Schw76a]H.D. Schwetman and S.C. Bruell, "When to Stop a Simulation Run: A Case Study," *Symp. on the Simulation of Computer Systems IV*, pp. 131-137 (Aug. 1976).

[Seli79a]P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, pp. 23-34 (May 1979).

[Seli82a]P.G. Selinger, Personal communication. (Sept. 1982).

[Seve76a]D.G. Severance and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Trans. Database Syst. 1*, 3, pp. 256-267 (Sept. 1976).

[Skee83a]D. Skeen, Personal communication. (May 1983).

[Slot70a]D.L. Slotnik, "Logic Per Track Device," pp. 291-296 in *Advances in Computers*, ed. F. Alt,Academic Press, New York (1970).

[Song81a]S.W. Song, "A Survey and Taxonomy of Database Machines," *IEEE Database Engineering Bulletin 4*, 2, pp. 3-13 (Dec. 1981).

[Spit76a]J.F. Spitzer, "Performance Prototyping of Data Management Applications," *Proc. ACM 76 Annual Conf.*, pp. 287-297 (Oct. 1976).

[Ston75a]M.R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proc. ACM-SIGMOD 1975 Int'l Conf. on Management of Data*, pp. 65-78 (June 1975).

[Ston76a]M.R. Stonebraker, E. Wong, and P. Kreps, "The Design and Implementation of INGRES," *ACM Trans. Database Syst. 1*, 3, pp. 189-222 (Sept. 1976).

[Ston80a]M.R. Stonebraker and K. Keller, "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," *Proc. ACM-SIGMOD 1980 Int'l Conf. on Management of Data*, pp. 58-66 (May 1980).

[Ston81a]M.R. Stonebraker, "Operating System Support for Database Management," *Commun. ACM 24*, 7, pp. 412-418 (July 1981).

[Ston81b]M.R. Stonebraker, "Hypothetical Data Bases as Views," *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp. 224-229 (May 1981).

[Su75a]S.Y.W. Su and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," *Proc. 1st Int'l Conf on Very Large Data Bases*, pp. 456-472 (Sept. 1975).

[Svob81a]L. Svobodova, "A Reliable Object-Oriented Data Repository for a Distributed Computer System," *Proc. ACM-SIGOPS 8th Symp. on Operating Systems Principles*, pp. 47-58 (Dec. 1981).

[Thom78a]R.H. Thomas, "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control," BBN Rep. 3340, Bolt, Beranek and Newman Inc. (July 1978).

[Upch79a]E.T. Upchurch, J.R. Bitner, D.P.S. Charlu, and A.G. Dale, "A Reconfigurable Database Machine: Architecture and Algorithms," Inst. Computer Science and Computer Applications, U. Texas at Austin (March 1979).

[Verh78a]J.S.M. Verhofstad, "Recovery Techniques for Database Systems," *ACM Computing Surveys 10*, 2, pp. 167-195 (June 1978).

[Yao77a]S.B. Yao, "Approximating Block Accesses in Database Organizations," *Commun. ACM 20*, 4, pp. 260-261 (April 1977).