# FAILURE RECOVERY IN MEMORY-RESIDENT TRANSACTION PROCESSING SYSTEMS

Kenneth Salem

(Thesis)

CS-TR-188-88

December 1988

# FAILURE RECOVERY IN MEMORY-RESIDENT TRANSACTION PROCESSING SYSTEMS

*Kenneth Salem*

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JANUARY 1989

© Copyright by Kenneth Salem 1988
All Rights Reserved

# ACKNOWLEDGEMENTS

A great many people have contributed to this dissertation. For fear that I may unintentionally lapse, I will not attempt to list them all here. However, I cannot help but mention a few names.

This dissertation owes the most to my advisor, Hector Garcia-Molina. For the last five years, he has provided motivation, guidance, constructive criticism, and encouragement, each in the proper dosage. For all that he has done for me, I am very grateful. One could not hope for a better advisor.

To Rafael Alonso and Jeff Naughton, my thanks for taking the time to read this dissertation.

To the G-people (Gene, Geree, Gwen, Grace), and to Ruth, Winnie, and (especially) Sharon, who have always been ready to help, I am repeatedly indebted. Thanks to them, things tend to have happy endings.

Thanks also to pep, Mr. Beck, kennedylemkennedylemke, and Larry Rogers, who answered so many questions.

Finally, it isn't any good without someone to share it with. To my family and friends who have shared in some or all, my thanks for putting up with me when I was difficult to put up with. Everyone should be so lucky.

# ABSTRACT

A main memory transaction processing system holds a complete copy of its database in semiconductor memory. We present and compare, in a common framework, a number of strategies for recovery management in main memory transaction processing systems. These include strategies for asynchronously checkpointing the primary (main memory) database copy, and for maintaining a transaction log. Though they are not directly concerned with recovery management, we also consider strategies for updating the primary database, since they affect the performance of the recovery manager.

The recovery strategies are compared using an analytic performance model and a testbed implementation. The model computes two performance metrics: *processor overhead* and *recovery time*. Processor overhead measures the impact of a recovery strategy during normal operation, i.e., the cost of preparing to recover from a failure. Recovery time is a measure of the cost of recovery once a failure has occurred. Generally, it is possible to reduce processor overhead by increasing recovery time, and vice versa. The model captures this tradeoff, the exact nature of which depends on which recovery strategies are used.

Many of the recovery strategies have been implemented in a testbed, a working transaction processing system. The testbed allows recovery strategies to be combined and tested. It has been used to verify the performance model and to study other aspects of performance not considered in the model, such as data contention and transaction response times. The testbed runs on a large-memory VAX 11/785 using services provided by the Mach operating system.

Our results indicate that the selection of a checkpointing strategy is the most critical decision in designing a recovery manager. In most situations, *fuzzy*, or unsynchronized, checkpointing strategies outperform highly sychronized alternatives. This is true even when synchronized checkpoints are combined with efficient logical logging strategies, which cannot be used with fuzzy checkpoints.

# Table of Contents

**Abstract** ............ i

**Acknowledgements** ............ ii

**Chapter 1: Introduction** ............ 1

**Chapter 2: Checkpointing the Database** ............ 9

2.1 Checkpointing Strategies ............ 9

2.2 Fuzzy Checkpoints ............ 11

2.3 Black/White Checkpoints ............ 13

2.4 Copy-on-Update Checkpoints ............ 14

2.5 Failure Recovery ............ 16

2.6 Performance Model ............ 16

2.7 Checkpoint Comparisons ............ 18

**Chapter 3: Other Recovery Issues** ............ 27

3.1 Secondary Database Management ............ 27

3.2 Logging ............ 32

3.3 Other Issues ............ 35

3.4 Recovery ............ 37

**Chapter 4: A Performance Model** ............ 40

4.1 Transaction Processor Overhead ............ 41

4.1.1 Synchronous Overhead ............ 41

4.1.2 Asynchronous Overhead ............ 48

4.1.3 Restart Costs ............ 53

4.2 Recovery Costs ............ 58

**Chapter 5: A Recovery Testbed** ............ 63

5.1 Process Architecture ............ 63

5.2 The Log Server ............ 66

5.3 Message Servers ............ 67

5.4 Checkpoint Server ............ 68

5.5 Transaction Server ............ 69

5.6 Implementation ............ 70

5.7 Transaction Management ............ 70

5.8 Memory Management ............ 72

5.9 Log Management ............ 75

5.10 Log Device Management ............ 75

5.11 Checkpoint Management ............ 76

5.12 Backup Management ............ 77

5.13 Lock Management ............ 77

5.14 Low-Level Support ............ 78

5.15 Application Libraries ............ 78

**Chapter 6: Model Verification** ............ 80

6.1 Parameter Determination ............ 81

6.1.1 Static Parameters ............ 81

6.1.2 Dynamic Parameters ............ 84

6.1.3 Other Parameters ............ 85

6.2 Recovery Overhead ............ 88

6.3 Verification ............ 89

**Chapter 7: Performance Studies** ............ 95

7.1 Logging Strategies ............ 95

7.2 Spooling Checkpoints ............ 105

7.3 Response Times ............ 106

7.4 Backup Strategies ............ 110

7.5 Update Strategies ............ 116

**Chapter 8: Conclusions and Future Work** ............ 121

8.1 Hardware ............ 122

8.2 Multiprocessors ............ 123

8.3 Concurrency Control ............ 123

8.4 The "Almost" MTPS ............ 124

**References** ............ 126

# CHAPTER 1
# INTRODUCTION

Year by year, semiconductor memory is becoming cheaper and memory chip densities are increasing. It is now possible to pack more memory into less space for less money than ever before. As a result of these trends, researchers have begun to consider transaction processing systems in which a complete copy of the database resides in main (semiconductor) memory.† Memory-resident databases can mean greatly improved performance for transaction processing systems. In current systems, much of a transaction's lifetime is spent waiting to access data on disks. Furthermore, much of the complexity of the system itself can be attributed to disk latencies.

In this dissertation, we will consider the design of transaction processing systems for memory-resident databases. The simplest way to design such a system is to borrow the design of a disk-based transaction processor. In a disk-based system, a memory buffer is used to hold portions of the database that are currently being accessed or that have recently been accessed by transactions. A memory-resident system (MTPS) can be viewed as a disk-based system (DTPS) with a buffer that happens to be large enough to hold the entire database. One problem with this approach is that it fails to capitalize on many of the potential advantages that memory-residence offers. For this reason, researchers have begun to re-examine some of the components of a traditional systems, with an eye towards design for memory-resident databases. Some of the components that have been considered are index structures [Lehm85a] and query processing [Bitt87a, Lehm86a].

† We do not rule out the existence of slow archival storage. One can think of a system as having two databases (as in IMS/VS Fastpath [Gawl85a] ): one memory-resident that accounts for the vast majority of accesses, and a second on archival storage [Ston87a].

Another component that should certainly be re-examined is the recovery manager. The purpose of a recovery manager is to ensure that an up-to-date copy of the database can be reconstructed in case the existing database is damaged or destroyed. To accomplish this, the recovery manager maintains a backup copy of the database on a stable storage medium, such as magnetic disks. Since the backup can only be accessed via expensive I/O operations, keeping the backup completely up-to-date can be prohibitively expensive. To avoid this problem, recovery managers normally also maintain a *log*, a compact record of recent modifications to the database. Before a transaction can commit (i.e., complete and return a value), a record of its activities must be in the log. Since the log is available, the backup database need not be kept completely up-to-date. In case of a failure, an up-to-date copy can be recreated using the backup and the information stored in the log.

As transactions run and the backup database becomes more out of date, the log grows in size. Recovery from a failure becomes more time consuming because there is more information in the log that must be processed in order to bring the backup database up-to-date. To limit the growth of the log, recovery managers take periodic *checkpoints*. During a checkpoint, transaction processing (and hence, logging) is temporarily halted while the backup database is brought up-to-date by forcing modified data in the buffer back out to the disks. Once a checkpoint is completed, existing log information is no longer needed (activity recorded there is already reflected in the backup database). After the checkpoint, transaction processing can resume, with the recovery manager recording the activity in a now-empty log.

Memory-resident databases will affect recovery management in a number of ways, some of which are discussed next.

- In a MTPS, the transactions' data requirements can be satisfied without disk I/O, since a copy of the database is memory-resident. However, recovery management requires access to disks (or other non-volatile storage) so that the backup database can be updated. The recovery manager's I/O requirements should be satisfied without reintroducing disk latency into the critical paths of transactions. In particular, this means that the recovery manager should do as little *synchronous* I/O as possible. Such practices as forcing a transaction's updates to disk before it commits, or checkpointing the database while transactions wait,

should be avoided. Halting transaction processing during checkpointing is particularly bad because of the large amount of data (possibly the entire database) that must be flushed to the disk during the checkpoint. Thus, a MTPS must update its backup database asynchronously, i.e., without halting transaction execution.

- The recovery manager's log is normally kept on disk. Since a transaction must have an entry in the log before it can commit, log I/O is difficult to remove from transactions' critical paths. In a MTPS, log I/O is the only I/O for which a transaction must wait. Therefore, logging must be done with care if it is not to become a transaction bottleneck. In particular, steps may have to be taken to reduce the volume of log I/O.
- The least expensive and most dense form of semiconductor memory is volatile. When volatile memory is used, the entire primary copy of the database is lost in the event of a loss of power. At recovery time, the recovery manager must restore a complete copy of the database, not just those portions that have been modified since the last checkpoint. Furthermore, the database to be restored and brought up-to-date resides in memory, and not on disk.
- The *relative* contribution of recovery management to the total cost of executing a transaction will increase. As a simple example, consider a "typical" transaction in a DTPS that costs about 20,000 instructions and makes 20 database references. In a memory-resident system, that same transaction may cost only half as much. The savings will come from such areas as reduced disk I/O cost (if half of the database references would have caused I/O activity, that alone is a substantial savings at 1000 instructions per I/O), lower concurrency control costs (e.g., fewer lock conflicts, deadlocks, and rollbacks), and reduced or eliminated buffer management costs. The recovery manager, on the other hand, must still perform expensive operations like disk I/O. This implies that in a MTPS the performance of the recovery manager will be more critical to the overall performance of the system.

In this dissertation, we will focus on recovery management in a MTPS in light of these differences.

The block diagram in Figure 1.1 illustrates the organization of those portions of an MTPS that are relevant to the recovery manager. Transactions update the primary (memory-resident)

copy of the database through the primary database manager. Their activities are recorded in the log by the log manager. The checkpointer is responsible for updating the backup database copy (through the secondary database manager) to reflect changes made to the primary by the transactions.

We consider recovery management to include the activities of the checkpointer, secondary database manager, and log manager. Though not considered part of the recovery manager, the primary database manager affects recovery management in a variety of ways. In this dissertation, we will present and compare various strategies for implementing each of these components in a MTPS. The goals of the strategies are described briefly in the following.

- *Checkpointing.* A checkpointing strategy determines which parts of the primary database should be *propagated*, or *flushed*, to the backup copy. Since checkpointing is accomplished asynchronously, the checkpoint strategy also determines how propagation is synchronized with transactions' updates to the primary database, which are occurring concurrently.
- *Logging.* A logging strategy determines the contents of the log, i.e., how transactions and their updates are represented in the log.
- *Secondary Database Management.* A SDBM strategy determines how the backup database is stored and how updates propagated (by the checkpointer) from the primary database are installed.
- *Primary Database Management.* A PDBM strategy determines how the primary database is stored and how it is updated by the transactions.

In general, the various strategies will not be independent of one another. For example, the way checkpointing is synchronized with transaction updates will affect the representation of transactions in the log. As we discuss and compare the recovery strategies, we will study their interrelationships and interactions with other parts of the system.

After presenting the recovery strategies, we will present a model of their performance, that is, a model of how different strategies affect the performance of the system. There are at least two aspects of performance. One is the recovery time, the time taken to restore normal operation after the primary database is damaged or destroyed. A second is the magnitude of the overhead of recovery management during normal operation of the system, i.e., the cost of preparing for a

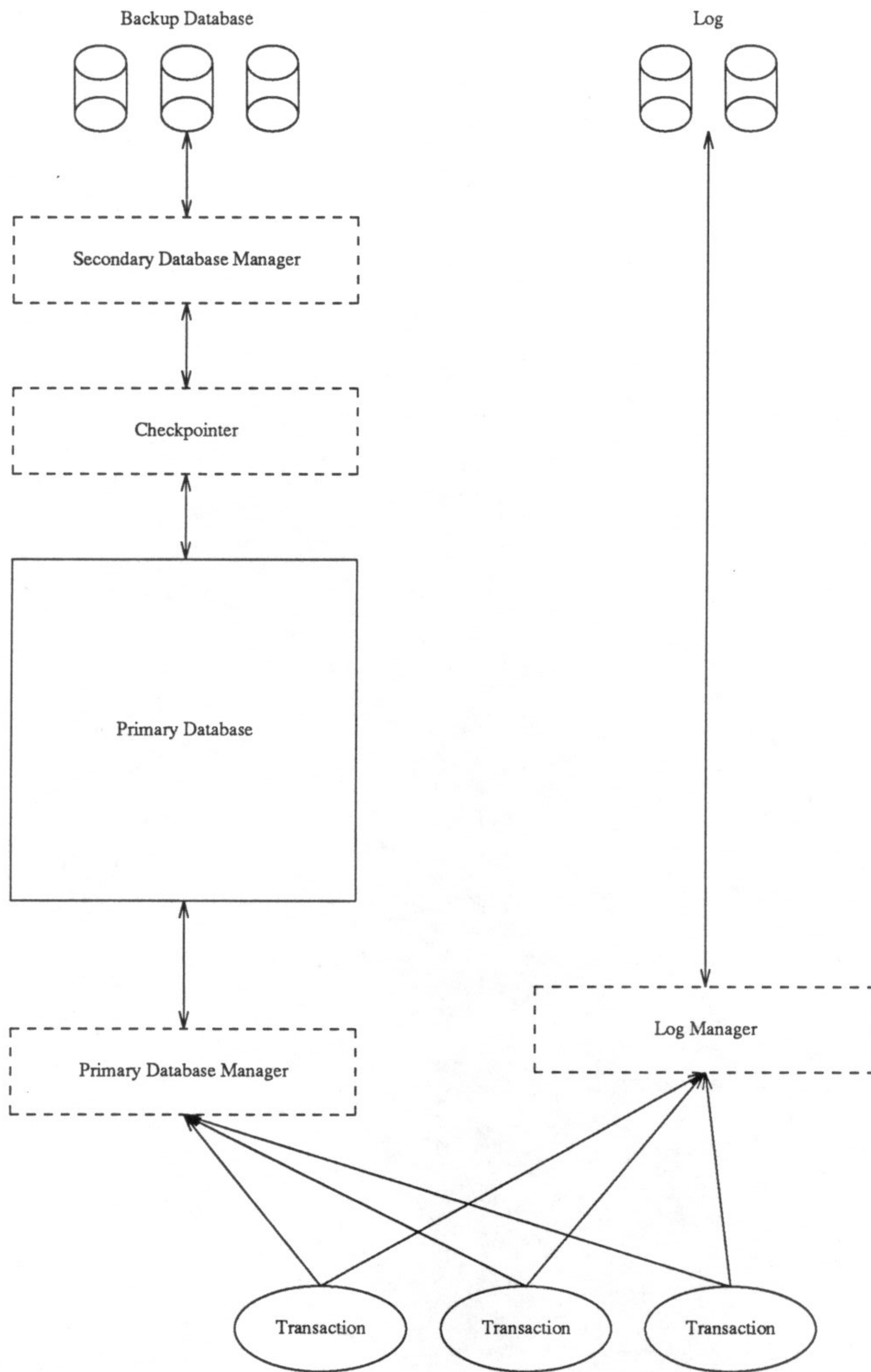



Figure 1.1 - Main Memory Transaction Processor

failure. In general, there is a tradeoff between these two costs. The model includes both aspects of performance and thus provides a useful illustration of the tradeoff.

An interesting feature of a MTPS is that the I/O bandwidth to the backup database disks should not become a bottleneck for transaction processing, since transactions require no access to the secondary database. Performance models for disk-based systems often estimate recovery overhead by counting disk I/O operations, since I/O can be a bottleneck in a DTPS. However, evaluating "I/O cost" is not a good way of measuring recovery overhead in a MTPS. (This is not to say that the I/O bandwidth is unimportant. As we will see, it affects recovery time in a number of ways.)

In a MTPS, the processor is likely to be the critical resource. Thus, our model estimates the "processor cost" of a recovery strategy. Recovery strategies generate processor costs when they perform activities such as initiation of disk I/O's, data movement (in memory), and locking or other synchronization with transaction processing activities. The fact that processor costs rather than I/O costs are the critical performance factors is another interesting aspect of recovery management in a MTPS, and is one of the reasons we believe this work to be important.

A number of other recent studies have looked at recovery for memory resident databases, or have suggested techniques that are applicable to MTPS recovery. A technique for producing asynchronous checkpoints is described in [DeWi84a]. This paper also suggests the use of non-volatile memory to hold the log tail (the most recently created part of the log). Recovery management in IMS/VS FastPath, which supports memory-resident databases, is described briefly in [Gawl85a]. The recovery mechanism proposed in [Hagm86a] stresses fast, unsynchronized checkpoints and log compression to provide fast recovery from system failures.

Several papers have suggested recovery techniques that make use of dedicated or special-purpose hardware. [Lehm86b] proposes a recovery processor that operates in parallel with the database processor to flush data from a non-volatile memory-resident log tail to log disks. The recovery processor proposed in [Eich86a] merges log information into the backup database copy to keep it as current as possible. Both proposals rely on the existence of large amounts of non-volatile main memory, i.e., memory whose contents are not destroyed by a power failure. HALO, a hardware logging device that logs automatically and is transparent to the system, is described

in [Garc83a] and [Sale86a].

The study presented here draws on ideas from most of these papers, and from others as well. Our emphasis is on algorithmic alternatives rather than special hardware. We will not consider any recovery mechanisms that rely on the existence of non-volatile memory or on special purpose or functionally segregated processors.† This is not to suggest that such strategies are not feasible or desirable. However, it is important to understand first the algorithmic alternatives in order to judge the value of non-volatile memory or additional hardware support.

There have been a number of studies of recovery mechanisms for disk-based databases, e.g., [Reut84a, Agra85a]. However, we are aware of only two comparative studies of recovery mechanisms for memory resident databases. A taxonomy of main memory recovery techniques is presented in [Eich87a] along with an analytic performance model. Unlike those presented here, most of the techniques considered there involve special hardware and/or non-volatile memory. This is also true of the study presented in [Sale86a].

The contributions of this dissertation are fourfold.

- We present, in a common framework, a number of recovery strategies. Included are new strategies as well as others that have appeared in the literature.
- We present and use a modeling methodology which we believe to be well-suited to main memory database systems.
- Most of the strategies that will be presented have been implemented in a recovery testbed. The testbed allows experimental verification of the performance model and provides insights into the behavior of the strategies that would have been difficult to ascertain without implementation.
- Our results indicate that there are significant differences in performance among the strategies. They also show how the differences are affected by changes in the environment and the workload.

---

† Note that even if all of main memory is non-volatile, some form of backup is necessary to protect against other types of failures, e.g., corruption due to software errors.

The rest of the dissertation is organized as follows. The next chapter provides an introduction to MTPS recovery by focusing on one of its most critical aspects, checkpointing. We present a variety of checkpointing strategies and compare their performance using the model. The third chapter describes recovery strategies for aspects of recovery management other than checkpointing. These include logging and management of the primary and secondary database copies. We also discuss how these strategies interact with checkpointing. The fourth and fifth chapters describe the performance model and the testbed. Chapter 6 describes the validation of the model using experimental results from the testbed. In Chapter 7 we compare the performance of the recovery strategies presented in the third chapter, based on data from the model and the testbed. We also show how these strategies affect the checkpointer's performance, and study the effects of changing workload and system parameters on our comparisons. Finally, Chapter 8 contains conclusions and a discussion of some issues for further study.

# CHAPTER 2
# CHECKPOINTING THE DATABASE

In this chapter, as an introduction to recovery in a MTPS, we will focus on one critical aspect of recovery, namely the maintenance on disk of the secondary copy of the database. We term this process *checkpointing*, although checkpointing may be realized quite differently in a MTPS than in a DTPS. We will describe a number of possible algorithms for *asynchronous* checkpointing, and compare them using a simple analytic model.

Several strategies for asynchronous maintenance of a secondary database copy have appeared in the literature [DeWi84a, Eich87a, Hagm86a, Lehm87a, Pu86a]. The checkpointing strategies that we will consider are based on ideas drawn from that work. Our emphasis is on algorithmic alternatives. We have not considered checkpointing strategies that rely on the existence of special purpose or functionally segregated processors, nor those that require large quantities of non-volatile primary memory.

The rest of the chapter is organized as follows. We will first describe a variety of checkpointing strategies that could be used in a MTPS. Afterwards we will give a summary of the recovery management performance model (which will be discussed in more detail in Chapter 4) and will use it to compare the performance of the various strategies.

## 2.1 Checkpointing Strategies

It is difficult to separate one part of the recovery manager from the others and from the rest of the transaction processing system. For the purposes of this chapter we will make several assumptions about the behavior of the rest of the system. It is important to note that there are alternatives. We are not arguing that those selected are the best; we are simply choosing representative and reasonable alternatives so we can study checkpointing strategies independently. In Chapter 7 we will eliminate these assumptions and consider other choices and how

they affect the checkpointer.

- We assume that two *complete* backup databases are maintained on disks and that a *ping-pong* update scheme is used. Only one of the two copies is updated during a single checkpoint, and successive checkpoints alternate between the copies.

  The database is composed of records grouped onto fixed-size regions of main memory called *segments*. During a checkpoint, only those segments that have been updated are written out to the backup database. To implement this, database segments in memory include two *dirty bits*. When a transaction modifies a segment, it sets both bits. When the checkpointer flushes a modified segment to one of the backups, it resets one of the bits. When it flushes it to the second copy, it resets the second bit. Thus, the checkpointer will only ignore segments that have been flushed to both copies. When a checkpoint begins, it enters a *begin-checkpoint* marker in the log. When it completes, the current checkpoint copy is noted at a known location on disk we call *home*. The home block also contains a pointer to the begin-checkpoint log entry made by this completing checkpoint. At recovery time, the home block is used to select the most recently completed checkpoint copy.

- We assume that transactions use a *shadow-copy* update strategy similar to that employed by IMS/VS Fastpath [Gawl85a] and proposed by others [Eich87a]. Updates are stored in a buffer local to the updating transaction until the transaction commits. At that point, updates are installed in the database by overwriting the old version of the record with the new. Transactions use REDO-only logging. UNDO logging (i.e., logging old versions) is not necessary because old versions are not overwritten in the database unless a positive commit decision is made for the transaction.

- We assume that record-level, value, REDO logging is used, i.e., log data consists of the new versions of modified records.

The checkpointing strategies we will consider vary according to the consistency of the backup copy they produce. We will consider fuzzy, action-consistent (AC), and transaction-consistent (TC) checkpoints.

Consider a transaction that updates records $R_1$ and $R_2$ with two update actions. A TC backup will reflect transaction activities atomically, i.e., the backup will contain either the old

versions of $R_1$ and $R_2$ or their new versions, but not one old and one new.

An AC backup may contain the old version of $R_1$ and the new version of $R_2$ (or vice versa). However, each action will be reflected atomically. That is, neither record will be found in a partially updated state. Finally, a fuzzy backup makes no guarantees about the atomicity of transactions or actions.

As we will see, consistent checkpoints are more costly to produce than fuzzy ones. However, an important advantage of consistent backups is that they permit the use of logical logging,† as opposed to value logging. With logical logging, operations like ''insert this new record'' or ''update this field of this record'' are recorded. Any associated changes in the database access structures are not recorded. Value logging, on the other hand, records all changes made to memory in the course of the action. We will consider the effect of logical logging in Chapter 7.

## 2.2 Fuzzy Checkpoints

Fuzzy checkpoints require little or no synchronization with executing transactions. Fuzzy checkpoints are suggested for recovery in main memory databases in [Hagm86a].

We call our fuzzy checkpointing strategy FUZZY. It begins a checkpoint by entering its begin-checkpoint marker in the log, along with a list of currently committing transactions. (A transaction is committing if it is in the process of placing its updates in the database.) Once the marker is in place, the checkpointer processes database segments. A segment is processed by carefully examining and clearing the appropriate dirty bit, and flushing the segment to secondary storage if it was dirty. Locks and other transaction activity are ignored. Once all segments have been processed, the (in-memory) log tail is flushed to disk and the new current checkpoint is noted (as described in Section 2.1).

If one is not careful, fuzzy checkpointing may in general lead to violations of the *log write-ahead protocol* [Gray78a]. (Such a violation occurs if a transaction's updates are reflected in a checkpoint but not in the log.) However, because we are using two ping-pong backup copies, the problem does not arise. While a checkpoint is in progress, a transaction's updates may indeed

† Logical logging is also known as *transition* [Haer83a] or *operation* logging.

appear in one of the backups before they do in the log. Nevertheless, since the checkpoint is incomplete, all such updates will be ignored at recovery time. It is only when the checkpoint completes that the updates in it become valid. (If two backup databases are not used, then a fuzzy checkpointer must copy the data to a main memory buffer before flushing it, adding overhead to the checkpointer. See Chapter 3.)

One difficulty with the FUZZY strategy is that in general the checkpointer can "see" updates made by uncommitted transactions. This is not an issue for the comparisons we will make in this chapter because we have assumed a shadow-copy update strategy, under which transactions do not install their updates until they have committed. However, under other update strategies (some of which will will consider in Chapter 3), the recovery manager may have to log UNDO as well as REDO information to eliminate the unwanted updates if necessary.

A related checkpoint strategy that avoids this problem is the segment-consistent (SC) strategy. The SC checkpoint behaves like FUZZY, except that segments are locked by the checkpointer before being processed. Once the segment has been processed, it is immediately unlocked. The lock prevents the checkpointer from seeing the effects of an incomplete transaction.

We have considered two variations on the SC strategy. They differ in how segments are processed. One option is to flush the segment immediately to the backup disks, as was done under FUZZY. The checkpointer locks each segment for the duration of the disk I/O operation. We call this type of checkpointer *SC/FLUSH*.

An alternative is to spool the I/O. Before flushing the segment, the checkpointer first copies it to a special buffer and then flushes the copy. The advantage of this alternative is that the segment can be unlocked as soon as it is copied; there is no need to maintain the lock through the disk I/O. However, since copying the segment to the special buffer is not free, there is a price paid in processor overhead. We will call the spooling alternative *SC/COPY*.

### 2.3 Black/White Checkpoints

So far, we have discussed three strategies (FUZZY, SC/FLUSH, and SC/COPY) for producing fuzzy checkpoints. As was described in Section 2.1, fuzzy checkpoints are inconsistent, i.e., they may not reflect atomically the effects of an action or a transaction. One way to produce a consistent backup is to treat the checkpoint operation as a (long-lived) transaction. The checkpointer acquires a lock on each segment before flushing and holds the locks until the checkpoint is complete. Clearly, this method will result in unacceptably frequent and long lock delays for other transactions. (At some point during each checkpoint the checkpointer will have all of the dirty database segments locked simultaneously.) An alternative, which produces consistent backup copies but requires that locks be held on only one segment at a time, is presented in [Pu86a]. (It can also be viewed as a special case of the "altruistic" locking protocol described in [Sale87a]. ) The strategies we will describe next are variants of the mechanism proposed in that paper.

The basic strategy described in [Pu86a] proceeds as follows. There is a "paint bit" for each segment which is used to indicate whether or not that segment has already been included in the current checkpoint. Assuming that all segments are initially colored white (i.e., paint bit = 0), checkpointing is accomplished by the strategy shown in Figure 2.1.

```
WHILE there are white segments DO
        find a white segment that is not locked
        IF there are none THEN
                request lock on any white segment and wait
        ELSE
                lock the segment
                process the segment
                paint the segment black (set paint bit = 1)
                unlock the segment
END-WHILE
```

Figure 2.1 - Black/White Checkpoint

The strategy can be used to produce either a TC or an AC backup. To ensure that the checkpointer produces a TC backup, no transaction† is allowed to access both white and black records. (A record is the same color as the segment it is a part of). Any transaction that attempts to do so is aborted and restarted. Similarly, an AC backup can be produced by ensuring that no

action accesses both black and white segments. (Note that a transaction may contain a mix of black-accessing and white-accessing actions.) A transaction is aborted if any of its actions attempt to access both white and black records.

As for segment-consistent checkpoints, we have considered variations on the basic black/white checkpoint that differ in whether or not segment I/O is spooled. Since the black/white strategy can be used to produce TC or AC checkpoints, the result is four possible variations on the strategy, which we will call TCBW/FLUSH, TCBW/COPY, ACBW/FLUSH, and ACBW/COPY.

### 2.4 Copy-on-Update Checkpoints

Copy-on-update checkpointing forces transactions to save a consistent "snapshot" of the database, for use by the checkpointer, as they perform updates. The advantage of copy-on-update (COU) checkpointing is that it does not cause transactions to abort, as do the black/white strategies. On the other hand, primary storage is required to hold the snapshot as it is being produced. Potentially, the snapshot could grow to be as large as the database itself. The COU mechanisms we will describe are inspired by the technique described in [DeWi84a], the "initial value" method of [Rose78a], and the "save-some" method of [Pu86a].

To begin a COU checkpoint, the database must first brought into a state of the desired consistency (either action-consistent or transaction-consistent). A simple way to achieve a TC database state is to quiesce the system: the updates of all currently *committing* transactions are completed, while no new transaction are allowed to commit. (Note that running transactions that are not in the process of committing are allowed to continue. All their updates are private and can be ignored at this point.) To achieve an AC state, all update actions (e.g., install record) are completed while new actions are disallowed.

When the database is quiescent a begin-checkpoint record is written to the log, and the log tail is flushed to non-volatile storage. The consistent database state that exists in primary memory is the "snapshot" that will be flushed to secondary storage by the checkpointer. Once the

† A *read-only* transaction is permitted to read both black and white records.

begin-checkpoint entry is in the log, transaction committing can resume.

The algorithm uses a paint bit per segment in much the same way as the black/white strategies (the bit determines whether or not the segment has already been included in the current checkpoint). In addition, each segment has a pointer which will be used to point at the "snapshot" copy of the segment, if one exists.

Checkpointing is accomplished by the algorithm shown in Figure 2.2. (As before, we assume that all segments are initially colored white.)

```
WHILE there are white segments DO
  find a white segment (S) that is not locked
  IF there are none THEN
      request shared lock on any white segment and wait
  ELSE
      lock S
      IF S has a pointer to a "snapshot" copy S' THEN
         paint S black
         save pointer to S'
         unlock S
         IF S' is dirty THEN
               flush S' to the backup
         free S'
      ELSE
         process S
         unlock S
END_WHILE
```

Figure 2.2 - COU Checkpointing

The transactions are responsible for saving snapshot copies of segments when necessary so that the consistency of the snapshot is preserved. When a transaction wishes to update a segment that the current checkpointer has not reached (a white segment), it first makes a copy of the old version of the segment if such a copy does not already exist. The segment's pointer is set to point at the newly-created copy.

When the checkpointer processes a segment which does not have a "snapshot" copy it has the same two options as did the black/white and segment-consistent strategies. It can flush the segment while retaining its lock, or spool the segment so that the lock need not be held for the duration of the I/O operation. (Note that if segment $S$ already has a snapshot copy $S'$, the lock on $S$ can be released immediately without creating another copy of $S$. The existing copy $S'$ can be spooled instead.) Thus there are four variations of the COU strategy, differing in the consistency

of the checkpoint and the spooling decision. These will be called TCCOU/FLUSH, TCCOU/COPY, ACCOU/FLUSH, and ACCOU/COPY.

### 2.5 Failure Recovery

A *system failure* [Gray78a], results in the loss of part or all of the primary database copy. After a system failure, the recovery manager has at its disposal a backup copy of the database and a transaction log on non-volatile storage. In a disk-based system, the log is used to bring the stable database copy to a consistent state. In a MTPS, the stable database copy and the log are used to recreate a consistent primary database copy in main memory.

The recovery procedure is to first read the backup database into main memory, and then to apply the log to the new primary database to bring it into an up-to-date consistent state. Applying the log to the database means the following. Recall that the location of the begin-checkpoint log marker of the most recently *completed* checkpoint is stored in the home block. Thus it is not necessary to scan the log backwards to find the begin-checkpoint marker. From that point the log is scanned forwards. The recovery procedure is discussed in more detail in Chapter 3.

### 2.6 Performance Model

In this section we will briefly describe the model used to compare the performance of the various checkpoint strategies. The model, and the testbed implementation used to verify the model, will be presented in detail in Chapters 4 and 5. Here we will give only an overview before presenting some comparisons. We are presenting performance results at this point to demonstrate the utility of the model before we describe it in detail in Chapter 4, and because we have found that the checkpointing strategy is one of the most critical parts of the recovery manager.

The model computes two performance metrics: processing overhead and recovery time. As the name suggests, processing overhead refers to the additional processor power (instructions executed) used for failure recovery preparations while transactions are running. Processing overhead is the cost over and above the cost of running the same transactions on a failure-free system. Recovery time is the time required to restore an up-to-date, consistent copy of the database in

memory after a failure, so that transaction processing can resume.

The modeling methodology involves analyzing the checkpointing strategies to determine the number of *primitive processor operations* they require to complete their task. The costs of the primitive operations are model parameters, and the total overhead is obtained by summing the costs of the primitive operations involved. The number of primitive operations depends on the particular checkpointing strategy being used as well as on some simple probabilistic analyses based on several assumptions regarding transaction data access patterns, transaction failures, checkpoint scheduling, etc.

The primitive operations represent the work that the processor must do to accomplish recovery preparations. They include:

- Synchronization: most of the checkpointing strategies require synchronization with transactions via locking. $C_{lock}$ is the cost of acquiring and releasing a lock.
- Storage Management: $C_{lalloc}$ represents the cost of preparing new pages in the log buffer. $C_{salloc}$ is the cost of acquiring new space for other uses, i.e., for database segment "snapshots" when copy-on-update checkpointing is used.
- I/O Initiation: $C_{io}$ is the processor cost of a disk I/O. We assume that the disk controllers support direct memory access, so that $C_{io}$ is independent of the amount of data being transferred.
- Data Movement: Memory to memory data copies have a fixed cost $C_{mv_fixed}$ plus a cost $C_{move}$ times the amount of data copied.

The parameters we have described are summarized in Table 2.1. The table also lists their default values. We believe that the defaults are realistic, at least for some types of hardware and applications. Of course, other values are possible. Later we will explore the effects of variations in some of the more critical parameters.

In addition to the processor model (the primitive operations), there are also simple models of the database, secondary storage, and the transaction load.

- The database consists of $S_{db}$ words of data, grouped into records of size $S_{rec}$. Records are grouped into segments of size $S_{seg}$.

| symbol | parameter | default | units |
|---|---|---|---|
| $C_{lock}$ | (un)locking overhead | 50 | instructions |
| $C_{lalloc}$ | log buffer maintenance | 200 | instructions |
| $C_{salloc}$ | space (de)allocation cost | 100 | instructions |
| $C_{io}$ | I/O overhead | 1500 | instructions |
| $C_{mv_fixed}$ | constant data movement cost | 10 | instructions |
| $C_{move}$ | proportional data movement cost | 1 | instructions |

Table 2.1

- Segments are the units of transfer to the backup disks. The bandwidth to the backup disks is expressed by $1/(a_{back} + b_{back}S_{seg})$. Bandwidth to the log disks is given by $1/(a_{log} + b_{log}S_{lpg})$, where $S_{lpg}$ is the size of a log page.
- A transaction consists of a collection of actions. A transaction consists of $i$ actions with probability $N_{act}(i)$. Of the $i$ actions, $R_{up}i$ of them are update actions, e.g. insert, modify. Each action affects $R_{rpa}$ records and $R_{spa}$ segments. Excluding the effects of recovery management, a transaction fails (aborts) with probability $p_{fail}$. Excluding recovery management costs, successful transactions have a processor cost of $C_{trans}$. Transactions arrive at the system at a rate $\lambda$.
- The checkpoint interval (time between the initiation of successive checkpoints) is $t_{icp}$. The minimum possible checkpoint interval is a function of the I/O bandwidth and the transaction load.

These model parameters are summarized in Table 2.2. The full analysis, along with complete list of model parameters and their default values, can be found in Chapter 4.

### 2.7 Checkpoint Comparisons

Figure 2.3 shows processor overhead and recovery time for each of the checkpointing strategies. The data were obtained assuming that the checkpoints duration was as short as possible. In the figure (and in all other figures), the processor overhead is a per-transaction cost, i.e., the total overhead cost distributed evenly over all transactions.

Several points are apparent from Figure 2.3. Most obvious is the relatively high cost of the TC black/white checkpoint strategies compared to the corresponding COU strategies. Most of the additional cost comes from rerunning transactions that are aborted for violating the black/white

| symbol | parameter | default | units |
|---|---|---|---|
| $a_{back}$ | backup I/O fixed time | 1.5 | milliseconds |
| $b_{back}$ | backup I/O linear time | 0.15 | μseconds |
| $a_{log}$ | log I/O fixed time | 4 | milliseconds |
| $b_{log}$ | log I/O linear time | 0.4 | μseconds |
| $S_{db}$ | database size | 256 | Mwords |
| $S_{rec}$ | record size | 32 | words |
| $S_{seg}$ | segment size | 8192 | words |
| $S_{lpg}$ | log page size | 1024 | words |
| $\lambda$ | arrival rate | 1000 | transactions/second |
| $p_{fail}$ | failure probability | 0.05 | -- |
| $N_{act}(i)$ | probability of $i$ updates | 0.1 for $0 < i \leq 10$ | -- |
| $R_{up}$ | update probability | 0.33 | -- |
| $R_{spa}$ | segments per action | 1.1 | segments/action |
| $C_{trans}$ | transaction processor cost | 10000 | instructions |

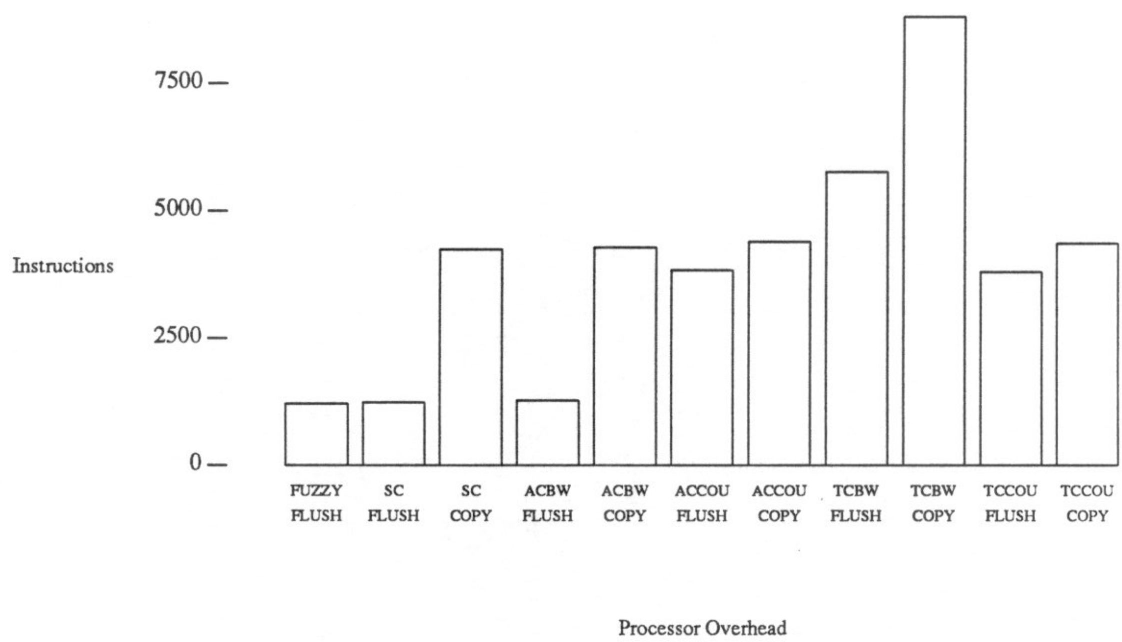
Table 2.2

restriction. It is also apparent that spooling adds substantially to the cost of a checkpoint (e.g., compare ACBW/FLUSH with ACBW/COPY). Of course, spooling strategies lock segments for shorter periods, but this is not reflected in our overhead metric. We will return to the issue of lock times in Chapter 7.

AC checkpointing can be done almost as cheaply as FUZZY. Though the FUZZY strategy need not lock pages and never causes transaction aborts, ACBW/FLUSH does not cause too many aborts. As we shall see shortly, this gap widens as actions become more complex ($R_{spa}$ grows), since more complex actions are more likely to violate the black/white restriction.

Because we are using shadow-copy updates, a ping-pong backup, and value logging, recovery times are not affected by checkpoint strategy. Under different assumptions, the checkpointer can affect the recovery time. We will return to this point in Chapter 7 when we consider the effects of other recovery strategies on performance.

Although recovery times do not vary with changes in the checkpoint strategy, they can be made to vary by controlling the checkpoint duration. In fact, for a given checkpoint strategy there is a tradeoff between processor overhead and recovery time that can be controlled by varying the checkpoint duration. This tradeoff is illustrated in Figure 2.4 for two of the checkpoint strategies, TCBW/FLUSH and TCCOU/FLUSH. The two solid curves represent the trajectory of TCBW/FLUSH and TCCOU/FLUSH through the processor overhead/recovery time space as the



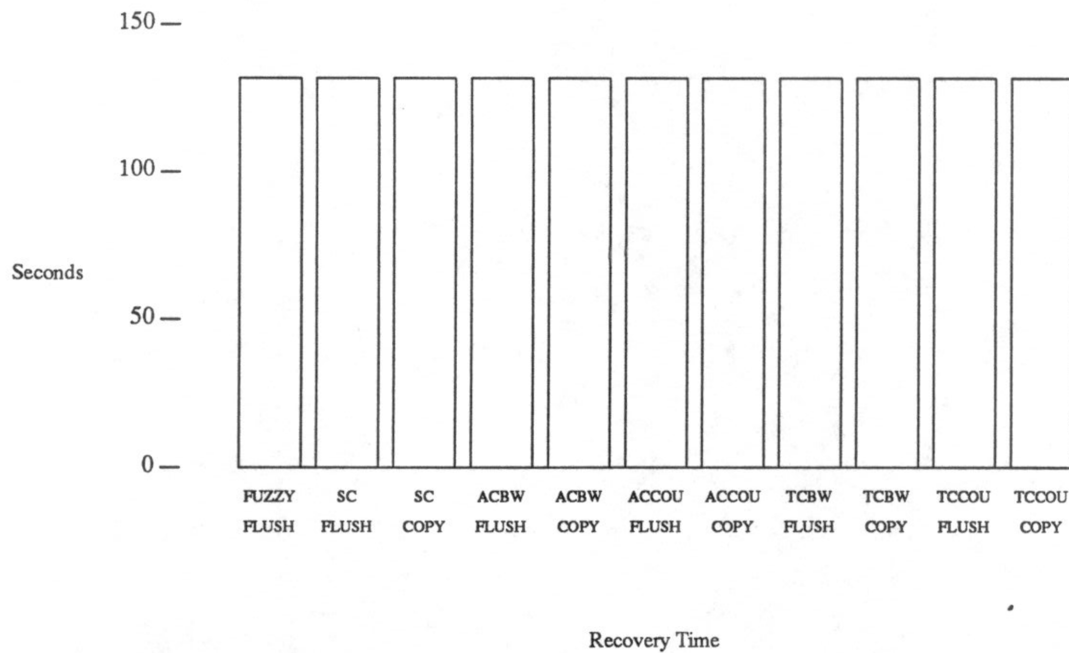




Figure 2.3 - Processor Overhead and Recovery Time

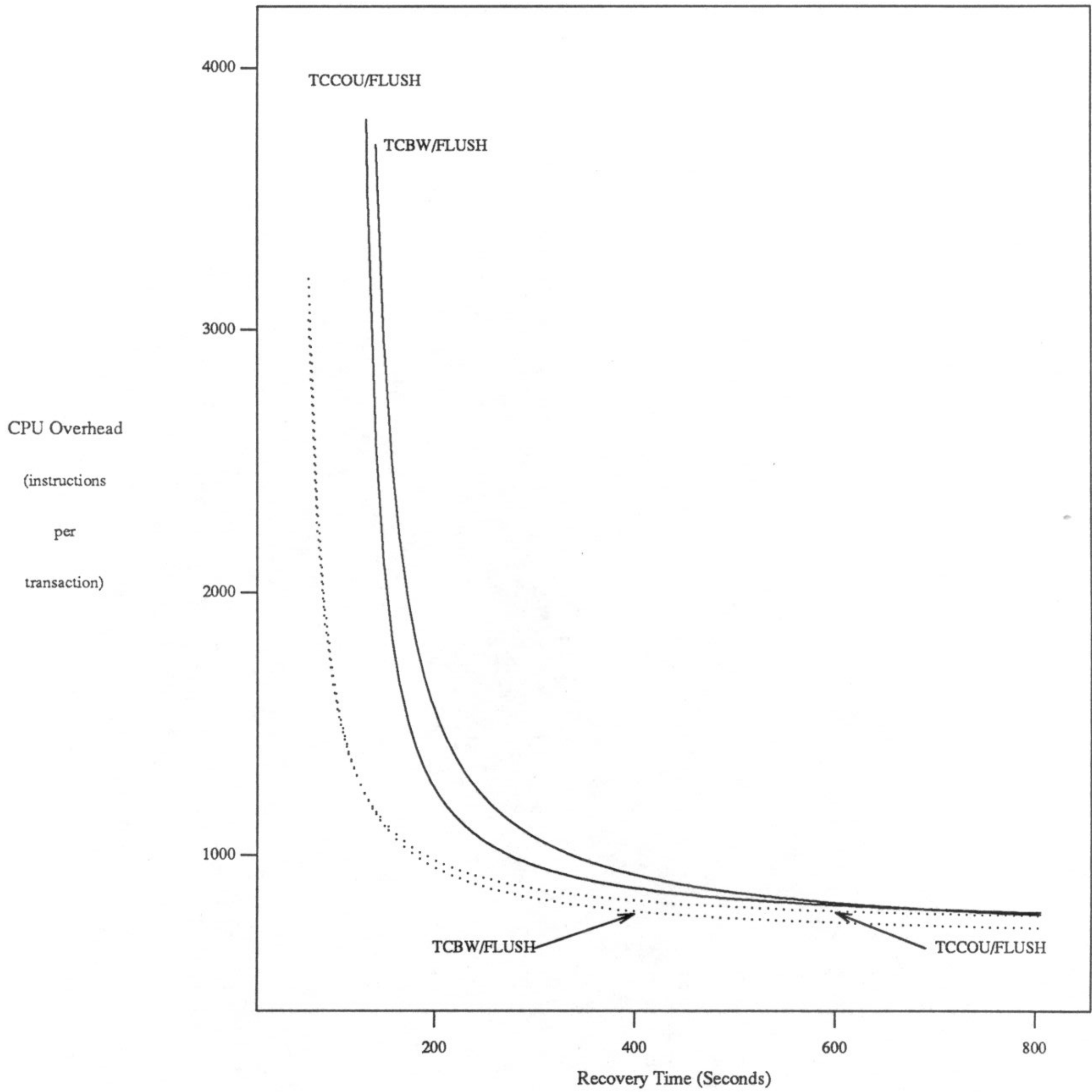



Figure 2.4 - Processor Overhead/Recovery Time Tradeoff

checkpoint duration is varied. The checkpoint duration is smallest at the left end of each curve and increases to the right. Thus, by increasing the checkpoint duration, it is possible to drive processor overhead down at the cost of increased recovery time.

The dotted lines in the figure represent the same experiment except that the bandwidth from primary memory to the backup disks has been doubled (by halving $a_{back}$ and $b_{back}$). The dotted lines extend further to the left than their solid counterparts because the higher bandwidth permits a lower minimum checkpoint interval. Thus, greater bandwidth allows the designer of a memory-resident database system greater range of processor overhead/recovery tradeoff.

It is also interesting that the increased bandwidth is much more beneficial to TCBW/FLUSH than to TCCOU/FLUSH. Though the black/white strategy is more costly in the original experiment (particularly with fast checkpoints), its performance is indistinguishable from TCCOU at the higher bandwidth. This is because of reductions in the number of transactions that must be rerun because of violations of the black/white constraints. As the bandwidth increases, the checkpointer requires less time to update the backup copy. As a result, an incoming transaction is less likely to encounter an ongoing checkpoint and, consequently, a black/white constraint violation.

Figure 2.5 describes the effect of transaction load, $\lambda$, on processor overhead for four of the strategies. The general trend is for decreasing per-transaction cost with increasing load, because the cost of a checkpoint is distributed over a greater number of transactions as the load increases. In particular, the spooling strategies (dotted lines) are much more expensive at low loads than their non-spooling counterparts. However, at high loads they are comparable. This is because at low loads the cost of spooling dirty segments (which changes little with the load) is shouldered by fewer transactions in a lightly loaded system.

We have already seen that checkpointing overhead can be controlled by varying the checkpoint interval. Figure 2.6 describes the effect of another parameter, the segment size ($S_{seg}$), assuming that the checkpoint duration is as short as possible. (Recall that segments are the units of transfer to secondary storage.)

The variety of behavior exhibited by the different strategies arises from a combination of two effects. First, as segments get larger, the total number of segments in the database decreases. Thus, checkpoints can be produced with fewer *per-segment* overhead charges. For example, fewer I/O's need to be initiated since each I/O moves more data.

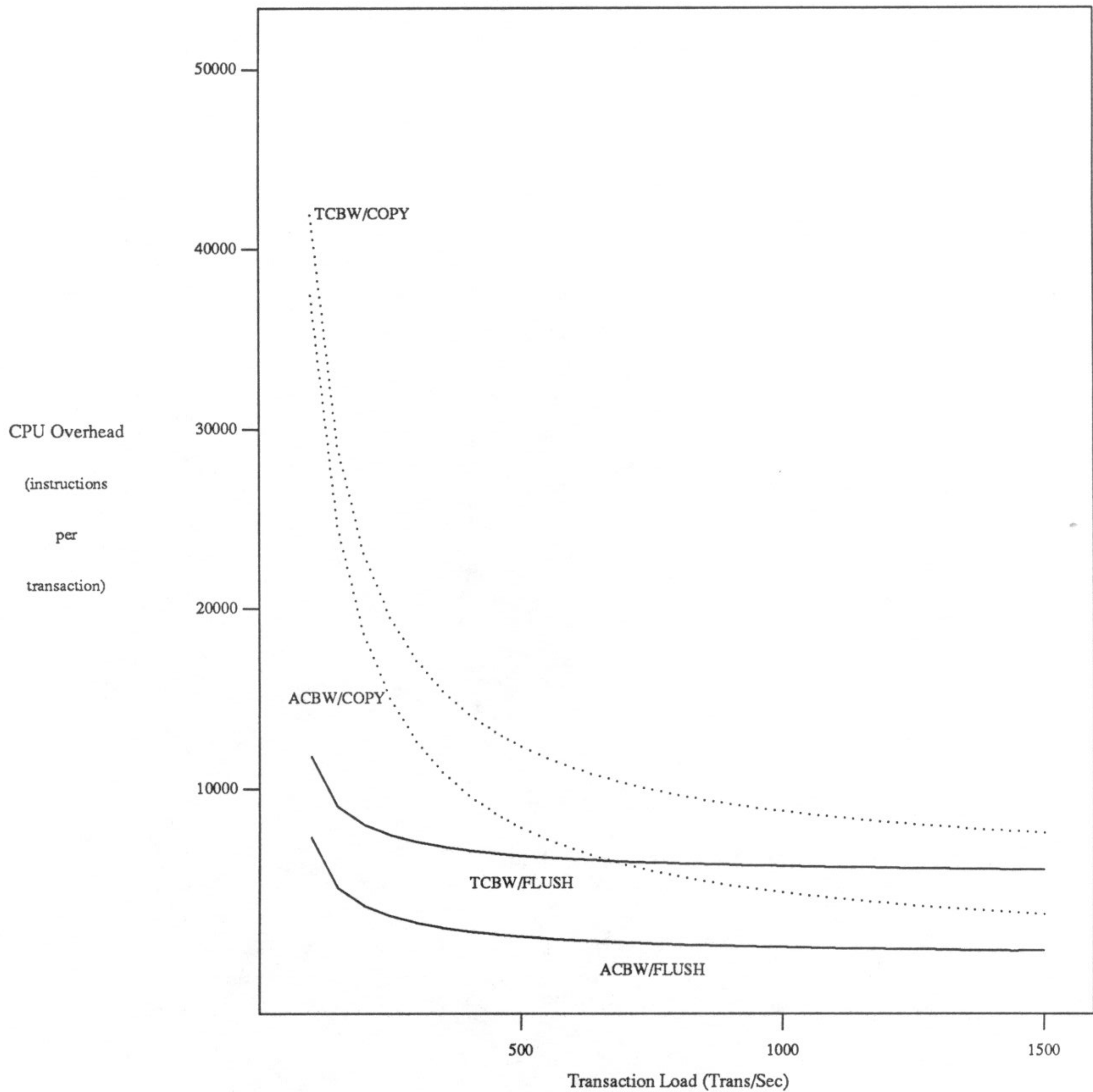



Figure 2.5 - Effect of Varying Transaction Load

Second, larger segments mean more efficient disk I/O and hence faster checkpoints. This tends to increase per-transaction overhead since relatively fixed components of the checkpoint overhead, such as copy costs, must be shared by fewer transactions. (Note that it also reduces recovery time for all of the strategies, though recovery times haven't been plotted here.)

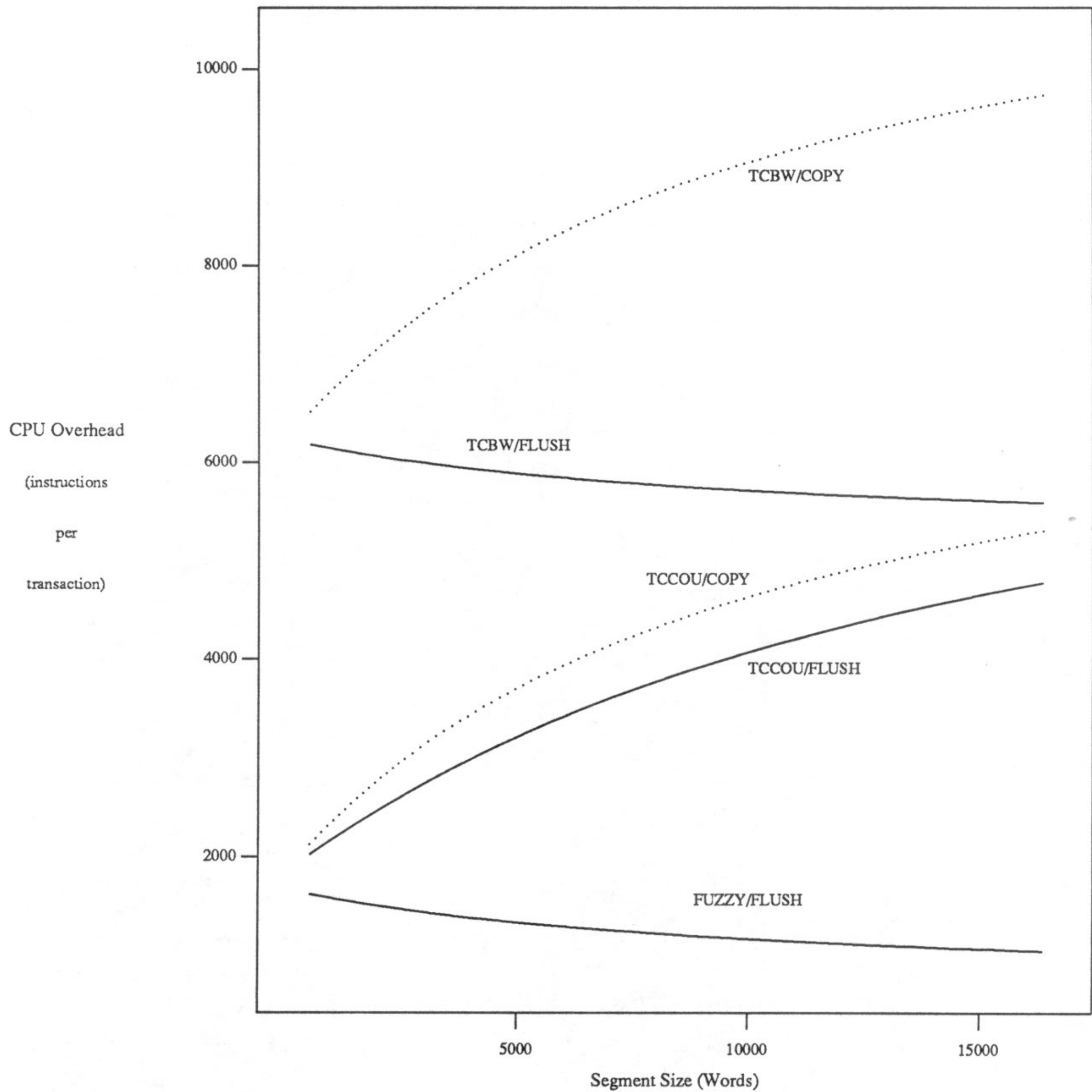



Figure 2.6 - Effect of Varying Segment Size

Spooling strategies (e.g., the two dotted curves in the figure) are affected most strongly by this second effect. Their per-transaction overhead costs increase with the segment size as a result. TCCOU/FLUSH, which does not spool but which still requires a significant amount of data copying, is affected in the same way though not as strongly. The performance of the non-spooling

strategies is dominated by the first effect and their overhead costs are lower for larger segments.

Finally, Figure 2.7 looks at the effect of increasing $R_{spa}$, the complexity of (i.e., the number of segments accessed by) a database action. The most strongly affected strategy is the black/white AC strategy, whose performance suffers because more complex actions are more likely to violate the two-color restriction, causing transaction rollback and restart. The cost of TC black/white checkpoints increases for a similar reason: more complex actions mean more complex transactions which are more likely to violate the two-color constraint.

### Summary

We have presented a variety of strategies for checkpointing memory-resident databases, and have used a model to compare their performance. Our results indicate that there are significant differences in performance among them.

So far, we have considered checkpoint strategies independently of the other components of the transaction processing system. Later (Chapter 7), we will explore the interactions between the checkpointer and some of the other components, namely logging and storage management of both primary and secondary storage. In some cases, more expensive checkpointing strategies may actually prove to be beneficial because they can be used in conjunction with less costly logging or storage management techniques.

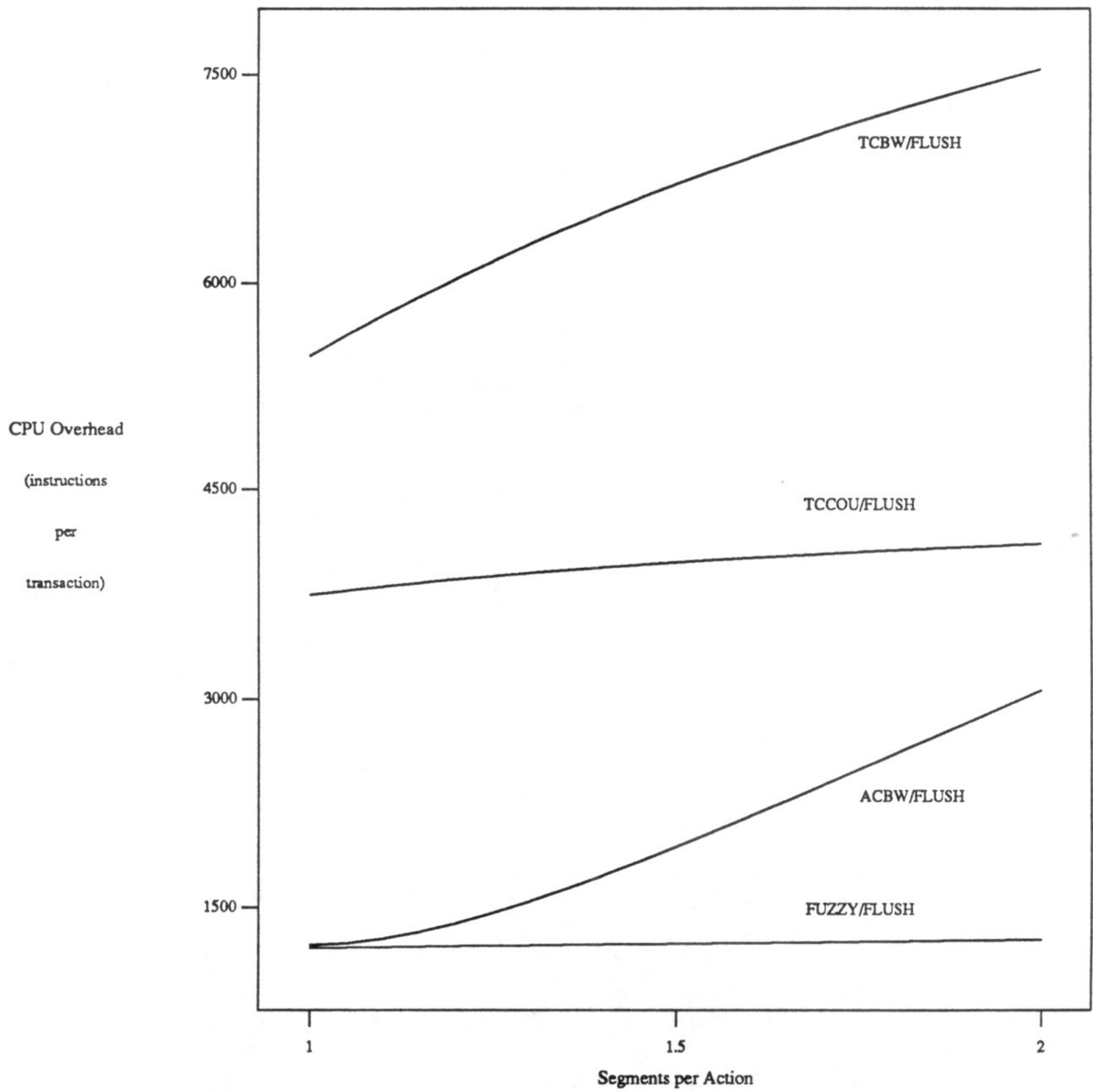



Figure 2.7 - Effect of Varying Action Complexity

# CHAPTER 3
# OTHER RECOVERY ISSUES

In the last chapter we considered checkpointing strategies, strategies for deciding when and how to propagate updates from the primary database copy to the backup. In this section we will consider other aspects of recovery. In particular, we will discuss strategies for:

- Secondary Database Mangement. We have already discussed checkpoint strategies, which determine which dirty segments are to be propagated to the secondary database. The secondary database management (SDBM) strategy determines how the dirty segment is actually installed in the secondary database. There are a number of alternatives, including strategies for updating single-copy (*monoplex*) and double-copy (*duplex*) secondary databases.
- Logging. Logging strategies determine what is written into the log, and when log pages are moved from the log buffer to the log disks.
- Restart. Restart strategies describe how the primary database is restored to a consistent, up-to-date state after a failure. To accomplish its task, a restart strategy has at its disposal the secondary database copy (or copies) and the log created during the period before the failure.

In this section we will also consider other parts of the transaction processing system whose behavior directly affects recovery strategies or recovery performance. In particular we will consider the primary database manager, since the way transactions update the primary database affects the work of the recovery manager.

## 3.1 Secondary Database Management

The SDBM strategy determines how the the secondary copy of the database is updated. We will examine five possibilities here. Two of the strategies we will discuss are *duplex* strategies, meaning that two complete copies of the database exist on secondary storage. Duplex strategies

are necessary if the checkpoint is to be consistent (i.e, if it is produced by a COU or black/white checkpointer). Monoplex strategies can only be used to support fuzzy checkpoints.

One advantage of duplex strategies over single copy, or *monoplex*, strategies is their increased tolerance of media failures. (We will not discuss media failures further here.) As we will discuss (Chapter 4), having two secondary copies can also eliminate synchronization problems between the checkpointer and the logger. Of course, the disadvantage of duplex backups is the extra disk space they require.

In the following we give brief descriptions of each of five SDBM strategies. To simplify the discussion, we will assume that database segments are numbered 1 to $N_{seg}$. Secondary storage is composed of $B$ segment-sized slots, or blocks, and $L_i$ is the $i$th block. The value of $B$ (i.e., the amount of secondary storage required) is dependent on the SDBM strategy. (Some of the SDBM strategies require small amounts of storage in addition to the $B$ blocks.)

Each of these strategies relies on the assumption that if a write to secondary storage fails (because of a system failure) then the corrupted block is detectable through a checksum or some other error detection mechanism.

### 3.1.1 Fixed Monoplex SDBM

Fixed monoplex (FMONO) SDBM is the most straightforward type. Each segment is assigned a single location in secondary storage, e.g., $S_i$ is assigned to $L_i$. In addition, a segment-sized write buffer ($L_{N_{seg}+1}$) is available on the disks. Thus $B = N_{seg} + 1$. The chief advantage of FMONO backups is that they require relatively little secondary storage space.

Two writes are required to propagate segment $S_i$. First, the segment is written to $L_{N_{seg}+1}$ (the write buffer). $S_i$ is then written to its assigned location $L_i$. This ensures then an uncorrupted version of $S_i$ is always available.

Recovery using FMONO is also straightforward. Each segment $S_i$ is read from $L_i$. If $L_i$ is corrupted, an uncorrupted version of $S_i$ is available in the write buffer.

### 3.1.2 Sliding Monoplex SDBM

The sliding monoplex (SMONO) SDBM strategy is a modified version of FMONO in which secondary updates need only be made once. SMONO backups require $B = N_{seg} + 1$ blocks of secondary storage, plus space for a *base register*. The base register holds a pointer to one of the blocks $L_i$. In addition, space is required within each block to store a *checkpoint-identifier* to indicate which checkpoint wrote that block most recently.

Segments need be written only once because the block assigned to each segment is changed each time a new checkpoint is started. Segments are assigned blocks as follows. The base register indicates the location currently assigned to $S_1$. Initially, $S_1$ is assigned $L_1$. If $S_i$ is assigned $L_i$, $S_{i+1}$ is assigned $L_{i+1}$. At the beginning of each checkpoint, the location assigned to $S_1$ is "slid" by one block, i.e., if $S_1$ was assigned to $L_i$, its new assignment is $L_{i-1}$. (The base register is carefully updated to reflect the new assignment for $S_1$.) Once $S_1$ is reassigned, the remaining segments are again sequentially assigned to the remaining blocks. Thus, the assignments of all segments are effectively slid by one block with each checkpoint.

The checkpointer must perform a full checkpoint, i.e., *all* segments are propagated to secondary storage whether they are dirty or not. Segments are propagated in order from $S_1$ to $S_{N_{seg}}$. The checkpoint is assigned a timestamp when it begins, and the timestamp is stored on each segment before it is propagated.

By "sliding" the location of each segment with each new checkpoint, the SMONO strategy ensures the the old version of a segment is still available in case the write of the new version fails. For example, if, during the second checkpoint, the propagation of $S_i$ (to $L_{i-1}$) fails, the old version of $S_i$ is still available in $L_i$.

Recovery using an SMONO backup is relatively simple. Blocks are read in sequentially starting from the block pointed at by the base register. If a corrupted block is read it is ignored. As described above, the next block contains the old version of the corrupted block and that version is restored instead. If two consecutively read uncorrupted blocks have different checkpoint timestamps, then the second block is ignored. This condition indicates that the system failed after writing the first block and before writing the second. Thus the two blocks contain the new and old versions of the same segment.

### 3.1.3 Shadowed SDBM

Shadow (SHAD) SDBM uses $B = N_{seg} + N_{shadows}$ blocks, where $0 < N_{shadows} < N_{seg}$, plus space for an indirection table $T$ containing pointers to $N_{seg}$ blocks. $T[i]$ indicates the block assigned to segment $S_i$. A copy of $T$ is also maintained in primary storage, along with two free lists, *current* and *next*. (A free list is a list of blocks that are not pointed at by $T$). Only one complete backup copy of the database exists.

To propagate a segment $S_i$ to the secondary database, the checkpointer selects any block from *current* and writes the segment to that block. The block formerly assigned to $S_i$ (indicated by $T[i]$) is added to *next* and $T[i]$ is updated to indicate the new assignment. After every $N_{shadows}$ updates, the main memory copy of the indirection table is carefully (two I/O operations) written to the disks. List *current* is replaced by *next* and *next* is cleared.

Recovery using SHAD involves restoring $T$ and using $T[i]$ to locate each segment. In addition, the initial *current* must be constructed from the indirection table.

### 3.1.4 Ping-Pong SDBM

Ping-pong is the SDBM strategy that was assumed for the checkpointing comparisons that were made in Chapter 2. Ping-pong (PIPO) uses $B = 2N_{seg}$ blocks to maintain two complete secondary copies of the database. Segments $L_1$ through $L_{N_{seg}}$ hold one copy of the database, and the remaining segments hold the other. Segment $S_i$ is assigned permanently to blocks $L_i$ and $L_{N_{seg}+i}$.

A flag is indicating which of the two database copies is "current" is maintained in primary storage. Secondary storage includes a flag indicating which of the two copies holds the most recently *completed* checkpoint.

To begin a checkpoint, the primary flag is toggled. Segments $S_i$ are propagated either to $L_i$ or to $L_{N_{seg}+i}$ depending on the flag's setting. When the checkpoint has completed, the secondary flag is toggled.

PIPO SDBM requires that each segment in primary memory be equipped with two dirty bits, one for each of the secondary database copies. This was discussed in Chapter 2 but is now briefly reviewed. When a transaction updates a segment, both dirty bits are set. When a page is

flushed by the checkpointer, only the dirty bit corresponding to the current backup database is cleared since the update will not yet have been reflected in the other backup copy. The second dirty bit ensures that updates are eventually propagated to both secondary database copies.

To recover using a PIPO, the secondary flag is examined to determine the most recent backup database copy, and all segments are read in from that copy. The primary flag is set to match the secondary flag.

### 3.1.5 Twist SDBM

The TWIST SDBM we will consider is based on a scheme suggested in [Reut80a] for disk-based databases. TWIST SDBM is essentially a variation of ping-pong in which the notion of "current" is maintained on a per segment rather than a per database copy basis.

Like ping-pong, a TWIST backup maintains space for two complete secondary database copies. We will assume that segments are assigned to backup blocks as they are in ping-pong.[†] To implement TWIST, each segment carries a timestamp which will be used during recovery to indicate which of the two blocks for that segment was most recently updated. In addition, a flag per segment (one bit, in primary memory only) is used to determine which of a segment's two blocks should be the target the next time the segment is propagated to secondary storage.

When a checkpoint begins it is assigned a timestamp. During checkpointing, segment $S_i$ (if it is to be propagated) is marked with the checkpointer's timestamp and is written to the least recently updated of its two blocks as indicated by its flag. The flag is then toggled. When the checkpoint completes, its timestamp is carefully stored to a known location (the *home* block, see Chapter 2) on secondary storage.

To recover from a TWIST backup, the timestamp ($\tau$) of the most recently completed checkpoint is retrieved first. For each $S_i$, both $L_i$ and $L_{N_{seg}+i}$ are read. The block with the largest timestamp less than $\tau$ is selected and the segment is restored from that block.

† Of course, blocks are logical entities. We have said nothing about how the blocks should be physically arranged on the disks. The ideal physical layout may be very different for PP and TWIST style backups. In particular, a TWIST backup should probably have blocks $L_i$ and $L_{N_{seg}+i}$ physically contiguous on the disks, since both will have to be read in and compared during recovery.

## 3.2 Logging

Thus far we have considered checkpointing and SDBM strategies. Together, checkpointing and SDBM provide a secondary database copy which may not be up-to-date. The log, a history of all recently executed transactions, provides a means of bringing the secondary database up to date when necessary, i.e., when recovering from the loss of the primary database.

Just as there are a variety of ways to checkpoint and update the secondary database, there are also a variety of ways to maintain the log. In this section we will consider a number of options and examine their relationships to checkpointing strategies. We will consider what should be logged, and how the log information should be maintained.

### 3.2.1 Log Data

The purpose of the log is to provide a record of all transaction activity so that, if necessary, the effects of transactions can be recreated during recovery. When a transaction executes, what should be written into the log so that the effects of the transaction can be recreated? One possibility is to record the location and new value of any part of the database modified by the transaction. Then, the transaction's effects can be recreated by restoring the new values, overwriting whatever exists at that location. This type of logging is called *state*, or *value*, logging.

Instead of recording the effect of the transaction, we could record a description of the transaction itself. Then, if the secondary database copy does not include the effects of that transaction, it can be rerun to produce them again. This is an example of *logical*, or *operation*, logging.

For example, imagine a transaction that transfers money between accounts in a banking system. A state log record for such a transaction would include the new balances of the two accounts involved in the transfer. An operation log record might include a code indicating the type of transaction (i.e., funds transfer) and any input parameters needed to rerun it (i.e., two account numbers and the amount to transfer).

Transactions are normally composed of a sequence of sub-operations or *actions*, e.g., insert-record, delete_record, modify_record. Operation logging can be done at the transaction level, as we have just described, or at the action level. The funds transfer transaction in our example might consist of two modify_record operations. Instead of making a single log entry for

the transaction, two log entries can be made, one for each of the modify_record operations. Like the transaction entry, each action log entry would include any parameters necessary to rerun the action, e.g., an account number and an amount to add or subtract from the account balance. To differentiate between these two types of operation logging, we will term the former *transaction logging* and the latter *action logging*.

The log strategy (value, action, or transaction) affects system performance both directly and indirectly. The log strategy determines the log bulk, i.e., the amount of data written to the log per transaction. Greater log bulk means more costly logging, since the CPU must spend more time moving data into the log and flushing the log to disk. Greater log bulk also means that more time is required to bring the log in from disk at recovery time. In most cases, operation logs result in less log bulk than value logs. However, when operation logs are used, actions or transactions must be re-run at recovery time. Thus, recovery of an operation log may be more costly even though the log has less bulk.

The log strategy is closely tied to checkpointing, and for this reason it affects performance indirectly as well. Recall that recovery using a transition log involves rerunning transactions or sub-operations to recreate their effects. If the results of the rerun operations are to make sense, the database against which they are run must be in a consistent state. In particular, a transaction log requires a transaction-consistent state and an action log an action consistent state. Thus, operation logging requires the use of consistent checkpoint strategies. As a general rule, the greater the consistency required, the more costly the checkpoint is to run.

### 3.2.2 UNDO Logging

Thus far we have considered the log a repository of information for recreating the effects of committed transactions, i.e., a *REDO* log. In some cases it is necessary to log information so that the effects of aborted transactions can be erased from the database. Such a log is called an *UNDO* log. (Both types of information can also be combined in a *REDO+UNDO* log.) Whether or not UNDO logging is necessary depends on how database updates are managed. Primary database updates are discussed later in Section 3.3.

Like REDO logs, UNDO logs can be value or operation logs. However, operation UNDO logging is complicated by the fact that an operation (such as a transaction) can fail at many points, and thus can have many possible inverse operations. We do not consider operation UNDO logging further and will assume instead that if UNDO logging is needed it is done by value.

### 3.2.3 Log Propagation

For reasons of space and efficiency, the log is kept in two parts. The most recently created portion of the log, the log tail, is maintained in a memory-resident buffer. The remainder of the log resides on the log disks. The *log propagation* strategy determines when log data is flushed from the buffer to disk.

Normally, the log is broken into fixed size log pages. The standard log propagation technique is to flush a log page when a transaction's commit record is written to it. (To ensure transaction durability, a transaction cannot externalize until its log record is on stable storage, i.e., the log disks.) Alternatively, log page flushes can be delayed until the page is full. Pages flushed using this strategy may contain commit records from more than one transaction. It has thus earned the name *group commit* [DeWi84a, Gawl85a]. We will call the other technique *single commit*.

Single commits can introduce inefficiencies since log pages may only be partially full when they are flushed. Grouping commits reduces the amount of disk traffic, thus reducing the overhead associated with disk I/O and the bandwidth required to the log disks. For high performance systems (thousands of transactions per second), these savings (particularly in log disk bandwidth) become critical. For this reason we will consider only group commit log propagation.

Note that grouping commits may introduce a transaction response time penalty, since transactions must wait for a log page to fill up before they can externalize, i.e., send an output message. However, at high load this is unlikely to be a major factor since log pages will fill quickly. Furthermore, this delay may be offset by reduced queueing delays at the log disks due to the lower I/O request rate of the group commit strategy.

Note that if non-volatile memory is available to hold the log tail, there is no longer any reason to consider a single commit strategy. The response time penalty of group commits is eliminated since transactions can externalize as soon as their commit records are in non-volatile RAM.

### 3.3 Other Issues

Thus far we have considered the two major components of recovery management, namely the maintenance of the log and the secondary database. Other parts of the transaction processing system can affect the performance of the recovery manager as well, though they themselves are not directly concerned with recovery. In this section we will consider two such areas: primary database management, and the transaction commit logic.

The primary database manager (PDBM) handles modifications to the primary database copy requested by the transactions. The are several aspects to PDBM that affect recovery management. We will consider two of them:

- Update Timing. Updates can be installed as they are requested, or they can be buffered outside the database until the requesting transaction commits, at which point they are installed. The first type we will call *immediate* updates, the latter *delayed* updates. The checkpoint comparisons made in Chapter 2 assumed delayed updates.
- Update Mechanism. When an update is installed, the new version can overwrite the old, or the new version can be written to a new location. In the latter case, the current location of each database object is stored in an indirection table. The first strategy will be called *in-place* updates, the other *shadow* updates. In-place updates were assumed in Chapter 2.

Immediate, in-place updates are relatively inexpensive since new versions do not need to be buffered and there is no indirection table to be maintained. However, UNDO logging must be used (in addition to REDO logging) so that the database can be restored to its original state in case a transaction aborts.

It is not necessary to propagate the UNDO log to the log disks unless the UNDO information might be needed at recovery time. This can only happen if the effects of partially completed transactions are seen in the secondary database copy, which in turn can only happen if fuzzy

checkpointing is used. Therefore, unless fuzzy checkpointing is used, the UNDO log can be maintained separately from the REDO log, with only the REDO log being propagated. With fuzzy checkpoints the logs can be merged into a single REDO+UNDO log since all of the information must be propagated.

Shadow updates suffer somewhat from the fact that the database is constantly shifting. Some mechanism, such as an indirection table, must be used to indicate which parts of the database are currently valid and which are shadows. The table or other mechanism is part of the database, which means that it must be checkpointed and its modifications must be logged. In the case of value logging, this results in additional log bulk.

Like PDBM, the transaction commit logic can affect the recovery manager. We distinguish between two types of transaction commits, namely *standard commits* and *pre-commits*. The principal difference between the two strategies is in when they release database locks. When using pre-commits, a transaction releases its locks as soon as its commit record is in the log, even if that part of the log has not yet been flushed to the log disks [DeWi84a]. With standard commits, locks are held until the commit record has been propagated to the log disks.

Pre-commits are a useful tool for reducing data contention since they reduce lock durations. However, when pre-commits are used, locking checkpointers can not rely on locking to prevent violations of the log write-ahead protocol. The checkpointer may "see" a transaction's updates and propagate them to the secondary database before the transaction's commit record is on non-volatile storage. Thus if monoplex backups are used in conjunction with a pre-commit strategy, it is necessary for the checkpointer to explicitly synchronize with the log manager. Specifically, the checkpointer must delay a segment's propagation until all log entries affecting the segment have been propagated to the log disks. Note that fuzzy checkpointers used in conjunction with monoplex backups must use explicit log synchronization whether or not pre-commits are used, since transaction locks are ignored.

### 3.4 Recovery

Thus far we have devoted most of our discussion to recovery preparations. In this section we will consider the steps taken to restore normal transaction processing once a failure has occurred.

After a system failure, the recovery manager has at its disposal a backup copy of the database and a transaction log on non-volatile storage. In a disk-based system, the log is used to bring the non-volatile database copy to a consistent state. In a MTPS, the non-volatile database copy and the log are used to recreate a consistent primary database copy in main memory.

One possible recovery strategy, a straightforward extension of the disk-based strategy, is to bring the backup database to a consistent state using the log and then to load the backup database into primary memory. However, this results in a large amount of unnecessary disk I/O. A faster strategy is to first read the backup database into main memory, and then to apply the log to the new primary copy. We will assume that this latter method is used.

The method of reading the backup database depends on the SDBM strategy; the various techniques were described in Section 3.1. Applying the log means applying each relevant log entry in the order in which they appear in the log. In the case of value logging, an entry is applied by restoring the stored value to the specified location in the database. For transition logging, application means rerunning the transaction or action with the parameters stored in the log entry. This leaves us with the question of which log entries are "relevant".

Log entries can be divided into four classes:

1) Committed: entries for transactions with commit records in the log.

2) Aborted: entries for transactions with abort records in the log.

3) Indeterminate: entries for transactions with neither a commit nor an abort record in the log. These transactions were active (or had recently finished) at the time of the failure.

4) Begin and end checkpoint markers.

A relevant log entry is one:

- that was logged by a transaction that later committed.

- who's effect is not already recorded in the secondary database. We will call such entries *uninstalled.*

Note that applying a value log entry to the database is an idempotent operation. If a value log entry is already installed, installing it again does not lead to an incorrect state (though it is a waste of processor time). However, installation of operation log entries is not, in general, idempotent. When operation logging is used, it is important that only uninstalled log entries be installed, and that they be installed exactly once.

The determination of which log entries are uninstalled is checkpoint-dependent. However, in all cases, entries logged by transactions whose commit records are logged before the beginning of the most recently *completed* checkpoint are installed in that checkpoint, and thus can be ignored. In the following, we consider only log entries from transactions which committed after the checkpoint began.

- Copy-on-Update Checkpoints: COU checkpoints quiesce activity at the beginning of the checkpoint and force transactions to save the database state as it exists at the beginning of the checkpoint. All log entries after the the begin checkpoint marker are uninstalled.
- Black/White Checkpoints: Each checkpoint has a target color, the color it paints database segments once they are propagated to the backup. Log entries also have a color, indicating whether they apply to black or white data. Log entries of the target color are uninstalled and must be installed. Entries of the other color are already reflected in the backup. Note that if there is a begin-checkpoint log entry for an incomplete checkpoint (no end-checkpoint marker exists) in the log, all log entries after that point are uninstalled, regardless of their color.
- Fuzzy and Segment-Consistent Checkpoints: These types of checkpoints do not provide a mechanism for determining which log entries are already installed. However, this is not a problem since both types of checkpoints are used only with value logging. All log entries made by transactions that commit after the beginning of the checkpoint are treated as uninstalled.

Thus far we have determined which log entries are uninstalled. To decide whether or not to apply a log entry, we must also know whether the transaction that wrote the entry later

committed. Since a transaction's commit entry is the last entry it makes in the log, a simple forward (chronological) order log scan will not provide the necessary information at the proper time.

There are many solutions to this problem. One possibility is to scan the log twice, once backwards to determine which entries belong to committed transactions, and once forward to apply them. Other strategies use only a single pass and maintain auxiliary data structures to achieve the same effect. When modeling the recovery time in the next chapter, we will assume that a one-pass strategy is used. We will not consider other aspects of the log scan in detail.

# CHAPTER 4
# A PERFORMANCE MODEL

In this chapter we present a performance model for recovery managers. The model is an analytic tool that can be used to predict the performance of various combinations of recovery strategies. A brief summary of the model was presented in Chapter 2. The model parameters described here are a superset of the parameters presented in Chapter 2.

A recovery manager is active while transactions are being processed as well as after a failure has occurred. Thus there are two aspects to a recovery manager's performance:

- the degradation in transaction processing performance because of resources consumed by the recovery manager in preparation for a failure
- the time required to resume transaction processing once a failure has occurred.

As we shall see later, a recovery manager manager provides a tradeoff between these two aspects.

The model that we will describe computes three recovery metrics:

- *Transaction Processor Overhead*: transaction overhead is the amount of processor power (i.e., number of instructions executed per transaction) dedicated to recovery preparations during normal transaction processing.
- *Recovery Processor Cost*: the recovery processor cost is the amount of processor power required to restore normal transaction processing after a failure has occurred.
- *Recovery I/O Cost*: the recovery I/O cost is the time required to retrieve data from stable storage (e.g., the log disks and backup disks) during failure recovery.

These metrics are simpler to compute than higher-level metrics such as transaction throughput, and yet are closely related to them. The first metric is indicative of performance degradation during transaction processing. A higher transaction overhead results in lower transaction throughput because the processor is the critical resource in a memory-resident system.

The second and third metrics are indicative of performance after a failure. If the processor's speed is known, they can be used to determine the recovery time, i.e., how quickly transaction processing can be restored. (The performance comparisons presented in Chapter 2 considered transaction processor overhead and recovery time.) In the remainder of this chapter, we will develop expressions for each of these three metrics.

### 4.1 Transaction Processor Overhead

The transaction processor overhead is the work done by the processor to prepare for failure recovery. Such preparations occupy time that could otherwise have been spent processing transactions, thus a recovery manager with a high transaction processor overhead is detrimental to transaction processing performance. In this section we will describe a procedure to determine, given the strategies used in the recovery manager, the transaction processing overhead.

We will divide the overhead into three components. The first of these is the cost of making a checkpoint, i.e., of sweeping through the primary database propagating segments to the backup. We will call this the *asynchronous* cost (because checkpointing is accomplished asynchronously). The second will encompass costs incurred as each transaction is run, in particular the cost of maintaining the log. This will be termed the *synchronous* cost. The final component is the *restart* cost, the cost of rerunning transactions that must be aborted *because of recovery management activities*. In the rest of this section we will develop expressions for these costs, and then combine them into a single expression for the transaction processing overhead. The combined cost is the overhead metric used in the performance comparisons presented in Chapters 2 and 7.

#### 4.1.1 Synchronous Overhead

Synchronous overhead will be computed on a per-transaction basis. As we will see, a transaction's overhead cost depends on whether the transaction is successful (commits) or unsuccessful (aborts for any reason). We will develop separate expressions for $C_{ok_synch}$ and $C_{fail_synch}$, the overhead costs of successful and unsuccessful transactions.

A model parameter, $p_{fail}$, gives the probability that a transaction will abort voluntarily, e.g., if an incorrect account number is supplied. Using $p_{fail}$, we can combine the expressions for

successful and unsuccessful transactions into a single expression for synchronous overhead:

$$C_{synch} = p_{fail} C_{fail_synch} + (1 - p_{fail}) C_{ok_synch}$$

Any transaction, whether it ultimately commits or voluntarily aborts, may be involuntarily aborted and restarted one or more times because of recovery constraints, i.e., for attempting to access both black and white data when a black/white checkpointer is being used. Such recovery-induced restarts consume processor resources and thus contribute to the total transaction overhead. However, we will not consider this cost in our computation of $C_{synch}$. Instead, we will consider it separately in Section 4.1.3. In the remainder of this section we will develop expressions for $C_{ok_synch}$ and $C_{fail_synch}$.

#### 4.1.1.1 Successful Transactions

The principal component of the synchronous overhead is the cost of maintaining the transaction log. Log maintenance includes log buffer management (e.g., determining when buffers are filled and where the next log entry should go), flushing buffers to the log disks, and the cost of copying log data into the buffers. The synchronous overhead may also include an additional cost, the copy-on-update cost. We will write

$$C_{ok_synch} = C_{log} + C_{cou}$$

and consider each component cost separately. Table 4.1 lists the model parameters that will be used in the expression for $C_{ok_synch}$, including many of the parameters that already appeared in Tables 2.1 and 2.2. Table 4.1 also shows the default parameter values used in Chapters 2 and 7.

Central to the calculation of $C_{ok_synch}$ is the model of transactions. A transaction consists of a collection of database actions (e.g., reads and writes). The number of actions can vary from transaction to transaction. A function $N_{act}$ determines the number of actions per transaction in the model's load. $N_{act}(i)$ gives the fraction of the transaction load having $i$ actions. $N_{act}$ is specified as a model parameter.

The fraction of actions that are updates is given by $R_{up}$. The (expected) number of update actions per transaction can then be calculated using

$$N_{upact} = R_{up} \sum_i i N_{act}(i)$$

| SYMBOL | DESCRIPTION | DEFAULT | UNITS |
|---|---|---|---|
| $S_{rec}$ | record size | 32 | words |
| $S_{lent}$ | log entry overhead | 4 | words |
| $S_{op}$ | operation log entry size | 32 | words |
| $S_{lpg}$ | log page size | 1024 | words |
| $S_{seg}$ | segment size | 8192 | words |
| $N_{act}(i)$ | probability of $i$ actions | $0.1\ 0 < i \leq 10$ | |
| $N_{upact}$ | update actions per transaction | calculated | actions/transaction |
| $R_{up}$ | frequency of update actions | 0.33 | |
| $R_{spa}$ | segments per action | 1.1 | segments/action |
| $R_{rpa}$ | records per action | 1 | records/action |
| $N_{ru}$ | records updated per transaction | calculated | records/transaction |
| $N_{su}$ | segments updated per transaction | calculated | segments/transaction |
| $N_{rent}$ | REDO log entries per trans. | calculated | entries |
| $N_{uent}$ | UNDO log entries per trans. | calculated | entries |
| $N_{ritems}$ | REDO log items per trans. | calculated | items |
| $N_{uitems}$ | UNDO log items per trans. | calculated | entries |
| $D_{redo}$ | REDO log bulk per trans. | calculated | words |
| $D_{undo}$ | UNDO log bulk per trans. | calculated | words |
| $C_{mv_fixed}$ | constant data movement cost | 10 | instructions/word |
| $C_{move}$ | linear data movement cost | 1 | instructions/word |
| $C_{io}$ | I/O initiation cost | 1500 | instructions |
| $C_{lalloc}$ | log maintenance cost | 200 | instructions |
| $C_{salloc}$ | segment (de)allocation cost | 100 | instructions |
| $C_{lock}$ | locking (synchronization) cost | 50 | instructions |
| $C_{logopen}$ | per-entry log cost | 50 | instructions |
| $C_{logend}$ | per-transaction log cost | 100 | instructions |
| $N_{coutrans}$ | segments copied-on-update | calculated | segments/transaction |
| $p_{fail}$ | probability of voluntary abort | 0.05 | |

Table 4.1

Each update action affects $R_{rpa}$ records and $R_{spa}$ segments ($R_{rpa} \geq R_{spa}$). Thus a transaction updates $N_{ru} = R_{rpa} N_{upact}$ records and $N_{su} = R_{spa} N_{upact}$ segments.

A transaction's log overhead cost $C_{log}$ is directly related to the number of log *entries* and to its *log bulk*, the total volume of log data it produces. A log entry consists of a *header* and zero or more *items* of data. (For example, in a REDO/UNDO log, an entry might consist of a header plus

two items, the old and new versions of a modified record.) $S_{lent}$ is the size of a log header.

The size of each log item, the number of items per entry, and the number of entries per transaction, all depend on the logging strategy:

- For VALUE logging, each transaction logs the new version of each updated record. In addition, the transaction must log any incidental modifications to affected segments (e.g., modification of a primary index to indicate the new location of a record). Finally, each transaction logs a commit record to indicate the end of its log activity.
- If action operation (AOPER) logging is used, transactions log the type and input parameter(s) of each action executed (e.g., insert record), plus a commit record.
- For transaction operation (TOPER) logging, each transaction makes a single log entry, describing the type of transaction executed and any input parameters to the transaction.

Table 4.2 presents the expressions for $N_{rent}$, $N_{ritems}$, and $D_{redo}$, the number of entries, items, and total log bulk per transaction for the various logging strategies. We assume that

- Incidental modifications made to segments affect a single word, e.g., a pointer is modified to reflect the new location of an updated record.
- The size of a commit record is roughly the size of a log header ($S_{lent}$).
- $S_{op}$ is the size of the input parameters for a transaction or an action.

| Logging Strategy | $D_{redo}$ | $N_{rent}$ | $N_{ritems}$ |
|---|---|---|---|
| VALUE | $N_{ru}\ (S_{lent} + S_{rec}) + N_{su}(S_{lent} + 1) + S_{lent}$ | $N_{ru} + 1$ | $N_{ru} + N_{su}$ |
| AOPER | $N_{upact}\ (S_{lent} + S_{op}) + S_{lent}$ | $N_{upact} + 1$ | $N_{upact}$ |
| TOPER | $S_{lent} + S_{op}$ | 1 | 1 |

Table 4.2

If an in-place update strategy is used, transactions must log UNDO data as well. Letting $N_{uent}$, $N_{uitems}$, and $D_{undo}$ represent the number of entries, items, and total bulk of the UNDO component of the log, we can write

$$D_{undo} = N_{ru}\ (S_{lent} + S_{rec}) + N_{su}S_{lent}$$

$$N_{uent} = N_{ru}$$

$$N_{uitems} = N_{ru} + N_{su}$$

If updates are not made in-place, $D_{undo} = N_{uent} = N_{items} = 0$.

Once the log bulk is determined, we can compute the log cost, $C_{log}$, which has several components:

- Transactions must move their log data into the log. The cost of moving a single item is taken to be

  $$C_{mv_fixed} + C_{move}(size_of_item)$$

  Since each transaction logs $N_{ritems} + N_{uitems}$ items, whose total size (including the log headers) is $D_{redo} + D_{undo}$, the data movement component of the log cost is taken to be

  $$C_{mv_fixed}(N_{ritems} + N_{uitems}) + C_{move}(D_{redo} + D_{undo})$$

- Log buffer management and buffer flushing costs are charged in proportion to the number of log pages a transaction fills. The log page size is $S_{lpg}$, so the cost due to flushing and buffer management is

  $$C_{io}\left[\frac{D_{redo}}{S_{lpg}}\right] + C_{lalloc}\left[\frac{D_{redo} + D_{undo}}{S_{lpg}}\right]$$

  This expression assumes that only the REDO portion of the log is flushed to disk, i.e., the UNDO entries are kept in a separate log. For some combinations of recovery strategies (e.g., FUZZY checkpoints combined with immediate, in-place updates) the UNDO entries must go to disk as well. In those cases, an extra factor of

  $$C_{io}\left[\frac{D_{undo}}{S_{lpg}}\right]$$

  must be added to $C_{log}$ to account for the cost of flushing the UNDO log bulk.

- A constant per-entry overhead cost is used to account for such activity as determining where the entry should be place on the log page, and synchronization of multiple loggers. This adds a cost of

  $$C_{logopen}(N_{rent} + N_{uent})$$

- A per-transaction overhead cost adds an additional $C_{logend}$ to the log cost.

We now have a complete expression for $C_{log}$, but have still to consider $C_{cou}$. $C_{cou}$ represents additional costs incurred by transactions when copy-on-update checkpointing is used. If the checkpoint strategy is not copy-on-update, $C_{cou} = 0$. The additional costs come from two activities. One is the cost of making "snapshot" copies of database segments for use by copy-on-update checkpointers. For each segment that needs to be copied, space must be allocated and the segment moved. If $N_{coutrans}$ is the number of segments copied-on-update by a transaction, the cost of making the snapshot copies is given by

$$N_{coutrans} \left( C_{salloc} + C_{mv_fixed} + C_{move} S_{seg} \right)$$

We will develop an expression for $N_{coutrans}$ in Section 4.1.2.

The second activity is arises because the checkpointer needs to bring activity to a transaction- or action-quiescent state before a new checkpoint can begin. Each operation (action or transaction) must check, before running, whether or not it is allowed to proceed immediately. Since this is a synchronization operation, we assess a cost $C_{lock}$ for this check. If the checkpointing strategy is ACBW, each update action must check before proceeding. Since there are $N_{upact}$ update actions per transaction, the total COU cost for a transaction is

$$C_{cou} = N_{coutrans} \left( C_{salloc} + C_{mv_fixed} + C_{move} S_{seg} \right) + N_{upact} C_{lock}$$

For TCBW checkpoints, only one check is made per transaction. The total COU cost for TCBW transactions is

$$C_{cou} = N_{coutrans} \left( C_{salloc} + C_{mv_fixed} + C_{move} S_{seg} \right) + N_{upact} C_{lock}$$

#### 4.1.1.2 Unsuccessful Transactions

In this section we develop an expression for $C_{fail_synch}$, the overhead cost of an unsuccessful transaction. Like $C_{ok_synch}$, $C_{fail_synch}$ consists of logging and copy-on-update costs. We will develop new expressions for $C_{log}$ and $C_{cou}$ and will combine them as before using $C_{fail_synch} = C_{log} + C_{cou}$. For simplicity, we will assume that failed transactions abort after completing half of their actions.

The log cost expression for unsuccessful transactions is the same as that for successful transactions; however the log bulk of unsuccessful transactions is different. If delayed updates are used, $D_{redo}$, $N_{rent}$, and $N_{ritems}$ are zero for unsuccessful transactions since no database modifications (or log entries) are made until the commit point. If immediate updates are used, the REDO log bulk is usually half that of a successful transaction (not including the commit/abort entry). Table 4.3 describes the the REDO log bulk for unsuccessful transactions (assuming an immediate update strategy).

| Logging Strategy | $D_{redo}$ | $N_{rent}$ | $N_{ritems}$ |
|---|---|---|---|
| VALUE | $\frac{N_{ru}(S_{lent}+S_{rec})+N_{su}S_{lent}}{2}+S_{lent}$ | $\frac{N_{ru}}{2}+1$ | $\frac{N_{ru}+N_{su}}{2}$ |
| AOPER | $\frac{N_{upact}(S_{lent}+S_{rec})}{2}+S_{lent}$ | $\frac{N_{upact}}{2}+1$ | $\frac{N_{upact}}{2}$ |
| TOPER[1] | 0 | 0 | 0 |

1) Under TOPER logging, transactions make a log entry only when they commit, so $D_{redo} = 0$.

Table 4.3

If in-place, immediate updates are used, unsuccessful transactions also have UNDO log bulk. The UNDO log bulk for an unsuccessful transaction is half that of a successful one.

The log cost expressions for unsuccessful transactions are nearly the same as those for successful ones. There are two exceptions:

- The log bulk expressions of Table 4.3 rather than Table 4.2 are used in the calculations.
- An unsuccessful transaction is charged additional overhead for undoing its effects by copying its UNDO log information from the log buffer back into the database. The cost of moving the data is given by

$$N_{uitems}C_{mv_fixed} + D_{undo}C_{move}$$

Of course, unless in-place updates are used, $D_{undo}$ and $N_{uitems}$ are zero and so is the additional undo cost.

Copy-on-update costs are as for successful transactions except that only half as many segments are copied:

$$C_{cou} = \frac{N_{coutrans}}{2}(C_{salloc} + C_{mv_fixed} + C_{move}S_{seg})$$

The quiescing cost is half that of a successful transaction if ACBW checkpointing is done, since only half as many actions are executed. For TCBW, the quiescing cost remains one.

### 4.1.2 Asynchronous Overhead

The asynchronous overhead is the cost of making a database checkpoint. Whereas $C_{synch}$ is a per-transaction overhead metric, $C_{asynch}$ will reflect the cost of the entire checkpoint, which spans many transactions. Later, we will combine $C_{synch}$ and $C_{asynch}$ into a single per-transaction metric by determining the time required to complete the checkpoint and dividing $C_{asynch}$ among all of the transactions that run during that interval. The new model parameters that we will use in determining $C_{asynch}$ are shown in Table 4.4.

| symbol | description | default | units |
|---|---|---|---|
| $N_{seg}$ | number of segments | calculated | segments |
| $C_{lsn}$ | check log seq. number cost | 50 | instructions |
| $N_{chk}$ | number of segments propagated | calculated | segments/checkpoint |
| $N_{chkio}$ | number of segment I/O's | calculated | segments/checkpoint |
| $N_{coutot}$ | segments copied-on-update | calculated | segments/checkpoint |
| $t_{icp}^{min}$ | minimum checkpoint interval | calculated | seconds |
| $t_{icp}$ | checkpoint interval | $t_{icp}^{min}$ | seconds |
| $a_{back}$ | backup I/O constant time | -- | seconds |
| $b_{back}$ | backup I/O linear time | -- | seconds/word |
| $N_{shadow}$ | number of backup shadow blocks | 10 | blocks |
| $\lambda$ | transaction arrival rate | 1000 | transactions/second |
| $n_{pieces}$ | number of hot/cold regions | 1 | |
| $\Delta(i)$ | fraction of database in $i$th region | 1.0 ($i = 1$) | |
| $\Phi(i)$ | fraction of updates to $i$th region | 1.0 ($i = 1$) | |
| $S_{db}$ | database size | 256 | Mwords |

Table 4.4

Making a checkpoint may involve locking, copying, and initiating the I/O of database segments (depending on which checkpointing strategy is being used). For now we will make use of $N_{chk}$, $N_{chkio}$, and $N_{coutot}$ without determining their values. Later, we will develop an expressions for them.

The principal checkpointing cost is the cost of propagating the dirty segments:

$$C_{asynch} = C_{io} N_{chkio}$$

To this is added a variety of other costs depending on which checkpointing and SDBM strategies are being employed.

- For non-FUZZY checkpoints, the checkpointer must lock each database segment. This adds a cost of $C_{lock} N_{seg}$
- If the checkpointing strategy requires explicit log synchronization (e.g., fuzzy checkpoints to a monoplex backup), the checkpointer must check the log sequence number of each propagated page at a cost of $C_{lsn} N_{chk}$
- If a copy-style checkpoint strategy is used, the checkpointer must copy segments to a special buffer before propagating them. Unless the checkpoint strategy is also copy-on-update, this adds a cost of $C_{move} S_{seg} N_{chk}$. In the case of copy-on-update checkpoints, the checkpointer need not re-copy segments that have already been copied by transactions. Thus the added cost for copy-on-update checkpointers is only $C_{move} S_{seg} (N_{chk} - N_{coutot})$
- If copy-on-update checkpoints are used, the checkpointer must release the segment copies produced by the transactions once they have been propagated to the backup. This adds a cost of $N_{coutot} C_{salloc}$.

Except for values for $N_{chk}$, $N_{coutot}$, and $N_{chkio}$, we have a complete expression for $C_{asynch}$. To get expressions for the three remaining quantities, we must know $t_{icp}$, the checkpoint interval. The checkpoint interval is the time between the beginnings of checkpoints.

The checkpoint interval can be made arbitrarily long by inserting a delay between the completion of one checkpoint and the beginning of the next. (For this reason, $t_{icp}$ is a model parameter.) Thus a checkpoint interval can be divided into two sub-intervals, the *active interval*, during which checkpoint activity is actually occurring, and the *inactive* interval (the delay).

The checkpoint interval cannot be less than the active interval, and the duration of the active interval will be calculated from the model parameters. Thus, $t_{icp}$ cannot be made arbitrarily small. The duration of the active interval, $t_{icp}^{min}$, depends on how quickly the transactions dirty database segments and how quickly the checkpointer can clean them by propagating them to the

secondary database. We will next calculate $t_{icp}^{min}$.

We will assume that the checkpointer is I/O-bound, i.e., the duration of the active checkpoint interval is the time necessary to flush the dirty segments to the backup disks. The number of segments that can be written to the backup disks in time $t$ is

$$N_{io}(t) = \frac{1}{a_{back} + b_{back} S_{seg}}$$

Transactions update $N_{su}$ segments each. We assume a *piecewise-uniform* update probability across the database. This means that the database can be split into $n_{pieces}$ pieces such that all segments in a piece are equally likely to be updated. The size of each piece and the probability that an update will be to a segment in that piece are given by the functions $\Delta(i)$ and $\Phi(i)$. We will let $\Delta(i)N_{seg}$ and $\Phi(i)$ represent the size (number of segments) and update probability, respectively, of the $i$th piece. $N_{seg}$ is the total number of segments in the database, given by

$$N_{seg} = \frac{S_{db}}{S_{seg}}$$

Obviously, we will require that

$$\sum_{i=1}^{n_{pieces}} \Delta(i) = \sum_{i=1}^{n_{pieces}} \Phi(i) = 1$$

A segment update will miss an arbitrary segment in the $i$th piece with probability

$$\left[1 - \frac{1}{\Delta(i)N_{seg}}\right]$$

We will take the probability that a transaction $T$ will not update an arbitrary segment in the $i$th piece to be

$$\left[1 - \frac{1}{\Delta(i)N_{seg}}\right]^{\Phi(i)N_{su}}$$

(The expected number of segments from the $i$th piece updated by $T$ is $\Phi(i)N_{su}$.) Over a time interval $t$, $\lambda t$ transactions are processed. Then the probability that at least one of these will update the segment is

$$1 - \left[1 - \frac{1}{\Delta(i)N_{seg}}\right]^{\Phi(i)N_{su}\lambda t}$$

The expected number of segments dirtied during a time $t$ is thus

$$N_{dirty}(t) = \sum_{i=1}^{n_{pieces}} \Delta(i) N_{seg} \left[ 1 - \left[ 1 - \frac{1}{\Delta(i) N_{seg}} \right]^{\Phi(i) N_{su} \lambda t} \right]$$

The expected number of pages the checkpointer must flush to the backup disks during the checkpoint interval of duration $t$, written $N_{flush}(t)$, is a function of the number of dirty pages and of the SDBM, as described in the following:

- FMONO backups: each dirty segment flushed twice.
- SMONO backups: $N_{seg}$ (i.e., all) segments are flushed, regardless of how many are dirty
- SHADOW backups: each dirty segment flushed once, but the indirection table must be updated carefully (i.e., twice) for every $N_{shadow}$ pages that are flushed.
- TWIST backups: each dirty page flushed once.
- PIPO backups: Each of the two backup copies sees updates only during every other checkpoint. Thus the currently active copy sees as many dirty pages as a monoplex backup would see during a checkpoint interval of twice the duration. Each dirty page is flushed once to the active backup.

These relationships between $N_{flush}$ and $N_{dirty}$ are summarized in Table 4.5.

| backup policy | $N_{flush}(t)$ |
|---|---|
| FMONO | $2N_{dirty}(t)$ |
| SMONO[1] | $N_{seg} + 2$ |
| SHADOW | $\left[1 + \frac{2}{N_{shadow}}\right] N_{dirty}(t)$ |
| PINGPONG[1] | $N_{dirty}(2t) + 2$ |
| TWIST | $N_{dirty}(t)$ |

1) The two extra segment flushes charged to SMONO and PIPO backups are for the one-time (per checkpoint) careful updating of the base (SMONO) or current backup (PIPO) pointers on disk.

Table 4.5

We find $t_{icp}^{min}$, by setting $N_{flush}(t_{icp}^{min}) = N_{io}(t_{icp}^{min})$ and solving for $t_{icp}^{min}$. When this equality holds, the system is flushing dirty segments at the same rate they would be expected to be created by the transactions. The equality results in an expression with the general form $a t_{icp}^{min} + log(t_{icp}^{min}) + b = 0$,

where $a$ and $b$ are constants. We solve this expression numerically to arrive at a value for $t_{icp}^{min}$.

Once the checkpoint interval is known, we can get the desired expressions for $N_{chk}$, $N_{chkio}$, and $N_{coutot}$. Fortunately, we have done must of the work during our calculation of $t_{icp}^{min}$. Unless PIPO backups are used, the expected number of segments to checkpoint is simply the number dirtied during the checkpointer interval, so

$$N_{chk} = N_{dirty}(t_{icp})$$

In the case of PIPO backups, segments dirtied during the last two checkpoint intervals must be checkpointed, so

$$N_{chk} = N_{dirty}(2t_{icp})$$

We have also already determined the expected number of segment I/O operations performed by the checkpointer:

$$N_{chkio} = N_{flush}(t_{icp})$$

This leaves us with $N_{coutot}$, the expected number of segments copied on update during a checkpoint. Recall that a segment must be copied-on-update the first time it is updated after a checkpoint begins but before the segment has been examined by the checkpointer. Thus, copies-on-update can only occur during the active interval. To simplify our calculations, we will assume that the checkpointer proceeds at a uniform rate through the database while it is active, i.e., that it examines the $n$th segment at time

$$t_{ck}(n) = \frac{(n-1)}{N_{seg}} t_{icp}^{min}$$

after the beginning of the checkpoint.

We will compute $p_{cou}(n)$, the probability that the $n$th segment visited by the checkpointer will first have been copied-on-update. We observe that this is the same as the probability of a segment being updated at least once (dirtied) in time $t_{ck}(n)$. Since we have already calculated $N_{dirty}(t)$, we will express this as

$$p_{cou}(n) = \frac{N_{dirty}(t_{ck}(n))}{N_{seg}}$$

We can now write the desired expression for $N_{coutot}$:

$$N_{coutot} = \sum_{n=1}^{N_{seg}} p_{cou}(n)$$

Note that earlier, in Section 4.1.1, we made use of $N_{coutrans}$, the number of pages copied on update per transaction, without determining its value. To get an expression for $N_{coutrans}$, we will spread the cost of copying-on-update evenly over all transactions. Since we expect $\lambda t_{icp}$ transactions during a checkpoint interval, we will write:

$$N_{coutrans} = \frac{N_{coutot}}{\lambda t_{icp}}$$

### 4.1.3 Restart Costs

Restart costs are incurred when a transaction must be aborted and restarted due to activities of the recovery manager. The only recovery strategies that cause transactions to restart are the black/white checkpointing strategies. Unless black/white checkpointing is used, $C_{restart} = 0$. Parameters that will be used in determining the restart costs of non-black/white checkpoints are shown in Table 4.6.

| symbol | description | default | units |
|---|---|---|---|
| $C_{trans}$ | raw transaction cost | 10000 | instructions |
| $f_{retry}$ | retry frequency | 0.25 | |
| $p_{restart}$ | recovery-induced restart prob. | calculated | |
| $N_{tries}$ | number of attempts to complete | calculated | |

Table 4.6

As noted in [Agra85a], the *entire cost*, up to the point of restart, of executing a transaction aborted by the recovery manager must be considered part of the overhead of the recovery mechanism. Assuming as before that aborted transactions abort halfway through, restarting a transaction incurs a cost of

$$\frac{C_{trans}}{2} + C_{fail_synch}$$

in addition to the overhead of the restarted transaction. If a transaction requires $N_{tries}$ attempts to complete (including the successful attempt), we take the restart cost to be:

$$C_{restart} = (N_{tries} - 1)\left[\frac{C_{trans}}{2} + C_{fail_synch}\right]$$

We are left with the task of computing $N_{tries}$. We will consider transaction-consistent black/white (TCBW) checkpoints first, followed by action-consistent black/white checkpoints.

Transactions can only violate the black/white restriction, and thus can only be restarted, while the checkpointer is active. The checkpointer imposes an order on the database segments, namely the order in which it examines them. So we can think of the segments as being labeled from 1 to $N_{seg}$ by the checkpointer. Figure 4.1 shows the labeled segments and a transaction $T$ which requires access to three of them.

What is the likelihood that $T$ will have to be restarted for violating the black/white restriction of a TCBW checkpointer? It is the same as the likelihood that the checkpointer will be examining a segment after segment 3 and before segment $N_{seg} - 1$ when $T$ runs. If we assume for simplicity that:

- the checkpointer proceeds at a uniform rate through the database
- the active segment (the segment currently being examined by the checkpointer) does not change during $T$'s lifetime (This is a simplifying assumption which is reasonable as long as most transactions are short-lived.)

then the probability of restart for $T$ is simply

$$\frac{(N_{seg} - 1) - 3}{N_{seg}}$$

In general, if the difference between a transaction's highest- and lowest-numbered segments is $D$, the restart probability $p_{restart}$ is

$$\frac{D}{N_{seg}}$$

If we assume that the checkpointer chooses its next database segment at random from among those it has not examined, then the ratio above, for a transaction accessing $i$ segments, is simply the expected distance between the largest and smallest of $i$ points chosen at random on (0,1). Knowing that the probability of $i$ such points being less than $x$ (where $0 \le x < 1$) is $x^i$, we can determine the expected positions of the greatest ($\bar{T}$) and least ($\bar{B}$) of points to be:

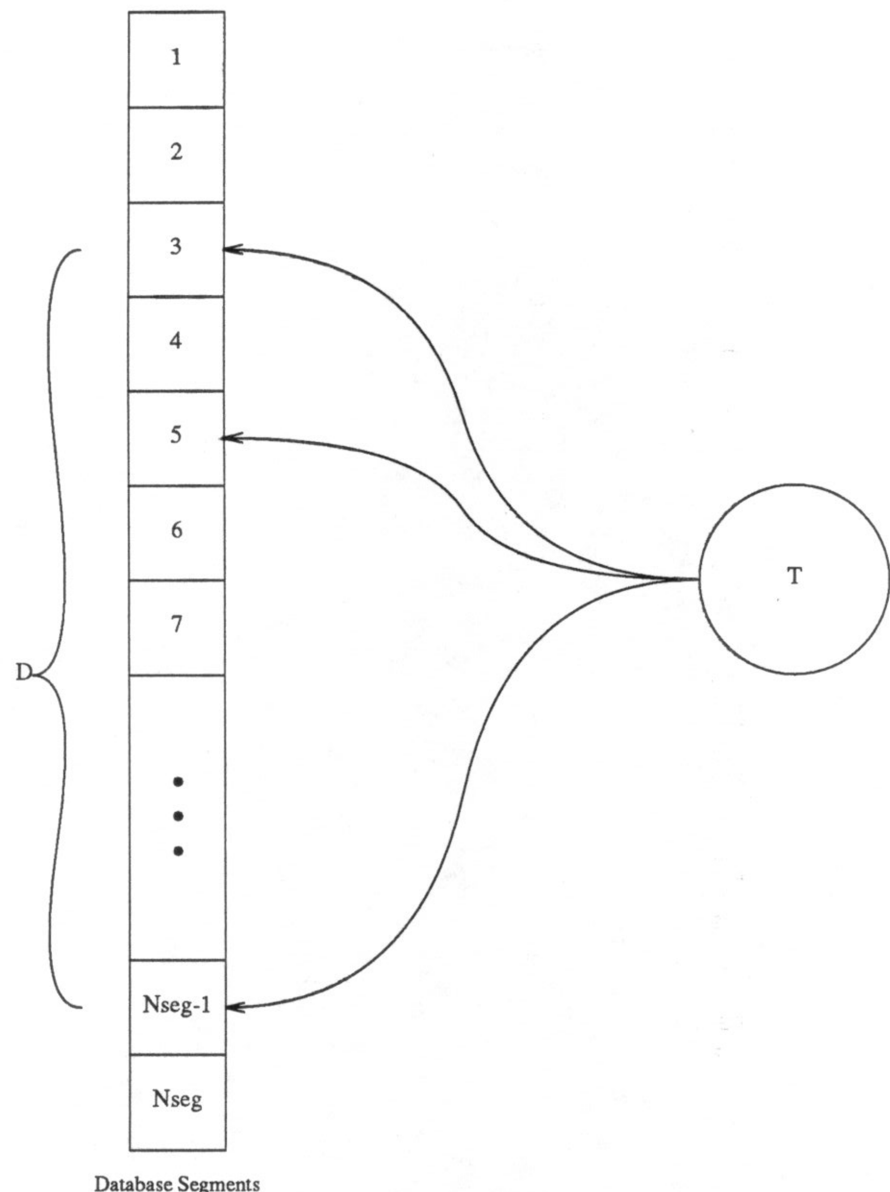



Figure 4.1 - Transaction Data Access and Restart

$$\overline{T} = \frac{i}{i+1} \quad \overline{B} = \frac{1}{i+1}$$

Their difference gives us the desired expression for $p_{restart}$:

$$\frac{D}{N_{seg}} = \overline{T} - \overline{B} = \frac{i-1}{i+1}$$

How many times must a transaction be restarted before it can run without violating the color restrictions? The *retry frequency*, $f_{retry}$, specifies how frequently the system will attempt to rerun a transaction that is to be restarted. It is expressed as a fraction of the checkpoint interval. Alternatively (given our assumption of uniform checkpointer progress), we can say that the system attempts to restart aborted transactions each time the checkpointer completes another $f_{retry}N_{seg}$ segments.

In the worst case, if a transaction is restarted once it will have to be restarted as many times as the restart interval $f_{retry}N_{seg}$ occurs in $D$, or

$$\left\lceil \frac{D}{f_{retry}N_{seg}} \right\rceil \text{ times}$$

In the best case a single restart is sufficient. Taking advantage of our "uniform progress" assumption again, we will use half of the worst case number of retries. Thus the expected number of attempts per transaction (including the successful attempt), is given by

$$1 + \frac{p_{restart}}{2} \left\lceil \frac{D}{f_{retry}N_{seg}} \right\rceil$$

A transaction with $i$ actions accesses $iR_{spa}$ segments. As we have already seen, for a transaction accessing $iR_{spa}$ segments we expect:

$$\frac{D}{N_{seg}} = \frac{iR_{spa} - 1}{iR_{spa} + 1}$$

We now have an expression for the number of attempts per transaction when the checkpointer is active and when the transaction accesses $i$ segments. We make this independent of $i$ by considering $N_{act}(i)$, the likelihood of an $i$-action transaction and arrive at an expression for $N_{tries}$:

$$N_{tries} = \sum_i N_{act}(i) \left[ 1 + \frac{p_{restart}}{2} \left\lceil \frac{iR_{spa} - 1}{f_{retry}(iR_{spa} + 1)} \right\rceil \right]$$

When the checkpointer is inactive, $N_{tries} = 1$. Since the checkpointer is active for a period $t_{icp}^{min}$ out of every checkpoint interval $t_{icp}$, we modify our expression for $n_{tries}$ to account for this:

$$N_{tries} = 1 + \sum_i \frac{N_{act}(i) p_{restart}}{2} \left\lceil \frac{iR_{spa} - 1}{f_{retry}(iR_{spa} + 1)} \right\rceil$$

For ACBW checkpoints we develop a similar expression. A transaction must be restarted if any of its update actions violate the black/white restriction. An action touches $R_{spa}$ segments, so during the active checkpoint interval an action will not violate the restrictions with probability

$$1 - \frac{R_{spa} - 1}{R_{spa} + 1} = \frac{2}{R_{spa} + 1}$$

Transaction consist of $N_{upact}$ actions, so a transaction will not violate the restrictions with probability

$$\left[ \frac{2}{R_{spa} + 1} \right]^{N_{upact}}$$

The restart probability is then given by

$$p_{restart} = \left[ 1 - \left[ \frac{2}{R_{spa} + 1} \right]^{N_{upact}} \right]$$

When ACBW checkpointing is used, a restarted transaction cannot complete successfully until all segments affected by the violating action are the same color.† As we did for TCBW, we will use half of the worst-case number of restart intervals $f_{retry} N_{seg}$ that will occur in this time. For ACBW, this is given by

$$\frac{1}{2f_{retry}} \left[ \frac{R_{spa} - 1}{R_{spa} + 1} \right]$$

The number of attempts to complete is then given by

$$N_{tries} = 1 + \frac{t_{icp}^{min}}{t_{icp}} \frac{p_{restart}}{2f_{retry}} \left[ \frac{R_{spa} - 1}{R_{spa} + 1} \right]$$

We have now developed complete expressions for the three overhead costs. It is convenient to merge these into a single per-transaction overhead metric, $C_{tot}$. We can do so by dividing the asynchronous cost over all transactions that ran during the checkpoint interval. Since there are $\lambda t_{icp}$ transactions during the interval, we can write

† We are ignoring here the increase in retry count that can be expected if more than one of a transaction's actions violate two-color rule.

$$C_{tot} = C_{synch} + C_{restart} + \frac{C_{asynch}}{\lambda t_{icp}}$$

## 4.2 Recovery Costs

The overhead metric, $C_{tot}$, provides an indication of how recovery management affects transaction processing. In this section we will develop expressions for the two remaining metrics. Together they are indicative of the performance of the recovery manager once a failure has occurred. Table 4.7 shows new parameters that will be used in the recovery cost calculations.

| symbol | description | default | units |
|---|---|---|---|
| $a_{log}$ | log I/O constant time | -- | seconds |
| $b_{log}$ | log I/O linear time | -- | seconds |
| $t_{ioback}$ | database restoration I/O time | calculated | seconds |
| $t_{iolog}$ | log replay I/O time | calculated | seconds |
| $C_{cpulog}$ | log replay processor cost | calculated | instructions |
| $C_{cpuback}$ | database restoration processor cost | calculated | instructions |
| $t_{crash}$ | rollback time | calculated | seconds |
| $N_{\log}$ | log pages to recover | calculated | pages |
| $N_{ent}$ | log entries per trans. | calculated | entries |
| $N_{appl}$ | applicable entries per trans. | calculated | entries |
| $C_{scan}$ | log entry scan cost | 100 | instructions |
| $C_{act}$ | raw action cost | 1500 | instructions |
| $M$ | processor speed | 20 | MIPS |

Table 4.7

Failure recovery encompasses a wide variety of activities. The failure must be detected, the disks spun-up (if power failed), and communications and primary memory must restored [Hagm86a]. We will consider only the restoration of an up-to-date, consistent, primary database copy in our model, since it is this part of the recovery process that is directly affected by the recovery strategies that we have discussed.

The recovery process involves both processor and I/O activity. We will develop two metrics, one for each of those resources. $t_{iorecov}$, the recovery I/O cost, is the amount of time required to read in the necessary information from the backup and log disks. $C_{recov}$, the recovery

processor cost, is the amount of processor power required to request the I/O activity and to process the data once it is brought in. We will start with $t_{iorecov}$, and then consider $C_{recov}$.

### 4.2.1 Recovery I/O Costs

Recovery I/O activity includes copying a secondary database copy into main memory and reading in enough of the log to allow the new primary copy to be brought up to date. Thus we will write

$$t_{iorecov} = t_{ioback} + t_{iolog}$$

where $t_{ioback}$ and $t_{iolog}$ are the backup database and log I/O times. $t_{ioback}$ is relatively straightforward to compute. Unless a TWIST backup strategy is used, there are (roughly) $N_{seg}$ segments of size $S_{seg}$ to be restored from the backup disks. This can be accomplished in time

$$t_{ioback} = N_{seg}(a_{back} + b_{back}S_{seg})$$

If TWIST backups are used, both copies of each segment must be copied. Assuming that each segment's copies are stored contiguously on the backup disks, this requires

$$t_{ioback} = N_{seg}(a_{back} + 2b_{back}S_{seg})$$

Determining $t_{iolog}$, the time to read in the log, is somewhat more complicated. The first question to answer is how far back in time the log must be scanned so that the new primary copy and be brought properly up to date. As we have already discussed, any log entry after the beginning of the most recently *completed* checkpoint may be applicable and must be read in from the log disks.

Assuming that a failure is equally likely at any time during a checkpoint interval (of duration $t_{icp}$), we expect the end of the most recently completed checkpoint to have occurred a time

$$\frac{t_{icp}}{2}$$

before the crash. The beginning the checkpoint was a time $t_{icp}^{min}$ before that. Thus we expect to have to scan log pages written during the period

$$t_{crash} = \frac{t_{icp}}{2} + t_{icp}^{min}$$

before the failure.

How many log pages were written during the interval $t_{crash}$? We have already calculated $D_{redo}$ and $D_{undo}$, the per transaction REDO and UNDO log bulk, for successful and unsuccessful transactions. $N_{log}$, the expected number of log pages to be read in at recovery time is given by

$$N_{log} = t_{crash}\lambda\left[(1-p_{fail})\frac{D_{redo\,(succ)}}{S_{lpg}} + (p_{fail}+N_{tries}-1)\frac{D_{redo\,(unsucc)}}{S_{lpg}}\right]$$

if transactions wrote only REDO data to disks, or

$$N_{log} = t_{crash}\lambda\left[(1-p_{fail})\frac{D_{redo\,(succ)}+D_{undo\,(succ)}}{S_{lpg}} + (p_{fail}+N_{tries}-1)\frac{D_{redo\,(unsucc)}+D_{undo\,(unsucc)}}{S_{lpg}}\right]$$

if both REDO and UNDO data went to the log disks. We can now write

$$t_{iolog} = N_{log}(a_{log} + b_{log}S_{lpg})$$

for the log I/O time.

### 4.2.2 Recovery Processor Costs

Next we would like to develop an expression for the processor cost of restoring the primary database copy. Like the I/O cost, the processor cost can also be split into costs for restoration of the primary database and replay of the log. We will write

$$C_{recov} = C_{cpuback} + C_{cpulog}$$

Restoration of the backup involves initiating an I/O for each segment that must be read in from the backup copy, and allocating space for it in primary memory. Since there are $N_{seg}$ segments in the backup, the cost for restoration of the primary copy is

$$C_{cpuback} = N_{seg}(C_{io} + C_{salloc})$$

The log replay costs can be broken down into four components:

- Each I/O request ($N_{log}$ of them) costs $C_{io}$.
- Memory management costs $C_{lalloc}$ per log page.
- Each entry in the log must be examined to determine whether or not it must be applied to the database, at a cost of $C_{scan}$ for each entry.

- If an entry must be applied, an additional cost is incurred depending on the type of data in the log. In a VALUE log, an applicable entry contains the new version of a record along with incidental updates, all of which must be copied into place in the database. This costs $C_{mv_fixed} + C_{move}S_{rec}$ to copy the record, plus

$$\frac{N_{su}}{N_{ru}}(C_{mv_fixed} + C_{move})$$

to copy the incidental updates. (Incidental update items are one word each. See Table 4.2.) Applicable operation log entries describe actions or transactions which need to be rerun against the database. The cost is $C_{act}$ or $C_{trans}$ each.

To get a complete expression for $C_{cpulog}$ we must determine the number of entries and the number of applicable entries in the log. Earlier (Tables 4.2 and 4.3) we determined the number of REDO and UNDO log entries for successful and unsuccessful transactions. Using the transaction failure and restart probabilities, which are known, we can write an expression for $N_{ent}$, the expected number of log entries per transaction (assuming UNDO log information has been logged to disk):

$$N_{ent} = (1 - p_{fail})(N_{rent\,(succ)} + N_{uent\,(succ)}) + p_{fail}(N_{rent\,(unsucc)} + N_{uent\,(unsucc)})$$

To account for restarted transactions, we add an additional

$$(N_{tries} - 1)(N_{rent\,(unsucc)} + N_{uent\,(unsucc)})$$

to $N_{ent}$. If only the REDO portion of the log is sent to disk, then

$$N_{ent} = (1.0 - p_{fail})N_{rent\,(succ)} + p_{fail}N_{rent\,(unsucc)} + (N_{tries} - 1)N_{rent\,(unsucc)}$$

There are $\lambda t_{crash}$ transactions in the part of the log that must be examined, so the total number of entries is $\lambda t_{crash} N_{ent}$.

REDO entries from unsuccessful transactions and UNDO entries (if any) from successful transactions need not be applied to the database. The commit/abort entries also need not be applied. Thus the number of *applicable* entries per transaction is given by

$$N_{appl} = (1 - p_{fail})(N_{rent\,(succ)} - 1) + p_{fail}(N_{uent\,(unsucc)} - 1) + (N_{tries} - 1)(N_{uent\,(unsucc)} - 1)$$

if UNDO entries are logged to disk, else

$$N_{appl} = (1.0 - p_{fail})(N_{rent(succ)} - 1)$$

The total in the log is $N_{appl} \lambda t_{crash}$.

The number of applicable entries must be modified slightly if a black/white checkpoint strategy is used. With black/white checkpointing, log entries that do not match the target color of the most recently completed checkpoint are not applicable at recovery time (their effects are already reflected in the database). $\lambda t_{icp}^{min}$ transactions ran while the last checkpointer was active. Assuming that half of these transactions will be "in front of" and the other half "behind" the checkpoint, we subtract

$$N_{appl} \lambda \frac{t_{icp}^{min}}{2}$$

from the total number of applicable log entries when black/white checkpoints are used.

We now have a complete expression for $C_{cpulog}$ and thus for $C_{recov}$. As a final note, we mention that if the processor speed available at recovery time is known, we can combine the pieces of $C_{recov}$ and $t_{iorecov}$ into a single metric, the recovery time $t_{recov}$, using the expression

$$t_{recov} = \max\left[t_{ioback}, \frac{C_{cpuback}}{M}\right] + \max\left[t_{iolog}, \frac{C_{cpulog}}{M}\right]$$

where $M$ is the processor speed. This assumes that restoration of the backup must be completed before the log can be replayed, and that during each phase processor and I/O operations can occur concurrently.

# CHAPTER 5
# A RECOVERY TESTBED

In this section we will consider the design and implementation of the recovery testbed. The testbed is an implementation of a transaction processing system. Its purpose is to allow us to test and compare recovery strategies, and to provide verification for the recovery model. In effect, the testbed is many transaction processing systems in a common framework. Various recovery strategies can be put together in different combinations into a working system.

Though the testbed is truly a transaction processing system, and not a recovery manager with "simulated" input from other parts of the system, some parts of the testbed are more highly developed than others. Our emphasis is on recovery, and the implementation reflects this. Many parts of the testbed directly related to recovery are implemented several ways. Other parts of the system, such as network support, access paths, ad-hoc query capabilities, etc., are present in a more primitive form or not at all. This is not to suggest that such components are unimportant to a transaction processing system. In fact, other researchers have already begun to consider some of them in the context of memory-resident databases, e.g., [Lehm86a].

The testbed is implemented on a VAX 11/785 with a 128 Mbyte main memory and running the Mach [Acce83a] operating system. It contains very little machine-specific code, and in fact its use of special facilities provided by Mach is limited as well. (We will describe the testbed's use of operating system support later.) In the remainder of the section, we will present the architecture of the testbed and discuss its implementation.

### 5.1 Process Architecture

The testbed consists of a collection of processes (called *tasks* in Mach) operating on shared data structures, including the database itself. Each process acts as a server, accepting work requests and returning results. There are four types of servers that make up the testbed.

1) The transaction server (TS) accepts transaction requests from the transaction request queue and runs the requested transactions against the primary (main-memory) database. Running a transaction has a number of effects on system data structures:

   - the primary database may be modified
   - a record of the transaction's execution is placed in the log buffers
   - a transaction response message (containing any data to be returned by the transaction) is created

   As log buffers are filled, requests are placed into the log request queue to have them flushed to disk and cleared for reuse. To ensure the durability of a transaction, its response message should not be sent until its log record is safely on disk. Therefore, rather than sending the response upon completion of the transaction, the TS attaches the response message to the log buffer containing the transaction's record. The response is sent by the log server once it has successfully flushed the page to which it is attached.

2) The log server (LS) accepts log requests and flushes the appropriate log buffers to disk, as described above. If any transaction responses are attached to the log request, they are placed in the appropriate transaction response queues after the buffer is successfully flushed.

3) Message servers (MS) place transaction requests into the transaction request queue and take responses from a transaction response queue. There can be any number of message servers. All deposit requests into the same transaction request queue, but each has its own response queue. A transaction's response is always delivered to the response queue of the message server that requested the transaction.

4) The checkpoint server (CS) is responsible for flushing modified portions of the primary database copy to the backup disks. The CS makes periodic sweeps though the database to accomplish this.

Figure 5.1 shows the servers and their shared data structures. The "critical path" of a transaction through the system is illustrated by the double lines. In the following sections we describe each of the servers in more detail.

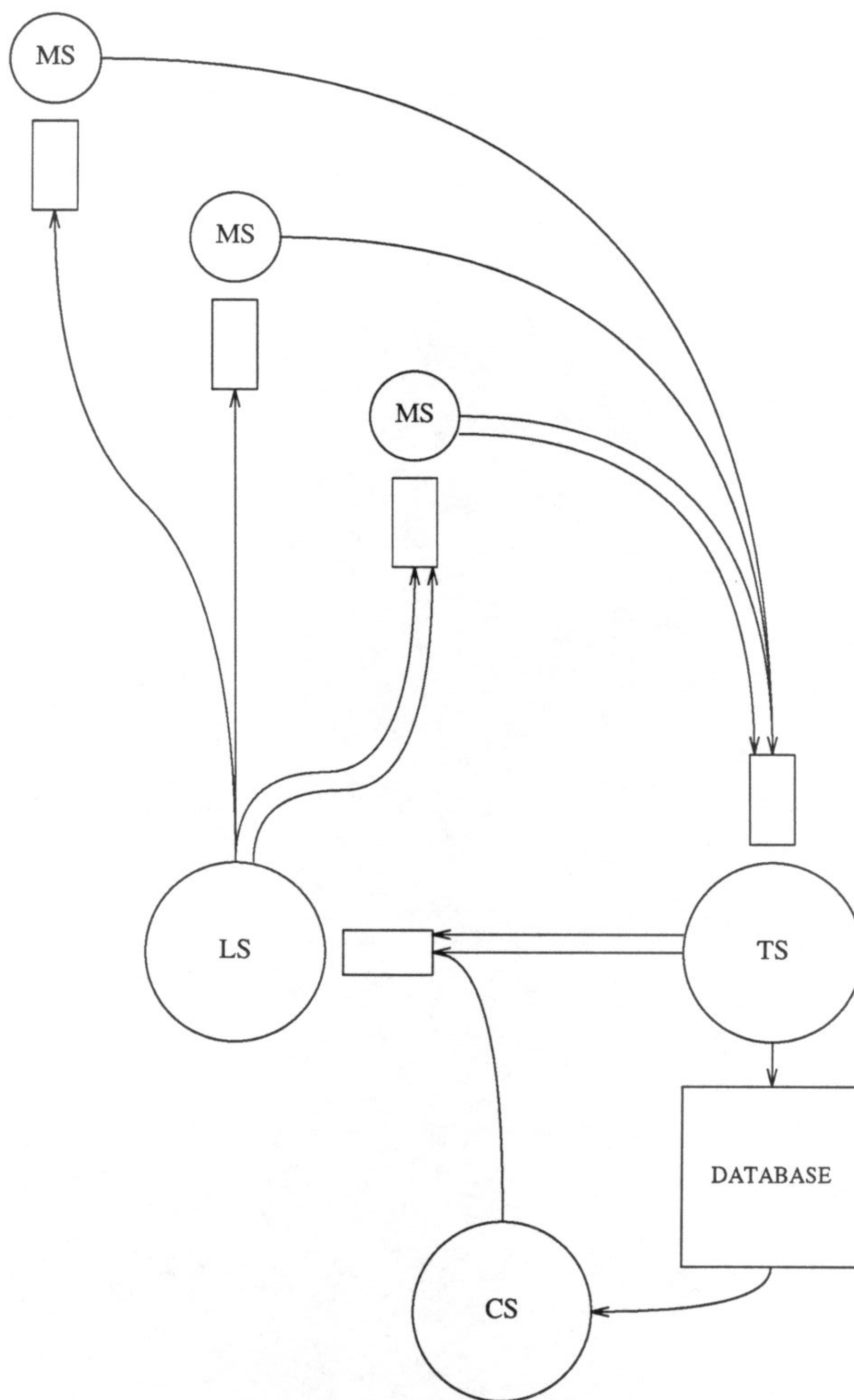



Figure 5.1 - Testbed Servers and Request Flow

### 5.2 The Log Server

We begin with the log server, as its function is the most straightforward. The system log is used to maintain a record of each transaction executed by the system, so that the effects of transactions can be reconstructed if necessary in case of a failure. The log itself consists of two parts, a memory-resident buffer and a disk-resident buffer. The memory-resident buffer holds the log tail, the most recently created part of the log. The principal job of the log server is to flush the log tail to disk, thereby freeing up space in the memory-resident buffer.

The LS and TS interact through several shared data structures: the memory-resident log buffer and the log request queue. The log buffer consists of a collection of log pages which are filled with log data by the TS. In addition, there is a queue associated with each log page. When the TS writes a transaction's log record into a buffer page (actually, when it places the *commit record* for the transaction onto a page), it places the transaction's response message on the queue for that page.

At certain times, such as when a page is full, the TS determines that the log page should be flushed to disk. To accomplish this it places a request in the log request queue, indicating which page is to be flushed. The LS takes requests from the queue and flushes the appropriate pages to disk. Normally, the I/O is done asynchronously with respect to the TS, i.e., the TS does not wait for the LS to flush the pages it has requested.

Once the I/O is complete, the LS examines the queue associated with the page that was just flushed. If there are any transaction response messages in the queue, the LS places them into the response queues of the message servers that originated the transactions. (Transaction response messages are tagged with a response queue identifier for this purpose.) Once the responses have been sent, the LS marks the buffer page as clean so that it can be reused by the TS.

The second task of the log server is the management of the disk-resident buffer. As they have been described thus far, the actions of the LS cause the disk buffer to grow monotonically as the system runs. The purpose of checkpointing is to limit the amount of log that must be replayed at recovery time. By interacting with the CS, the LS can determine which log pages are no longer needed for recovery and can free their space for reuse.

Interaction between the CS and the LS takes two forms. The CS maintains a record of its activity by placing entries in the log request queue. The CS makes a log entry at the beginning and end of each checkpoint sweep. The LS uses this information to determine which log pages are no longer needed. Typically, once a checkpoint completes, the LS can discard log pages created before that checkpoint began and can reclaim their space. However, this is somewhat dependent on the checkpointing algorithm used by the checkpointer. The LS maintains the disk-resident portion of the log as a circular buffer, with new pages added in response to requests from the TS, and old pages discarded as checkpointing progresses.

It is also possible for the CS to request "clearance" for a log page from the LS. The LS provides clearance if the log page in question has been successfully flushed to the log disks. If this is not the case, clearance is delayed (as is the checkpointer) until the log page is disk-resident. Depending on the checkpoint strategy, the CS makes use of the clearance facility to ensure that the log write-ahead protocol is not violated when a dirty segment is propagated to the backup database.

## 5.3 Message Servers

Message servers are responsible for feeding transaction requests to the TS and for handling the response messages from those transactions. There can be any number of message servers, each with its own response queue. A transaction's output message is routed (by the LS) into the response queue of the MS that requested it. The TS has a single transaction request queue into which all message servers deposit their requests.

Several different types of message servers have been implemented. They differ in how they produce transaction requests. Interactive message servers (IMS) prompt a user for transaction requests. Generative message servers (GMS) use random number generators to create requests. Generative servers are used to drive the recovery performance experiments that we will describe in Chapter 6. Both types of servers display and then discard response messages. Normally, all responses are displayed on the same output device (the terminal from which the testbed was initiated). If the X windows system is available, a simple facility is implemented which permits each message server to display its response messages in its own window. The windowing

facility, which can also display messages from the other servers (i.e., TS, CS, and LS) is useful primarily as a debugging tool.

Each interactive server operates synchronously. An IMS does not prompt for a new request until a response has been received for the current one. Generative servers operate asynchronously, continuously producing new requests and serving responses as they become available.

Other types of message servers are possible but have not been implemented. For example, a file message server could read pre-computed transaction requests from a file, or a network message server could accept requests generated (somehow) at remote machines. By employing several message servers, the TS can be fed requests from multiple sources.

### 5.4 Checkpoint Server

The checkpoint server is responsible for migrating changes in the primary database copy to the backup, which resides on disk. The CS operates by periodically sweeping through the database and copying dirty data to the disks. The minimum elapsed time per database sweep, called the checkpoint interval floor, is a parameter supplied to the CS. Should the actual sweep take less time than the interval floor the CS pauses for the remainder of the interval before starting a new sweep.

The exact method used to update the backup, and the method of synchronizing copying with database updates made by the TS, are determined by the checkpoint and backup algorithms selected for the CS. However, all of the algorithms have a few points in common. The CS transfers data to (and from) the backup in fixed-size blocks called *segments*. The checkpointer's access to database segments is read-only, except that some checkpointing algorithms require that the CS toggle a bit in the segment to indicate that it has been visited during the current sweep. Most of the checkpointing algorithms require that the CS lock segments before accessing them. These locks can conflict with database accesses by transactions running in the TS.

The checkpoint server keeps the LS informed of its progress by making entries in the log request queue. The CS enters a begin-checkpoint marker in the queue before the each sweep begins, and an end-checkpoint marker as soon as the sweep is completed. As we have already described, these markers are used by the LS to determine which log pages are no longer needed

for recovery.

Finally, some checkpointing algorithms require that database update activity be temporarily quiesced before the CS begins each database sweep. The CS accomplishes this by raising a flag which is monitored by the TS. The TS cooperates by quiescing its update activity, and raises a flag of its own once it has achieved a quiescent state. At this point the CS normally enters its begin-checkpoint marker into the log request queue and lowers its flag to indicate to the TS that update activity can proceed.

### 5.5 Transaction Server

The transaction server task performs the "useful work" of the transaction processing system. It executes transactions against the primary database in response to requests taken from the transaction request queue. In effect, the purpose of the rest of the system is to keep the TS as busy as possible while ensuring that transaction updates will not be lost in the event of a failure.

Though the TS is a single Mach task, it actually consists of a number of transaction servers, each of which is capable of serial transaction execution. We will term each sub-server a *transaction executor*, or TE. Transaction executors are implemented as coroutines within the TS task.

The purpose of having multiple TEs is so that the TS can continue to process transactions even if the currently executing transaction is forced to wait on a lock[†]. As long as an unblocked TE is available, the TS can continue to do useful work. However, unnecessary concurrency can hurt performance. Switching from TE to TE, though fast, consumes processor resources. As the number of active TEs increases, the likelihood of lock conflicts (which leads to more context switching) increases as well.

For this reason, the number of active TEs should be kept as small as possible while still ensuring that the TS is kept busy. To accomplish this, TEs are created in two flavors, *priority* and *standby*. Standby TEs are normally inactive. In case all active TE's are blocked, a single standby TE is activated to process a transaction. The standby TE is deactivated once its transaction has completed.

† Concurrency is also useful for promoting fair sharing of resources when there are long-lived transactions.

Normally, the TS consists of a single priority TE and one or more standby TEs. Transactions are processed serially by the priority TE until a lock (in most cases, a lock set by the CS) is encountered, at which point a standby is activated. Thus the TS makes use of low-overhead serial transaction execution as much of the time as possible.

Once a TE has a transaction request to process, it simply calls the application-defined transaction specified in the request. (Transactions are pre-defined, compiled, linked, and loaded together with the testbed.) A transaction terminates voluntarily by making a commit or abort request, or involuntarily if a conflict (e.g., a deadlock) arises during its execution. The TE ensures that involuntarily aborted transactions will be automatically restarted (though not necessarily by the same TE) by placing the unfulfilled transaction request in a restart queue.

## 5.6 Implementation

Thus far we have discussed the behavior of the testbed. In this section we will describe its implementation. The testbed is structured as a collection of libraries, many of which are used by two or more or the testbed servers. At the lowest level, libraries provide for server synchronization and basic data structures. Libraries also exist to manage log and backup disk storage, to perform transaction management, and to implement a simple set- and record-based data model. The application code (i.e., the transactions) is also a library. The application library is compiled together with the testbed to produce a working system. Figure 5.2 shows the various libraries and their relationships (an arrow from *A* to *B* indicates that *A* uses facilities provided by *B*). In the remainder of the section we will provide an overview of the major major parts of the testbed.

## 5.7 Transaction Management

The transaction management library provides facilities for beginning, committing, aborting, and ending transactions. Transaction commit and abort commands are available for voluntary use by application transactions, i.e., when a transaction finishes its application-defined tasks, it requests that its work be committed. Transactions may also be involuntarily aborted by the actions of other parts of the system (e.g., a deadlock will result in an aborted transaction).

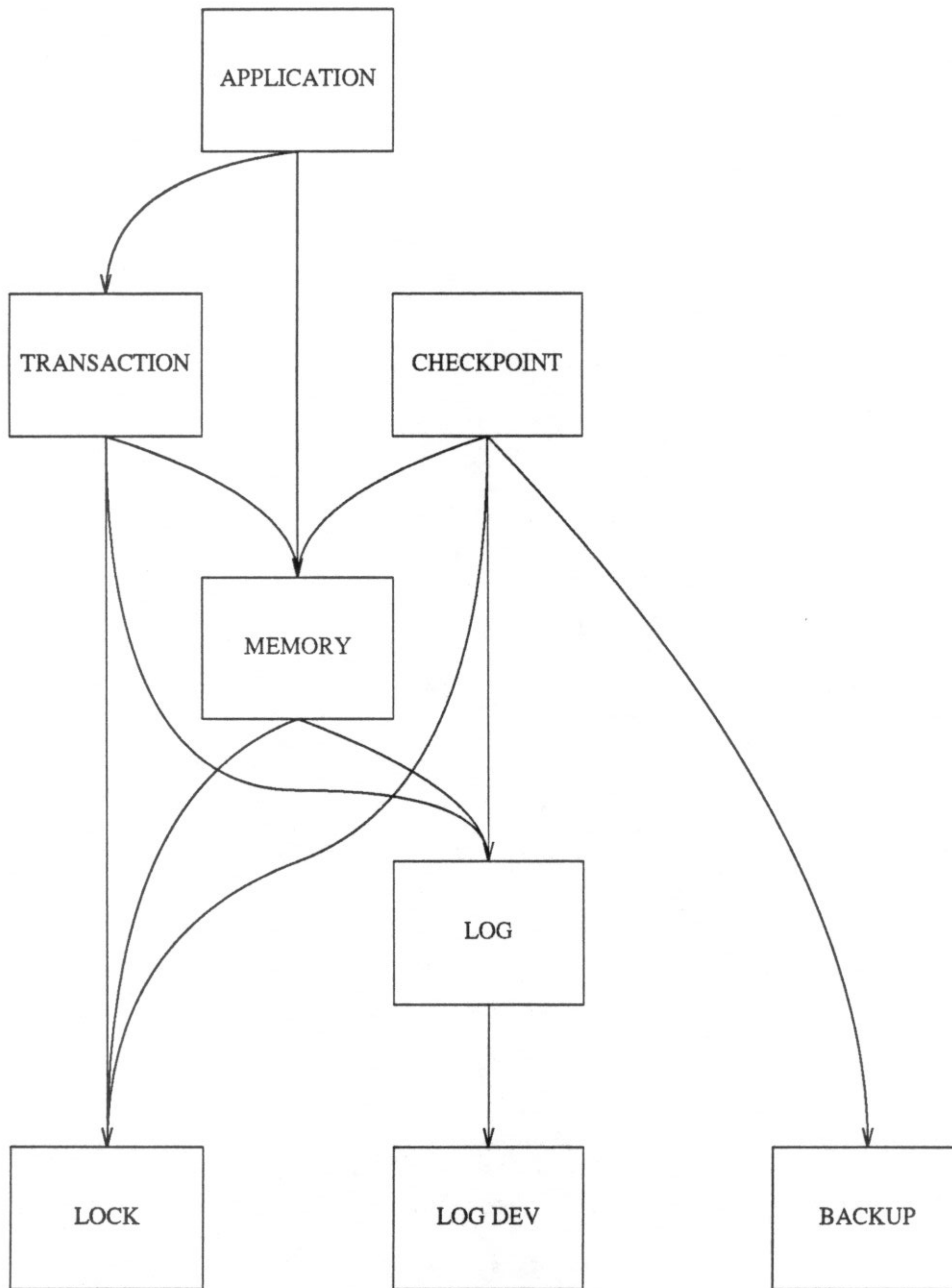



Figure 5.2 - Testbed Libraries and Call Graph

The duties of the transaction manager include:

- maintenance of data structures representing the each transaction current state, the location of its request and response buffers, etc.
- making abort and commit notations in the log through calls to the log manager.
- undoing, through calls to the memory manager, the effects of partially completed transactions should they abort
- discarding (also through calls to memory management) old versions of updated data no longer needed when a transaction commits

### 5.8 Memory Management

The memory manager implements a simple data model consisting of records and sets. A record is a collection of fields, which may be of fixed or variable length. Each record has a type, which determines the number of fields in the record and their lengths. The possible record types are determined by an application-specified record catalogue.

A set is a collection of records, each of which has a unique identifier. All of the records in a set are of the same type. The sets, and their respective record types, are determined by an application-specified set catalogue.

The memory manager supports retrieval, update, insert, and deletion of records. It also supports retrievals and updates of individual record fields. (Only fixed-length fields can be updated individually.) The record and field access commands are used by the application-defined transactions to manipulate the database.

Database access uses copy semantics, i.e., the caller receives a copy of the requested data, rather than a pointer into the database itself. An alternative would be pass a pointer to the application code. While this would save the expense of copying the requested data, it requires safeguards to ensure that the application does not disturb data other than that it was granted access to. One possibility is to build safeguards into the language in which the application is coded. These alternatives are discussed further in [Garc87a].

Figure 5.3 shows the memory organization maintained by the memory manager. A set's records are stored in a collection of fixed-size memory blocks called segments. (As we have already described, segments are the units of transfer from the primary database copy to the backup.) A set is represented by a table of pointers to the set's segments plus a record table containing a direct pointer to each of the set's records. The set's record and segment tables are not part of the database, i.e., modifications to the tables are not logged or migrated to the backup. After a failure, the tables are regenerated from scratch once the set's segments have been restored from the backup. Since record tables are regenerated after a failure, they can contain absolute (rather than segment-relative) record addresses despite the fact that segments are relocatable after a failure.

Each segment consists of a header plus free space for holding records. Each record is stored in a contiguous region of the segment. These regions may vary in size since records can contain variable-length fields. All records in a segment are connected by a singly-linked list anchored in the segment's header. The list is used to distinguish records from free space when the segment is restored after a failure.

Currently, record updates are done using *shadows* to eliminate the need for UNDO logging. (The in-place update strategy has not yet been implemented.) A record that is to be updated is not overwritten. Instead, space is allocated for a new copy. The old copy (the shadow) is unlinked from the record list in its segment, effectively removing it from the database. However, its space is not freed, and a pointer to it is saved in a *shadow table* maintained for the set. The shadow is not freed until the updating transaction commits. In case of an abort, the shadow is relinked into the record list, and its record table entry is restored from the shadow table.

The memory manager attempts to keep the record and its shadow in the same segment, but this is not a requirement. Segments are kept partially empty so that there will usually be room for both the shadow and the new version when an update occurs. The fraction of empty space that the memory manager attempts to preserve on each segment for this purpose is a parameter to the memory manager.

The memory manager maintains a variety of information in segment headers that is useful to other components of the system. Each segment has a color bit and a pair of dirty bits which

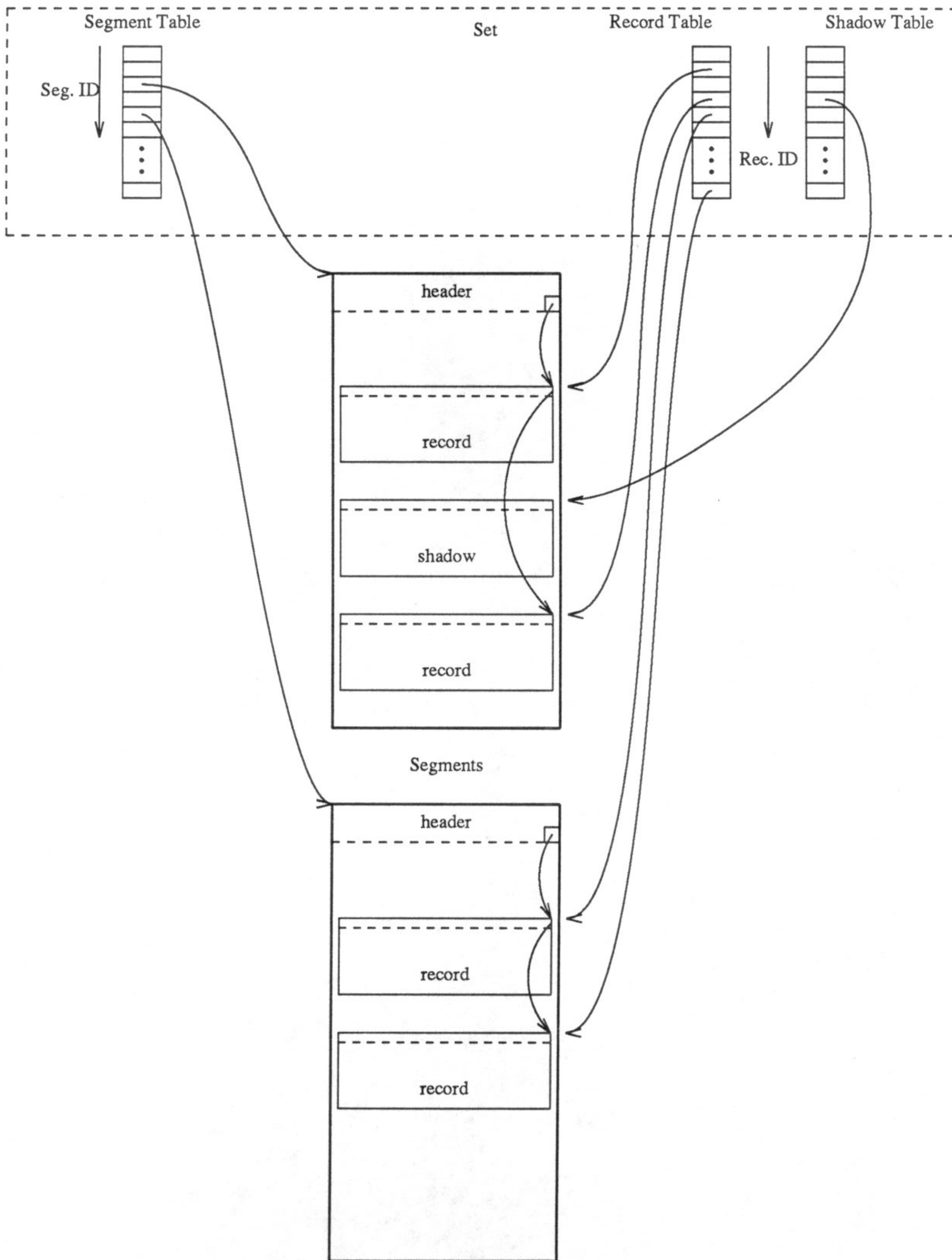



Figure 5.3 - Sets and Segments

are used for checkpointing algorithms. Each segment also has a log-entry number indicating the most recent log entry affecting the segment.

In addition to its own duties, the memory manager requests related activities from other parts of the system. The memory manager automatically requests database locks on behalf of the calling transaction. In the case of database modifications, it also arranges for a log record of the update to be generated (if the locking strategy requires it).

## 5.9 Log Management

The log management library is responsible for maintaining the log tail, the memory-resident portion of the log. The log tail is maintained as a circular buffer of log pages. Each page consists of a data region and a header which includes a queue for the response messages of transactions whose log entries are on the page.

Log users, e.g., the memory and transaction managers, interact with the log manager using the open and close commands. The open command allocates space of a given size on the current log page. If there is insufficient room on the current page, a request is placed to flush the page to disk, and a new page is opened. Once the log has been opened, the log user can write any data it desires into the allocated space as long as the total volume of data written is no more than the amount requested. Only a single user can have the log opened at any one time.

The log manager also needs to be informed of the beginnings and ends of transactions and checkpoints, and log management commands exist for each of these purposes. The transaction-ending command is accompanied by the response message of the terminating transaction, which is stored on the response queue of the current log page. Checkpoint commands cause checkpoint information to be passed through the log manager to the log device manager.

## 5.10 Log Device Management

The log device manager maintains the disk-resident portion of the log. Its principal task is to move log pages to disk in response to requests from the log manager. The log device manager is also capable of retrieving log pages from the disks in reverse or forward sequential order. In addition, each log page header contains an ID field (supplied by the log manager) and the device

manager can retrieve a page given its ID number.

The log disk is maintained as a circular buffer. The space occupied by old pages is freed in response to a *firewall* command. The firewall command is accompanied by a log page ID. It causes the log manager to discard from the log disk all pages with lower IDs. Firewall commands are normally issued by the log manager at the completion of a database checkpoint.

Log "devices" can either be files or Mach/Unix raw devices. Raw devices are preferable, as they permit I/O directly from the log buffer to disk. File system I/O requires the log pages be first copied to the file system's buffer cache before being written to disk. In addition, the device manager must issue an additional system call to ensure that the log page is written immediately from the buffer cache to disk. The file system provides no functionality useful to the device manager in return for these costs.

### 5.11 Checkpoint Management

The checkpoint management library implements the various checkpointing algorithms by making calls to the memory, log, and backup device managers. All of the checkpoint algorithms involve sweeping through the database and processing segments. This is accomplished set (see Section 5.8) by set, i.e., all of the segments of one set are processed before moving on to the next. Processing a segment means different things depending on which checkpoint algorithm is begin used. The various checkpoint algorithms are described in section two.

A secondary function of the checkpointer is raising the *checkpoint progress flag*. This flag is used by the transaction executors to determine when to attempt to retry transactions that have been aborted as a result of interactions with the checkpointer. (Only the black/white checkpointing algorithms cause transactions to be aborted.) The flag is raised each time the checkpointer finishes processing a certain number of segments; the exact number is a checkpoint parameter. (The flag is lowered by the TEs.)

## 5.12 Backup Management

The backup manager maintains the disk space used to store the secondary copy (or copies) of the database. Each database segment has a unique backup ID. Backup IDs are managed by the backup manager. Each time the memory manager creates a new segment, the backup manager is requested to issue it an ID.

The task of the backup manager is the migration of segments to (and from) the backup storage according to the particular backup algorithm that is being used. (Part of the specification of a backup algorithm is a mapping from backup IDs to disk locations.) Backup algorithms are described in Chapter three. Currently, only the fixed monoplex and pingpong (duplex) algorithms are implemented in the testbed.

The backup database includes a special segment that is used to store information pertaining to the backup. The contents of the special segment depend of the type of checkpointing that is used to create the backup. The special segment is updated in response to begin- and end-checkpoint commands issued to the backup manager. Typically, the special segment includes such data as the ID of the log page that was current when the checkpoint began and an identifier for the checkpoint.

As with log devices, the backup device can either be a file or a raw device. If available, the raw devices are desirable from a performance standpoint.

## 5.13 Lock Management

The lock manager provides locking capability for database segments and records. Locking is used by the checkpoint server (except with fuzzy checkpoints) and by the transaction executors via the memory manager. The lock manager provides shared and exclusive lock modes, deadlock detection, and lock escalation (shared to exclusive). Currently, lock acquisition time is reduced by pre-allocating lock data structures (i.e., "lock table" entries) for lockable objects.

Normally, a lock requester is blocked and queued in case of a conflict. However, locks can be requested in non-blocking mode. This causes the lock manager to return immediately in case of a conflict, with a status code indicating that the lock was not achieved. Blocking requests are used by both the TS and the CS. Non-blocking lock requests are used by the CS only..

### 5.14 Low-Level Support

The testbed makes use of a modified version of the CThreads [Coopa] coroutine library to implement TEs within the transaction server. The CThreads library provides a support for non-preemptively scheduled coroutines within a Mach task. The library has been modified in several ways for use with the testbed:

- New primitives have been added to allow synchronization of coroutines with Mach tasks. For example, if a TE (coroutine) attempts to lock a segment locked by the checkpoint server (a task), only the locking TE is blocked, and not the whole TS task. Similarly, if a CS lock request conflicts with an existing lock, the CS task may block.
- A two-level scheduling strategy is used to implement priority and standby coroutines.
- New primitives have been added that permit, in conjunction with the X windowing system, each coroutine (or task) to open its own window on the display. This is very useful in debugging a multi-threaded system.

The testbed makes use of Mach's virtual memory primitives to implement its shared data structures, namely the database and various request queues. The Mach kernel schedules (preemptively) the server processes and provides block and restart primitives.

Currently, disk storage for the log and the secondary database copy is accessed though Mach's raw device interface. Normal files can be used as well, though there is a performance penalty associated with doing so.

Finally, Mach system calls are used to gather performance (timing) information on the testbed's servers.

### 5.15 Application Libraries

In order to perform useful work, the testbed must be combined with an application library. An application library consists of several parts:

- Database Catalogue: The catalogue is a description of the records and sets that will make up the database. Record descriptions include the name and length (which may be variable) of each field in the record. A type is specified for each field as well, but this is used only in displaying records. Set descriptions include a name and the record type of the sets

members.

- Transactions: The application can include any number of transactions. Transactions can perform arbitrary computations on local data and can request database access (read, update, insert, delete) and transaction (abort, commit) services from the testbed. Transactions are supplied with input and output buffers when they are called. The input buffers contain any parameter values required by the transaction. Output buffers can be used to hold the transactions' return status or any other results of the computation
- Message Catalogue: The message catalogue is much like the database catalogue, but it describes transaction input and output message buffers rather than database records. Message descriptions are very much like record descriptions. Each specifies the number of fields in the buffer and their lengths. Each transaction can have its own input and output buffer descriptions, or the same descriptions can be used by several different transactions.

The application library (transactions and catalogues) is precompiled, linked, and loaded with the remainder of the testbed to produce a working system.

# CHAPTER 6
# MODEL VERIFICATION

Thus far we have described a performance model and a testbed implementation for MMDB recovery strategies. It is difficult to investigate the full spectrum of recovery strategies and changes in the environment (e.g., transaction load) with the testbed because of limitations of the hardware and the amount of time required for such studies. However, varying recovery strategies and the environment is a relatively simple matter with the performance model. For this reason, we will use the testbed to verify the performance model, and then use the model to make most of our comparisons. We will verify the model by comparing the measured performance of the testbed to the models predictions over a variety of recovery strategies. The purpose of these comparisons is to increase confidence in the accuracy of the model's predictions. The verification experiments are also useful and interesting in their own right.

All of the verification experiments described in this chapter were performed in a similar environment:

- The testbed ran on a VAX-11/785 equipped with 128 Mbytes of memory and running Release 1 of the Mach operating system.
- The disk-resident portion of the log and the backup database were kept (1 partition each) on two RA81 disk drives.
- The log and backup partitions were accessed using the "raw" I/O facilities provided by Mach, i.e., the file system is bypassed.

In the remainder of this chapter, we will discuss the model verification. It will be presented as a three-step process:

1) Determination of the parameter values to be used in the model.
2) Measurement of recovery overhead in the testbed.

3) Comparison of the measured values to those predicted by the model.

In the rest of this section we will describe each of the steps in more detail.

## 6.1 Parameter Determination

To verify the model against the testbed, it is important to use model parameter values which accurately reflect the characteristics of the testbed. The parameter values we used were derived from several sources:

- Some model parameters are also testbed parameters or can be obtained from a static analysis of the testbed or application code. Parameters such as segment, log page, and record sizes fall into this category.
- Other parameters describe dynamic attributes of the testbed. The testbed has a variety of built in instrumentation that allows some of these parameters to be measured while the testbed runs. Parameters describing I/O throughput and the number of segments accessed per action are example of parameters in this group.
- The final group of parameters consists mostly of the primitive operation costs. They are difficult to measure in the running testbed and do not lend themselves to accurate static analysis. Instead, they are measured with special test programs.

### 6.1.1 Static Parameters

Parameters such as the segment and log page sizes are compiled into the testbed and are independent of the application code it is supporting. The values of these parameters in the testbed that was used for the verification experiments are shown in Table 6.1. In the table, the record-per-action count is one because the three update actions provided by the testbed (update, insert, delete) modify a single record each. The log entry overhead is the size of the most common log entry header used by the log manager. Other log entry headers are slightly smaller.

The remaining static parameters describe application-dependent features such as record sizes. The application used to drive the testbed simulates a credit card data processing environment. It is based loosely on an application used to drive a recent set of benchmarks of IMS/VS FastPath [Vigu87a].

| description | model name | value | units |
|---|---|---|---|
| segment size | $S_{seg}$ | 1024 | words |
| log page size | $S_{lpg}$ | 1024 | words |
| log entry overhead | $S_{lent}$ | 6 | words |
| records per action | $R_{rpa}$ | 1 | record |

Table 6.1

An application defines the various collections (or *sets*) of records in the database and transactions to access those records. The credit card application defines four sets and eight different transactions. The sets include:

- *Account Set*: one record per account. Record fields include account number, credit limit, used credit, expiration date. 40,000 accounts; expected record size is 13[†] words.
- *Customer Set*: one record per customer. Records hold customer information such as name, address, social security number, and account number; 40,000 customers, expected record size is 50 words.
- *Hot Card Set*: one record for each stolen card reported. Records hold the account number, number of attempted uses of the stolen card, and the report date. Initially, 100 cards reported. Expected record size is 20 words.
- *Store Set*: one record for each point of sale (retail store). Record fields include store name and number and counters for various types of credit card activity at the store; 5000 stores, expected record size is 20 words.

Eight types of transactions are also defined:

- *BAL (Balance Check)*: Returns information about an account, including customer name and current balance.
- *CCCK (Credit Card Check)*: Checks the validity of an account and increments a counter at the store from which the transaction originated.

---

† This and all record sizes include four words of system header information.

- *CLCK (Credit Limit Check)*: Checks that there is sufficient credit available for a purchase. Also increments a counter in the originating store's record.
- *CHCUST (Change Customer)*: Modifies customer description, e.g., change of address.
- *DEBIT (Account Debit)*: Modify account information to reflect a purchase. Also increments a counter in the seller's record.
- *FOUND (Cancel Lost Card)*: Re-validate a card that had been reported missing.
- *LOST (Report Lost Card)*: Invalidate a card that is reported missing.
- *PAY (Make Payment)*: Make a payment on an account.

Table 6.2 gives the application profile, including transaction frequencies and the size of a transaction's input parameters. It also includes a call profile; a description of each transaction's access to the data sets. An R stands for read access, M for record modification, I for record insert, and D for record deletion.

| | | | Set Name | | | |
|---|---|---|---|---|---|---|
| type | input size | % of requests | Account | Customer | Hot Card | Store |
| BAL | 1 | 17 | R | R | | |
| CCCK | 2 | 20 | R | | R | RM |
| CLCK | 3 | 20 | R | | | RM |
| CHCUST | 44 | 1 | | RM | | |
| DEBIT | 18 | 20 | RM | | | RM |
| FOUND | 1 | 1 | R | | RD | |
| LOST | 16 | 1 | R | | RI | |
| PAY | 4 | 20 | RM | | | |

Table 6.2

We generate values for $R_{up}$, $S_{op}$, $S_{rec}$, and $N_{act}$ from the application profile. Recall that $N_{act}$ is a function that describes the number of actions per transaction. $N_{act}(i)$ gives the probability that a transaction will have $i$ actions. We determine the $N_{act}(i)$ using the actions listed in table 6.2 (Read, Modify, Insert, and Delete are actions) weighted by the frequency of occurrence of the different types of transactions. The result is shown in Figure 6.1

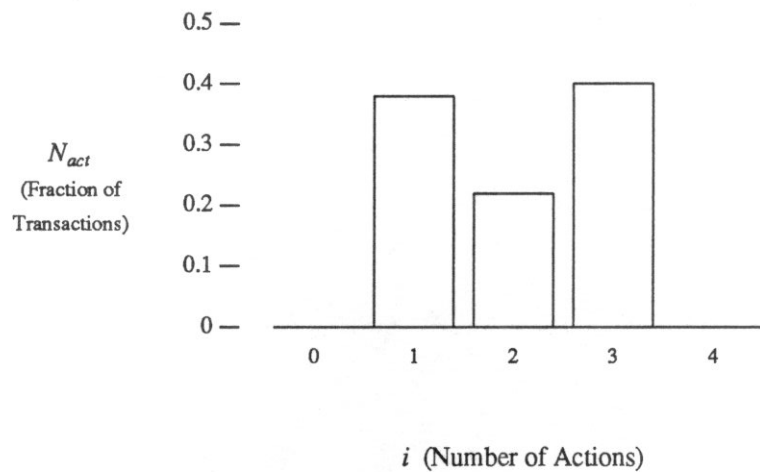



Figure 6.1 - Distribution of Number of Actions per Transaction

The update fraction, $R_{up}$, is the frequency-weighted fraction of the actions that are updates (anything but Reads). The operation log entry size, $S_{op}$, is taken to be the weighted average of the transaction input sizes (since the operation log entry must contain the information necessary to re-run the operation). Finally, the record size, $S_{rec}$, is taken to be the average record size of the four sets, weighted to reflect frequency of access. The actual values used are summarized in Table 6.3.

| parameter | description | value |
|---|---|---|
| $R_{up}$ | update fraction | 0.34 |
| $S_{op}$ | operation log entry size | 8 |
| $S_{rec}$ | record size | 18 |

Table 6.3

### 6.1.2 Dynamic Parameters

A number of parameters are measured by the running testbed. These parameters and their values are summarized in Table 6.4.

| parameter | description | value |
|---|---|---|
| $S_{db}$ | database size[1] | 4000000 |
| $a_{back}$ | throughput constant[2] | 0.1 |
| $b_{back}$ | throughput constant[2] | 0.0 |
| $\lambda$ | transaction input rate[3] | various |
| $R_{spa}$ | segments per action | 1.05 |
| $p_{fail}$ | transaction failure rate[4] | 0.0 |

1) The database size varies with time because of updates to variable-length fields, the shadow update strategy, and record insertions and deletions. The testbed reports the mean number of segments after each checkpoint. The value used is approximately that number times the segment size.

2) Throughput to the backup disks is measured by the testbed at close to ten segments per second. Recall that, in the model, segment throughput is determined by the parameters $a_{back}$ and $b_{back}$ using the expression

$$\frac{1}{a_{back} + b_{back}S_{seg}}$$

We have not determined the variation in throughput with segment size, so for verification purposes the linear coefficient ($b_{back}$) is zeroed and the constant term is set at 100 milliseconds to achieve ten segment per second throughput.

3) Input (throughput) rates vary from under thirty to over seventy transactions per second over the various test points, since each point represents a different combination of recovery strategies.

4) Transactions in the testbed's workload fail occasionally because of bad input parameters (e.g., the same card is reported FOUND twice). The failure rate is normally less than a tenth of one percent, so $p_{fail}$ is taken to be zero.

Table 6.4

### 6.1.3 Other Parameters

The remaining parameters, all primitive operation costs, cannot be easily measured within the testbed. They were measured instead with specialized test programs. Each test program consists of a loop which repeatedly calls the portion of the testbed responsible for implementing the primitive action. For example, to measure $C_{lock}$ the test program repeatedly acquires and releases a database lock. The results of these measurements are summarized in Table 6.5. All measured values are averages over 500,000 iterations (primitive operation executions). Times are given in microseconds.

Note that in the performance model, processor costs are measured in "instructions". An instruction can be thought of as an arbitrary unit of consumption of processor resources. For the purposes of verification, we take an instruction to be one µsecond (i.e., a 1 MIP processor). Of

course, this means that the absolute magnitude of the overhead predicted by the model is meaningless unless this scaling factor (the number of instructions per second) is known.† However, relative measurements (comparisons of overhead) are the same regardless of the scaling factor.

| parameter | description | measured value | model value |
|---|---|---|---|
| $C_{salloc}$ | segment allocation[1] | 267 | 125 |
| $C_{lalloc}$ | log buffer allocation | 1053 | 1000 |
| $C_{lsn}$ | log sequence number[2] | 81-148 | 100 |
| $C_{lock}$ | lock request[3] | 285 | 300 |
| $C_{io}$ | I/O request | 5207 | 5000 |
| $C_{mvfixed}$ | movement constant cost | ~60 | 60 |
| $C_{move}$ | movement linear coef. | ~1.5 | 1.5 |
| $C_{logopen}$ | log entry cost | 222 | 200 |
| $C_{logend}$ | log per-trans. cost | 283 | 300 |

1) The measured value includes segment allocation and deallocation. Since these costs are charged separately by the model (allocation is part of the synchronous cost, deallocation of the asynchronous cost), we take $C_{salloc}$ to be half of the measured value.

2) Log synchronization costs varied depending on the state of the log page that is being waited on. The lower number is achieved when the page is no longer in the main-memory log buffer. If the page has been flushed to disk but also remains in the buffer, the cost is higher. We chose a synchronization cost in between these values.

3) The lock request cost was measured for the most common case in which there are no other locks on the requested object.

Table 6.5

The data movement costs ($C_{move}$ and $C_{mvfixed}$) require some additional explanation. Data movement costs were measured for a variety of block sizes. The results are shown by the solid curve in Figure 6.2. In the figure, the data copy time less 60 μsec is plotted against the block size. Based on this curve, we chose $C_{mvfixed} = 60$ and $C_{move} = 1.5$. (Recall the model takes the cost of moving $w$ words of data to be $C_{mvfixed} + wC_{move}$.) The dotted line shows the predicted value (less 60 μsec).

---

† Actually, the scaling factor is used explicitly in the model. However, it is used only to determine whether the recovery processor overhead is sufficiently great to make the recovery time processor-bound (see Chapter 4). We will only consider the transaction processor overhead in the verification. The model parameter representing the scaling factor is $M$, the processor speed.

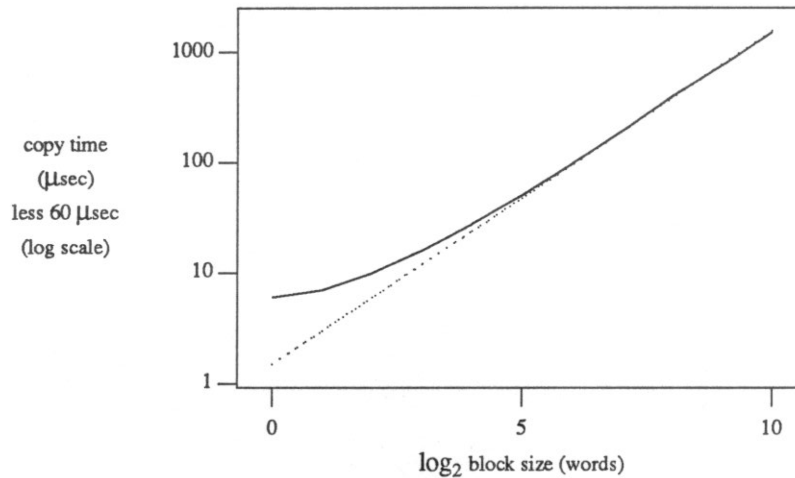



Figure 6.2

We have yet to determine a value for $C_{trans}$, the raw (without recovery) cost of running a transaction. This cost is measured by running the testbed with all recovery facilities turned off (i.e., no logging or checkpointing). This measurement was made using the same transaction mix as was used in the verification experiments.

The testbed reports total processor time (broken down into system and user time) for each of its four principal processes. In addition, the transaction server process reports the total transaction count. Table 6.6 shows the results of a thirty minute run with no recovery operations. (All times are reported in seconds.)

| server | user-mode time | system-mode time | total time | transaction count |
|---|---|---|---|---|
| transaction | 1269.41 | 34.45 | 1303.86 | 158781 |
| message | 339.08 | 121.14 | 460.22 | NA |
| log | 0.00 | 0.00 | 0.00 | NA |
| checkpoint | 0.00 | 0.00 | 0.00 | NA |

1) The message server time includes additional overhead for determining transaction response time (i.e., timestamping transaction request and response messages). As a result, the message server time is not as low as it might otherwise be. However, since message and network costs are normally high in transaction processing systems, we have chosen not to attempt to reduce further the message server overhead.

Table 6.6

The raw transaction cost is taken to be the transaction server time divided by the total transaction count. The resulting value for $C_{trans}$ is approximately 8200 μseconds.

## 6.2 Recovery Overhead

We are now in a position to present the model verification experiments. The testbed's performance was measured in a dozen experiments. Each measured the performance of a different combination of checkpointing and logging strategies.

The testbed configurations had a number of features in common across all of the experiments:

- non-spooling checkpointers
- asynchronous logging
- transaction group commit and pre-commit
- immediate, shadowed primary database updates
- a ping-pong backup strategy
- checkpoints run as quickly is possible, i.e., no delay between completion of one checkpoint and initiation of the next.

Each experiment measured the total processor time of each of the servers, the transaction throughput, and the total true transaction count. Throughput is measured in true transactions per second over 60 second (real time) intervals by counting the number of true transactions completed during the interval and dividing by 60. Each experiment covered 30 minutes (intervals). The true transaction count does not count restarted transactions as new transactions.

Server times are given as μseconds of processor time per true transaction. Server times are computed by dividing the total processor time used by the server (over the 30 minute experiment) by the total true transaction count. Processor time was measured using operating system process task timing facilities (via calls to the system routine "getrusage"). The results of these experiments are presented in Table 6.7.

| Experiment | | | transaction | checkpoint | log | message | transaction |
|---|---|---|---|---|---|---|---|
| checkpoint | log data | throughput | server | server | server | server | count |
| FUZZY | VALUE | 72.7 | 9658 | 275 | 802 | 2908 | 130877 |
| SC | VALUE | 71.2 | 9723 | 460 | 827 | 2935 | 128247 |
| ACCOU | VALUE | 65.7 | 10140 | 1335 | 744 | 2899 | 118338 |
| ACBW | VALUE | 71.5 | 9580 | 586 | 739 | 2981 | 128665 |
| TCCOU | VALUE | 63.8 | 10613 | 1315 | 740 | 2891 | 114863 |
| TCBW | VALUE | 41.61 | 18687 | 925 | 1128 | 3075 | 74885 |
| ACCOU | AOPER | 66.4 | 10295 | 1230 | 511 | 2928 | 119474 |
| ACBW | AOPER | 73.1 | 9619 | 493 | 553 | 2886 | 131557 |
| TCCOU | AOPER | 66.6 | 10186 | 1250 | 525 | 2907 | 119967 |
| TCBW | AOPER | 42.38 | 18591 | 836 | 700 | 3251 | 76298 |
| TCCOU | TOPER | 72.6 | 9103 | 1147 | 482 | 2941 | 130731 |
| TCBW | TOPER | 46.31 | 16990 | 786 | 566 | 3052 | 83382 |

Table 6.7

## 6.3 Verification

The total transaction overhead was computed by summing the times for the transaction, log, and checkpoint servers (Table 6.7) and subtracting the raw transaction cost, $C_{trans}$. Total overhead was also computed by the model for the same strategy combinations, and using the measured throughput for each combination as the value of $\lambda$ (see Table 6.7). Figure 6.3 shows the computed and predicted overhead costs for each of the experiments.

Although there are some minor disagreements over the relative ordering of strategies with very similar overhead costs, the model generally predicts lower overheads for those recovery strategies that showed the lowest costs experimentally. However, the figure shows two areas in which the model had greater difficulty in conforming to the measured values:

- Measured overhead costs for the four copy-on-update checkpoint scenarios showed a greater increase over faster strategies than did the predicted costs.
- The model significantly underestimates the overhead in the three experiments using TCBW (transaction-consistent, black/white) checkpointing.

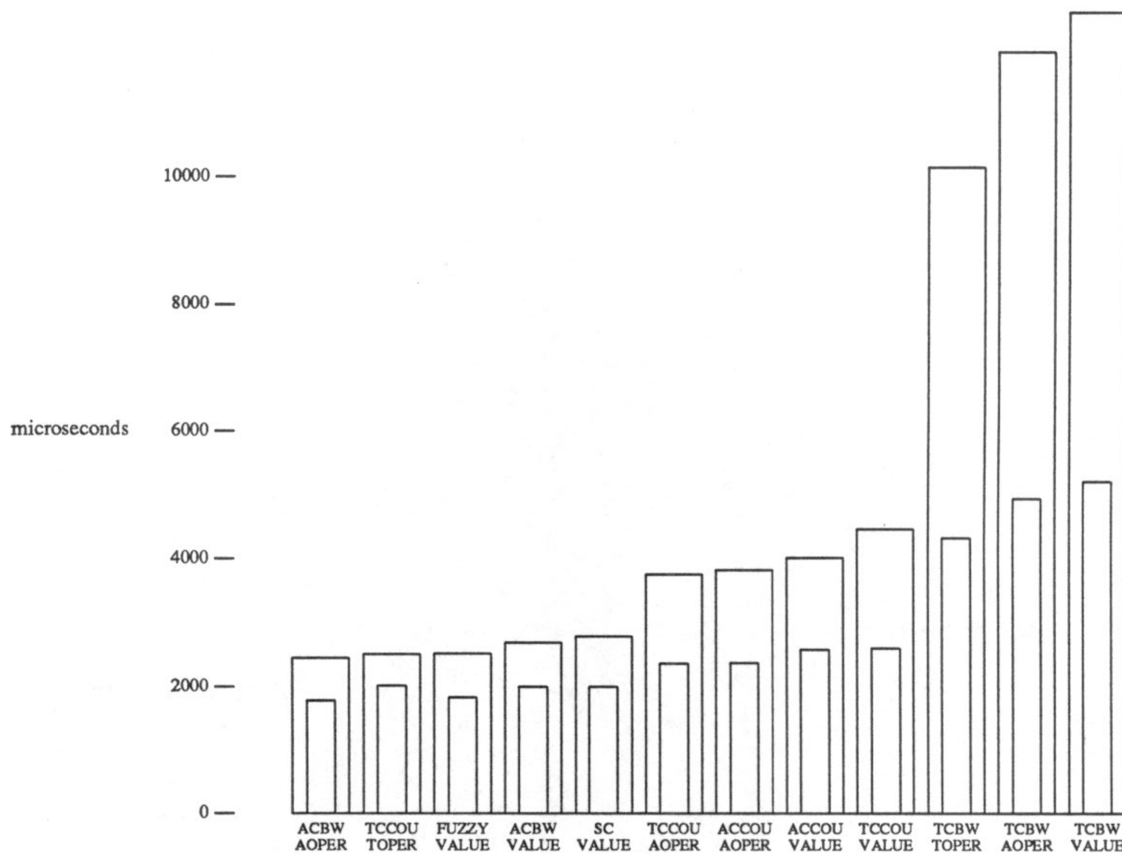



Figure 6.3 - Measured and Predicted Overhead

Before continuing, we will first consider the causes of these discrepancies, starting with the copy-on-update checkpoints.

Examination of the data in Table 6.7 indicates an abnormally large amount of processor time consumed by the checkpoint server in each of the COU checkpoints. Because of the statistical nature of the system's timing facilities and potential correlation between the testbed processes and the sampling clock, timings of individual processes taken in isolation must be interpreted with care. However, the large increase in checkpointing time did indicate that the checkpoint server was a reasonable place to look for the cause of the increase.

A copy-on-update checkpoint server would be expected to consume more processor time than checkpoint servers using other strategies, since a COU checkpoint involves freeing the "snapshot" segment copies (created by transactions) once they have been flushed to disk. However, the observed increase is larger than would be expected from this effect alone.

In the testbed, another difference between COU checkpointers and others is in the access patterns they exhibit in primary memory. All types of checkpointers examine database segments in the same logical order during each sweep through the database. Because of the way the segments are physically laid-out in primary memory when they are first read in from the secondary database, this logical sweep translates to a sequential† sweep though virtual (primary) memory. Recall, however, that COU checkpointers write the "snapshot" copy of a segment to disk if one is available. The snapshots are kept in a separate buffer pool. The sequential primary memory sweep of a COU checkpointer is disturbed by these diversions to the snapshot pool.

Several experiments were run to determine whether or not this may have had an effect on the processor cost of the COU checkpointer. Each of the experiments uses a sequential allocation of 4000 segment-sized blocks of memory (about the size of the database used in the verification experiments) plus an additional pool of 1000 blocks. In addition, disk space for 1000 blocks is available. Each experiment involves writing 1000 blocks from memory to the disk. In the first experiment, the next block to be written is chosen at random from blocks 1 to 4000. In the second experiment, the first block to be written to the disk is chosen at random from among the first four blocks (numbers 1 through 4). The next block is then chosen from among the next four blocks (5 to 8), and this process continues until 1000 blocks are written. The third experiment is a variation of the second. As in the second experiment, one block from each group of four is chosen for I/O. However, half of the time another block is written it its place. The other block is chosen at random from the last 1000 blocks (numbers 4001 to 5000), which are unused in the "sequential" part of the experiment.

The second experiment was designed to have a memory access pattern similar to that of a non-COU checkpointer. The final experiment is similar to a COU checkpointer, where the last

† This is not completely accurate. Newly allocated segments and locking conflicts on existing segments can alter the sequential pattern somewhat.

thousand blocks (4001 to 5000) correspond to the "snapshot" buffer pool from which a COU checkpointer sometimes takes a segment copy.

For each experiment, the mean total processor time was measured over 30 runs. Note that all three experiments have the same disk access pattern. Only the source of each disk write varies among the experiments. The target is the same in each case. The times are shown in Table 6.8.

| experiment | user processor time (milliseconds) | system processor time (milliseconds) | total processor time (milliseconds) |
|---|---|---|---|
| 1 (random) | 1.38 | 84.72 | 86.10 |
| 2 (sequential) | 1.40 | 44.77 | 46.17 |
| 3 (variant) | 2.09 | 101.71 | 103.80 |

Table 6.8

The third test, corresponding to the COU checkpointer, has significantly higher processor cost than the second. Interestingly, it's cost is even higher than that of the first test in which blocks were selected at random. Higher times might be expected for a more random memory access pattern if virtual memory paging was occurring. However, the experimental machine has more than enough physical memory to hold the virtual space used in these experiments. In addition, Mach's virtual memory performance monitoring facilities indicate that no paging occurs during the tests.

Though these experiments are not conclusive and we are uncertain as to the cause of the extra processor time, we believe that this effect (which is *not* modeled) accounts for most of the unexpected increase in COU overhead costs.

Next we will consider the discrepancy in the TCBW experments. Recall that the distinguishing characteristic of black/white checkpoints is that they can cause transactions to abort and restart for violating the two-color rule. The model computes the expected number of attempts required to complete a transaction. The mean number of attempts is also measured by the testbed. Table 6.9 shows the measured and predicted numbers of attempts for each of the three TCBW experiments.

| experiment | | number of attempts to complete | |
|---|---|---|---|
| checkpoint | log | measured mean | predicted mean |
| TCBW | VALUE | 1.84 | 1.52 |
| TCBW | AOPER | 1.95 | 1.52 |
| TCBW | TOPER | 2.04 | 1.52 |

Table 6.9

The great expense of aborting and restarting transactions means that a small increase in the number of attempts to complete results in a large increase in transaction overhead. Figure 6.4 shows what the results of the verification experiments would have been if the model had accurately predicted the mean number of retries in each of the black/white experiments. (To produce the model's bars in Figure 6.4, we substituted the measured mean number of retries for the predicted value in the model's calculations.)

The large increase in the model's predictions shown in Figure 6.4 indicates that the underestimation of the number of retry attempts is a principal cause of the low predictions for the TCBW checkpoint experiments. The model's difficulty lies in its assumption that the checkpointer chooses its next segment at random from among the segments it has yet to examine. In the testbed, the checkpointer's segment access is far from random. In fact, the testbed's checkpointer examines the segments one set at a time, and always moves through the sets in the same order: Account set, Customer set, Hot Card set, Store set. To make matters worse, the DEBIT, CCCK, and CLCK, transaction access records in both the Account and Store sets (see Table 6.2). As these sets are at the beginning and end of the checkpointer's sweep, the transactions are very likely to violate the black/white restriction. DEBIT, CCCK, and CLCK comprise 60% of the transaction load.

Modifying the model to account for this difference would involve eliminating the randomness assumption used in analyzing the checkpointer. Instead, the conditional probability of a portion of the database being accessed next, given location of the previous access, would have to be considered. This would detract considerably from the simplicity of the model. In addition, it is not clear how serious the underestimation would be for other transaction loads. Clearly, the load used to drive the testbed was particularly bad in this respect because of the high correlation of

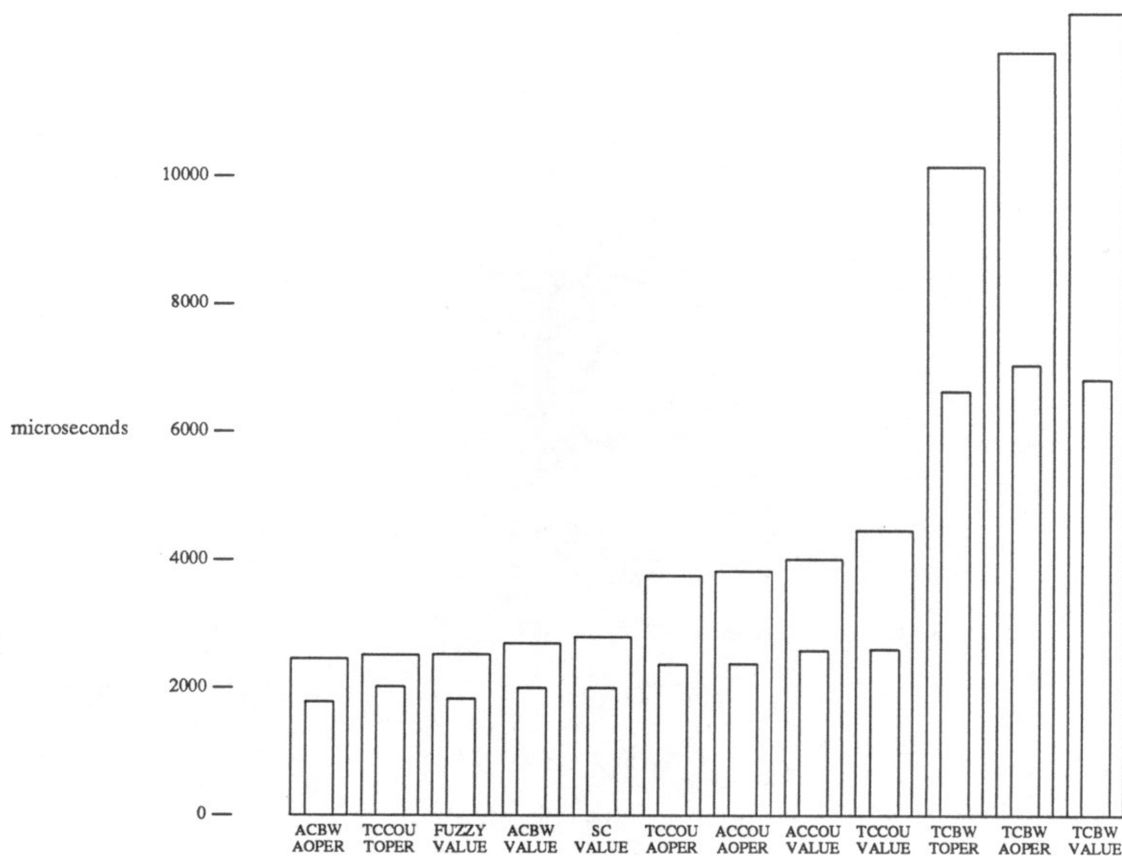



Figure 6.4 - Measured and Alternate Predicted Values

accesses to the Account and Store sets.

Even with the corrected retry counts, there is still a great deal of overhead not predicted by the model. There are a number of unmodeled costs which contribute to this difference, such as the cost of queueing the transactions that are waiting to be restarted. Fortunately, as we have seen in Chapter 2 and will discuss again in the next Chapter, TCBW checkpoints are generally predicted to have the highest overhead costs. The model's underestimation of these costs does not affect the *relative* performance of TCBW checkpoints in most situations.

# CHAPTER 7
# PERFORMANCE STUDIES

In Chapter 2 we compared the performance of a variety of checkpointing strategies. In this Chapter we will expand that analysis to include the performance of logging and storage management alternatives. We will also consider how these strategies can affect the performance of a checkpointer.

Most of the analyses in this chapter are based on data from the performance model and use the same default parameter settings are were used in the checkpointing analysis of Chapter 2. Others use experimental data from the testbed. The remainder of the Chapter is divided into three sections. The first section examines logging strategies and the interactions between logging and checkpointing. The second and third sections cover management of the secondary and primary database copies.

## 7.1 Logging Strategies

In Chapter 2, we considered the performance of various checkpointing strategies, assuming that VALUE logging was used. In this section we will eliminate that assumption, and re-evaluate the checkpointing strategies in combination with different logging strategies.

Recall that the various checkpointing strategies *permit* different logging strategies. A logging strategy is permitted by a checkpointer if the log it produces can be combined with the backup database produced by the checkpointer to create an up-to-date, consistent copy of the database after a failure. The logging strategies permitted by the three classes of checkpoint strategies are shown in Table 7.1.

To study the effects of the different logging strategies, we will reconsider the checkpointing comparisons made in Chapter 2, this time combining a checkpointer with an operation logging strategy if permitted. A checkpointing strategy will be combined with the most "abstract" (i.e.,

| checkpoint | logging strategy VALUE | AOPER | TOPER |
|---|---|---|---|
| fuzzy | permitted | -- | -- |
| action consistent | permitted | permitted | -- |
| transaction consistent | permitted | permitted | permitted |

Table 7.1

rightmost in Table 7.1) logging strategy it permits. In other respects, this comparison is the same as that that produced the graph of Figure 2.3., meaning

- checkpoints are taken as quickly as possible
- an immediate, shadow update strategy is used
- transactions are pre- and group-committed
- the backup database is maintained using the PING-PONG strategy.

The results are shown in Figures 7.2 and 7.3. For convenience, the wide, outer bars in each figure give represent the results of the original experiment (VALUE logging). The inner bars represent the results of the current experiment.

As Figure 7.2 indicates, the changes in overhead are not large. This indicates that the logging cost is not a dominant factor, at least for our default parameter set. Using TOPER logging with the transaction-consistent checkpoints saves 200-400 instructions per transaction. The change to AOPER logging saves even less.

As is indicated in Figure 7.3, changing log strategies affects recovery time as well. In fact, changing to operation logging tends to reduce *both* overhead and recovery time, although by small amounts. The exception is TCCOU, which sees an increase in recovery time with the switch from value to transaction logging. This is because the log replay time becomes processor-bound because of the high processor cost of re-running logged transactions. Note that the log replay time does not become I/O bound when logging switches from value to transaction logging under TCBW checkpoints. This is because a TCBW checkpointer produces a more up-to-date database snapshot than TCCOU. As a result, fewer of the transactions in the log need to be re-run.

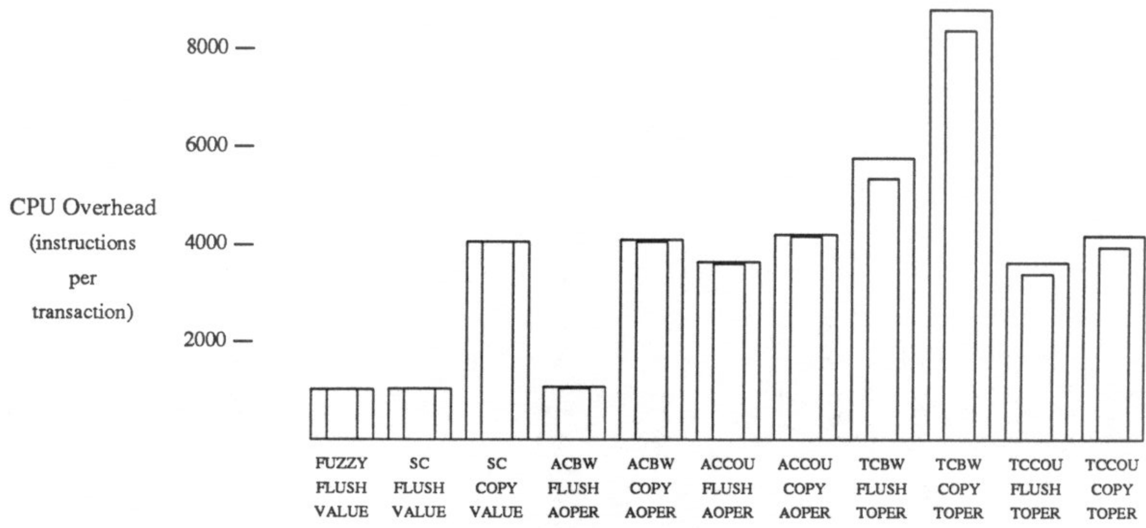



Figure 7.2 - Processor Overhead Using Various Checkpointers (varied logging)

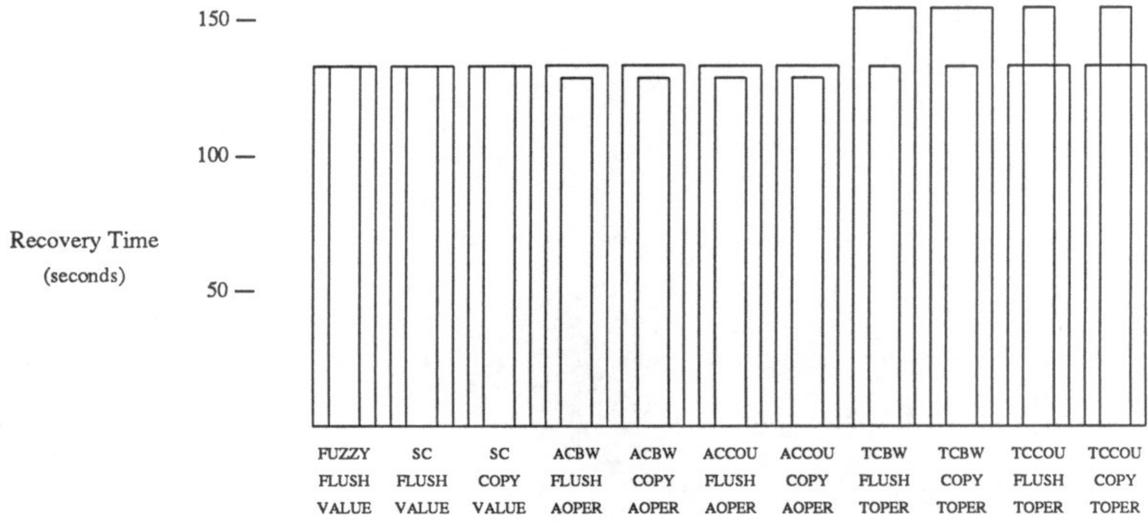



Figure 7.3 - Recovery Time Using Various Checkpointers (varied logging)

It would be possible to get a clearer picture of the total effect of a change in logging strategy if the overhead and recovery time changes were combined into a single number. As we showed in Chapter 2, recovery time can be traded-off for overhead costs (and vice-versa) by changing the checkpoint interval. To get a single measure of the effect of a change in logging strategy, we can adjust the checkpoint interval so that the recovery time is the same before and after the change. The overhead cost is then a useful single metric.

Figure 7.4 shows the processor overhead of each of the combinations when their recovery times have been adjusted to match the slowest of the recovery times (about 2.5 minutes, for TCBW checkpoints with VALUE logging) in Figure 7.3. As in Figure 7.2, the outer bars show the overhead of the checkpointer when combined with value logging (with suitably modified checkpoint interval).

Though this test shows a substantial reduction in overhead achieved by switching to from VALUE to TOPER logging under TCBW checkpoints in particular, the relative ordering of the checkpointing strategies is not greatly affected. Overhead costs are lower because switching to operation logging reduces recovery time (except under TCCOU checkpoints). When we equalize recovery time, the operation logging scenarios have more recovery time to "trade-off" for decreased overhead. In other words, (in most cases) operation logging can achieve the same recovery time as value logging, but with less frequent checkpoints.

Less frequent checkpoints benefit TCBW checkpoints more so than the other strategies because restart costs are reduced. (When checkpoints are less frequent, transactions are less likely to violate the black/white restriction of the TCBW checkpointer.) As a result, the overhead reduction shown for TCBW in Figure 7.4 is more dramatic than for the other checkpointing strategies.

When TCCOU checkpointing is used, recovery is faster with value log (because of the high cost of re-running transactions from a TOPER log). As as result, the trade-off works in the other direction under TCCOU checkpointing. To keep recovery time constant, checkpoints must be more frequent when operation logging is used than when logging by value. Thus, as shown in Figure 7.4, the per-transaction overhead is greater when TOPER logging is used.

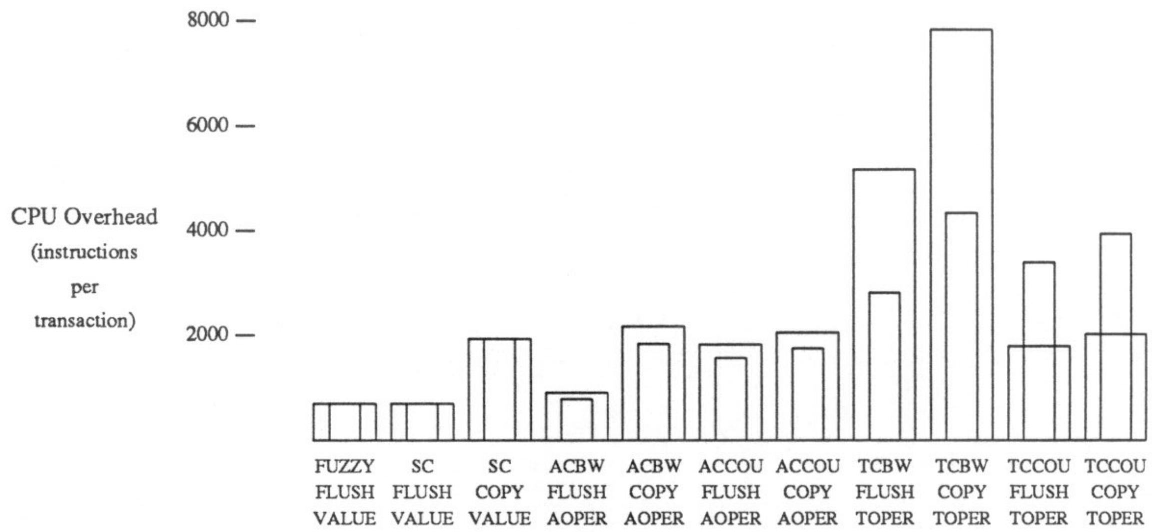



Figure 7.4 - Processor Overhead Using Various Checkpointers (varied logging & fixed recovery time)

Figure 7.5 shows in more detail the performance of the three logging strategies in combination with TCCOU checkpoints. The curves are produced by varying the checkpoint interval. The disadvantage of TOPER logging disappears if long recovery times (long intercheckpoint intervals) are acceptable. In fact, as the asynchronous component of the overhead becomes less and less significant (because slower checkpoints have their cost spread over more transactions), TOPER logging results in less overhead than the other strategies because of its lower synchronous costs.

We have seen that the same logging strategy can have qualitatively different effects on performance when combined with different checkpointing strategies. This has hinged on the fact that the replay time for TOPER logs can be CPU-bound given our model parameters. Next we will reconsider our comparison under the assumption that the processor is sufficiently fast to keep the replay time I/O-bound, even for TOPER logs.

Figure 7.6 is analogous to Figure 7.3, showing the recovery time for a variety of checkpointing and logging strategies. However, the recovery times in Figure 7.6 are I/O times; we are assuming that the processor is fast enough to keep up with the I/O rate. (The transaction overhead costs for these scenarios are the same as those in Figure 7.2.) As in Figure 7.3, the outer bars represent the VALUE logging case, and the inner bars AOPER or TOPER logging as appropriate.

Figure 7.6 shows that switching to TOPER logging now reduces the recovery time of the TCCOU (and TCBW) strategies. A TOPER log means significantly less log per transaction than a VALUE log. Next we equalize all of the recovery times of Figure 7.6 by adjusting the checkpoint interval and compare the overhead costs of the different strategies. Figure 7.7 shows the overhead of each of the strategies, with a constant I/O recovery time of just over 2.5 minutes.

Figure 7.7 shows that with I/O bounded recovery and the adjusted checkpoint interval TCCOU checkpoints with TOPER logging are no more expensive then FUZZY checkpoints. Even the overhead of TCBW checkpoints become reasonable.

Achieving I/O bounded recovery times in a high-performance system using TOPER logging requires fast processor. Table 7.2 shows the minimum processor speed required to keep up with the I/O rate during the replay of the log. The numbers in the table were derived assuming

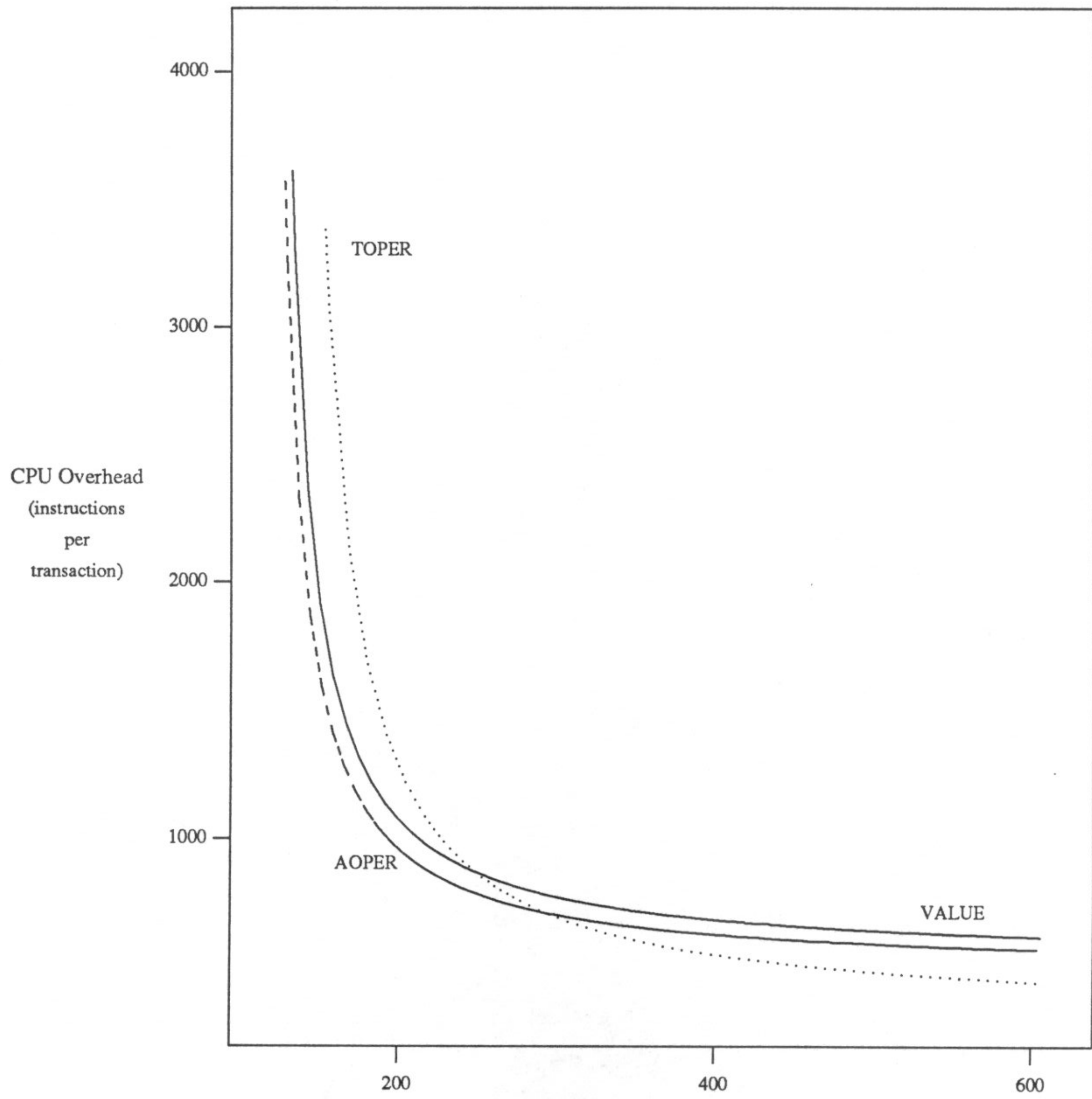



Figure 7.5 - Processor Overhead/Recovery Time Tradeoff: TCCOU Checkpoint

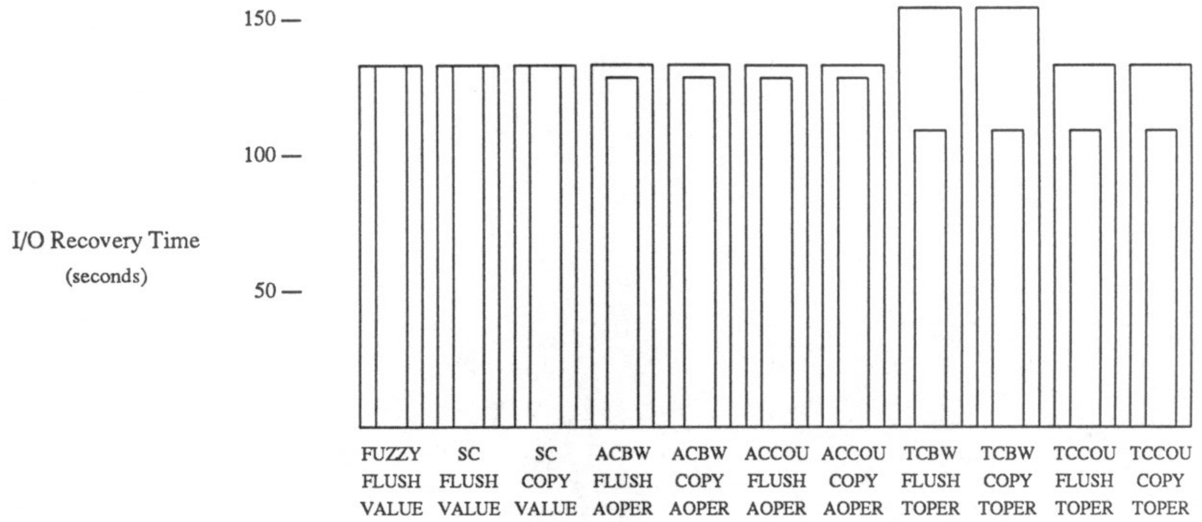



Figure 7.6 - I/O Recovery Time Using Various Checkpointers (varied logging)

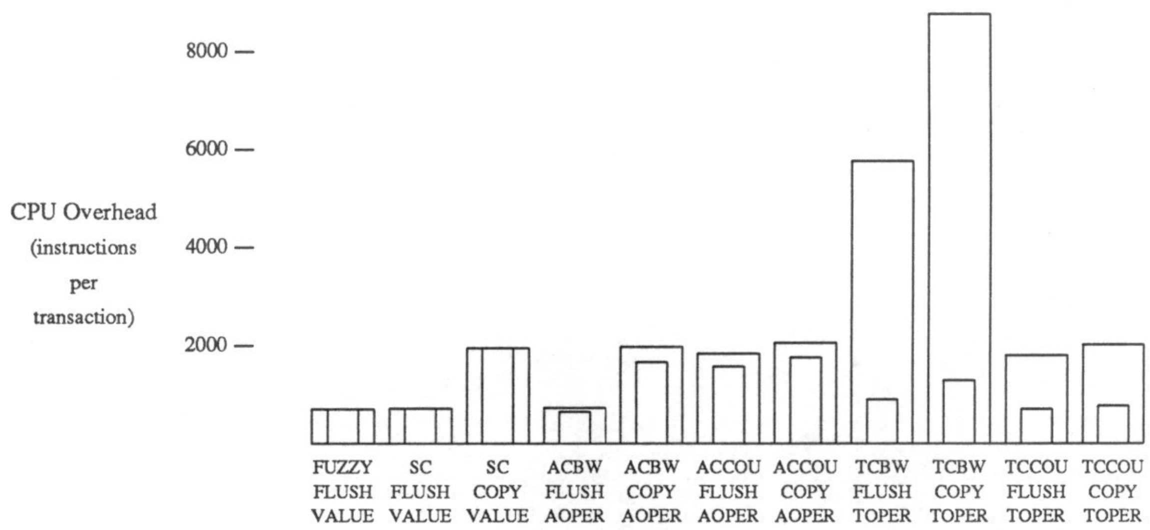



Figure 7.7 - Processor Overhead Using Various Checkpointers (varied logging & fixed recovery time)

PINGPONG backups and the TCCOU checkpoint strategy.

| logging strategy | processor speed (MIPS) |
|---|---|
| VALUE | 1.68 |
| AOPER | 10.51 |
| TOPER | 65.54 |

Table 7.2

### 7.2 Spooling Checkpoints

So far we have examined a variety of checkpointing and logging strategies. For each checkpointing strategy (except FUZZY) we have considered a spooling (COPY) and non-spooling (FLUSH) option. Recall that a spooling checkpointer copies segments to a primary memory buffer and releases its segment lock before writing the segment (copy) to the backup disks. The goal of this strategy is to reduce the amount of time the segment lock is held, thereby reducing the lock contention between the checkpointer and transactions. The model has shown (see Figures 7.2 and 7.7) that the cost of spooling checkpoints can be significantly greater than that of their non-spooling counterparts. However, since lock contention is not modeled, the benefits of spooling have yet to be determined.

The testbed monitors lock contention, and we have used this facility to study the advantages of spooling. Table 7.3 shows the lock contention rate for the spooling and non-spooling versions of several combinations of logging and checkpointing strategies. Lock contention here means the percentage of lock requests that could not be granted immediately. The data was generated over thirty or fifteen minute runs using the same transaction load used in Chapter 6.

The table shows that spooling checkpointers cause almost no lock contention. The lack of lock conflicts is not surprising given the way the checkpoint server process is scheduled by Mach. As long as the CS can find a segment to flush before the end of its normal time quantum (i.e., always), it relinquishes the processor as a result of I/O activity. A spooling checkpointer holds no locks during I/O, so the checkpoint server will almost never hold a lock when it relinquishes the processor.

| log strategy | checkpoint strategy | lock requests | | | | | |
|---|---|---|---|---|---|---|---|
| | | spooling | | | non-spooling | | |
| | | total | conflicting | contention | total | conflicting | contention |
| AOPER | ACBW | 459466 | 2 | 0.0004 | 979309 | 222 | 0.0227 |
| VALUE | ACBW | 459112 | 3 | 0.0007 | 916527 | 261 | 0.0285 |
| AOPER | ACCOU | 457316 | 10 | 0.0022 | 896807 | 211 | 0.0235 |
| TOPER | TCCOU | 488704 | 5 | 0.0010 | 957610 | 264 | 0.0276 |
| TOPER | TCBW | 596845 | 8 | 0.0013 | 1344143 | 543 | 0.0402 |
| VALUE | SC | 925722 | 11 | 0.0012 | 950993 | 266 | 0.0280 |

Table 7.3

The table also shows that non-spooling checkpointers exhibit only a tiny amount of contention. Well over 99% of lock requests can be satisfied immediately. Thus it appears that the extra cost of spooling checkpointers is not justified, at least in the testbed environment. We would expect to see higher lock contention with non-spooling checkpointers if transactions were more data intensive or longer-lasting, or in a multiprocessor environment in which the normal mode of transaction execution was not nearly serial. In such situations spooling checkpointers could prove more beneficial.

### 7.3 Response Times

Before we move on to discuss secondary database management (SDBM) strategies, we will first take a look at checkpointing and logging strategies from a different perspective. Thus far we have considered transaction overhead (a throughput metric) and recovery costs, and have not looked at transaction response times. Transaction response times are not included in the performance model. However, we can obtain data on response times from the testbed.

Table 7.4 shows the mean transaction response time for different combinations of checkpointing and logging strategies. The data were generated using the same transaction load as was used in Chapter 6 and to produce Table 7.3. To measure response time, message servers in the testbed timestamp each transaction request message when it is placed on the transaction request queue. Another stamp is generated when the response is taken from the message server's response queue, and the response time taken to be the difference between the stamps. Thus, response time includes the entire time spent in the system, including queueing delays.

| log strategy | checkpoint strategy | transaction response time (seconds) mean | 90% confidence interval min | max |
|---|---|---|---|---|
| VALUE | FUZZY | 3.40 | 3.40 | 3.41 |
| VALUE | SC | 3.48 | 3.48 | 3.48 |
| AOPER | ACBW | 3.86 | 3.85 | 3.86 |
| AOPER | ACCOU | 4.09 | 4.09 | 4.10 |
| TOPER | TCBW | 33.85 | 33.46 | 34.24 |
| TOPER | TCCOU | 3.93 | 3.93 | 3.93 |

Table 7.4

Except for TCBW checkpointers, most of the strategies exhibit comparable response times. Under TCBW checkpoints, many transactions suffer from repeated abort/delay/restart cycles from violation of the black/white restriction. When using TCBW, transactions with no black/white violations have response times comparable to transactions under other strategies. However, those with violations can take an extremely long time.

It is conceivable that the response time of transactions under the TCBW strategy might be improved by varying the retry interval. Recall that the retry interval is delay between the abortion of a transaction for black/white violations and it's restart. The retry interval is measured in terms of the checkpointer's progress as it sweeps though the database. In the testbed, the checkpoint server signals the transaction server to restart aborted transactions each time it examines another $R$ segments. The restart interval, $R$, is a controllable checkpoint parameter. Figures 7.8 and 7.9 show the variation of the mean transaction response time and throughput under a TCBW checkpointer and TOPER logging, as a function of the retry interval $R$.

The response time can be improved somewhat by reducing the retry interval ($R = 1000$ segments was used to produce the data presented in Table 7.4) at some loss of throughput (i.e., increase in transaction overhead). However, even with very frequent retry attempts, the response time is considerably worse than that of the other checkpointing strategies. Thus, even in situations where TCBW may have been a feasible checkpointing strategy when throughput and recovery time alone are considered (e.g., combined with TOPER logging and sufficient processor speed to make the log replay I/O-bound), TCBW must be ruled out on the basis of poor response time.

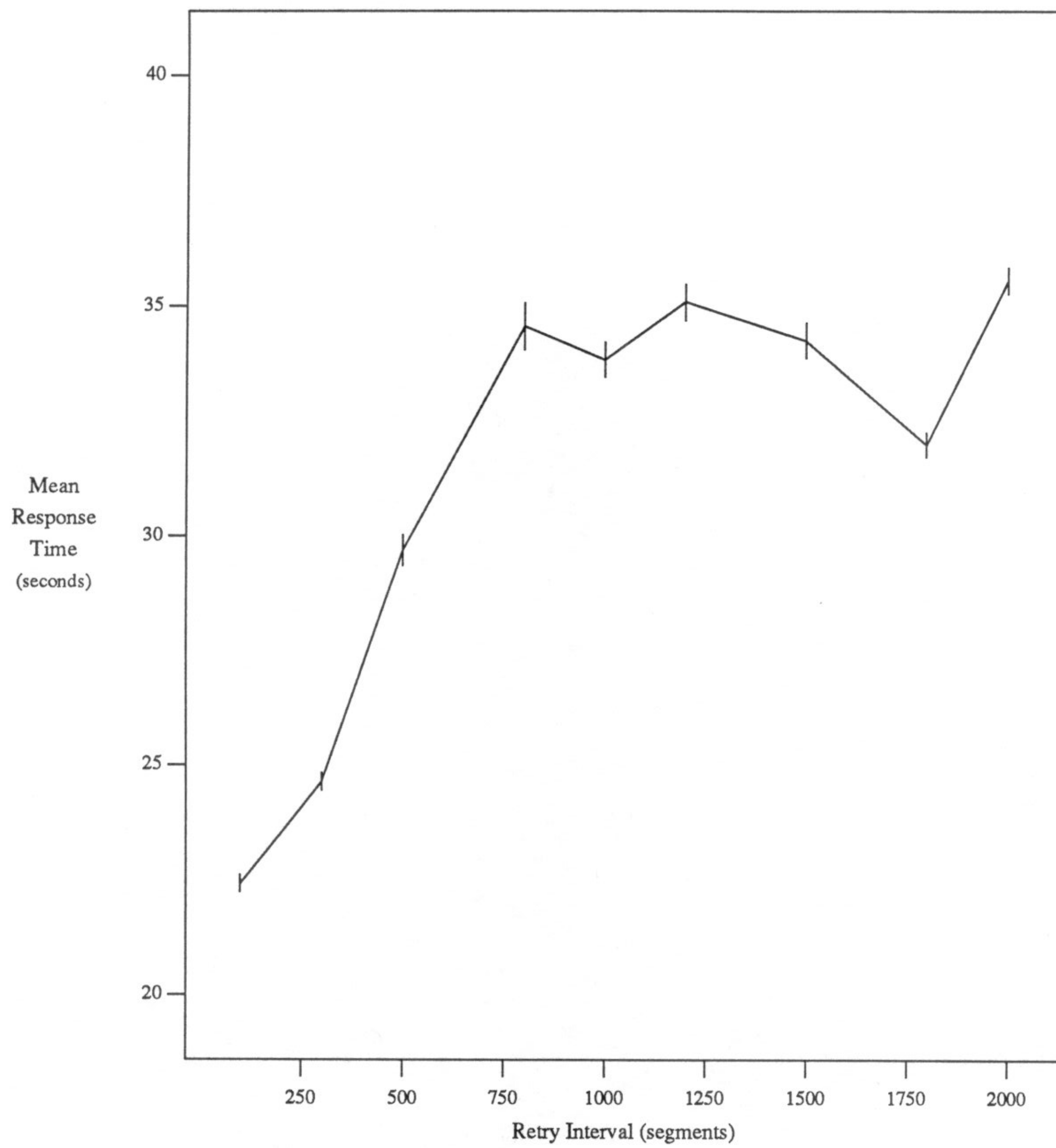



Figure 7.8 - Response Time Variation with Retry Interval

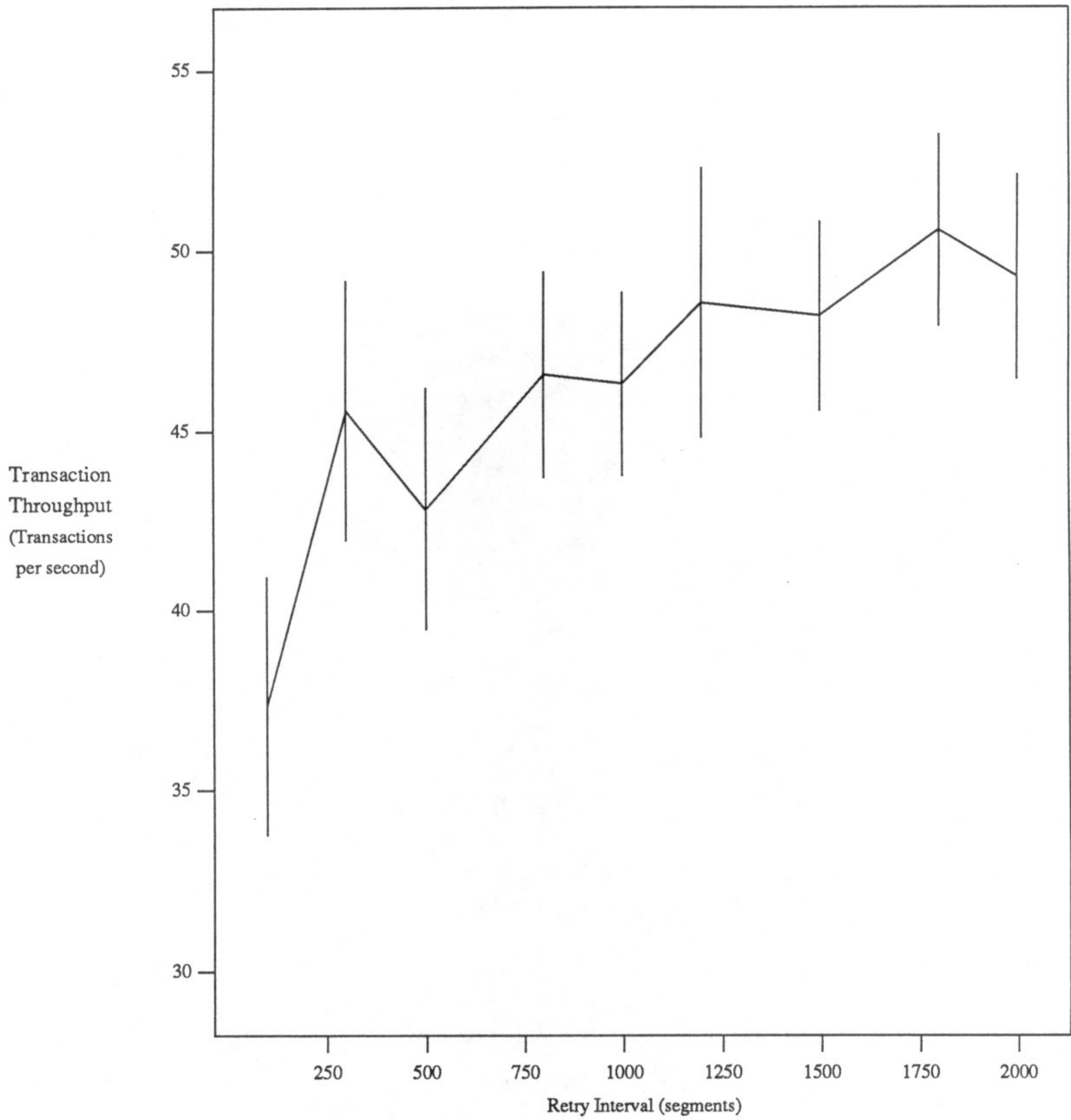



Figure 7.9 - Throughput Variation with Retry Interval

### 7.4 Backup Strategies

In the first section of this chapter and in Chapter 2 we have assumed that a PINGPONG strategy was used. In this section we will eliminate that assumption and consider the effect of the SDBM strategy on performance.

When checkpoints are taken as quickly as possible, i.e., a checkpointer is active constantly, the backup strategy has little effect on transaction overhead. Intuitively, this is because when the checkpointer is active, segments are always being flushed to the backup disks at the same rate. The rate is determined by the bandwidth to the backup disks, and does not depend on the checkpoint or SDBM strategy that is being used.

However, SDBM does affect recovery time by affecting the active *duration* of a checkpoint. If the total checkpoint interval is held constant while the SDBM strategy is varied, transaction overhead is affected because of the changing length of the active interval. For example, checkpoint sweeps initiated once every three minutes may take only one minute to complete under one SDBM strategy, but two minutes to complete under another.

Figure 7.10 shows the transaction overhead for the five SDBM strategies assuming a constant recovery time of just under three minutes. (This is the fastest recovery time possible under the FM strategy given our parameters.) Segment-consistent checkpoints and VALUE logging are combined with each of the SDBM strategies. The performance differences are small relative to the differences produced by changing checkpoint or log policies. However, the I/O cost of 1500 instructions in the default parameter set is lean, and the effect of SDBM changes will be more pronounced when I/O is more expensive.

The FM strategy costs more because each segment is flushed twice, resulting in longer-lasting checkpoints and more overhead. Single-flushing strategies, like SM, can recover more quickly than FM since their faster checkpoints mean a shorter log to replay at recovery time. If SM is to have the same recovery time as FM, it can afford to initiate checkpoints less frequently and thus reduce overhead.

TWIST also requires longer checkpoints, but not because segments are flushed twice. TWIST suffers in restoring the primary database copy after a failure, which takes it longer than any of the other SDBMs. (Recall that under TWIST, two copies of each segment must be read in

from the backup.) To equalize recovery times, TWIST must reduce its log replay time by checkpointing more frequently.

One interesting note in Figure 7.10 is that PINGPONG and SM backups perform about as well as SHD. This is an indication that the transaction load is producing a *saturated* database. We say that that database is saturated when the load is sufficient to dirty almost all of the segments during the checkpoint interval, i.e., a checkpoint normally must flush all $N_{seg}$ segments.

Under a SHD strategy dirty pages are flushed once each (along with occasional updates to the indirection table). Therefore we expect the length of a checkpoint to a SHD backup to decrease correspondingly as the database becomes less saturated. A PINGPONG checkpoint should get shorter as well, but not as quickly as for SHD. This is because PINGPONG uses two dirty bits, one for each secondary copy, and must sometimes flush twice a segment that would have been considered clean the second time by a SHD checkpointer. In a saturated database, these segments are always dirtied by a subsequent transaction before the second flush can take place, thus PINGPONG no longer does extra work in flushing them. On the other hand, checkpointing to a SM backup should get no faster with a less saturated database since SM backups require all segments to be written, dirty or not.

We would expect the database to become less saturated in a less heavily loaded system. Figure 7.11 shows the active checkpoint interval (checkpoint duration) under each of the SDBM strategies as a function of transaction load. In the figure, saturation is indicated by flattening of the active interval curve as the load increases. As expected, the active interval decreases more quickly under SHD than PINGPONG as the load drops, with the curves crossing at about 300 transactions/second. Also as expected, the active interval under SM is load-independent.

Figure 7.12 shows the overhead under each of the SDBM strategies as a function of the transaction load, with the checkpoint interval fixed at three minutes. As expected, the differences among the SDBM strategies are more pronounced when the database is not saturated.

Another factor that can have an effect on SDBM performance is the segment size. Figure 7.13 shows the recovery time under each of the SDBM strategies as the segment size is varied. The curves are governed by two effects. First, in a saturated database, larger segments mean more efficient I/O and thus faster checkpoints. At recovery time this translates into less log to be

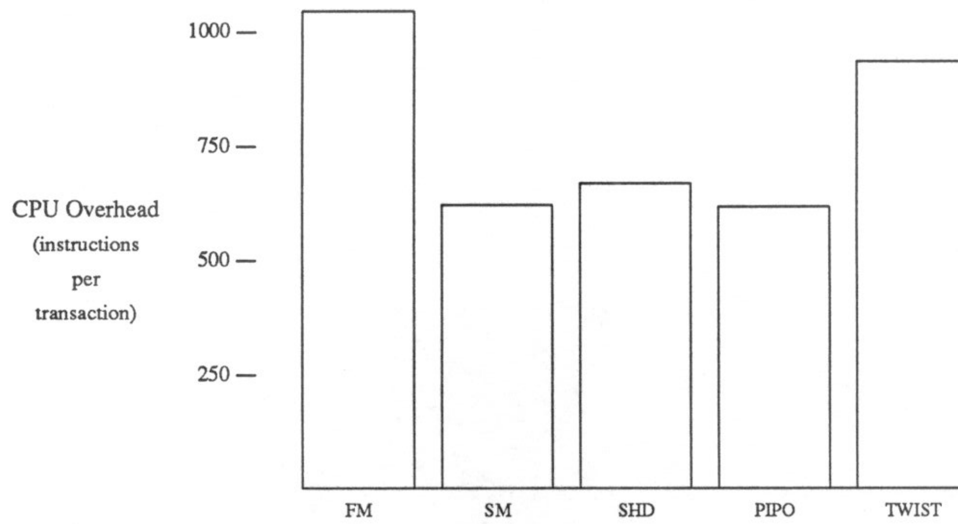



Figure 7.10 - Processor Overhead Using Various SDBM (fixed recovery time)

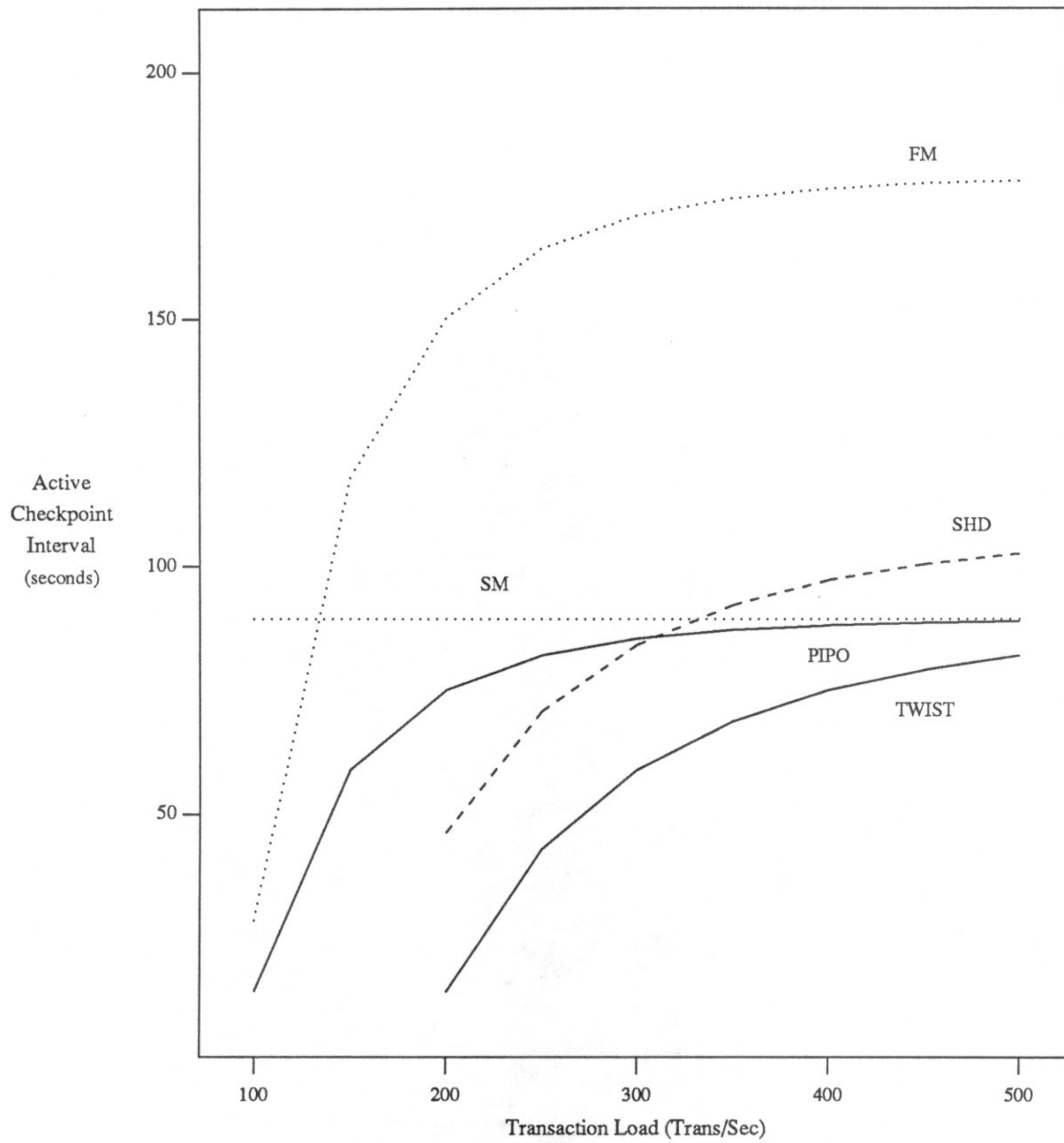



Figure 7.11 - Change in Checkpoint Interval with Load

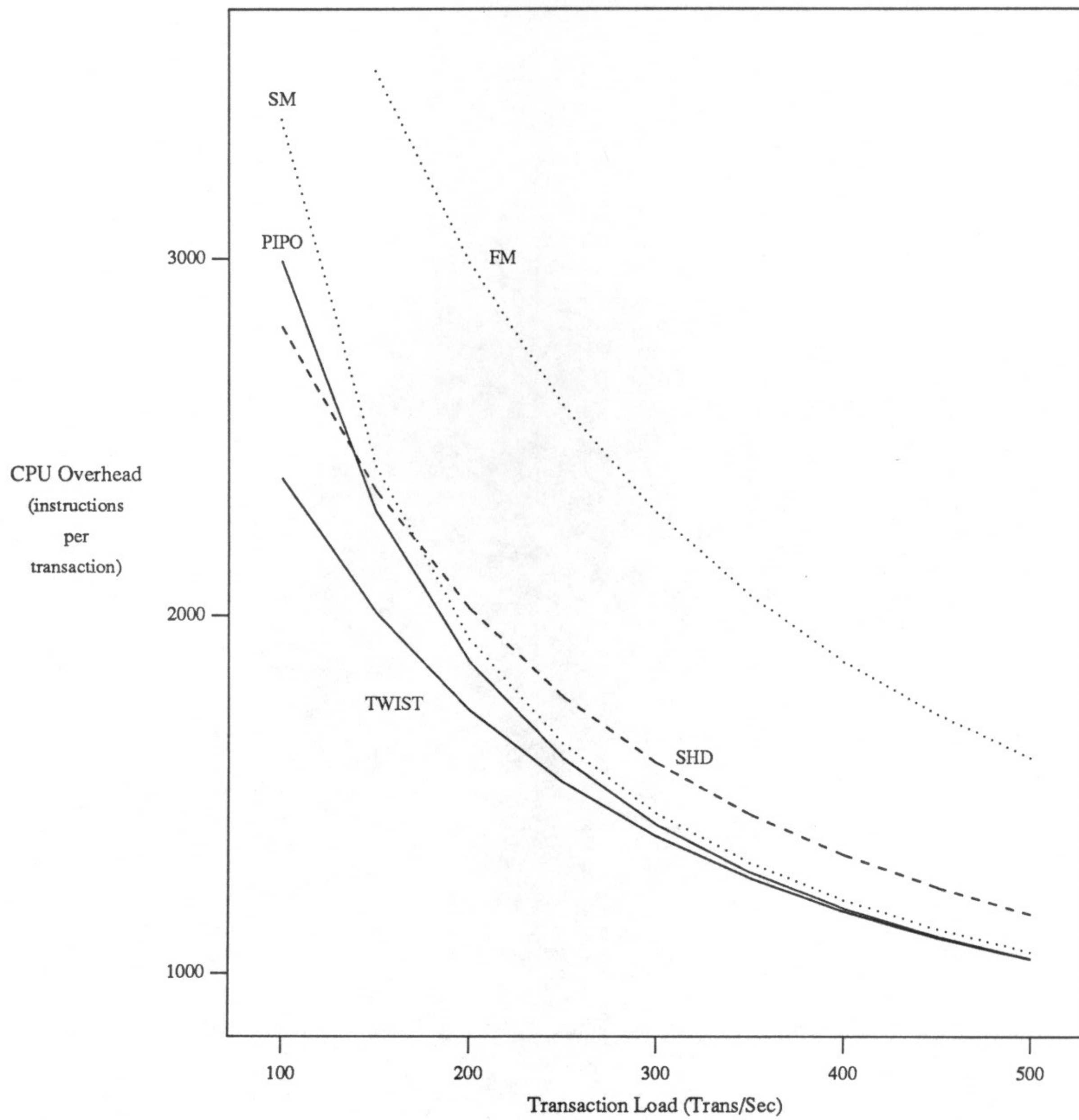



Figure 7.12 - Effect of Varying Transaction Load

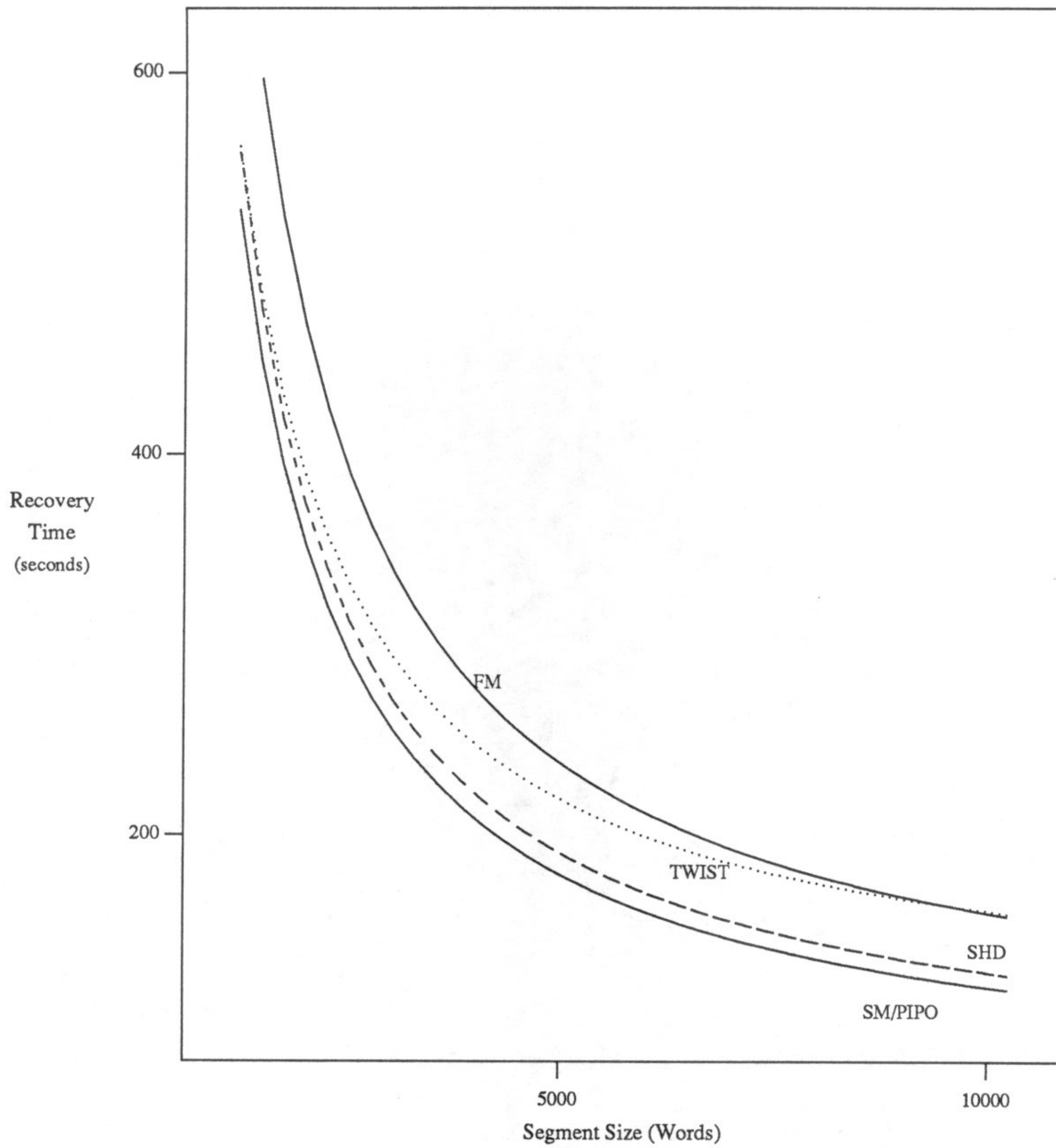



Figure 7.13 - Effect of Varying Segment Size

replayed, and thus shorter recovery times. All of the the SDBM strategies show this effect, although the FM strategy has longer checkpoints than the others (see Figure 7.11), accounting for its greater recovery times.

The second factor is the speed with which the primary database copy can be restored after a failure. Because of more efficient I/O with larger segments, all strategies can restore the primary more quickly. TWIST, which restores two copies of each segment (as one double-sized block) behaves differently from the other strategies. When segments are small, a double-sized segment can be read from the backup disks almost as quickly as a single segment because of the fixed overhead associated with each I/O operation. For small segment sizes TWIST performs nearly as well as PINGPONG. For larger segment sizes, TWIST's penalty is fully felt and its performance degrades.

### 7.5 Update Strategies

In this section we consider the effects of primary memory management on the recovery manager's performance. Although the cost of primary memory management is not directly included in the recovery overhead, the memory management strategy does affect it. For example, the primary memory management strategy determines whether or not UNDO logging is necessary. We will consider all four combinations of update location strategies (INPLACE and SHADOWS) and update delay strategies (IMMEDIATE and DELAY).

Figure 7.14 shows the recovery time/transaction overhead tradeoff for each of the four combinations. The curves were obtained by varying the checkpoint interval.

The difference between SHADOW and INPLACE strategies is not large unless recovery time is very short. The slightly higher cost of the INPLACE strategy arises from logging UNDO information in the log. This difference is probably even less significant than it appears since we do not consider the cost of the update strategies themselves (e.g., the cost of allocating shadow pages in primary memory) in the model. We can expect that SHADOW updates would be more costly to implement than INPLACE updates, thus negating any reduction they permit in recovery overhead.

One distinction SHADOW and INPLACE updates is in the way they handle aborted transactions. Aborting a transaction when INPLACE updates are used is likely to be costly because old values of modified records must be restored from the log. We would expect to see a greater difference between the SHADOW and INPLACE strategies in an environment with many aborted transactions. TCBW checkpointers produce such an environment. As we have already seen, TCBW checkpointers cause frequent transaction restarts, particularly when checkpointers occur frequently.

Figure 7.15 shows the *increase* in the transaction overhead when INPLACE rather than SHADOW updates are used under segment-consistent and TCBW checkpointers. As expected, the difference is greater under TCBW, and it increases as the recovery time is reduces, since to do so the checkpointer must be run more frequently. However, the increase is not large and would probably not be significant unless transactions updated a large number of records.

Figure 7.14 showed even less of a difference between the IMMEDIATE and DELAY update time strategies. As with the SHADOW/INPLACE comparison, this difference can be expected to get somewhat larger when there are many aborted transactions, since the DELAY strategy results in almost no log data for an aborted transaction. However, any advantage this accrues is likely to be offset by the additional (unmodeled) expense of implementing the DELAY strategy. DELAYed updates are more costly to implement because modifications must be buffered until the updating transaction commits.

One situation where the DELAY strategy does prove advantageous is when fuzzy checkpoints are combined with INPLACE updates. In that case, UNDO log information must be sent to the log disks since transaction locks do not prevent the checkpointer from seeing the results of partially completed transactions. Since the DELAY strategy avoids the creation of UNDO log information, this cost is avoided. Figure 7.16 shows this situation. It is similar to Figure 7.14, except that fuzzy rather than segment-consistent checkpoints are being taken.

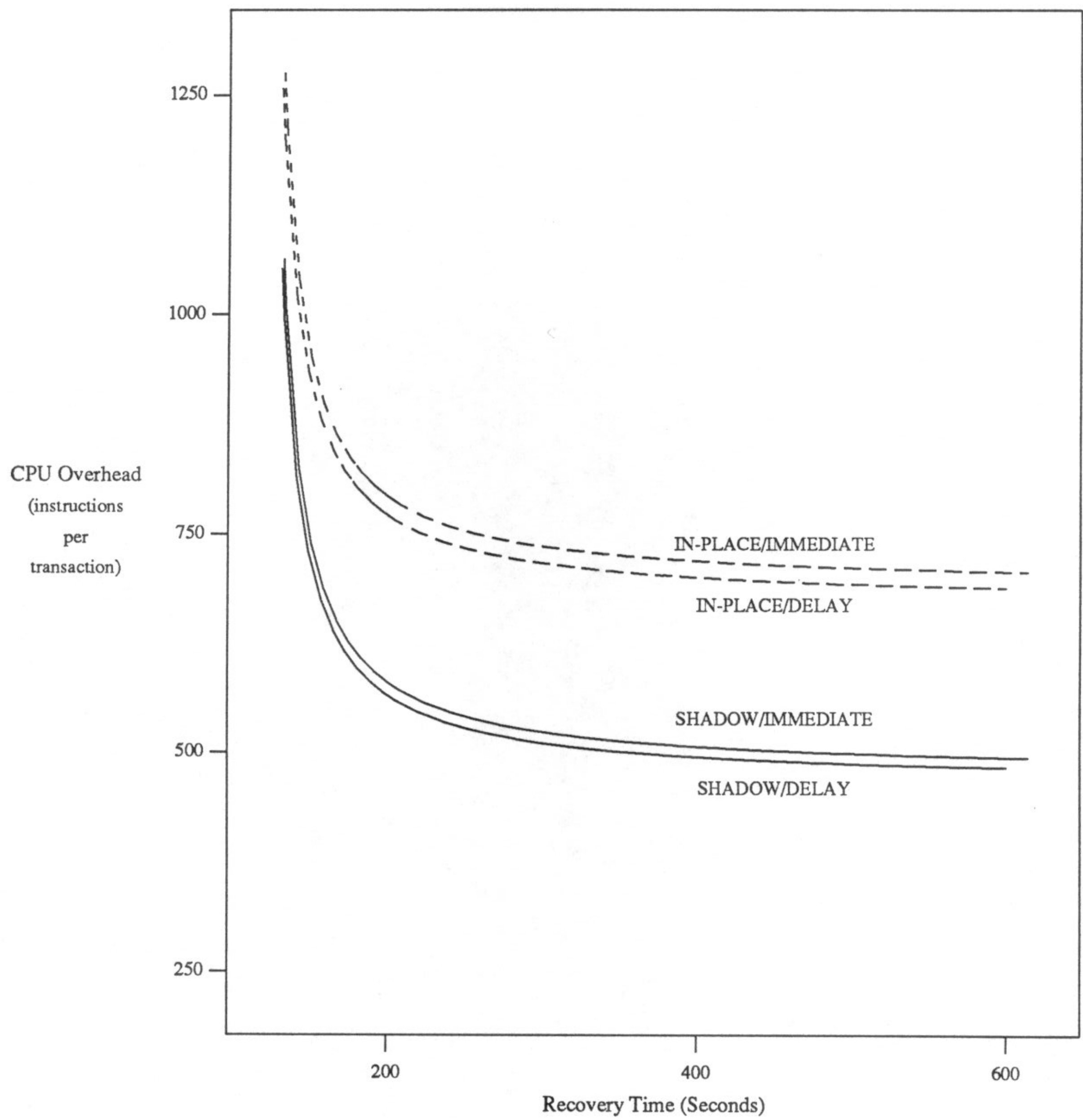



Figure 7.14 - Processor Overhead/Recovery Time Tradeoff (SC checkpoint)

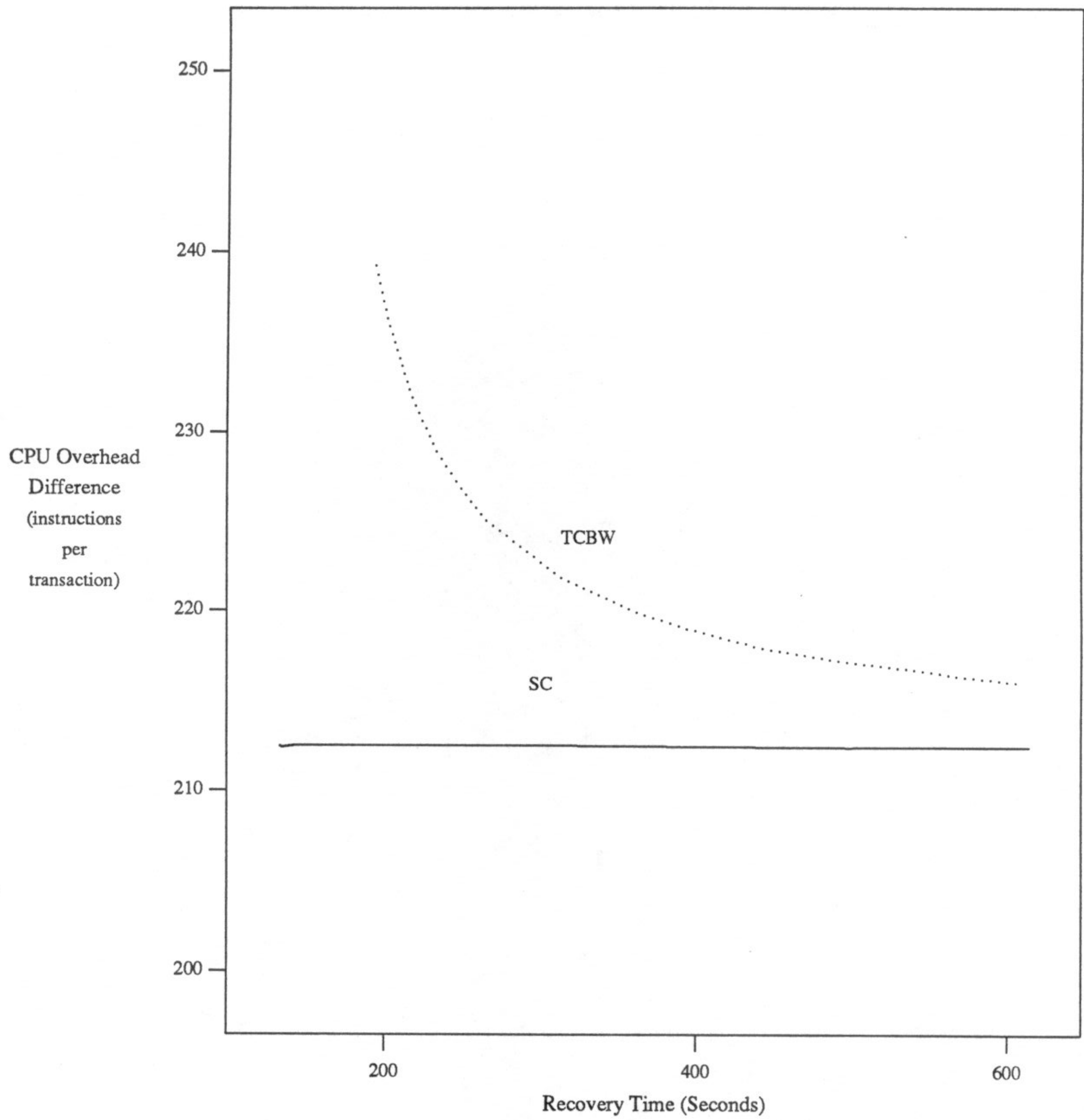



Figure 7.15 - Overhead Difference (SC and TCBW checkpoints)

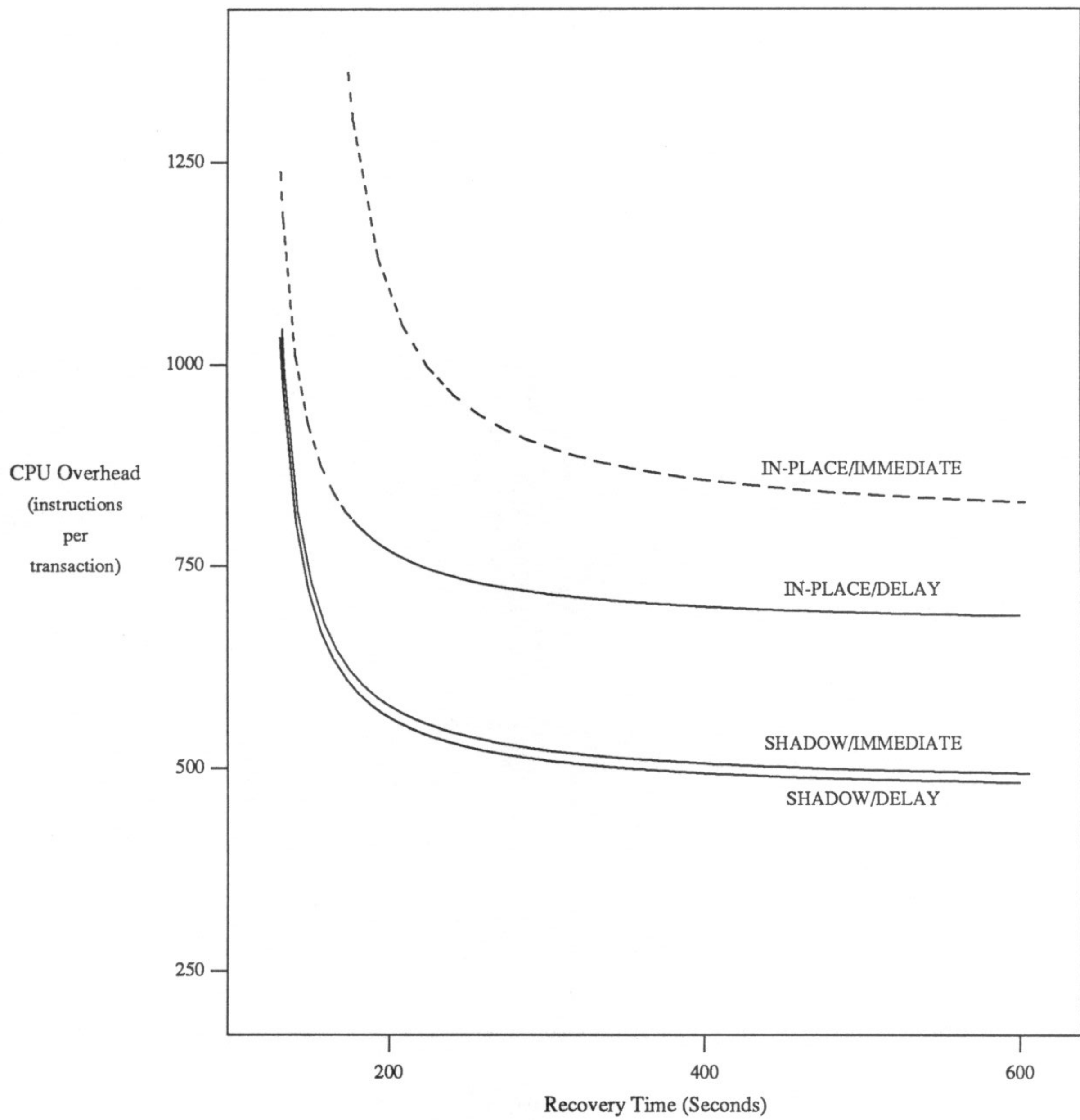



Figure 7.16 - Processor Overhead/Recovery Time Tradeoff (Fuzzy Checkpoint)

# CHAPTER 8
# CONCLUSIONS AND FUTURE WORK

We have presented a collection of strategies for recovery management in memory-resident databases. The collection includes strategies for maintaining a secondary copy of the database, for logging, and for updating the primary database copy as requested by running transactions. We have described an analytic performance model and the implementation of a recovery testbed used to compare the performance of various combinations of strategies. The performance metrics included the impact of recovery strategies during normal transaction processing as well as the speed of recovery after a failure.

While all of the strategies affect performance in one way or another, the most critical decision is the choice of checkpointing strategy. Checkpointing costs are normally the biggest factor in the impact of recovery management on transaction processing speed. Under conditions that could be expected in a high-performance memory-resident system, FUZZY checkpointing strategies consistently prove to be the best choice. This is true despite the fact that other strategies, which produce more consistent backups, and can take advantage of less-costly operational logging. FUZZY checkpoints have the additional advantage that they can be used with fast monoplex backup strategies for even better performance.

The performance advantages of FUZZY checkpointing hold up under a number of variations in our high-performance scenario. One scenario in which its advantages disappear is when log bandwidth is very limited and a great deal of processor power is available for recovery (i.e., the processor is under-utilized during normal transaction processing). That scenario favors checkpointing strategies which can be combined with operation logging, since operation logging requires relatively little I/O bandwidth but more processor time than value logging.

Among the backup strategies, PINGPONG, shadow, and sliding monoplex performed well. Both the fixed monoplex and TWIST strategies suffer from longer recovery times. PINGPONG

and sliding monoplex perform well in large part because the database is saturated by the transaction update load. The performance of both of these strategies can be expected to degrade when the database is unsaturated. The database may fail to saturate because of a lighter load (fewer transactions or fewer updates per transaction), a larger database or a database with large cold areas, or when segment sizes could be made smaller without impacting the efficiency of I/O operations (e.g., if the backup medium was stable RAM rather than disk).

Normally, the update strategies (for modifications to the primary database copy) have the least effect on performance of all of the strategies studied. Furthermore, those strategies which result in lower overhead must be used with more expensive storage management strategies. Thus their benefits will be at least partially negated. However, when combined with certain recovery and backup strategies, the updates strategies can prove to be more critical. This is particularly true when FUZZY checkpointing is used with a monoplex backup. Using a delayed update strategy in that case avoids the need for a large increase in the cost of logging.

Despite the variety of strategies we have considered, there remain a number of issues to be explored. Though we (and others) have begun exploring some of them, others have yet to be studied. Some of these issues are discussed briefly in the following sections.

## 8.1 Hardware

This thesis has presented and compared a variety of *algorithmic* recovery alternatives. We did not consider the feasibility or benefits of special purpose or dedicated recovery hardware. For example, several architectures for MMDB transaction processing have been proposed which rely on a large amount of non-volatile RAM [Eich87a, Lehm87a]. These proposals also include dedicated recovery processors. Other possibilities have been suggested as well. For example, HALO, a special hardware device which creates a word-level value log transparent to the database processor is proposed in [Sale86a]

To determine the feasibility of these ideas, it is necessary to know what performance advantages they can provide over similar systems without special hardware. The comparisons presented here play a useful roll in that decision, since they can be used to establish the best performance of the base case. Another interesting issue is whether the strategies presented here can

be adapted to work with special hardware, and if so how their relative performance will differ.

## 8.2 Multiprocessors

The recovery performance model presented in Chapter 4 does not distinguish between uni- and multiprocessor transaction processors. Instead, the transaction processor is treated as a generic resource capable of supplying processor cycles. In reality, differences can be expected between single and multiple processor implementations. The question of how best to use a multi-processor MMDB is an interesting one that remains to be studied.

One alternative is to functionally segregate the processors, e.g. the checkpointing, logging, and transaction execution tasks could be assigned to different processors. This approach has been taken in a transaction processing system built recently at Princeton on Firefly multiprocessors [Li88a]. Alternatively, processors could be treated uniformly, with all processors performing (potentially) all functions. The latter approach has the advantage of even distribution of the workload, even if the different functions differ greatly in their demand for processor time. One the other hand, functional segregation can reduce the task switching overhead that is present processors perform multiple duties.

## 8.3 Concurrency Control

The same concurrency control algorithms used in a DTPS can be used in a MTPS. However, the best strategy may not be the same one in both cases. In a MTPS, most transactions have much shorter lifetimes than they would in a disk-based system because I/O delays are no longer in their critical path. Furthermore, the degree of multiprogramming, i.e., the number of active transactions, in a MTPS can be kept much lower than in a disk-based system without decreasing the utilization of the system's processing power. One comparison of concurrency control in disk and main-memory databases has appeared in [Lehm86a].

Another interesting concurrency issue for MTPS is *long-lived* transactions, i.e., transactions whose lifespans are much greater than most others running in the system. Long-lived transactions are a problem in disk-based as well as main-memory, databases. Arguably, however, the problem will be more widespread in the case of main-memory. In a MTPS, most transactions

will be very short, much shorter than in a disk-based system. As a result, *unusual* transactions, e.g., those which perform special activities such as user interaction, disk access (see Section 8.3 below), or device control during their lifetimes, will be relatively longer-lived in a MTPS than in a disk-based system. Handling such *long-lived transactions* gracefully has been a topic of interest for some time [Gray81a].

We have considered two approaches for coping with long-lived transactions: *sagas* and *altruistic locking* protocols. A saga [Garc87a] is a particular class of long transaction that appears common in many transaction processing applications. By replacing the normal atomicity guarantee of a transaction with a weaker (but nonetheless useful) *semantic atomicity guarantee*, sagas can be scheduled much more freely than other types of long transactions.

The second approach retains all of the guarantees normally associated with transactions. Instead, additional concurrency is gained by employing a locking protocol than can take advantage of access pattern information provided by transactions. By using knowledge of what data a transaction will no longer access, and what data it may access in the future, these altruistic locking protocols [Sale87a] can permit greater concurrency than other locking protocols without sacrificing the serializability of the transaction schedule. The altruistic protocols are general purpose, i.e., they can be used to schedule any set of transactions, not just those with long-lived transactions. However, the increase in concurrency that can be achieved with the protocol is likely to be greatest when there is a mix of long and short transactions.

## 8.4 The "Almost" MTPS

It is likely that there will always be some databases that cannot feasibly be kept entirely memory-resident. However, by keeping the most frequently accessed portions of the database in memory we may be able to achieve most of the performance possible with a MMDB without eliminating access to the remainder of the data. Some interesting aspects of "almost" memory-resident databases are discussed in [Garc87b]. The transaction processing system for such a database behaves like a DTPS in that it caches the frequently accessed portions of the database in memory for fast access. However, because of the large amount of main memory, the recovery manager behaves like a MTPS, using an asynchronous checkpointer to update the secondary

database copy.

**References**

Acce83a.

Accetta, Mike, Robert Baron, Willian Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernal Foundation For UNIX Development," unpublished draft, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA, May, 1983.

Agra85a.

Agrawal, Rakesh and David J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, vol. 10, no. 4, pp. 529-564, December, 1985.

Bitt87a.

Bitton, Dina, Maria Butrico Hanrahan, and Carolyn Turbyfill, "Performance of Complex Queries in Main Memory Database Systems," *Proceedings of the Third Int'l. Conference on Database Engineering*, pp. 72-81, Los Angeles, CA, February, 1987.

Coopa.

Cooper, Eric C., "C Threads," unpublished report, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA. Implementation overview and manual.

DeWi84a.

DeWitt, David J., Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood, *Implementation Techniques for Main Memory Database Systems*, ACM, 1984.

Eich86a.

Eich, Margaret, "Main Memory Database Recovery," *Proc. ACM-IEEE Fall Joint Computer Conference*, 1986.

Eich87a.

Eich, Margaret, "A Classification and Comparison of Main Memory Database Recovery Techniques," *Proc. 3rd Int'l Conf. on Data Engineering*, pp. 332-339, Los Angeles, CA, February, 1987.

Garc83a.

Garcia-Molina, Hector, Richard J. Lipton, and Peter Honeyman, "A Massive Memory Database System," unpublished report, Dept. of Elec. Eng. and Computer Sci., Princeton University, Princeton, NJ, September, 1983.

Garc87a.

Garcia-Molina, Hector and Kenneth Salem, "High Performance Transaction Processing with Memory Resident Data," *Proc. Int'l. Workshop on High Performance Computer Systems*, Paris, December, 1987.

Garc87b.

Garcia-Molina, Hector and Kenneth Salem, "Sagas," *Proc. ACM SIGMOD Annual Conference*, pp. 249-259, San Francisco, CA, May, 1987.

Gawl85a.

Gawlick, Dieter and David Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *Data Engineering Bulletin*, vol. 8, no. 2, pp. 3-10, June, 1985.

Gray78a.

Gray, Jim, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. G. Seegmüller, pp. 393-481, Springer-Verlag, 1978.

Gray81a.

Gray, Jim, "The Transaction Concept: Virtues and Limitations," *Proceedings of the Seventh Int'l. Conference on Very Large Databases*, pp. 144-154, IEEE, Cannes, France, Sept., 1981.

Haer83a.
Haerder, Theo and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery," *Computing Surveys*, vol. 15, no. 4, pp. 287-317, ACM, December, 1983.

Hagm86a.
Hagmann, Robert B., "A Crash Recovery Scheme for a Memory-Resident Database System," *IEEE Transactions on Computers*, vol. C-35, no. 9, pp. 839-843, September, 1986.

Lehm86b.
Lehman, Tobin J., "Design and Performance Evaluation of a Main Memory Relational Database System," CS Technical Report #656, Computer Sciences Department, University of Wisconsin, Madison, WI, August, 1986.

Lehm85a.
Lehman, Tobin J. and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc. Int'l Workshop on High Performance Transaction Systems*, Asilomar, CA, September, 1985. also, CS Technical Report #605, Computer Sciences Department, University of Wisconsin, Madison, WI, July, 1985.

Lehm86a.
Lehman, Tobin J. and Michael J. Carey, "Query Processing in Main Memory Database Management Systems," *Proc. ACM-SIGMOD Conference*, pp. 239-250, Washington, DC, 1986.

Lehm87a.
Lehman, T. J. and M. J. Carey, "A Recovery Algorithm for a High-Performance Memory-Resident Database System," *Proc. ACM SIGMOD Annual Conference*, pp. 104-117, San Francisco, CA, May, 1987.

Li88a.Li, Kai and Jeffrey Naughton, "Multiprocessor Main Memory Transaction Processing," CS-TR-159-88, Department of Computer Science, Princeton University, Princeton, NJ, June, 1988.

Pu86a.Pu, Calton, "On-the-Fly, Incremental, Consistent Reading of Entire Databases," *Algorithmica*, no. 1, pp. 271-287, Springer-Verlag, New York, 1986.

Reut80a.
Reuter, Andreas, "A Fast Transaction-Oriented Logging Scheme for UNDO Recovery," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 4, pp. 348-356, July, 1980.

Reut84a.
Reuter, Andreas, "Performance Analysis of Recovery Techniques," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 526-559, December, 1984.

Rose78a.
Rosenkrantz, Daniel J., "Dynamic Database Dumping," *Proc. SIGMOD Int'l Conf. on Management of Data*, pp. 3-8, ACM, 1978.

Sale86a.
Salem, Kenneth and Hector Garcia-Molina, "Crash Recovery Mechanisms for Main Storage Database Systems," CS-TR-034-86, Dept. of Computer Science, Princeton University, Princeton, NJ, 1986.

Sale87a.
Salem, Kenneth, Hector Garcia-Molina, and Rafael Alonso, "Altruistic Locking: A Stratagy for Coping with Long-Lived Transactions," *Proc. 2nd Int'l Workshop on High Performance Transaction Systems*, Asilomar, CA, September, 1987.

Ston87a.
Stonebraker, Michael, "The Design of the POSTGRES Storage System," *Proc. 13th VLDB Conference*, pp. 289-300, Brighton, England, 1987.

Vigu87a.
Viguers, Dave, "IMS/VS Version 2 Release 2 Fast Path Benchmark (ONEKAY)," *Proc. of*

*the Second Int'l Workshop on High Performance Transaction Systems*, Asilomar, CA, September, 1987.