CHECKPOINTING MEMORY-RESIDENT DATABASES

Kenneth Salem
Hector Garcia-Molina

CS-TR-126-87

December 1987

Revised June 1988

Checkpointing Memory-Resident Databases

Kenneth Salem
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

A main memory database system holds all data in semiconductor memory. For recovery purposes, a backup copy of the database is maintained in secondary storage. The checkpointer is the component of the crash recovery manager responsible for maintaining the backup copy. Ideally, the checkpointer should maintain an almost-up-to-date backup while interfering as little as possible with the system's transaction processing activities. We present several algorithms for maintaining such a backup database, and compare them using an analytic model. Our results show some significant performance differences among the algorithms, and illustrate some of the tradeoffs that are available in designing such a checkpointer.

Checkpointing Memory-Resident Databases

Kenneth Salem
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

1. Introduction

The cost per bit of semiconductor memory is decreasing and chip densities are rising. As a result of these trends, researchers have begun to consider database systems in which all of the data resides in main (semiconductor) memory.† Memory-resident data can mean large performance gains for database systems. In current systems, much of a transaction's lifetime is spent waiting to access data on disks. In addition, much of the complexity of the database system itself can be attributed to the long delays associated with the disks.

The simplest way to design a main memory database management system (MMDBMS) is to borrow the design of a disk-based database manager. A MMDBMS can be viewed as a disk-based DBMS 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 DBMS with memory-resident data in mind. Some that have been considered are index structures [Lehm85a, DeWi84a, Thom86a], query processing [Lehm86a, Bitt87a, DeWi84a] and (primary) memory management [Eich87a].

One component of a DBMS that might be particularly difficult to transfer from a disk-based to a memory-resident system is the recovery manager. From the point of view of the recovery manager, there are several interesting aspects of memory-resident

† We do not rule out the existence of slow archival storage. One can think of a system as having two databases (as in IMS Fastpath [Gawl85a]): one memory-resident that accounts for the vast majority of accesses, and a second on archival storage [Ston87a]. In this paper we focus on the main memory database since its performance is critical.

databases:

- At recovery time, the focus of the recovery manager must be the restoration of the primary (memory-resident) database, rather than the disk-resident database, to a consistent state. Since the primary database can be lost during a failure (e.g., a memory failure or power loss), it must be reconstructed from a backup copy on secondary storage.
- In a MMDBMS, the transactions' data requirements can be satisfied without disk I/O. However, to manage the backup database the recovery manager requires access to disks (or other non-volatile storage). The recovery manager's I/O requirements should be satisfied without sacrificing the performance advantages that memory resident data can bring to transaction processing. In particular, this means that the recovery manager should do as little *synchronous* I/O as possible. Such practices as forcing transaction updates to disk before commit, and flushing dirty pages to disk (while transactions wait) at checkpoint time should probably be avoided.
- 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 disk-based system that costs about 20,000 instructions (without recovery) and makes 20 database references, half of them updates. In a memory-resident system, that same transaction may cost only half as many instructions. 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 the performance of the recovery manager will be more critical to the overall performance of a DBMS when data is memory resident than when it is disk resident.

In this paper we will focus on one critical aspect of crash recovery in a MMDBMS, namely the maintenance on disk of the up-to-date secondary copy of the database. We term this process *checkpointing*, although checkpointing may be realized quite differently in a MMDBMS than in a disk-based DBMS. We will describe a number of possible

algorithms for *asynchronous* checkpointing, and compare them using a simple analytic model.

An interesting feature of a MMDBMS 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. Similarly, I/O latency should not be a problem if I/O is done asynchronously, because asynchronous I/O is not likely to be in the critical execution path of any transaction. Thus evaluating "I/O cost", as is commonly done for disk-based systems, is not a good way of measuring the impact of checkpointing on transaction processing in a MMDBMS. This is not to say that the I/O bandwidth is not important to the system's performance. As we will see, it affects recovery time in a number of ways.

What does appear to be a useful checkpointing performance metric in a MMDBMS is processor overhead. Checkpointing processor overhead results from such activities as initialization of disk I/O's, data movement, and locking or other synchronization with transaction processing activities. Checkpointing can also indirectly affect the overhead costs of other system activities, such as logging. The fact the CPU costs rather than I/O costs may be the critical performance factors in a MMDBS is one of the reasons we believe the model presented here is important.

Several algorithms for asynchronous maintenance of a secondary database copy have appeared in the literature [DeWi84a, Eich87a, Hagm86a, Lehm87a, Pu86a]. The checkpointing algorithms that we will consider are based on ideas drawn from that work. Our emphasis in this paper is algorithmic alternatives. We have not considered checkpointing mechanisms that rely on the existence of special purpose or functionally segregated processors, nor those that require large quantities of stable primary memory. However, in Section 6 we will consider the effect of a stable log tail, i.e., the availability of enough stable RAM to hold the in-memory portion of the log.

The rest of the paper is organized as follows. In the next section we discuss the assumptions made and the failures considered. In Sections 3 and 4 we describe the checkpointing algorithms. Section 5 presents our performance model and analysis, while Section 6 gives some of results.

2. Assumptions

The hardware underlying the MMDBMS consists of one or more processing units (CPUs), volatile memory, and disks, all linked by one or more data channels. There is enough primary memory to hold a complete copy of the database (the primary copy) plus any additional data structures that are required by the system, e.g., page tables.

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.

During a checkpoint, only those portions of the database that have been updated are written out to their corresponding position on 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 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.

Finally, we assume that transactions use a *shadow-copy* update scheme similar to that employed by IMS/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 (copying) 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.

It is important to note that there are other approaches besides shadow-copy updates and ping-pong backups. We are not arguing that the two selected are the best; we are simply choosing representative and reasonable alternatives so we can study checkpointing algorithms independently of these other components.

2.1. Failures

There are numerous types of failures that can occur in a transaction processing system. We will concentrate on recovery from *transaction failures* and *system failures*. As defined in [Gray78a], a transaction failure occurs when a particular transaction must be aborted, either because of some internal condition or because of external intervention. We are particularly concerned with transactions that fail as a result of actions of the checkpointer. The probability of a checkpoint-induced failure, $p_{restart}$, will be computed in Section 5 as a function of the checkpoint algorithm.

A system failure results in the halt of the system and the loss of the contents of volatile memory, followed by system restart. One of the performance measures we consider is the time for recovery from a system failure. The recovery time is discussed in more detail in Section 4.

In our model we do not explicitly consider recovery from *media failures* [Gray78a]. Provided there is extra memory available, and provided that the failed portion of memory can be mapped-out transparently to the database system, media failures in primary memory can be treated like system failures. There are also interesting aspects to secondary media failures in a MMDBMS. Recovery from such failures may be easier than in a DBMS because the lost data will be available in primary memory (provided that a system failure does not occur simultaneously). Dumping of the backup database (e.g., to tape) may also be easier because of the more predictable disk access patterns of a MMDBMS. We will not discuss these issues further here.

3. Checkpointing Algorithms

In this section we describe a number of checkpointing algorithms. The algorithms 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 transaction 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.

3.1. Fuzzy Checkpoint Algorithm

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 algorithm FUZZY. It begins a checkpoint by entering a *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 flushes dirty segments from main memory to secondary storage. Locks and other transaction activity are ignored. Once the dirty segments have been flushed, the (in-memory) log tail is flushed to disk and the new current checkpoint is noted (as described in section 2).

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 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 [Sale87a].)

---

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

3.2. Black/White Algorithms

One way to produce a consistent backup is to treat the checkpointing process as a (long-lived) transaction. The checkpointer acquires a lock on each segment before flushing and holds the locks until the checkpoint is complete. We assume that 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]. The algorithms we will describe next are variants of the mechanism proposed in that paper.

The basic algorithm described in [Pu86a] proceeds as follows. There is a "paint bit" for each database segment which is used to indicate whether or not a particular 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 algorithm in Figure 3.1.

The algorithm 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 single transaction may contain both black-accessing and white-accessing actions.) A transaction is aborted if any one of its actions attempts to access both white and black records.

The "processing" of a segment can occur in two ways. One option is to simply schedule the segment to be flushed to the backup disks. The checkpointer locks each segment for the duration of the disk I/O operation. We call this type of checkpointer *BW/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 for this advantage. When

```
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 3.1 - Black/White Checkpoint

checkpointing is handled in this fashion we say that the checkpoint style is *BW/COPY*.

3.3. Copy-on-Update Algorithms

Copy-on-update checkpointing forces transactions to save a consistent "snapshot" of the database, for use by the checkpointer, as they perform updates. The principal advantage of copy-on-update (COU) checkpointing is that once the checkpoint has started, it will not cause transactions to abort, as do the black/white algorithms. 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). In this case it is almost as easy to get a TC state as an AC state[†], so we will only consider TC COU checkpoints in the rest of the paper. 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.)

---

† This is true because we are assuming database updates are not installed until the commit point. If updates are installed before commit, AC states are easier to achieve than TC.

When the database is quiescent a begin-checkpoint record is written to the log, and the log tail is flushed to stable 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 begin-checkpoint entry is in the log, transaction committing can resume.

The algorithm uses a paint bit per segment in much the same way the black/white algorithm (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 3.2. (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 from S
        unlock S
        IF S' is dirty THEN
            flush S' to the backup
        free S'
      ELSE
        process S
        unlock S
END_WHILE
```

Figure 3.2 - 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. A pointer in the segment is set to point at the newly-created copy.

When the checkpointer processes a segment which does not have a "snapshot" copy it has two options, much as the black/white checkpointer did. 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. The former strategy will be termed COU/FLUSH, and the latter COU/COPY. (Note that if segment S already has a snapshot copy S', then neither locking for the duration of the I/O nor copying is necessary.)

4. System Failure Recovery

After a system failure, the recovery manager has at its disposal a backup copy of the database and a transaction log on stable storage. In a disk-based system, the log is used to bring the stable database copy to a consistent state. In a MMDBMS, 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 (as discussed in Section 2), 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. (With the FUZZY algorithm, the log must be scanned backwards a short ways from the begin-checkpoint marker to retrieve all updates made by transactions that were committing at the time the checkpoint started.) From that point the log is scanned forwards. If the log is a value log, new values of modified records are written in place in primary memory. Otherwise, the logged actions are rerun against the database[†].

5. Performance

In this section we consider the performance of the various checkpoint algorithms that were presented. The performance metrics that we will consider are processing overhead and recovery time (from system failures).

---

† In general, actions may not be idempotent. We assume that an identifier is associated with the log entry for each action (much like a *log sequence number* [Gray78a]). This identifier is stored with each segment affected by the action and is used to ensure that an action is applied exactly once to a segment.

5.1. Performance Model

The first step is to model the hardware, database, and the arriving transactions. Each CPU is able to perform certain operations at a cost of some instruction executions. A discussed earlier, synchronization (for consistent checkpoints) is accomplished through locking. C_{lock} is the cost to lock and unlock a database object. Storage management costs are represented by C_{alloc}, which is charged for allocating (and later freeing) of a block of memory. 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. The cost of data movement in memory is taken to be proportional to the number of words moved, with constant of proportionality one instruction per word. (Note that the number of processors is not needed since we will only compute overhead per transaction.)

The parameters we have described, together with the rest we will cover, are summarized in Table 5a. The table also lists the default values used. We believe that the default values are realistic, at least for some types of hardware and applications. Of course, other values are possible. In the next section we will explore the sensitivity of our results to variations in some of the more critical parameters.

symbol	parameter	default	units
C_{lock}	(un)locking overhead	50	instructions
C_{alloc}	buffer (de)allocation overhead	200	instructions
C_{io}	I/O overhead	1500	instructions
T_{seek}	I/O delay time	0.03	seconds
T_{trans}	transfer time constant	3	μseconds/word
N_{bdisks}	number of backup disks	20	disks
N_{ldisks}	number of log disks	5	disks
S_{db}	database size	256	Mwords
S_{rec}	record size	32	words
S_{seg}	segment size	8192	words
S_{lent}	log entry overhead	4	words
λ	arrival rate	1000	transactions/second
p_{fail}	failure probability	0.05	no units
N_{act}	number of actions	5	actions/transaction
R_{spa}	segments per action	1.1	segments/action
C_{trans}	transaction processor cost	10000	instructions

Table 5a — Parameters and their defaults

The disks are used to hold the secondary database copies and for logging. N_{bdisks} is number of disks available for backups. Disks are modeled as simple servers that can transfer d words of data in time $T_{seek} + T_{trans}\, d$. We assume that the transfer bandwidth scales linearly with the number of disks, i.e., we do not consider interference caused by bus contention or secondary reference locality† Note that I/O to the backup disks in a MMDB is likely to be better behaved than I/O in a disk-based system since I/O in a MMDB is done only by the checkpointer. Thus we might expect seek delays to be somewhat shorter for a MMDB than for a disk-based system.

The database is assumed to contain S_{db} words of data, grouped into records of size S_{rec}. The record is the granule at which the transaction interface operates, i.e. the primitive actions of a transaction are record reads and writes. Records are stored on larger physical blocks, called *segments*, for efficient transfer to the backup disks. S_{seg} is the segment size, which can be any multiple of S_{rec}. S_{lent} is the space overhead of each log entry.

For simplicity we assume that all transactions running against the database are identical. They are assumed to arrive at the system at the rate of λ transactions per second. With probability p_{fail} transactions voluntarily abort (e.g., insufficient funds found in account). The model treats the execution of a transaction, much like a basic operation. The cost of executing a transaction is C_{trans}. This is the cost of executing the transaction *exclusive of recovery costs*, i.e., as if the transaction were running in a failure-free environment.

Each transaction consists of N_{act} actions, each of which modifies a single record. In many cases, the modifications made by an action will be contained within a single segment. However, in other cases, a single action may affect more than one segment. (For example, if the record update makes it grow, it may have to be moved to another segment.) We let R_{spa} represent the expected number of segments that will be modified by each action. The update probability is distributed uniformly across all of the database segments.

Note that we have chosen a R_{spa} relatively close to 1. This is because we assume that secondary indexes are not checkpointed (and their changes are not logged). In a

---

† This may require a high performance IO subsystem. An example of such a subsystem is found on the Convex C-1. It can support up to 160 I/O controllers though five buffered I/O processors onto an eighty megabyte per second bus to the main memory [Dozi84a].

memory resident database, secondary indexes can easily be recreated as the database is loaded up after a failure. Thus, there is no need to consider them part of the recoverable database. In a conventional system, this is not true and we would expect R_{spa} to be larger. In any case, we will consider the impact of R_{spa} in Section 6. Also note that a uniform distribution for segment updates is in a sense a "worst case" assumption. If some type of hot spots exist, the number of segments dirtied during a checkpoint interval will be less, and the checkpointer will have less work to do.

5.2. Performance Analysis

To compute the cost of running each transaction we compute a *synchronous* overhead (i.e., extra work done while running a transaction) and an *asynchronous* overhead, the cost of checkpointing the database. Synchronous overhead represents recovery-related work done for each transaction (e.g., logging overhead). Asynchronous overhead is the cost of making a database checkpoint. We combine both overheads into a single cost measure (instructions per transaction) with the equation

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

where t_{icp} is the intercheckpoint interval and λ is the transaction processing rate.

The value t_{icp} is a model parameter, but it must be greater than t_{icpmin}, the minimum intercheckpoint interval that gives the system a chance to flush all dirtied pages. We now discuss how t_{icpmin} is computed. We start by computing the number of pages that are dirtied in an arbitrary time period, $N_{dirty}(t)$. Transactions update $N_{su} = N_{act} R_{spa}$ random segments. The probability that a particular segment will not be touched by the λt transactions executing in the period is

$$1 - \left[1 - \frac{N_{su}}{S_{db} / S_{seg}} \right]^{\lambda t}$$

The expected number of dirtied segments, $N_{dirty}(t)$ will be the above quantity multiplied by the number of segments, S_{db} / S_{seg}.

The number of segments that can be flushed by the checkpointer in an interval t is given by

$$N_{io}(t) = N_{bdisks} \frac{t}{T_{seek} + T_{trans} S_{seg}}.$$

In the minimum interval t_{icpmin}, a ping-pong checkpointer must flush all dirtied pages in the current and previous period (see Section 2). Thus,

$$N_{io}(t_{icpmin}) = N_{dirty}(2t_{icpmin}).$$

By solving this equation we determine t_{icpmin}. As discussed above, the actual checkpoint duration may be made longer than the minimum by inserting a delay between the completion of a checkpoint and the initiation of the next.

The synchronous costs are computed by

$$C_{synch} = (1 - p_{fail})C_{succ} + p_{fail}C_{fail} + p_{restart}(\frac{C_{trans}}{2} + C_{fail}),$$

where p_{fail} is the probability that the transaction aborts on its own, $p_{restart}$ is the probability that it must be restarted due to a checkpointer conflict, C_{succ} is a transaction's synchronous overhead during normal execution, and C_{fail} is the overhead in case it fails. Note that when a transaction is aborted by a restart we charge the expected wasted processing $C_{trans}/2$ to overhead (this assumes that transactions fail halfway through).

The synchronous overheads C_{succ} and C_{fail} include the costs of log maintenance and of making "snapshot" copies of segments (COU checkpoints only). Log costs are assigned for allocating, flushing, and copying data to the log, in proportion to the log bulk per transaction. For logical or value logging, each updated record must be written, including S_{lent} overhead. For value logs only, any incidental changes to segments (that occur when records are installed) must also be logged. We assume this involves S_{lent} extra words for each of the $R_{spa}N_{act}$ segments dirtied by each transaction. Unsuccessful transactions have no log bulk, since no log entries are made until commit.

Unless black/white checkpointing is used, $p_{restart}$ is zero. (FUZZY and COU checkpoints never cause transaction aborts.) For the TC black/white checkpointers, we compute $p_{restart}$ as follows. Let W be the fraction of the database colored white, and let $Pr[OK \mid W{=}\omega]$ be the probability that a transaction executes without being aborted for violating the color rule, given that $W{=}\omega$. (We assume that W remains constant throughout the execution of the transaction, reasonable when the database is large and the transaction small.) This occurs when all segments touched by the transaction are the same color, so

$$Pr[OK \mid W{=}\omega] = \omega^{N_{su}} + (1-\omega)^{N_{su}}$$

If checkpoints are always occurring ($t_{icp} = t_{icpmin}$) we can assume that W is uniformly distributed from zero to one. By integrating the above expression we can show that

$$Pr[OK] = \frac{2}{N_{su} + 1}$$

The restart probability, $p_{restart}$, is simply 1 - $Pr[OK]$. If $t_{icp} > t_{icpmin}$ then we multiply the value obtained by t_{icpmin}/t_{icp}, the fraction of the time the checkpointer is in operation and transaction in danger of being aborted.

For AC black/white checkpoints, we are interested in the probability that a single action violates the two color rule. An action affects R_{spa} segments on the average, so (by a similar calculation) we get

$$Pr[OK] = \frac{2}{R_{spa} + 1}$$

The asynchronous overhead costs, C_{asynch}, arise from the activities of the checkpointer. Consistent checkpointers lock and unlock each database segment at a cost of C_{lock} per segment. Dirty segments are flushed to the backup at a cost of C_{io} per segment. In addition, spooling checkpointers copy dirty segments before the I/O at a cost of S_{seg}. The number of dirty segments per checkpoint ($N_{dirty}(2t_{icp})$) was calculated earlier. We omit further details here, but the complete analysis is presented in [Sale87a]

The checkpoint duration is also used to determine recovery time, the other performance metric. Recovery time has a number of different components. The failure must be detected, the disks must be spun-up (if power failed), the backup database and the log must be read in off of the disks, and communications must be restored [Hagm86a]. We will consider only the restoration of the database from the backup and the log in our measure of response times. The other components, while possibly introducing significant delays, are not likely to be affected by the transaction processing system.

We assume that recovery time is dominated by I/O time. In particular, we take the recovery time to be the time necessary to read the backup database copy into main memory, plus the time to read the appropriate portion of the log. The time to read in the backup copy (or log) is determined by the size of the database (or log) and the bandwidth to the backup (or log) disks. The log size is computed as $t_{icp}\lambda B$, where B is the log bulk per transaction.

6. Results

Figure 6a shows processor overhead and recovery time for each of the checkpointing algorithms. The data were obtained assuming that the checkpoints duration was as short as possible (no time between checkpoints) and using the basic operation costs given in Section five.

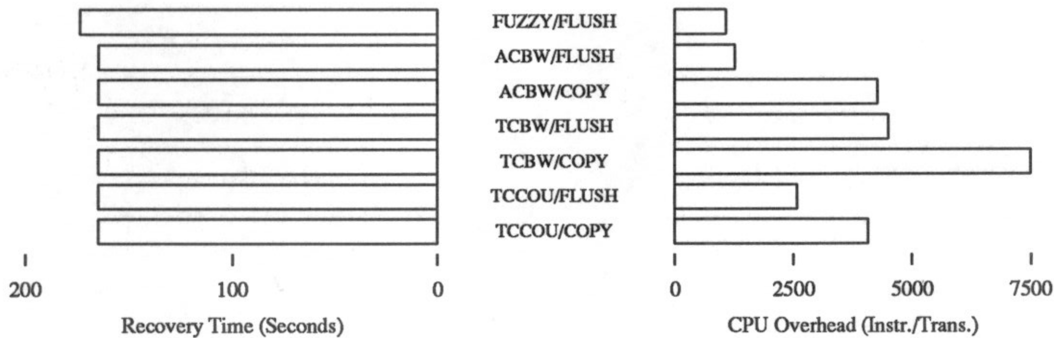

Figure 6a - Processor Overhead and Recovery Time

Several points are apparent from Figure 6a. Most obvious is the relatively high cost of the TC black/white checkpoint algorithms compared to the corresponding COU algorithms. Most of the additional cost comes from rerunning transactions that are aborted for violating the black/white 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 algorithms lock segments for shorter periods, but this is not reflected in our overhead metric.

AC checkpointing (with logical logging) can be done almost as cheaply as FUZZY. Though the FUZZY algorithm need not lock pages and never causes transaction aborts, ACBW/FLUSH does not cause too many aborts and is able to take advantage of less-costly logical logging. As we shall see shortly, this gap widens as actions become more complex (access more segments).

Recovery times vary little. The slightly longer time for the FUZZY algorithm arises from the greater log bulk (per transaction) of value logging. This difference would be much greater if R_{spa} were larger, i.e., if activities such as secondary index modifications

had to be logged (in a value log). (See Section 5.1.)

Although recovery times do not vary significantly with changes in the checkpoint algorithm, they can be made to vary by controlling the checkpoint duration. In fact, for a given checkpoint algorithm there is a tradeoff between processor overhead and recovery time than can be controlled by varying the checkpoint duration. This tradeoff is illustrated in Figure 6b for two of the checkpoint algorithms, TCBW/FLUSH and TCCOU/FLUSH.

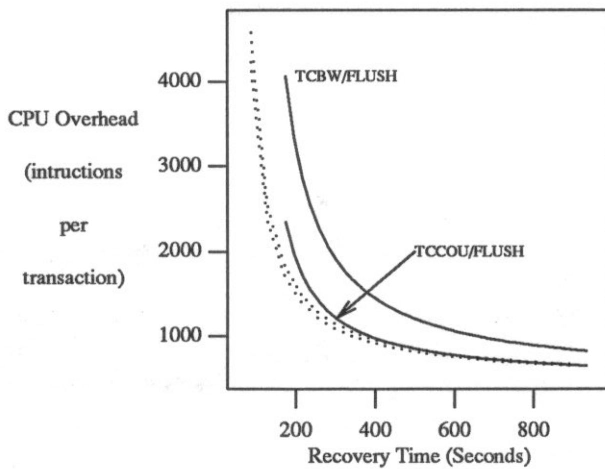

Figure 6b - Processor Overhead/Recovery Time Tradeoff

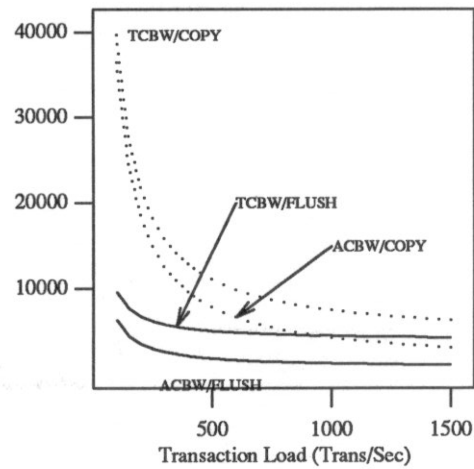

Figure 6c - Effect of Varying Transaction Load

The two solid curves represent the trajectory of TCBW/FLUSH and TCCOU/FLUSH through the processor overhead/recovery time space as the 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 adding more disks). 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 algorithm 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 6c describes the effect of transaction load, λ, on processor overhead for four of the algorithms. 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 algorithms (dotted lines) are much more expensive at low loads than their non-spooling counterparts. However, at high loads they are more 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.

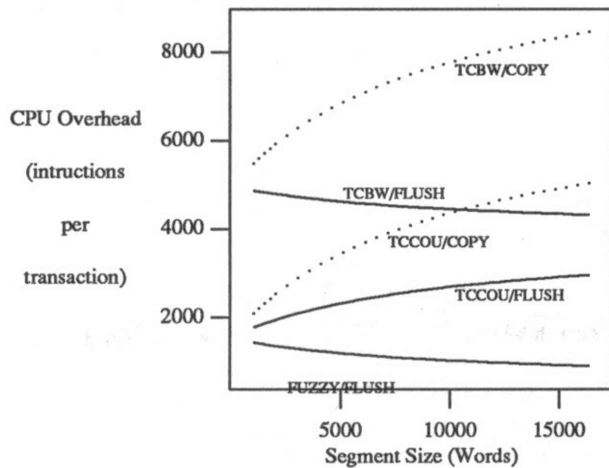

Figure 6d - Effect of Varying Segment Size

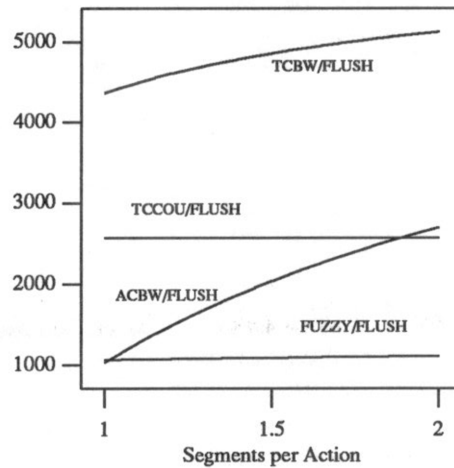

Figure 6e - Effect of Varying Action Complexity

We have already seen that checkpointing overhead can be controlled by varying the checkpoint interval. Figure 6d describes the effect of another parameter, the segment size (S_{seg}), assuming $t_{icp} = t_{icpmin}$. (Recall that segments are the units of transfer to secondary storage.)

The variety of behavior exhibited by the different algorithms 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.

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 algorithms, though recovery times haven't been plotted here.)

Spooling algorithms (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 other non-spooling algorithms is dominated by the first effect and their overhead costs are lower for larger segments.

Finally, Figure 6e looks at the effect of increasing R_{spa}, the complexity of (the number of segments accessed by) a database action. The most strongly affected algorithm is the black/white AC algorithm, 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.

In closing this section, let us consider how the availability of stable main memory to hold the log tail affects the cost of checkpointing. A stable log tail makes it possible to commit transactions without waiting for a log flush. This improves response time somewhat, but does not reduce the CPU overhead of checkpointing. Thus, we expect performance results with a stable log to be essentially equal to those of Figure 6a. In terms of transaction throughput and recovery time, a stable log tail has no advantages.

It is worthwhile pointing out that with other database backup strategies, a stable log can help. For example, if there is a single backup database, then the fuzzy checkpointer we described here can violate the log write-ahead rule. The problem can be avoided with

a stable log tail or by making the checkpointer spool segments, delaying their I/O until the appropriate log pages are on stable storage. Clearly, the stable log alternative reduces the overhead substantially.

7. Conclusions

We have presented a performance model for an important aspect of crash recovery in memory-resident databases. We have used the model to compare several checkpointing algorithms. Our results indicate that there may be significant differences in performance among them.

The absolute and relative performance of checkpointing algorithms is not an intrinsic property of the algorithm. As we have seen, an algorithm's performance depends on the system and environment of which it is a part (e.g., transaction load, checkpoint interval). However, it seems safe to say that fuzzy checkpointing is the most efficient, although it does require pure physical afterimage logging. For consistent checkpoints, a strategy like ACBW+FLUSH seems to have the least overhead. For dual backup database copies, a stable log tail has minimal impact on transaction throughput.

We have considered checkpoint algorithms independently of the other components of the transaction processing system. In [Sale87a], we 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 algorithms may actually prove to be beneficial because they can be used in conjunction with less costly logging or storage management techniques.

We are currently implementing a testbed with which we will be able to experimentally evaluate the algorithms presented here, as well as other aspects of crash recovery in memory-resident databases. We hope to able to measure synchronization and other delays using the testbed, as well as to verify the processor overhead and recovery time models used here.

References

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.

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.

Dozi84a.
Dozier, Harold and et al, "Super Supercomputer!," *Computer Systems Equipment Design*, pp. 17-22, November, 1984.

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.

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. Seegm"uller, pp. 393-481, Springer-Verlag, 1978.

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.

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.

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

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

Sale87a.
Salem, Kenneth and Hector Garcia-Molina, "Crash Recovery for Memory-Resident Databases," CS-TR-119-87, Dept. of Computer Science, Princeton University, Princeton, NJ, 1987.

Ston87a.
Stonebraker, Michael, "The Design of the POSTGRES Storage System," *Proc.*

13th VLDB Conference, pp. 289-300, Brighton, England, 1987.

Thom86a.
Thompson, William C., III, ''Main Memory Database Algorithms for Multiprocessors,'' PhD Dissertation, University of California, Davis, CA, June, 1986.