CHECKPOINTING MEMORY-RESIDENT DATABASES

Kenneth Salem
Hector Garcia-Molina

CS-TR-126-87

December 1987

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 performance 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, a number of researchers have begun to re-examine some of the components of a traditional DBMS with memory-resident data in mind. Some of the components that have been considered are index structures [Lehm85a, DeWi84a, Thom86a], query processing [Lehm86a, Bitt87a, DeWi84a], and (primary) memory management [Eich86a].

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 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

† 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.

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 the checkpointer 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 the processor overhead of the checkpointer, since processors are critical resources shared by both the checkpointer and transactions. The checkpointer's processor overhead is produced by a number of different activities, including the initialization of disk I/O's, data movement, and locking or other synchronization with transaction processing activities. The fact the CPU costs rather than I/O costs may be the critical performance factors is another interesting aspect of recovery management in a MMDBS, and 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, Eich86a, Hagm86a, Lehm87a, Pu85a]. 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 4 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 present our model of the database, the system architecture, and transactions. Section 3 describes the various checkpointing algorithms. In Section 4 we present the results of our comparisons. Section 5 presents a summary of our results, and some conclusions.

2. System and Load Models

In this section we describe our models of the components of the system that impact checkpointing. In particular we will describe the system's hardware resources (processors and storage) and the structure of the database. We also present a simple model of the transaction load, and describe the types of failures that will be considered. Finally, we describe our assumptions about related system activities, such as logging and storage management. Details of the model can be found in [Sale87a].

The hardware underlying the MMDBMS consists of processors, volatile main memory, and disks, all linked by one or more data channels. The next several sections are devoted to descriptions of our models of each of these components.

2.1. Processors

The system includes one or more processing units (CPUs). We model the collection of processors as a single server which is able to perform certain operations at a cost of some number of instructions per operation. The basic operations are synchronization, data movement (within primary memory), I/O initiation, and storage management, i.e., dynamic allocation and deallocation of storage.

Table 2a describes the model parameters related to the basic operations, and gives their default costs. Synchronization is accomplished through locking and log sequence numbers (LSN) [Gray78a]. C_{lock} is the cost of each lock or unlock operation. C_{lsn} is the cost of checking or maintaining a log sequence number (LSN). C_{lsn} is charged (under some checkpoint policies) to update a LSN when a transaction makes an update, and to check a LSN when the checkpointer flushes data to the backup disks. Storage management costs are represented by C_{alloc}, which is charged for the dynamic (de)allocation 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.

Not shown in the table is the cost of the final operation, data movement. The cost of data movement is taken to be proportional to the number of words moved, with constant of proportionality one instruction per word.

symbol	parameter	default	units
C_{lock}	(un)locking overhead	20	instructions
C_{alloc}	buffer (de)allocation overhead	100	instructions
C_{io}	I/O overhead	1000	instructions
C_{lsn}	maintain LSNs	20	instructions

Table 2a - Basic Operation Costs

2.2. Storage

Primary storage is assumed to be volatile RAM. 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. Secondary storage consists of magnetic disks, although the system may also use other media, such as tapes or optical disks, to store archival data. The disks are used to hold the secondary database copy (and also for logging). N_{bdisks} is number of disks available.

Disks are modeled as simple servers that can transfer d words of data in time $T_{seek} + T_{trans}\, d$. For simplicity 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. Table 2b shows the model parameters related to the disks.

symbol	parameter	default	units
T_{seek}	I/O delay time	0.03	seconds
T_{trans}	transfer time constant	3	μseconds/word
N_{bdisks}	number of disks	20	disks

Table 2b - Disk Model Parameters

2.3. Data Channels

Data movement, in particular the movement of data between various levels of the memory hierarchy, is important to any computer system. Even for a main memory database system, I/O to secondary storage is important since the recovery mechanism relies on it. Too little bandwidth to secondary storage can limit transaction throughput if log I/O becomes a bottleneck, and can increase recovery time if I/O to the backup database is not fast enough.

A number of techniques exist for boosting I/O bandwidth. Multiple secondary storage devices can be used to handle several I/O requests in parallel. Alternatively, secondary storage devices can be interleaved, or striped [Kim86a,Sale86a]. When using striped disks, several devices service a single request in parallel thus decreasing the service time for that request.

Furthermore, it is becoming possible to configure systems to handle the bandwidth these techniques can achieve. The nominal bandwidths for well-known 32-bit buses range from twenty megabytes per second for Motorola's VME bus to over a hundred megabytes per second for Fastbus and the IEEE Futurebus [Borr85a]. Some systems support multiple, buffered channel interfaces and device controllers to prevent I/O bottlenecks. For example, the Convex C-1 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].

The bandwidth requirements of a MMDBMS during normal operation are significant but not outrageous. As a rough estimate, imagine that an entire 1 gigabyte database is to be checkpointed every 100 seconds (fast), requiring ten megabytes per second. To this we must add the bandwidth required for logging. Even if every transaction uses one 1024 word log page, then at 1000 transactions per second (and four bytes per word) logging requires an extra four megabytes per second. Thus, during normal operation, the bandwidth requirement may be on the order of fifteen megabytes per second. Similar bandwidth will be required during recovery.

Thus the I/O problem for MMDBMSs, though not trivial, does appear to be manageable. Because of space limitations, we will assume in this paper that sufficient bandwidth is available to secondary storage and concentrate instead on CPU overhead. In particular, we will assume that the time required to execute a series of I/O operations is inversely proportional to the number of disks that are available. However, we will make note of those checkpoint algorithms whose bandwidth requirements are higher than others'.

2.4. Database

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 grouped into larger units, called *segments*, for efficient transfer to the backup disks. S_{seg} is the segment size, which can be any multiple of S_{rec}. Table 2c summarizes the model parameters related to the database and gives their default values.

2.5. Transactions

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

symbol	parameter	default	units
S_{db}	database size	256	Mwords
S_{rec}	record size	32	words
S_{seg}	segment size	8192	words

Table 2c - Database Model Parameters

second. 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 updates N_{ru} distinct records. The update probability is distributed uniformly across all of the database records. Table 2d summarizes the model parameters related to the transactions.

symbol	parameter	default	units
λ	arrival rate	1000	transactions/second
N_{ru}	number of updates	5	records/transaction
C_{trans}	transaction processor cost	25000	instructions

Table 2d - Transaction Model Parameters

2.6. Other System Components

The checkpointer necessarily interacts with other components of the transaction processing system, such as the logging mechanism. So that we can study checkpointing algorithms independently of these other components, we will make several assumptions about how the other components operate.

Transactions assumed to use a *shadow-copy* update scheme similar to that employed by IMS/Fastpath [Gawl85a]. 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.

Finally, we assume that two backup databases are maintained on the backup 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 two copies. Thus, there is always a complete checkpoint available after a system failure.

2.7. 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* in this paper. 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 4 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 3.3.

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

A DBMS has two components, one that is active during normal transaction processing and one that is active after a system failure. Although all of the checkpoint algorithms we will present in this section affect the design of the former component, they affect the *performance* of both components. Checkpointing affects recovery performance because it affects the amount of work that needs to be done at recovery time. This is one of the more interesting performance tradeoffs in a DBMS: less overhead during normal processing results in longer recovery times from system failures.

As we have already described, the task of the checkpointer is the maintenance of an "almost up-to-date" backup copy of the database on stable storage (the backup disks). The checkpointer runs repeatedly, each time updating the backup database according to the algorithm determined by the checkpoint policy.

Checkpoint algorithms can be distinguished along at least two dimensions. For example, we can consider whether the entire database, or only those portions of the database that have been updated since the last checkpoint, are backed up on each iteration. Checkpoints of the first sort are called *full* checkpoints, those of the latter sort are called *partial*.

We will not discuss full vs. partial checkpointing in great detail as it is rather straightforward. To implement partial checkpoints, database segments can include a dirty bit which is set by transaction updates and cleared by the checkpointer. Checkpointers that produce partial backups have the additional overhead of checking the dirty bit of every database segment, but in general they will flush fewer pages to the backup disks.

Another distinction among checkpointers can be made according to the level of consistency of the backup database they produce. Three of the possibilities are fuzzy, action-consistent (AC), and transaction-consistent (TC). We will only consider fuzzy and TC checkpointing in our study, for reasons that we will discuss shortly.

In the remainder of this section, we discuss the checkpoint algorithms we have considered and discuss some of the implementation questions that arise. The algorithms describe how checkpoints are created. At the end of the section we discuss how the checkpoints are used to recovery from a failure.

3.1. Fuzzy Checkpoints

Fuzzy checkpoints require little or no synchronization with executing transactions. The backup database produced by such a checkpoint is called fuzzy because it may not contain an atomic view of database activities (e.g., storage operations such as reading and writing) that were occurring while the backup database was being produced. For example, if a transaction were updating a database records *R1* and *R2* while a fuzzy checkpoint was occurring, the backup database might contain the new value of *R1* but the old (pre-update) value of *R2* after the checkpoint completes. Fuzzy checkpoints are suggested for recovery in main memory databases in [Hagm86a].

A checkpoint begins by entering a ***begin-checkpoint*** marker in the log, along with a list of currently active transactions. Once the marker is in place, the most straightforward approach to creating the checkpoint is to flush the appropriate segments from main memory to secondary storage. (By appropriate segments, we mean the dirty segments if a partial checkpoint is being taken, or all of the segments if a full checkpoint is being taken.) The checkpointer ignores locks and other transaction activity and simply flushes the segments.

The biggest problem with this approach is that it may lead to violations of the *log write-ahead protocol* [Gray78a] There is no guarantee that records updated by a transaction will not be flushed before the log records for that transaction have been flushed to the log disks. Therefore, if such straightforward fuzzy checkpoints are taken then transactions must delay their updates until their log records have been flushed to disk (an undesirable situation) or stable main memory must be available to hold the log tail. We will delay further discussion of such checkpoints until the next section, when we will consider the effects of a stable log on the performance of the checkpointer.

A fuzzy checkpointing scheme which avoids these problems is FUZZYCOPY. FUZZYCOPY checkpointing is similar to the straightforward fuzzy checkpoint, except that instead of simply flushing segments to the backup disks, segments are first copied into a main memory I/O buffer. The buffered segment copy is not flushed to the backup database until the log records of any updates that are reflected in the segment have been flushed to the log disks. The checkpointer can determine when it is safe to flush the segment copy by using *log sequence numbers* [Gray78a].

3.2. Consistent Checkpoints

The alternative to fuzzy dumps is to produce consistent database backups. As we have already mentioned, such backups can be *action-consistent* (AC) or *transaction-consistent* (TC). Action-consistent backups are more costly to produce than fuzzy backups, and transaction-consistent backups are more costly than either. However, having a consistent backup may mean that less log information needs to be retrieved after a system failure. Another advantage of consistent backups is that they permit the use of logical logging†.

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

We will consider only TC checkpoints, not AC, although AC checkpoints may actually be more practical in a real system. In many ways TC checkpoints can be seen as extreme versions of AC checkpoints. Both require some form of synchronization to ensure that actions are reflected atomically in the checkpoint. The actions are simply more complex or more abstract in the TC case. Thus, many, but not all, of the comparisons we will make between TC and fuzzy checkpoints could be made with qualitatively similar results between AC and fuzzy checkpoints. We will consider two general methods for producing TC checkpoints, and within each of the methods examine two variations in implementation.

3.2.1. Two-Color algorithms

One way to produce a TC backup database is to treat the checkpointing process as a (long-lived) transaction. The checkpointer acquires a read lock on each segment before flushing and holds the locks until it finishes. 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 TC backup copies but requires that locks be held on only one segment at a time is presented in [Pu85a]. The two locking algorithms we will study are variants of the mechanism proposed in that paper.

The algorithm described in [Pu85a] 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. 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.

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. Log sequence numbers are used to determine when the segment can be flushed (as was done under the FUZZYCOPY algorithm). If the checkpointing is handled in this fashion we say that the checkpoint algorithm is *2CFLUSH*. 2CFLUSH checkpointing requires that segments be locked for the duration of a disk I/O operation, plus any delay that might be needed to satisfy the LSN condition.

```
WHILE there are white segments DO
    find a white segment that is not exclusively locked
    IF there are none THEN
        request read (shared) 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 - A Variation of Pu's Basic Checkpoint

An alternative is to first copy the segment to a special buffer and then to flush the buffer to the backup disks. (Again, LSNs are needed.) 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 checkpointing is handled in this fashion we say that the checkpoint style is *2CCOPY*.

3.2.2. Copy-on-Update algorithms

Copy-on-update checkpointing forces transactions to save a TC "snapshot" of the database, for use by the checkpointer, as they perform updates. The principal advantage of COU checkpointing is that once the checkpoint has started, it will not cause transactions to abort, as do the two-color algorithms. However, COU has its own disadvantages. First, transaction processing must be temporarily quiesced each time a checkpoint begins. Second, primary storage is required to hold the TC 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].

† One difference is that the technique in [DeWi84a] is suggested for producing AC, and not TC, backup copies. TC backups can be produced by requiring that the system be transaction-quiescent, rather than action-quiescent, when the checkpointing begins. In reality, this could be problematic if long-lived transactions are common.

The COU technique works as follows. When a checkpoint is to begin, the system is first brought into a transaction-consistent state. This can be accomplished by aborting currently executing transactions, or by simply quiescing the system (i.e., delaying the start of new transactions until all currently executing transactions have completed). The checkpoint is assigned a timestamp ($\tau(CH)$), a begin-checkpoint record is written to the log, and the log tail is flushed to stable storage. The TC database state that exists when transaction processing has been quiesced is the "snapshot" that will be flushed to secondary storage by the checkpointer. Once the timestamp is assigned and the begin-checkpoint entry is in the log, transaction processing can begin again.

Transactions are assigned timestamps when they begin, and database segments are marked with the timestamp of the most recent transaction to update them. When a transaction wishes to update a database segment that has not yet been dumped by the checkpointer and whose timestamp is less than $\tau(CH)$, it first copies the old version of the segment to a special buffer so that the consistency of the snapshot is preserved. A pointer in the segment is set to point at the newly-created old copy in the buffer.

The algorithm requires a main-memory buffer to hold old copies of those segments that are updated while the checkpointer is running. In addition, each segment S has pointer $p(S)$ that can be used to point at an old copy of the segment and a timestamp $\tau(S)$.‡ For ease of presentation, we also assume that database segments are ordered and the checkpointer backs up segments to secondary storage in this order. *CUR_SEG* indicates the segment that has most recently been backed up. The process for a transaction T to update a record R in segment S is summarized in Figure 3.2.

As usual, the checkpointer sweeps through the database checking for dirty segments to flush. If some segment has been dirtied since the checkpoint began, then an old copy of that segment will exist and that copy will be flushed by the checkpointer. Segments that are dirty, but that have not been dirtied since the checkpoint began, do not have old copies. In this case, the checkpointer has the same two options it had under the locking algorithms. It can either lock the segment while it flushes it to the backup disks, or it can lock the segment long enough to copy it to a special buffer and then flush the buffer. In the former case we say that the checkpointing is *COUFLUSH*, in the latter *COUCOPY*. Figure 3.3 summarizes the checkpointing process under a COU algorithm. In the figure, the timestamp of the previous checkpoint is remembered to ensure that dirty segments are not flushed more than once unless they are dirtied

‡ For simplicity we assume that $p(S)$ and $\tau(S)$ are stored and locked with S. However, storing and locking $p(S)$ and $\tau(S)$ separately from S may provide better performance in a real implementation.

```
lock $S$ (and $R$) (exclusive)
IF ($S > CUR\_SEG$) AND ($\tau(S) \leq \tau(CH)$) THEN
        allocate buffer for $S$
        copy $S$ to buffer (including timestamp)
        set $p(S)$ to point at buffer
set $\tau(S) = \tau(T)$
unlock $S$
update $R$
```

Figure 3.2 - Transaction Updates under COU

again.

Note that under the COU algorithms, LSNs need not be maintained to ensure that the write-ahead log protocol is observed. Any updates seen by the checkpointer must have occurred before the checkpoint began. Thus their log records are already in stable storage.

3.3. 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, 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 several different things. First, the log must be scanned backwards until the begin-checkpoint marker of the most recently *completed* checkpoint is found in the log.† In the case of FUZZYCOPY checkpoints, the log must be scanned backwards even further, until the beginning of the earliest transaction in the active transaction list (stored with the begin-checkpoint marker) is found. Then the log is scanned forwards, and new values of modified records are written in place primary memory.

† This can be determined by placing explicit end-checkpoint markers in the log during normal operation of the system, or by skipping the first begin-checkpoint marker encountered during the back-

```
set $CUR\_SEG$ = first segment
quiesce transaction processing
log *begin-checkpoint* record and flush log tail
save timestamp of last checkpoint as $\tau(OLDCH)$
assign new timestamp $\tau(CH)$ to checkpoint
WHILE ($CUR\_SEG \leq$ number of segments in database) DO
    lock $CUR\_SEG$ (exclusive)
    IF ($\tau(CUR\_SEG) < \tau(CH)$) THEN
        IF ($\tau(CUR\_SEG) > \tau(OLDCH)$) THEN
            lock $CUR\_SEG$ (shared)
            IF (COUCOPY checkpoint) THEN
                copy $CUR\_SEG$ to special buffer
                unlock $CUR\_SEG$
                flush special buffer to backup disks
            ELSE
                flush $CUR\_SEG$ to backup disks
                unlock $CUR\_SEG$
    ELSE
        follow $p(CUR\_SEG)$ to old copy of segment, $OLD\_SEG$
        unlock $CUR\_SEG$
        IF ($\tau(OLD\_SEG) > \tau(OLDCH)$) OR (full checkpoint) THEN
            flush old copy of segment to backup disks
END_WHILE
```

Figure 3.3 - COU Checkpointing

Checkpointing affects recovery time by affecting the bulk of the log. The effective log bulk per transaction is increased by checkpointing algorithms (such as 2CCOPY and 2CFLUSH) that cause frequent transaction failures and restarts. The total log bulk is affected by the checkpoint duration. Delays in checkpointing result in a larger log to be processed at recovery time.

ward scan at recovery time.

4. Performance

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

The performance data presented in this section were obtained from the analytic performance model described in [Sale87a]. Briefly, the model calculates processor overhead by calculating *synchronous* overhead (i.e, work done on behalf of a particular transaction), and *asynchronous*, or checkpointing, overhead. Synchronous overhead arises from activities such as writing data into the log, initiating log I/O, and copying segments before update (assuming COU checkpoints are being used). Asynchronous overhead arises from activities of the checkpointer, such as flushing segments to the backup disks. Overhead costs are calculated in terms of the costs for basic operations. Note that in this paper we only consider overhead that is directly related to checkpointing. We do not include the other recovery costs, such as data movement for the creation of the log, which are covered by the model in [Sale87a].

To combine synchronous and asynchronous costs into a single measure, the asynchronous cost is divided by the number of transactions that run during the duration of the checkpoint and then added to the synchronous cost. (The checkpoint duration is the time from the beginning of one checkpoint to the beginning of the next.) The minimum possible checkpoint duration is a function of the bandwidth to the backup disks and the rate at which transactions dirty database segments. The former is estimated using model parameters, and the latter with model parameters and an assumption that record update probability is distributed evenly across all records in the database. 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. Thus, the actual checkpoint duration is a model parameter which can be set to any value greater than the calculated minimum.

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 have not modeled the recovery process itself in detail, as 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 is determined by the size of the database and the bandwidth to the backup disks. As described in the last section, the checkpointer does affect the log reading time by affecting the volume of log data that must be read.

Figure 4a 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 two.

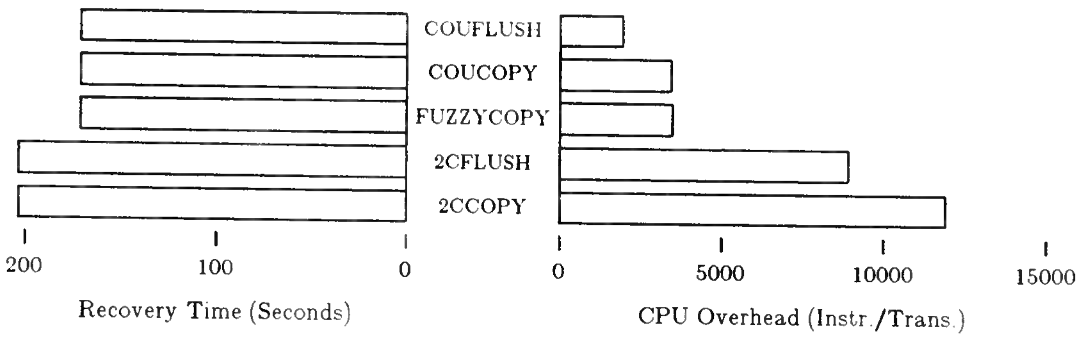

Figure 4a - Processor Overhead and Recovery Time

Several points are apparent from Figure 4a. Most obvious is the relatively high cost of the two-color checkpoint algorithms. Most of the cost comes from rerunning transactions that are aborted for violating the two-color restriction. The figure also shows that generating a transaction consistent backup with a COU algorithm is no more costly than generating a fuzzy backup. Recovery times seem to vary little from among the algorithms. The slightly longer times for the two-color algorithms arises from the added log bulk of transactions aborted by the two-color constraints.

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 trade-off between processor overhead and recovery time than can be controlled by varying the checkpoint duration. This trade-off is illustrated in Figure 4b for two of the checkpoint algorithms, COUCOPY and 2CCOPY. The two solid curves represent the trajectory of the 2CCOPY and

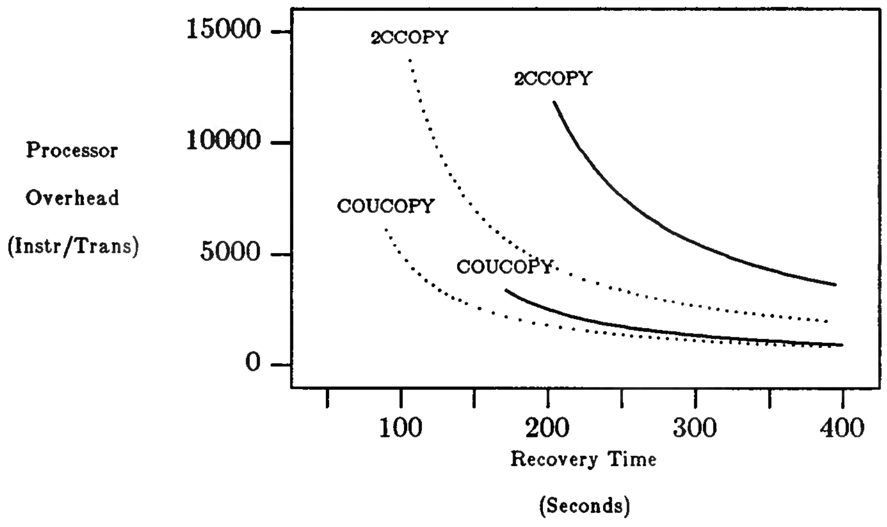

Figure 4b - Processor Overhead/Recovery Time Trade-off

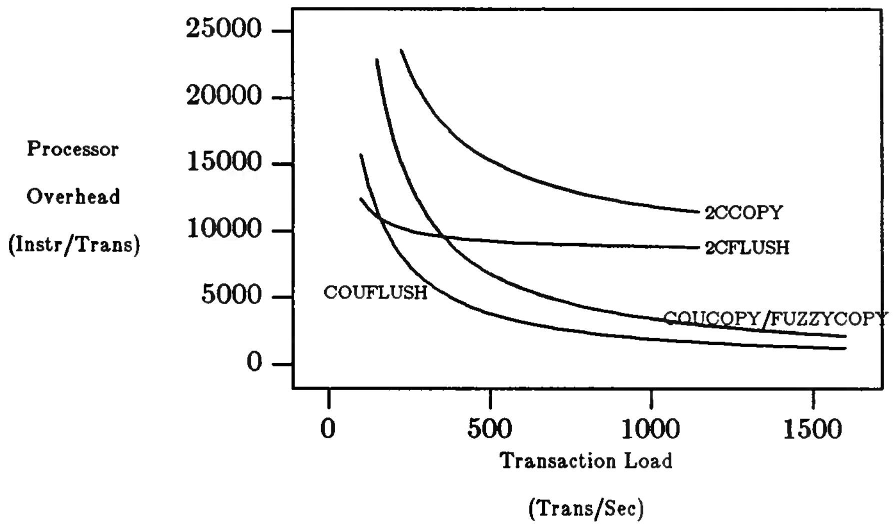

Figure 4c - Effect of Varying Transaction Load

COUCOPY algorithms 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 overhead.

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 2CCOPY than to COUCOPY. This is because of reductions in the number of transactions that must be rerun because of violations of the two-color 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 two-color constraint violation.

Figure 4c describes the effect of transaction load on processor overhead. 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. However, the effect is not uniform across checkpointing algorithms. In particular, 2CFLUSH is the least costly low-load alternative, yet is one of the most costly at high loads. The reason for this is that 2CFLUSH is the only algorithm which never requires segment copying in primary memory. Segment copying is expensive at lower transaction rates, since the cost of copying cannot be spread over many transactions.

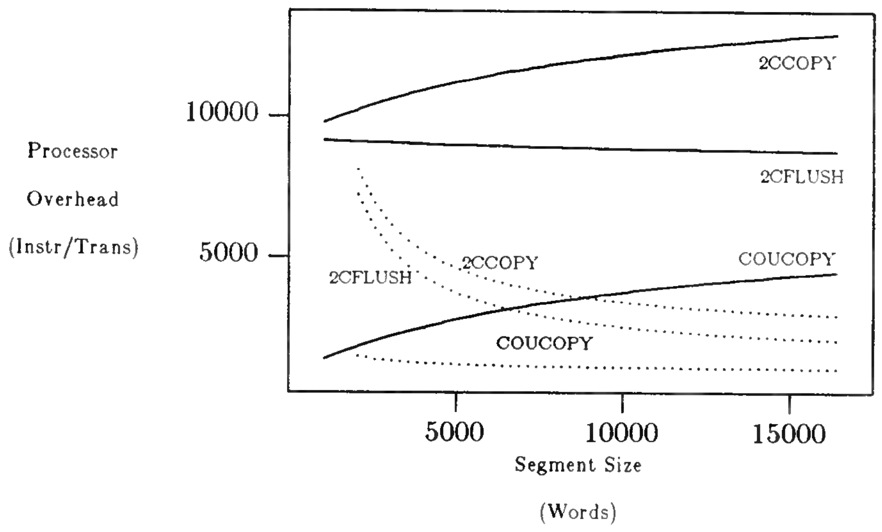

Figure 4d - Effect of Varying Segment Size

We have already seen that checkpointing overhead can be controlled by varying the checkpoint interval. Figure 4d describes the effect on checkpointing overhead of another parameter, the segment size. (Recall that segments are the units of transfer to secondary storage.) Two curves are plotted for each of three of the checkpoint algorithms. The dotted curves represent the case in which the checkpoint interval is held constant at 300 seconds as the segment size varies. The solid curves represent the case in which checkpoints are allowed to run as quickly as possible given the particular segment size.

Normally, bandwidth is greater, and thus checkpoints can complete more quickly, when segments are larger. Thus if the checkpoint duration is held constant while segment size increases, the fraction of each checkpoint interval during which the checkpointer is actually active decreases. One effect of this is that fewer transactions will need to be aborted for violating two-color restrictions. This effect is responsible for the decrease in the overhead of the 2CCOPY and 2CFLUSH algorithms (dotted curves). Under similar circumstances, COUCOPY (dotted curve) shows only minor variations with segment size.

Another effect of increasing the segment size is to decrease the number of segments in the database, resulting in decreased per segment overhead costs such as disk I/O initiation. However, when checkpoints are allowed to run as quickly as possible (solid curves), the cost of the whole checkpoint must be shared by fewer transactions when segments are bigger. These conflicting tendencies affect the algorithms differently. Algorithms with costly copy overhead, namely 2CCOPY, COUCOPY, and FUZZYCOPY (not shown) suffer from the latter tendency and show higher overhead as segment sizes increase. 2CFLUSH, which never copies data, actually exhibits lower overhead with bigger segments.

With our final figure we consider how the availability of stable main memory to hold the log tail affects the cost of checkpointing. As we mentioned in Section 3, straightforward fuzzy checkpointing (without violation of the log write-ahead protocol) is possible when the log tail is stable. The straightforward algorithm flushes dirty segments to disk without first copying them to a special buffer; we shall term this the FASTFUZZY algorithm here. The other algorithms change only in that log sequence numbers are no longer needed to synchronize the checkpointer with the logging mechanism.

Figure 4e shows the recovery overhead of the algorithms assuming a stable log tail. (Checkpoints are taken as quickly as possible.) Clearly, FASTFUZZY is an appealing algorithm in this case. The cost of maintaining the backup is only a few

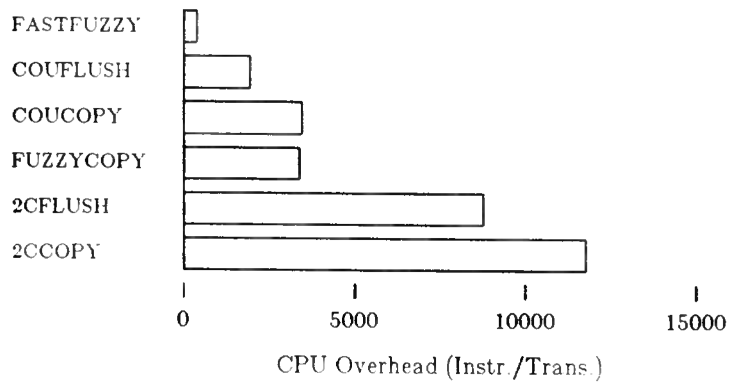

Figure 4e - Processor Overhead with Stable Log Tail

hundred instructions per transaction. The costs of the other algorithms are nearly identical to those from Figure 4a, since the savings in log synchronization costs is not significant.

5. 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 five checkpointing algorithms. Our results indicate that there may be significant differences in performance among them algorithms.

In general, the copy-on-update algorithms can produce consistent backups for about the same cost as a fuzzy backup, while the two-color algorithms are more costly. If stable memory is available to hold the log tail, faster fuzzy dumps are possible. However, by considering the effects of such variables as transaction load and the checkpoint interval, we have shown that the absolute and relative performance of checkpointing algorithms is not an intrinsic property of the algorithm. An algorithm's performance depends on the system and environment of which it is a part.

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.

Borr85a.

Borrill, Paul B., "A Comparison of 32-Bit Buses," *IEEE Micro*, pp. 71-79, Dec., 1985.

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.

Eich86a.

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

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.

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.

Kim86a.

Kim, Michelle Y., "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, vol. C-35, no. 11, pp. 978-988, November, 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.

Pu85a.

Pu, Calton, "On-the-Fly, Incremental, Consistent Reading of Entire Databases," *Proc. Int'l Conf. on Very Large Databases*, pp. 369-375, Stockholm, 1985.

Sale86a.

Salem, Kenneth and Hector Garcia-Molina, "Disk Striping," *Proc. Int'l. Conf. on Data Engineering*, pp. 336-342, IEEE-CS, Los Angeles, CA, Feb., 1986.

Sale87a.

Salem, Kenneth and Hector Garcia-Molina, "Crash Recovery for Memory-Resident Databases," CS-TR-???-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.