# CRASH RECOVERY FOR MEMORY-RESIDENT DATABASES

Kenneth Salem
Hector Garcia-Molina

CS-TR-119-87

November 1987

# Crash Recovery for 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 volatile semiconductor memory. The crash recovery manager of such a system logs changes to disk and periodically checkpoints the database to non-volatile storage. The manager is different from that of a disk-based system because after a failure the main memory and not the disk must be restored to a consistent state. Furthermore, the performance of the recovery manager is more critical since it is the only component of the system that performs expensive I/O operations. In this paper we study the algorithms for performing main memory crash recovery. Their performance is compared via a detailed model of the critical resources for this environment: CPU overhead and disk bandwidth. The performance results suggest a set of "rules of thumb" for selecting a crash recovery strategy.

# Crash Recovery for 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, query processing, and (primary) memory management.

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:

- The least expensive and most dense form of semiconductor memory is volatile. When volatile memory is used, the entire primary copy of the database is lost in the event of a loss of power. At recovery time, the focus of the recovery manager must therefore be on the restoration of the primary (memory-resident) database, rather than the disk-resident database, to a consistent state.

---

† 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.

- In a MMDBMS, the transactions' data requirements can be satisfied without disk I/O. However, to maintain transaction durability 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 the remainder of the paper, we will examine crash recovery in a MMDBMS in light of these differences. We have distilled from existing and proposed MMDB recovery mechanisms a set of recovery *issues*. A recovery issue is an aspect of the design of a MMDBMS which affects the performance of the recovery manager. For example, we consider such issues as how updates are made to both the primary and secondary database copies, how the secondary database copy is organized, what type of log information is generated, and how it is propagated to stable storage. For each of the recovery issues, we consider a number of possible *policies* that a MMDBMS might adopt. In general the policies, and thus the issues, are not independent. For example, the organization of secondary storage will affect how updates to the backup database copy are produced and propagated.

There are at least two aspects to the performance of a recovery manager. One is the recovery time, the time taken to restore normal operation after a system failure occurs. A second is the magnitude of the overhead of recovery management on transaction processing during normal operation of the system. 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. Thus evaluating "I/O cost", as is commonly done for disk-based systems, is not a good way of measuring the overhead of recovery management in a MMDBMS. This is not to say that the I/O bandwidth is not important to the performance of the recovery manager. As we will see, it affects recovery time in a number of

ways.

What do appear to be the potentially important overhead costs of recovery in a MMDBMS are CPU overhead and I/O bandwidth to the log disks. In this paper we will consider both of these costs, with particular emphasis on CPU overhead. CPU overhead is produced by a number of different recovery 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.

A number of other recent studies have looked at recovery for memory resident databases. A technique for producing asynchronous action-consistent checkpoints is described in [DeWi84a]. This paper also suggests the use of stable memory to hold the log tail and pre-committed transactions. Recovery management in IMS/VS FastPath, which also makes used of pre-committed transactions, is described briefly in [Gawl85a]. The recovery mechanism proposed in [Hagm86a] stresses fast, asynchronous, fuzzy checkpoints and log compression to provide fast recovery from system failures.

Several papers have suggested recovery techniques that make use of dedicated or special-purpose hardware. [Lehm86a] proposes a recovery processor that operates in parallel with the database processor to flush data from a stable memory-resident log tail to log disks. The recovery processor proposed in [Eich86a] merges logged after-images into the backup database copy to keep it as current as possible. [Garc83a] and [Sale86a] describe a hardware logging device that logs automatically. It operates transparently to the database processor(s) and at the word level.

The study presented here draws on ideas from most of these papers. A number of policies that we will consider have been suggested in one or more of them. Our emphasis is on algorithmic alternatives rather than special hardware. We do not consider here any recovery mechanisms that rely on the existence of some amount of stable primary storage or on special purpose or functionally segregated processors. Such mechanisms could be modeled with little difficulty within the framework we will present, but we have chosen not to do so for lack of space. Note incidentally that even if all of main memory is non-volatile some form of logging is probably necessary. This is to protect against the CPU corrupting the main memory database during a software failure. Thus, special purpose hardware may change the parameters for recovery I/O but not entirely eliminate the need for it.

There have been a number of studies of recovery mechanisms for disk-based databases, e.g., [Reut84a, Agra85a]. However, we are only aware of two comparative studies of recovery mechanisms for memory resident databases. A taxonomy of MMDBMS recovery mechanisms is presented in [Eich87a] along with an analytic performance model. Unlike the study presented here, most of the alternatives presented in that taxonomy concern the availability of dedicated hardware and stable memory. This is also true of the study presented in [Sale86a].

The contributions of this paper are threefold. First, we present, in a common framework, a number of recovery algorithms and policies. These include both new

policies and others that have appeared elsewhere in the literature. Second, we present and use a modeling methodology which we believe to be well-suited to main memory database systems. Lastly, the results of our study provide an indication of the performance that can be expected from the recovery policies and allow us to draw some conclusions about which policies are most appropriate for a MMDBMS.

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 recovery issues and policies. In Section 4 we present the results of our comparisons, focusing on comparisons of the various issues and policies and on the effects of changes in model parameter values. 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 recovery management. 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. In each section we introduce any relevant model parameters. For convenience, all of the model parameters are listed together in Appendix A.

### 2.1. System Components

Figure 2a is a block-level description of the system components we will assume in our discussion. Processors, memory, and disks are linked by data channels. The next several sections are devoted to descriptions of our models of each of these components.

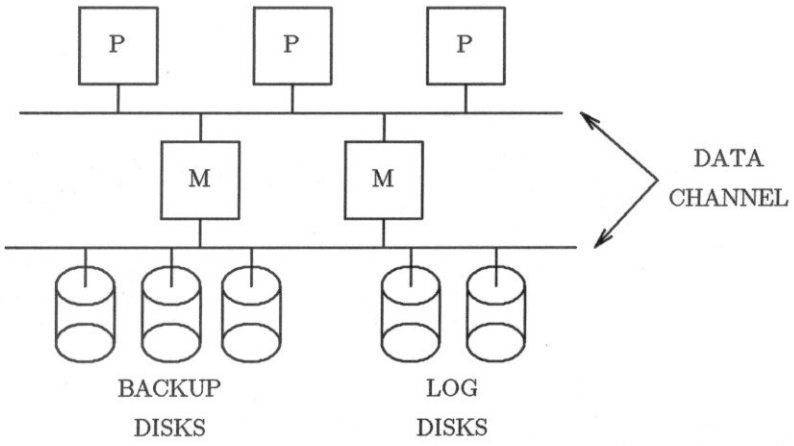


Figure 2a - System Components

### 2.1.1. Processors

The system includes one or more processing units (CPUs). We model the collection of processors as a server which is able to perform certain operations at a cost of some number of instructions per operation.

Table 2a describes the CPUs' operational capabilities which are relevant to our model and their costs. $C_{lock}$ is the cost of locking or unlocking a database entity. $C_{alloc}$ is the cost of allocating or deallocating a block of memory (of any size). $C_{io}$ is the 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. $C_{lsn}$ is the cost of checking or maintaining the log sequence number (LSN) [Gray78a] for a database entity. $C_{lsn}$ will be charged (under some recovery policies) to update a LSN when a transaction makes an update, and to check a LSN when an entity is scheduled for I/O to the backup disks. Finally, $C_{trans}$ represents the cost of executing a transaction *without recovery overhead.* In other words, a transaction (see Section 2.2) would require $C_{trans}$ instructions to execute in a failure-free system without recovery management.

Not shown in the table is the cost of one final operation, memory to memory copies. The cost of a memory to memory copy is taken to be proportional to the number of words copied, 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 |
| $C_{trans}$ | transaction cost | 25000 | instructions |

Table 2a - CPU Operation Costs

### 2.1.2. Primary 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.

It has been suggested that the availability of some amount of non-volatile primary memory would be beneficial to a MMDBMS. For example, non-volatile primary memory can reduce transaction response time by allowing externalization without a wait for I/O to the log disks. Since the performance metrics used here (e.g., CPU overhead) are at a "lower" level than transaction response time and throughput such benefits would not be apparent in our results. However, we will note, in the next section, those recovery policies that may be able to take advantage of non-volatile

memory in this way.

In some cases, the availability of non-volatile memory will can affect more than transaction response time. For example, the recovery mechanism proposed in [Lehm86a] depends on non-volatile RAM for correct operation. There are also certain combinations of recovery policies presented in this paper that would not function correctly without at least enough non-volatile RAM to hold the memory-resident portion of the log. (An example is the combination of FASTFUZZY checkpointing and immediate updates. See Section 3.) In keeping with our "standard hardware" assumption we do not consider these mechanisms. As mentioned in the final section, the benefits of non-volatile memory are a topic of continuing study.

### 2.1.3. Secondary Storage

The secondary storage media of interest are magnetic disks, although the system may also use other media, such as tapes or optical disks, to store archival data. Magnetic disks are used for logging and to hold the backup (secondary) database copy. $N_{disks}$ is the total number of disks available, and $f_{\log} N_{disks}$ $(0 \leq f_{\log} \leq 1)$ is the number used for logging. The remaining $(1 - f_{\log}) N_{disks}$ disks hold the backup database.

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. We also assume that the constants $T_{seek}$ and $T_{trans}$ are the same whether the disk is being used for the log or the backup database. Again, this is a simplifying assumption since we might expect the log disks to be somewhat faster because of the sequential nature of log access. However, 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 (Section 3.5).

Table 2b shows the model parameters related to the disks. The large number of log disks (60*0.67 = 40 log disks) is used so that the SINGLE log propagation policy (Section 3.4) can be included in the performance model. Obviously, the high log bandwidth requirements of that log propagation policy make it an undesirable choice in a high performance environment.

| symbol | parameter | default | units |
|---|---|---|---|
| $T_{seek}$ | I/O delay time | 0.03 | seconds |
| $T_{trans}$ | transfer time constant | 3 | $\mu$seconds/word |
| $N_{disks}$ | number of disks | 60 | disks |
| $f_{\log}$ | fraction of disks for logging | 0.67 | |

Table 2b - Disk Model Parameters

### 2.1.4. 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, Sale86b]. 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 our 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 recovery policies whose bandwidth requirements are higher than others'.

## 2.2. 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 also assumed to be the granules for log entries.

Records are grouped into larger units called pages and segments. Pages are the granules used for shadowing in primary memory, if shadowing is used. The page size is $S_{spg}$, which may be any multiple of $S_{rec}$. Records are also grouped into units, called segments, for efficient transfer to the backup disks. Such I/O is done using granules of size $S_{seg}$ which again can be any multiple of $S_{rec}$. The segment is the granule of storage organization in secondary storage.

Log records are collected into log pages of size $S_{lpg}$. These fixed-size log pages are the granule of transfer to and from the log disks.

Table 2c summarizes the model parameters related to the database and gives their default values.

| symbol | parameter | default | units |
|---|---|---|---|
| $S_{db}$ | database size | 256 | Mwords |
| $S_{rec}$ | record size | 32 | words |
| $S_{spg}$ | shadow size | 1024 | words |
| $S_{lpg}$ | log page size | 1024 | words |
| $S_{seg}$ | segment size | 8192 | words |

Table 2c - Database Model Parameters

### 2.3. Concurrency Control

We assume throughout the paper that transactions (Section 2.4) use locking as a synchronization mechanism. Some of the checkpointing policies we discuss in Section 3.5 make use of locks to synchronize secondary storage updates with transactions as well. Synchronization of recovery activities (e.g., checkpointing) without locking is a subject of current investigation.

The locking facilities are assumed to be hierarchical [Gray76a], i.e., it is possible to correctly lock database objects of several different granularities. This assumption affects only our performance model, and all of the recovery policies described here can perform correctly without hierarchical locking.

### 2.4. 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 $\lambda$ transactions per second. The multiprogramming level, i.e., the number of simultaneously active transactions in the system, is taken to be constant at $M$. As already described, executing a transaction requires $C_{trans}$ CPU instructions, excluding the overhead of recovery. 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 |
|---|---|---|---|
| $\lambda$ | arrival rate | 1000 | transactions/second |
| $N_{ru}$ | number of updates | 5 | records/transaction |
| $M$ | degree of multiprogramming | 10 | transactions |

Table 2d - Transaction Model Parameters

For the purposes of our model, a successful transaction will consist of a number of distinct phases. Transaction is initiated by an initiation message, and a "begin transaction" mark is made in the system log or logs. The transaction then produces a sequence of update requests. Once the updates are complete, the transaction submits a commit request to the system. During the *commit processing* phase, the system may take a number of actions on behalf of the transaction, such as releasing locks, flushing log pages, or installing updates. The actions taken, and the order in which they occur, depend on the recovery policies that are in use.

Once commit processing is finished the transaction will receive a commit acknowledgement. At that point the transaction is free to *externalize* (send an output message), however, it can request no further updates.

For various reasons, not all transactions will be successful. In the following section we discuss failures and describe how they are included in our model.

### 2.5. Failures

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 assume that the ultimate fate of a transaction, either success or failure, is determined independently of the recovery mechanism. In particular, a transaction has a probability $p_{fail}$ of experiencing transaction failure.† We assume that those transactions that fail do so halfway through their execution.

A system failure results in the halt of the system and the loss of the contents of volatile memory, followed by system restart. System failures might be caused by uncorrectable errors in the system software, or by power failures. One of the performance measures we consider is the time for recovery from a system failure. The

† A transaction may also fail as a result of actions of the recovery manager. In these cases the transaction can usually be restarted. We treat recovery induced failures independently of other failures. The probability of a recovery induced failure, $p_{restart}$, will be computed in Section 4 as a function of the recovery policies. A transaction's probability of *ultimate* failure remains $p_{fail}$, no matter what the value of $p_{restart}$.

recovery time is discussed in more detail in Section 3.10.

In this paper 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. The log can be duplexed to provide some protection against the failure of a log disk.

The strategies we will consider for managing the backup (secondary) database copy will have some impact on the system's ease of recovery from failures of the backup disks. In Section 3.8, where backup policies are described, we consider the implications of the various policies for media fault tolerance, although we do not include media recovery time as a performance metric. In that section, we also briefly discuss the differences between memory-resident and disk-based databases with respect to media failures in secondary storage.

## 3. Recovery Issues and Policies

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 recovery issues we present in this section are decisions that affect the design of the former component, they affect the *performance* of both components. Recovery policies that determine behavior during normal processing affect system recovery time because they affect 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. We will postpone further discussion of post-failure recovery management until the end of this section, after we have presented the issues.

For the purposes of our discussion, it is convenient to divide the recovery issues into two groups: *synchronous* and *asynchronous*. The synchronous issues concern recovery-related activities that are coordinated with transaction updates, commits, and aborts. The asynchronous issues impact the update and maintenance of the secondary database copy. Of the five recovery issues we consider, three are synchronous issues. We will discuss these issues in the following section.

### 3.1. Synchronous Recovery Issues

The synchronous recovery issues are concerned with the maintenance of the primary database copy and the log and with recovery from transaction failures. In particular:

- The *update policy* determines how and when transaction updates are installed in the database. Updates can be made "in-place", or some form of shadowing can be used. Similarly, updates can be made when they are requested, or can be delayed until the commit processing phase.
- The *logging policy* determines what kind of REDO information is logged for each transaction. REDO logging can be done by value or by operation. We assume

that REDO logging is the preferred method of ensuring transaction durability. (Other methods are possible, e.g., all segments updated by a transaction could be forced to the backup database copy before the transaction is externalized. However, this results in a relatively large amount of disk I/O synchronous with each transaction. In addition, it does not lend itself to the use of non-volatile log buffers when they are available.)

- The *log propagation policy* determines when in-memory log pages are forced to the log disks. During commit processing, a transaction's commit record can be forced immediately to the log disks, or the I/O can be delayed in the hope of committing more than one transaction with a single I/O operation.

### 3.2. Logging Policies

We begin the discussion of logging policies with a few general words about logging in memory-resident databases. As we have already noted, we assume that the recovery manager uses REDO logging. UNDO logging may or may not be used, depending on other recovery issues that we will describe shortly.

If UNDO logging is used, the recovery manager may maintain separate logs for REDO and UNDO information, or it may combine the two into a single log. Whether one or two logs are used depends on the checkpoint policy, to be discussed shortly. Not all checkpoint policies require that UNDO information be logged on stable storage (i.e., the log disks). In these cases, maintaining separate logs for REDO information (which is always logged on stable storage) and UNDO information can reduce the amount of log I/O traffic.

The logging policy determines the *nature* of the log information that is stored when records are updated by a transaction. We consider two possible policies, *value logging* and *operation logging.*† If value logging is used, the REDO information consists of copies of the new versions of modified records. Operation logging avoids storing copies of the records in the log. Instead of logging new values, an operation (action) that can be used to recreate those values is stored.

Typically, a database manager will support many different types of operations on the database. Complex, abstract operations are built up by combining simpler operations at a lower level of abstraction. For example, at a level of abstraction called the *data manipulation* level, operations such as UPDATE might be supported. An UPDATE might consist of a modification to a record in a database table plus modifications to the index structures that are used to access that table. The transactions supported by the system can be considered the highest level operations.

The REDO operations logged when an operation logging policy is used might come from any of these levels of abstraction. For the purposes of our model, the operations that we will consider are the transactions themselves, i.e. we will consider operation logging at the *transaction level.* Commonly, operations below the transaction level will be

---

† These are termed *state* and *transition* logging in [Reut84a, Haer83a].

used, e.g., data manipulation level operations. We consider transaction level operations because they enable us to study the effect of operation logging without modeling in detail the lower level operations of which the transactions are composed.

REDO operation logging is relatively simple. After a crash, to recreate the new values of a transaction $T$ we rerun $T$. Provided the transaction code is available, the operation-style log entry for $T$ would include a transaction type identifier and a set of values for the input parameters.† For example, if $T$ is a debit/credit type transaction [Gray85a] depositing ten dollars into an account in a bank database, $T$'s log entry would include the identifier for the CREDIT transaction type plus the account number and dollar amount. $T$'s log entry would be made during its commit processing phase.

UNDO operation logging is complicated by the fact that an operation such as a transaction can fail at many points, and thus can have many possible inverse operations. We do not consider UNDO operation logging further and assume instead that if UNDO logging is needed it is always done by value.

A possible disadvantage of operation logging is that recovering a transaction involves executing it again, instead of simply restoring its after-image. Thus CPU-intensive transactions that access little data may take longer to recover. A second disadvantage is that the correctness of operation logging depends on serializable transaction execution. Transactions being re-run after a failure may produce different results than they did after their initial execution if the original execution schedule was not serializable. Finally, operation REDO logging requires that checkpoints be made at the same level of consistency as the operations being logged. Thus, if transaction level REDO operation logging is being used, the checkpoints must be transaction-consistent. (i.e., transactions must be represented atomically in the checkpoint). If partial results from transactions survive a crash, rerunning the transactions will not in general produce the same results as the original executions. We will discuss checkpoints and checkpoint consistency further in Section 3.5.

### 3.3. Update Policies

The update policy determines *when* and *where* primary database updates are made. The answer to the "where" question is determined by the organization of the primary database. We consider two possibilities:

- Records are assigned a single, fixed location in main memory. Records can then be identified by their location.

---

† We are assuming here that transactions are not conversational (i.e. that they receive only the message that causes their initiation). It is possible to use operation logging with conversational transactions if all of the input messages are logged (e.g., see [Borg83a] ), but it is probably simpler to use value logging for this type of transaction. Value and operation log entries (for different transactions of course) can be mixed on the same REDO log, so conversational transactions could be using value logging while simple transactions are logged by operation.

- Records are addressed indirectly through a mapping table, thus allowing the location of the record to vary over time.

The first type of database organization we will term *static*, the latter *dynamic*.

Indirection under the dynamic policy is done in granules called *shadow pages*, i.e., it is the shadow pages that are mapped through the indirection table. Shadow pages may be the same size as the database records, or several records may be grouped onto a single page. (In that case, records would be identified by a page identifier plus an offset within that page.) The advantage of larger shadow pages is that the indirection table can kept small, thus reducing the amount of primary storage needed by the DBMS. The disadvantage, as we shall see, is that large pages can result in prohibitively high CPU overhead for copying pages.

We will also consider two answers to the "when" question:

- Records can be updated in-place as the updates are generated by transactions. These are termed *immediate* updates.
- Updated records can be stored in a special per-transaction buffer until the transaction's commit processing phase. These we will term *delayed* updates.

We consider below three of the four possible update policies implied by these questions. Our model does not distinguish between the performances of static and dynamic immediate policies since from the point of view of the recovery manager there is little difference between them.†

### 3.3.1. Immediate Update Policies

An immediate update policy implies that transactions update database records as the updates are generated, i.e., without waiting for commit processing. Since transaction failures are possible, it is necessary to record UNDO as well as REDO information in the log when immediate updating is to be used (although the UNDO log need not always be flushed to the log disks). UNDO and REDO information is copied to the (in-core) log tail before the database record is updated.

The use of immediate updates places some restrictions on the checkpointing policy that may be used by the recovery manager. In particular, FASTFUZZY checkpointing cannot be used in conjunction with immediate updates. We will discuss the reason for this restriction in section 3.6, when we consider the fuzzy checkpointing policies.

### 3.3.2. Delayed Dynamic Update Policy

Under a dynamic update policy, database records are grouped into pages which are then referenced indirectly through a page mapping table. Note that when dynamic updates are used, locking must be done with page rather than the record granularity. If there are several records per page, this may reduce the transaction concurrency

---

† Of course, the choice of static or dynamic updates may have other effects on the database that are outside the scope of this study. Some of the problems and benefits of direct (static) addressing are discussed in [Lehm86a].

possible in the system. Dynamic updates can also cause difficulties for the checkpointer. We will discuss this further in Section 3.9.

Pages are addressed indirectly through a global (memory-resident) mapping table, *GLOB_TAB*. The physical (primary memory) address of the $i$th page is stored at $GLOB\_TAB[i]$. In addition, each transaction maintains a local mapping table, *LOC_TAB*, which is used to address the shadows of pages updated by that transaction.

Every data reference by a $T_i$ is made indirectly through one of *GLOB_TAB* or $LOC\_TAB_i$. When transaction $T_i$ updates page $j$ and $LOC\_TAB_i[j]$ is null, a free page is allocated as a shadow and page $j$ is copied to the shadow. All further references by $T_i$ to page $j$ are made through $LOC\_TAB_i[j]$.

$T_i$ is aborted by freeing the space occupied by $LOC\_TAB_i$ and all of the pages to which it points. Thus if dynamic delayed updates are used, the recovery manager need not log UNDO information for transaction updates. If $T_i$ is to be committed, the updates must be installed in the global table. In a memory-resident database this operation is relatively simple because both tables are in volatile storage. Each non-null $LOC\_TAB_i[j]$ is copied to $GLOB\_TAB[j]$.

### 3.3.3. Delayed Static Updates

The delayed static update policy is a compromise between the delayed dynamic and immediate static update policies. As with the former policy, transactions maintain local mapping tables for their shadows. The global table is not needed, however, since updates are eventually installed on top of the original version of the data object. There is no need to group records onto pages with a delayed static policy since the local tables are likely to be much smaller than a global table would have to be. Thus the local indirection tables can be maintained at the record level.

Record updates and transaction aborts are handled in much the same way as they were under the delayed dynamic policy. When $T_i$ updates record $j$ and $LOC\_TAB_i[j]$ is null, space for a new record is allocated as a shadow and record $j$ is copied to the shadow. Further references by $T_i$ to that record are made through $LOC\_TAB_i[j]$. $T_i$ is aborted by freeing up the space allocated for the updated record copies and *LOC_TAB*. If $T_i$ commits it copies each of its shadow records to its permanent location and frees up its space and the space occupied by $LOC\_TAB_i$.

## 3.4. Log Propagation Policies

The log propagation policy determines *when* log data is sent to the log disks. Log data can be flushed to the disks after every transaction or delayed until a log page has been filled. Depending on the log page size and the log volume of the transactions, pages flushed using the latter method may contain commit records from more than one transaction. It has thus earned the name *group commit* [DeWi84a, Gawl85a]. We will call the other policy *single commit.*

Flushing the log after every transaction can introduce inefficiencies since log pages may only be partially full when they are flushed. Grouping commits reduces the

amount of log disk traffic, thus reducing the overhead associated with disk I/O. However, grouped transactions may pay a response time penalty since they cannot be externalized until enough log data has accumulated to permit the propagation of their log entries to stable storage.

Note that if stable RAM is available to hold the log (or at least the log tail), then there is no reason to flush a log page after every transaction. The response time penalty of group commits is removed since transactions can externalize as soon as their commit records are in stable RAM.

### 3.5. Asynchronous Recovery

The asynchronous recovery issues concern the maintenance of an "almost up-to-date" backup copy of the database on stable storage (the backup disks). In a MMDBMS, it is desirable to avoid transaction-synchronous I/O. For example, "forcing" updated records to the backup disks during commit processing should be avoided, as should periodic *checkpoints* which cause transaction processing to halt while the backup database is synchronized with the primary copy. Instead, the secondary database is maintained by one or more asynchronous processes which continually flush updates to the disks.

The granule of transfer between the primary and secondary databases is the *segment*, a logical collection of one or more database records. Since disk I/O is assumed to transfer data from a physically contiguous region of primary memory to the disks, the records in a segment must be physically contiguous, or must be made physically contiguous, before the segment can be flushed. When a static update policy is used this is not a problem, since the location of a record is fixed over time. However, dynamic updates can complicate the situation. We discuss this further in Section 3.10 when we consider some the inter-dependencies among the recovery policies.

We have studied two issues related to this asynchronous checkpointing function:

- The *checkpointing policy* determines the degree of consistency of the backup database, and consequently the amount of processor overhead and interference with transaction processing that is required to produce the backup.
- The *backup policy* determines how the secondary database copy is organized and how it is updated.

We next describe these two issues in more detail, and discuss the policies we have considered.

### 3.6. Checkpoint Policies

Checkpoint policies determine the level of consistency of the backup database. This, in turn, has implications for how the backup is maintained and updated. Checkpoint policies also affect many other aspects of recovery, e.g., the logging style.

The checkpointer runs repeatedly, each time updating the backup database according to the algorithm determined by the checkpoint policy. Checkpoint policies 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. However, one of the secondary storage management policies we will consider requires full checkpoints.

Another distinction among checkpointing policies 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 policies we have considered and discuss some of the implementation questions that arise. We consider two ways of producing fuzzy checkpoints, and four ways to produce TC checkpoints. We assume throughout that partial checkpoints are taken if they are permitted by the backup policy.

### 3.7. 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].

Synchronization can be costly. Because fuzzy checkpoints require no synchronization, fuzzy backups are the cheapest backups to produce. We consider two ways to do so. In both cases, the checkpoint is begun by entering a *begin-checkpoint* marker in the system log, along with a list of the currently active transactions. We then consider two ways to actually produce the backup copy.

The simplest method we call the FASTFUZZY policy. Under the FASTFUZZY policy, the appropriate segments from main memory are simply flushed to their proper locations on secondary storage, as determined by the backup policy. (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. For this reason, the use of FASTFUZZY checkpointing places restrictions on other recovery policies that can be used. Immediate update policies cannot be used,

since that would imply that UNDO information would have to flushed to the log *before each update* (to avoid possible violations of the *log write-ahead protocol* [Gray78a] ). This would result in a large amount of synchronous I/O for logging.

FASTFUZZY checkpoints also cannot be used in conjunction with a group commit log propagation policy.† Under group commit, transactions may have released their locks (and thus have installed their updates) before their log information is flushed to stable storage. As with immediate updates, this could result in violations of the log write-ahead protocol by causing uncommitted updates to be installed into the backup database before UNDO information for those updates has been propagated to the log disks.

A second fuzzy checkpointing scheme is called FUZZYCOPY. FUZZYCOPY checkpointing will result in fewer transaction delays than FASTFUZZY checkpointing, since transactions will be able to make their updates available quickly to other transaction. The price is additional CPU overhead paid to generate the checkpoint.

FUZZYCOPY checkpointing is similar to FASTFUZZY, 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 copied 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 employing *log sequence numbers* [Gray78a]. As the system creates new (in-memory) log pages, they are assigned monotonically increasing log sequence numbers (or LSNs). When a transaction updates the database, that update is associated with the LSN of the log page which holds its REDO/UNDO log record. With each segment is stored the largest LSN associated with any update that has affected the segment. The segment acquires the LSN of an update to a record in that segment if the update's LSN is greater than the current LSN of the segment. By copying the segment and then checking the LSN, the checkpointer eliminates the write-ahead logging problems that are possible with FASTFUZZY.

In case of a system failure, fuzzy checkpoints produced by either method can be used for recovery in the same way. The most recent copy of each database segment is first brought into main memory to create a new, but possibly inconsistent, primary database. (The mechanism for doing this is specified by the backup policy.) The log is then then processed against the new database to bring it into a consistent state.

With FUZZYCOPY checkpoints, log roll-back/roll-forward must be used, since the new primary database may contain updates from aborted transactions. Log rollback must continue until the beginning of the oldest transaction that was active at the time the most recent *completed* checkpoint began. This point can be deduced from information (e.g., the active transaction list) that is recorded in the log when a checkpoint

---

† Actually, the restriction must be even stronger. If FASTFUZZY checkpoints are used, transaction updates may not be *installed* until their log records are on stable storage. Since installation comes before lock release, transactions cannot use a pre-commit strategy [DeWi84a]. This precludes group commits, which would normally be used in conjunction with pre-commitment.

begins.

### 3.8. 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. Consistent backups also permit the use of operation-style REDO logging.

We will consider only TC checkpoints, not AC, although AC checkpoints may actually be more practical in a real system. By doing so we reduce to a more manageable number the already large number of recovery policies we have considered. Also, 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.

Under our model AC checkpoints are more expensive to produce than fuzzy checkpoints, yet they offer no advantages. There are two reasons for this. First, the database must be returned to a *transaction* consistent state after a system failure. This means that UNDO operations may be required whether the checkpoint is AC or fuzzy. Second, we consider operation REDO logging at the transaction level. Again, a transaction consistent checkpoint is needed, this time so that REDO operations can be correctly applied.

AC checkpoints may offer advantages over fuzzy checkpoints in certain situations. For example, if operation-style logging were done using action-level (rather than transaction-level) operations then AC backups would be useful. However, as we have already mentioned, this advantage can be studied (in an extreme sense) by comparing TC and fuzzy checkpoint policies.

We will consider two general methods for producing TC checkpoints, and within each of the methods examine two variations in implementation. The first general type we call *two-color* policies. They are based on the two-color mechanism presented in [Pu85a]. The second type of policies are called *copy-on-update* (COU) policies. They are based on the recovery scheme presented in [DeWi84a].

#### 3.8.1. Two-Color Policies

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. 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 policies 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.

```
WHILE there are white segments
DO BEGIN
    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
```

Figure 3.1 - A Variation of Pu's Basic Checkpoint

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. If a group commit log propagation policy is used, log sequence numbers are used to determine when the segment can be flushed (as was done under the FUZZYCOPY policy). If the checkpointing is handled in this fashion we say that the checkpoint policy 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.

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 if group commits are used.) 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 CPU overhead for this advantage. When checkpointing is handled in this fashion we say that the checkpoint style is *2CCOPY*.

### 3.8.2. Copy-on-Update Policies

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 policies. 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 based on the technique described† in [DeWi84a].

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

---

† One major 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.

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

```
lock S (and R) (exclusive)
IF (S > CUR_SEG) AND (τ(S) ≤ τ(CH)) THEN
        allocate buffer for S
        copy S to buffer (including timestamp)
        set p(S) to point at buffer
set τ(S) = τ(T)
unlock S
update R
```

Figure 3.2 - Transaction Updates under COU

not have old copies. In this case, the checkpointer has the same two options it had under the locking policies. 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 policy. 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 again.

Note that under the COU policies, 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.

### 3.9. Backup Policies

The backup policy determines how the the secondary copy of the database is managed. In particular, the backup policy determines how database segments are mapped onto secondary storage. This, in turn, determines the cost transferring segments to and from primary memory on behalf of the checkpointer and at recovery time.

We will examine five backup policies. Two of these are *duplex* policies, meaning that two complete copies of the database exist on secondary storage. One advantage of duplex policies over *monoplex* policies is their increased tolerance of media failures.

An interesting aspect of main-memory databases is that (secondary) database segments lost as a result of a disk media failure are available in primary storage (provided that a system failure does not occur simultaneously). Thus one way to recover from a media failure in a MMDB is to map the failed disk segment to a new location on the disks, write the primary copy of the segment to the new location, and then possibly use the log to restore the newly copied segment to a state consistent with the rest of the backup database.

Alternatively, the primary copy of the segment might not be flushed immediately to secondary storage. Instead, the primary copy can simply be marked as dirty so that it will be flushed by the checkpointer during its next pass over the database. This

```
set CUR_SEG = first segment
quiesce transaction processing
log begin-checkpoint record and flush log tail
save timestamp of last checkpoint as τ(OLDCH)
assign new timestamp τ(CH) to checkpoint
WHILE (CUR_SEG ≤ number of segments in database) DO
    lock CUR_SEG (exclusive)
    IF (τ(CUR_SEG) < τ(CH)) THEN
        IF (τ(CUR_SEG) > τ(OLDCH)) OR (full checkpoint) 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 (τ(OLD_SEG) > τ(OLDCH)) OR (full checkpoint) THEN
            flush old copy of segment to backup disks
END_WHILE
```

Figure 3.3 - COU Checkpointing

increases the failure window for the segment, but it permits recovery from the media failure with less effort on the part of the system. If checkpoints are repeated frequently it may be a viable recovery option.

Another way provide increased protection against media failures is to take regular dumps of the secondary database (perhaps onto magnetic tape). This is particularly easy to do in MMDBs since all access to secondary storage is by the checkpointer rather than by transactions. Synchronizing tape dumps and the checkpointer is easier than synchronizing tape dumps and transactions for two reasons. First, the access pattern of the checkpointer on secondary storage will be much more predictable than that of a set of concurrent independent transactions. Thus it should be possible to run the dump either "ahead of" or "behind" the current access point of the checkpointer. Secondly, any synchronization delays that are encountered are not as important, since neither the checkpointer nor the tape dumper will operate with the same tight response time constraints that are common with transactions.

In any event, we will not attempt to quantify the relative tolerances to media failures of the backup policies we present. Instead we will concentrate on comparing

the various policies in terms of CPU overhead and recovery time. In the following, we give brief descriptions of each of the five backup policies. To simplify the discussion, we will assume that database segments are numbered 1 to $N_{seg}$, where $N_{seg}$ is $S_{db}$ / $S_{seg}$, the number of segments in the database. $S_i$ is the $i$th segment. Secondary storage is composed of $B$ segment-sized slots, or blocks, and $L_i$ is the $i$th block. The value of $B$ is dependent on the backup policy. (Some of the backup policies require small amounts of storage in addition to the $B$ blocks.) We also assume that if a write to secondary storage fails (because of a system failure) then the corrupted segment is detectable through a checksum or some other error detection mechanism.

### 3.9.1. Fixed Monoplex Backups

FM backups are the most straightforward type. Each segment is assigned a single location in secondary storage, i.e. $S_i$ is assigned to $L_i$. In addition, a segment-sized write buffer ($L_{(N_{seg}+1)}$) is available on the disks. Thus $B = N_{seg} + 1$.

Since updates to $S_i$ are always written to $L_i$ they will overwrite the old version of $S_i$. To ensure that an uncorrupted version of $S_i$ always exists, updates must be made twice under this scheme. The first time, the update is made to the write buffer, the second time to the proper home of the segment. If the second write fails, the corrupted block can be restored from the correct version in the write buffer once the system is restarted. The chief advantage of FM backups is that they require relatively little secondary storage space. Like all monoplex backups, FM backups are not suitable for TC checkpoints since there is no guarantee that the entire checkpoint will be atomic.

### 3.9.2. Sliding Monoplex Backups

SM backups are a modification to FM backups so that secondary storage updates need to be made only once. SM backups require $B = N_{seg} + 1$ blocks of secondary storage, plus space for a *base pointer*. The base pointer points at one of the blocks $L_i$. In addition, space is required to store a *checkpoint-identifier* for each block to indicate which checkpoint wrote that block most recently.

By "sliding" the position of each segment each time a new checkpoint is started, SM backups allow a block to be updated with only a single I/O operation. The checkpointer does a full checkpoint, dumping the database segments in order from $S_1$ to $S_{N_{seg}}$. The base register indicates which block $S_1$ will be written to; it is decremented at the beginning of each checkpoint. (Decrementing the base pointer must be done carefully, using two I/O operations.) Starting with the block pointed at by the base pointer, blocks are filled sequentially by the dumped segments. For example, if segment $S_i$ gets written to location $L_i$ during the first checkpoint then it will be written to $L_i - 1$ during the second. (The subscript subtraction is done modulo $B$.) Thus if during the second checkpoint the dump of $S_i$ to $L_{i-1}$ fails, the old version of $S_i$ is still available in $L_i$. Note that SM backups support only full checkpoints. All of the $S_i$ must be written to disk during every checkpoint. In addition, because the backup is monoplex, only fuzzy checkpoints are supported.

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

### 3.9.3. Shadowed Backups

We have already discussed the use of shadow pages in primary memory. Shadow pages can be used to manage secondary storage as well. Shadow (SH) backup schemes use $B = N_{seg} + N_{shadows}$ blocks, where $0 < N_{shadows} < N_{seg}$, plus space for an indirection table containing pointers to $N_{seg}$ blocks. ($N_{shadows}$ is a model parameter.) We assume that an indirection table is also maintained in primary storage, along with two free lists, *current* and *next*. (A free list is a list of backup blocks that are not pointed to by the indirection table). Only one complete backup copy of the database exists. Thus, like other monoplex backups, SH backups are useful only for fuzzy dumps.

To dump a segment to secondary storage, the checkpointer selects the top spot from the current free list and writes the segment to that location on the backup disks. The block that holds the old version of the segment is added to the next free list and the indirection table entry (in main memory) for the segment is changes to point to the new block. After every $N_{shadows}$ updates, the main memory copy of the page table is carefully (two updates) written to the backup disks. The next free list becomes the current free list, and the current free list is cleared and used as the next.

Recovery using an SH backup is a matter of reading in the indirection table and using the pointers there to read in each of the database segments. In addition, the initial free list must be constructed from the indirection table after it is read in.

### 3.9.4. Ping-Pong

Ping-pong backups use $B = 2N_{seg}$ blocks on secondary storage to maintain two complete copies of the database. Segments $L_1$ through $L_{N_{seg}}$ hold one copy of the database, and the remaining segments hold the other. Segment $S_i$ is assigned permanently to blocks $L_i$ and $L_{N_{seg}+i-1}$. A flag is maintained on disk which indicates which of the two copies is "current".

To start a checkpoint, the current flag is toggled to switch the current database. Segments are written (once each) to their location in the current database. The non-current segments hold a complete copy of the database as it was after the previous checkpoint, thus ping-pong (PP) backups are suitable for TC or fuzzy checkpointing.

PP backups can be used for partial or full checkpoints. If partial checkpoints are being used, each segment in primary memory must be equipped with two dirty bits, one for each of the secondary database copies. When a transaction updates a segment,

both dirty bits are set. When a page is flushed by the checkpointer, only the dirty bit corresponding to the current backup database is cleared since the update will not yet have been reflected in the other backup copy. Thus each updated segment will eventually be flushed twice to the backup disks.

### 3.9.5. Twist

The twist (TW) backups we will consider are based on a scheme suggested in [Reut80a] for disk-based databases. TW backups can be thought of as a variation of ping-pong in which each updated database segment is written to the backup disks only once.

Like ping-pong, a TW backup maintains space for two complete copies of the database on secondary storage. We will assume that segments are assigned to backup blocks exactly as they are in ping-pong.† To implement TW, each segment in primary memory is augmented with a dirty bit (if partial checkpoints are being taken) and another bit to indicate which of the two backup blocks for that segment was most recently updated. Each backup block on secondary storage has room for a timestamp in addition to the segment itself.

When a checkpoint begins it is assigned a timestamp. During checkpointing, segment $S_i$ (if it is to be dumped) is written to the least recently updated of its two backup blocks as indicated by its associated flag in primary memory. The checkpoint's timestamp is stored with the flushed segment. The dirty bit (if any) of the segment is cleared. When a TC checkpoint is completed, its timestamp is carefully stored on disk to indicate that the checkpoint has been successfully completed. (The timestamp is not needed for fuzzy checkpoints.)

To recover a TC checkpoint, the special record is first read to indicate the appropriate checkpoint. For each $S_i$, both $L_i$ and $L_{N_{seg}+i}$ are read. The block with the largest timestamp less than or equal to the checkpoint timestamp is chosen and the segment is restored in primary memory from that block. Recovering fuzzy backups is similar, except that there is no special record containing a checkpoint timestamp. Instead, for each $S_i$ the block with the largest timestamp is chosen, and the segment is restored from that block.

## 3.10. Recovery Policy Interactions

Normally, the recovery issues we have presented are independent, i.e. choosing a policy for one recovery issue does not affect the possible choices for other issues. However, in some cases there are constraints, most of which we have already mentioned. The following are brief descriptions of each of these constraints their justifications.

---

† Of course, the backup blocks are logical entities. We have said nothing about how the blocks should be physically arranged on the disks. The ideal physical layout may be very different for PP and TW style backups. In particular, a TW backup should probably have blocks $L_i$ and $L_{N_{seg}+i}$ physically contiguous on the disks, since both will have to be read in and compared during recovery [Reut80a].

- Monoplex backups support only fuzzy checkpointing. The monoplex backup schemes are FM, SM, and shadows. Transaction-consistent checkpointing cannot be supported by monoplex backups because a system failure during checkpointing will leave the backup in a state where neither the old nor the new copy will be TC.
- Fuzzy checkpointing schemes support only value (REDO) logging. REDO operations expect to see a consistent database state when they begin and fuzzy backups cannot guarantee that this will be the case. Note that this restriction and the previous one imply that monoplex backups and operation logging cannot be used together.
- If dynamic updates cannot be used in conjunction with 2CFLUSH or COUFLUSH checkpoints unless $S_{seg} = S_{spg}$. Since pages (and the records contained in them) change their physical location over time, there is no way to guarantee that the records in a segment will be physically contiguous if a segment contains more than one (primary) shadow page. Note that this is not a problem for 2CCOPY or COUFLUSH checkpoint policies because they first copy the records in a segment to a special buffer. In the buffer the records are physically contiguous and can be flushed with a single I/O operation.
- Dynamic updates and FASTFUZZY checkpoints cannot be used together at all, since it is possible that shadow pages will be flushed to disk by the checkpointer. This may violate the log write-ahead protocol because the FASTFUZZY checkpointer is not synchronized with logging.
- Dynamic updates and FUZZYCOPY checkpoints can be combined, but the records associated with each segment will vary over time. Because of the lack of synchronization between the checkpointer and transactions, some database records may end up represented more than once in the secondary database, or may not be represented at all. However, any record without a unique backup copy will be represented by REDO information in the log. Therefore, the database can be reconstructed after a crash despite missing records in the backup copy.
- If FASTFUZZY checkpoints are used, then delayed updates must be used. In fact, as we have already mentioned, transactions may not install their updates when FASTFUZZY checkpoints are being taken until their UNDO information is safely on stable storage. This precludes immediate updates.
- For similar reasons, FASTFUZZY checkpoints also preclude the use of group commits. Group commits imply that a transaction's updates may not be safely logged on stable storage before they are seen and flushed by the checkpointer.

### 3.11. 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.

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

The system's 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 recovery policies we have considered.

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. This is a reasonable assumption when value logging is used, since CPU activity will consist mostly of copying after-images to their proper location in main memory. The situation is less clear cut if operation logging is used, since transactions must be re-executed to reproduce their effects. However, we believe that unless transactions are very CPU intensive or the capacities of the processors and the log disks are unbalanced, I/O time will be the dominant component of recovery time.

The various recovery policies that we have described affect recovery time by affecting the size of the log and by affecting the amount of I/O necessary to read in the backup database copy. In addition, one checkpoint policy (FUZZYCOPY) requires that the log be read twice, once backwards for UNDO and once forwards for REDO. In appendix B we present our cost model for recovery time as a function of the recovery policies as well as our model for CPU overhead.

## 4. Results

In this section we present the results of our comparisons of the various recovery policies, and we examine the effects of some of the model parameters on recovery time and recovery overhead. We have already presented, predominantly in Section 2, the model parameters and their default values, which we will use in our comparisons of the recovery policies. All of the parameters and their defaults are listed together in Appendix A.

We have presented recovery policies for five recovery issues; at least two policies for each issue. A *policy combination* is a collection of recovery policies, one for each of the recovery issues. The number of possible policy combinations is quite large. Even considering the inter-policy restrictions, there are more than one hundred valid policy combinations. Obviously, we cannot present a detailed analysis of every combination. Instead, our approach will be to study the recovery issues independently of each other,

and then to use the results of that study to choose an interesting group of policy combinations to examine in more detail.

In general, the performance of a recovery policy is dependent on the other policies that make up a particular policy combination. Thus, it is difficult to make any kind of statement about a recovery policy's performance without considering it in the context of some policy combination. Our approach will be to consider the *relative* performance of pairs of policies (for the same recovery issue). In other words, we consider the advantage or disadvantage of switching from policy $X$ to policy $Y$ (e.g., from FASTFUZZY to COUCOPY checkpoints) without changing the remaining components of the policy combination. Since the advantage may vary depending on which policies make up the rest of the combination, each policy comparison results in an interval (i.e., a maximum and minimum possible difference in performance) rather than in specific value.

For example, Figure 4.1 shows a comparison of the CPU overhead of the three update policies that we considered, namely IMST, DLST, and DLDY. The interval plotted for each pair of policies represents range of the magnitude of the tradeoff between that pair of policies. The figure shows that switching, for example, from a DLDY update policy to an IMST update policy saves at least 5500 instructions per transaction in CPU overhead. The savings might also be higher, approaching 8000 instructions per transaction, depending on which log propagation, logging, checkpointing, and backup policies are used in combination with these update policies. (All of the calculations in this and the next several graphs were carried out using the default parameter values, and with checkpoints taken as quickly as possible given the policy combination.)

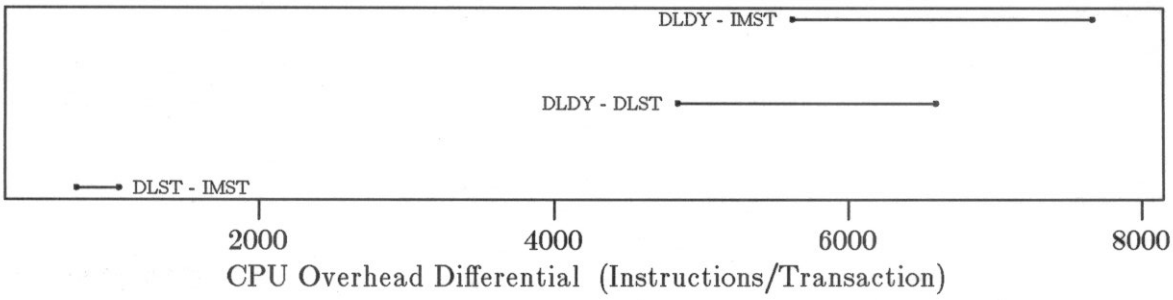


Figure 4.1 - CPU Overhead Comparison of Update Policies

Much can be learned from this type of graph. In particular:

- Performance-critical policy trade-offs are easily spotted. In the figure, switching from the dynamic policy to either static policy produces are large savings in CPU overhead, while switching between the two static policies has relatively little effect.
- Trade-offs that show a great deal of variance, i.e. that show a wide range between their minimum and maximum, are indicative of strong multi-policy dependencies. In other words, the advantage a switching between the policies is strongly

dependent on the remainder of the policy combination. Similarly, trade-offs with little variance are relatively independent. For example, switching between the two static update policies produces only a slight change in CPU overhead regardless of what other policies are used in conjunction with the static updates.

Figures 4.2-4 show similar trade-off graphs for the checkpoint, backup, logging and log propagation policies, respectively. (The logging and log propagation tradeoffs share Figure 4.4.) Note that the scales on these graphs are different so that the plots will be easier to read.

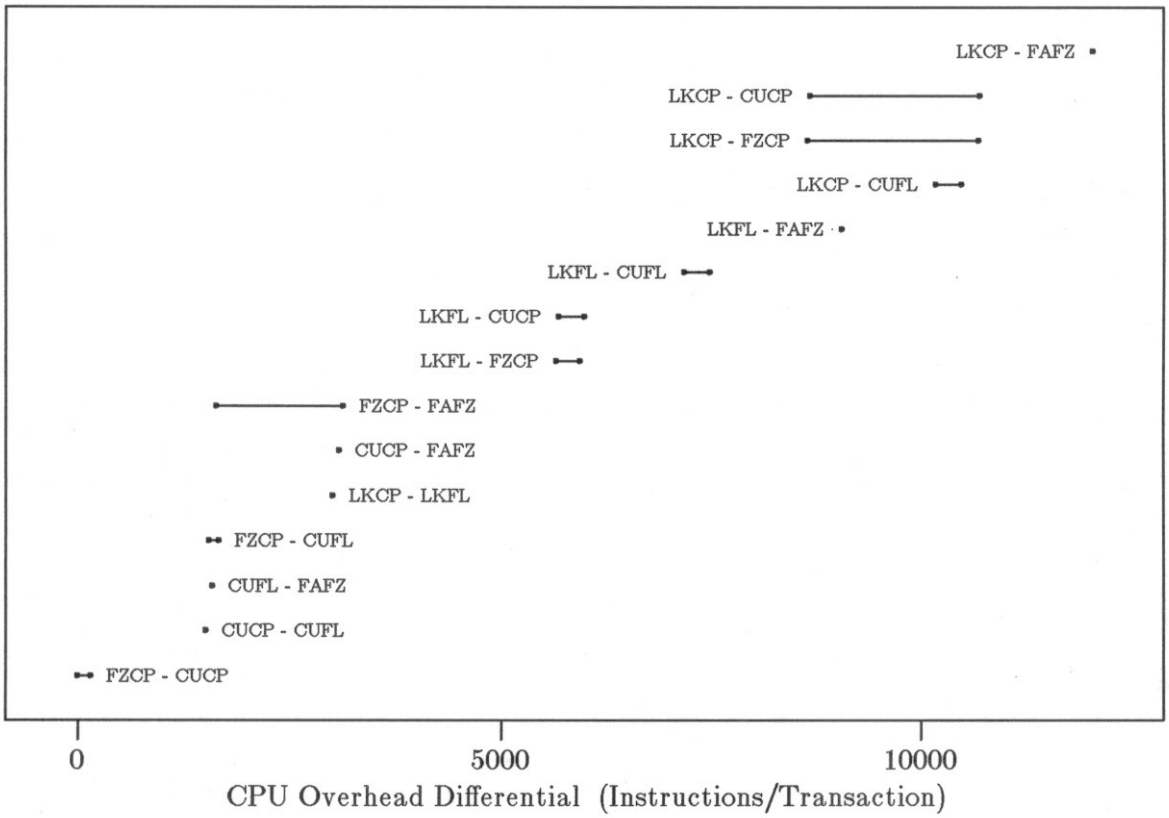


Figure 4.2 - CPU Overhead Comparison of Checkpoint Policies

An inspection of the figures indicates that the checkpointing and update policies normally provide the most significant CPU overhead trade-offs. However, the magnitude of the checkpoint policy trade-offs varies widely from policy pair to policy pair.

Further examination of Figure 4.2 shows that checkpoint schemes can be totally ordered according to their CPU overhead. Switching to a FASTFUZZY (FAFZ) policy from any other checkpoint policy always reduces recovery overhead. The next best is the COUFLUSH (CUFL) policy, which results in less overhead than any checkpoint policy except FASTFUZZY. Continuing in order of increasing CPU overhead we have FUZZYCOPY, COUCOPY, LOCKINGFLUSH, and LOCKINGCOPY.

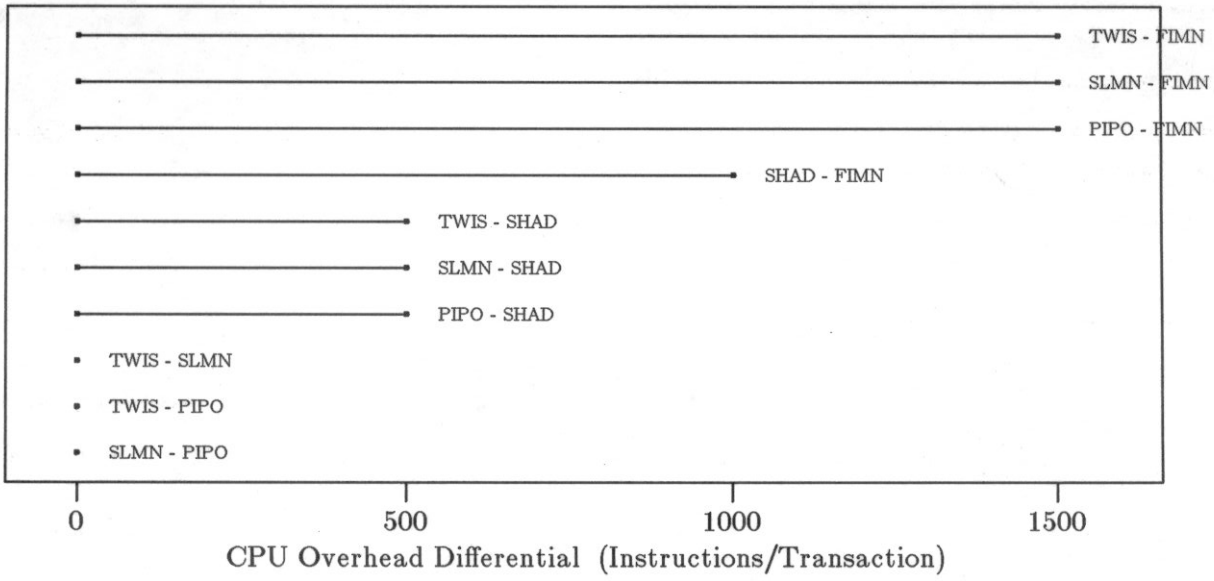


Figure 4.3 - CPU Overhead Comparison of Backup Policies

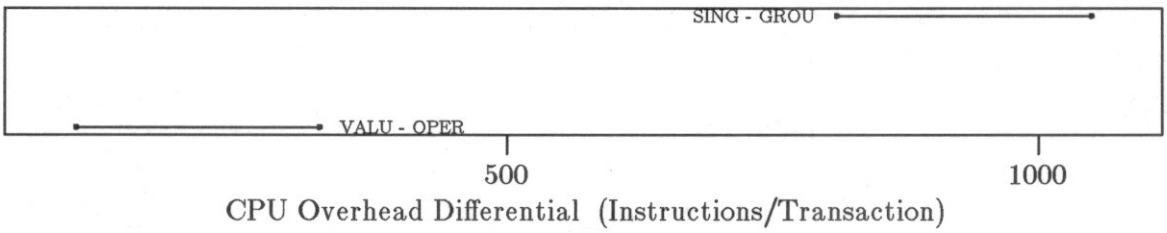


Figure 4.4 - CPU Overhead Comparison of Logging Policies

Figure 4.3 tells a very different story for the backup policy trade-offs. According to the figure, TWIST, PING-PONG, and sliding monoplex (SLMN) backups are indistinguishable in terms of recovery overhead. Switching to fixed monoplex (FIMN) or shadow (SHAD) backups reduces overhead on some cases. However, the differences are not large compared to those seen with the recovery and update policy trade-offs.

The principal reason that the backup style has such a weak effect on recovery overhead is that we have not fixed the intercheckpoint interval. Checkpoints occur as quickly as possible given the particular policy combination. Though changing the backup policy greatly effects the total work done by the checkpointer, it also affects the length of the checkpoint. Thus the overhead *per transaction* changes very little when the backup policy changes. We will see later in this section that if the intercheckpoint interval is held constant the performance differences among the backup policies become apparent.

### 4.1. Recovery Time

Thus far we have been concerned with CPU overhead as our performance metric. The performance model also allows us to study the changes in recovery time that result from policy trade-offs.

We have seen that when checkpoints occur as quickly as possible given the recovery policy combination, changing backup policies does not have a great effect on CPU overhead. However, we might expect that the backup policies would have a stronger effect on recovery time. Figure 4.5 is similar to figure 4.3 except that the performance metric is recovery time rather than CPU overhead. Although some of the trade-offs show a great deal of variance, it is apparent that the backup policy trade-offs are more significant here than they were in figure 4.3.

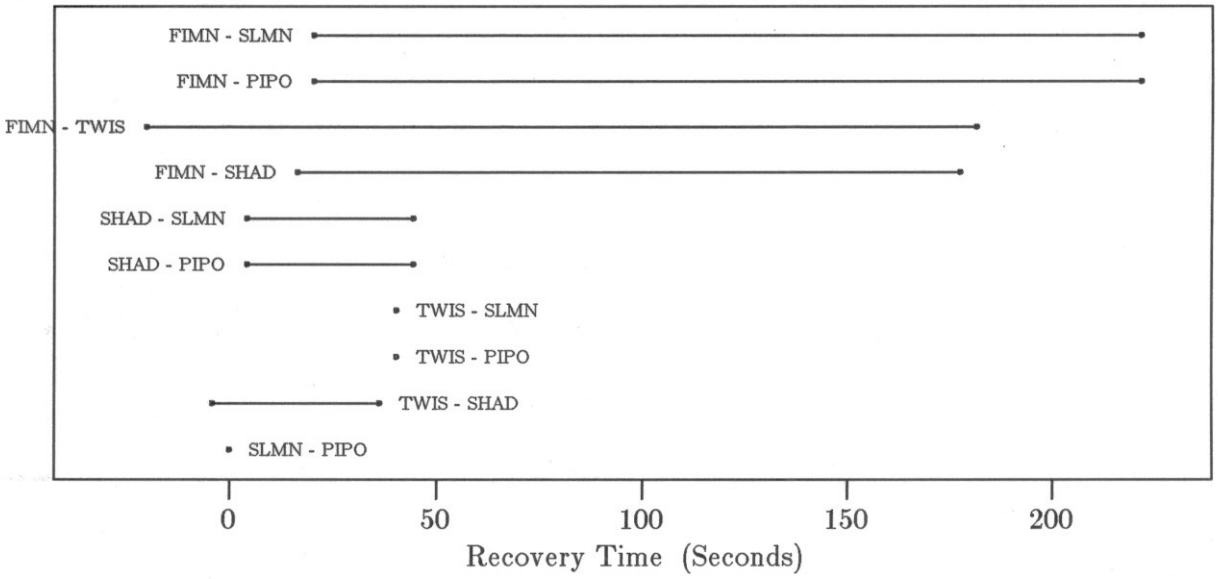


Figure 4.5 - Recovery Time Comparison of Backup Policies

Figure 4.5 is also interesting in that it contains an example of a policy trade-off that is qualitatively uncertain. The trade-offs that we have seen thus far are always beneficial in the same direction, the question is only "how much?". For example, we see that switching from fixed monoplex to sliding monoplex backups always reduces recovery time. However, switching from fixed monoplex to twist backups will reduce the recovery time in some cases, while in others it will actually lengthen it.

### 4.2. Policy Combinations

Thus far we have considered the relative importance of the various recovery policies. We have not considered the absolute effect of specific recovery policy combinations on system performance. If recovery is not a significant contributor to the total cost of running transactions, then it is not worthwhile to put a great deal of effort into picking and choosing recovery policies.

We have selected half a dozen policy combinations for further examination. These are described in Table 4.1. These six policy combinations were chosen for several reasons. First, their performance spans most of the range of performance we observed. Second, the collection of policy combinations illustrates a number of the interesting policy trade-offs that were revealed in the last section.

| name | backup | checkpoint | logging | log prop. | update |
|---|---|---|---|---|---|
| A | SLMN | FAFZ | VALU | SING | DLST |
| B | SHAD | FZCP | VALU | GROU | DLST |
| C | FIMN | FZCP | VALU | GROU | DLDY |
| D | PIPO | CUCP | VALU | GROU | DLDY |
| E | PIPO | LKFL | OPER | GROU | IMST |
| F | TWIS | CUFL | VALU | GROU | IMST. |

Table 4.1 - Recovery Policy Combinations

Figure 4.6 shows the recovery overhead and recovery time for each of the selected combinations. The processor overhead for even the cheapest of these policy combinations may be significant, particularly since CPU cost of the rest of the transaction is likely to be less than that in a disk-based database system. The most expensive policy combinations represent a potentially serious performance bottleneck.

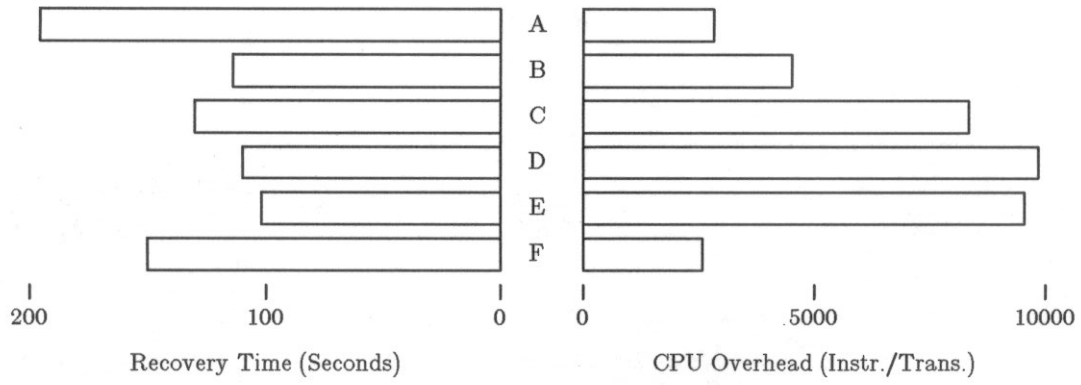


Figure 4.6 - Performance of Selected Recovery Policy Combinations

The significance of the recovery times depends, of course, on the frequency of failures and on the cost of system down time to the application. All of the recovery times reported here might be considered "fast" by the criteria used in [Hagm86a]. However, our recovery times only include those portions of the actual recovery time which are affected by the recovery policy; such delays as restarting the network and spinning

up the disks are not considered. More importantly, we have determined recovery times in an optimistic bandwidth-unlimited environment, i.e. bus contention has been assumed to be insignificant. Thus the relative performance of the various combinations may be a more useful metric than absolute performance, depending on how well the assumptions match actual operating conditions.

### 4.3. Parameter Variations

We are also interested in seeing how the selected policy combinations perform as some of the key model parameters are varied. We will consider variations in transaction load ($\lambda$) and segment size. We will also consider the effects of lengthening the intercheckpoint interval.

Figures 4.7-8 show the variation in CPU overhead and recovery time over a range of transaction loads. The decrease in per-transaction recovery overhead with increasing transaction load shown in the first graph can be ascribed to decreasing per transaction cost for checkpointing. Because checkpoints are taken to run as quickly as possible, the cost *per unit time* of checkpointing is roughly constant given a particular policy combination. Thus as the transaction load increases the per-transaction recovery overhead drop since the same checkpointing overhead is in effect spread over more transactions.

Of course, the total overhead required to complete a checkpoint may increase a great deal with increasing transaction load. This means that the total time to complete a checkpoint increases with the transaction load, which in turn implies that recovery time will suffer since a longer log will have to be processed after a system crash. Figure 4.8 verifies that this is indeed the case.

Thus far we have assumed that checkpoints are taken as quickly as possible, i.e., a new checkpoint begins as soon as the old one is complete. It may be desirable to take checkpoints less frequently, thus trading in increase in recovery time for a decrease in CPU recovery overhead during normal system operation. Figures 4.9-10 show the effects on CPU overhead and recovery time of fixing the intercheckpoint interval to some value in the range of one to ten minutes.

In the first figure we see that the reduction in CPU overhead that can be obtained by slowing the checkpointer is strongly dependent on the recovery policy combination. Combination *A* changes little across the spectrum of intercheckpoint intervals, while the overhead produced by combination *E* decreases rapidly as the intercheckpoint interval increases.

The relatively flat curve of combination *A* is due in large part to the FASTFUZZY checkpoints that are used. The overhead of each fastfuzzy checkpoint is low enough that the per transaction cost is negligible compared to synchronous overhead costs such as log I/O.

Combination *E*'s rapidly decreasing overhead can be attributed to decreasing overhead from transactions restarted for violating the black/white restriction of the *E*'s LOCKINGFLUSH checkpointer. As the checkpoint interval increases, the fraction of time during which the checkpointer is active decreases, thus reducing the likelihood

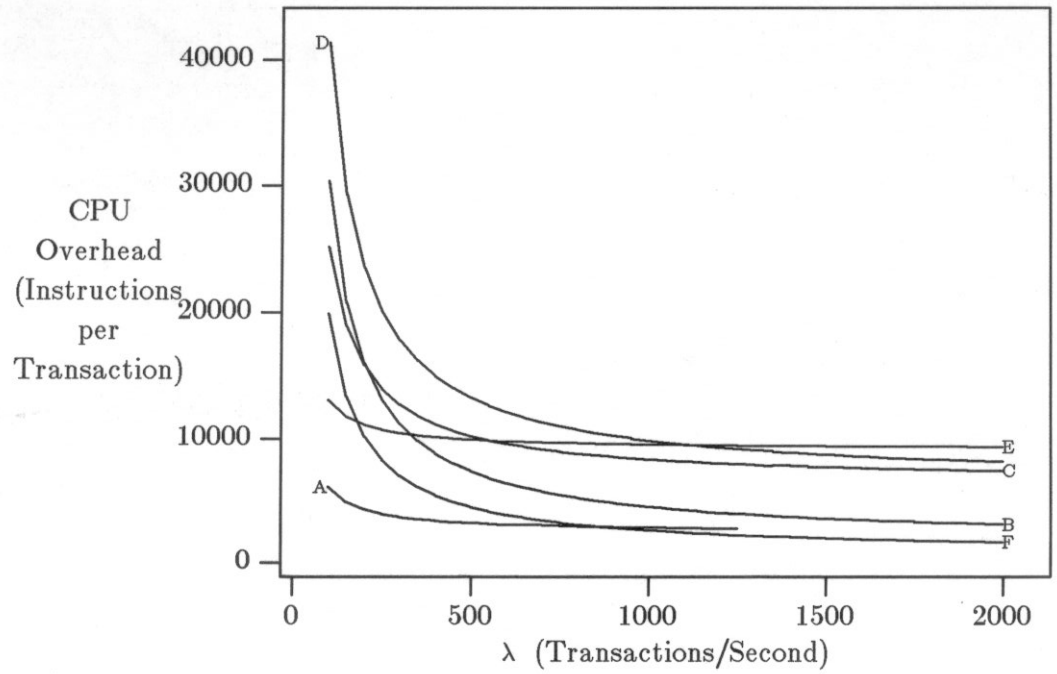


Figure 4.7 - CPU Overhead Variation with Transaction Load

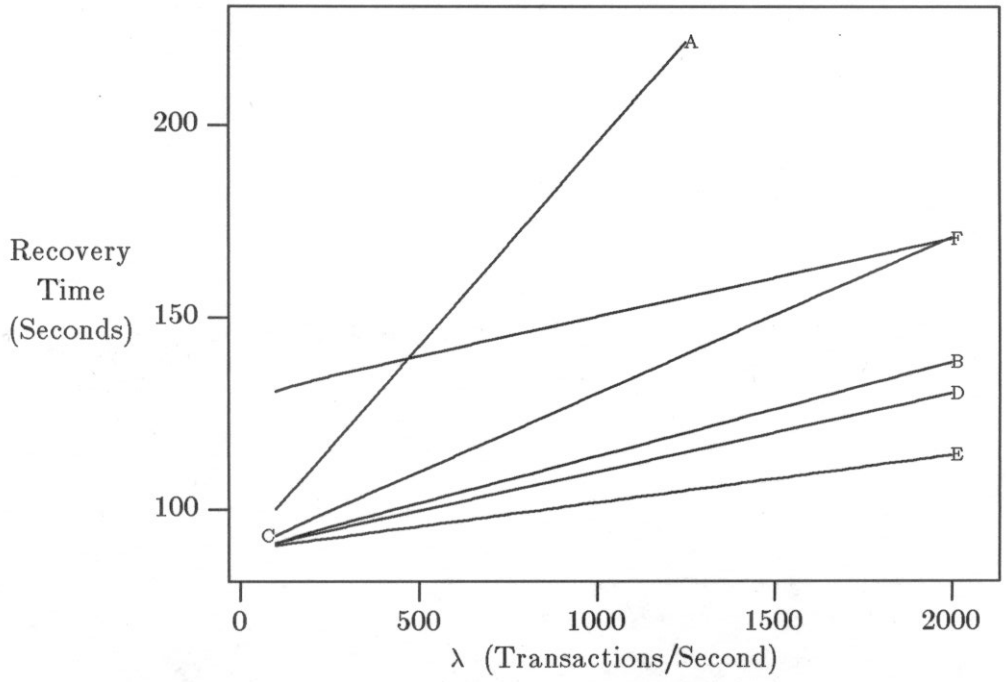


Figure 4.8 - Recovery Time Variation with Transaction Load

that black/white restrictions will be violated.

In figure 4.10, the slope of the rise in recovery time with the intercheckpoint interval is an indication of the volume of log data generated by a policy combination. Combination *A* has a bulky log since it uses a single commit log propagation policy. FUZZYCOPY checkpoints used in combination with an immediate update policy can also generate a relatively bulky log, since UNDO information will have to be flushed to the log disks.

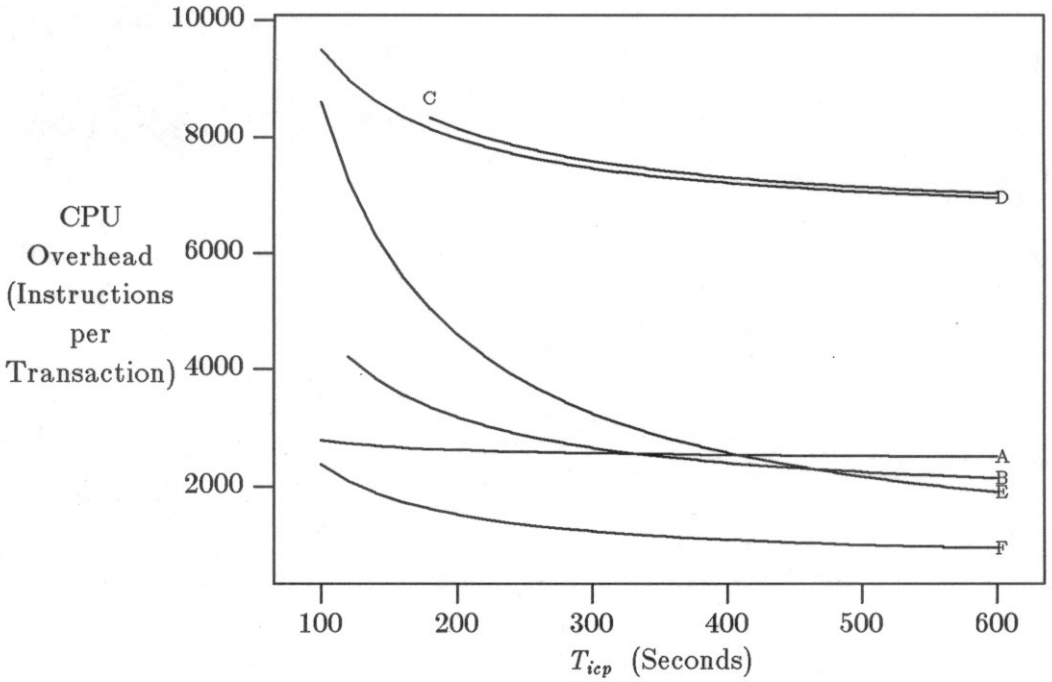


Figure 4.9 - CPU Overhead Variation with Intercheckpoint Interval

Our final experiment concerns the variation in CPU overhead as the segment size ($S_{seg}$) changes. Figure 4.11 shows that four of the combinations show increased overhead with increasing segment size. The increase is caused primarily by an increase in the checkpointing overhead, although several of the combinations suffer an increase in their synchronous recovery overhead as well.

Increasing the segment size produces conflicting effects which drive the per transaction checkpointing overhead in opposite directions. One the one hand, the total cost of a checkpoint is reduced since such overhead costs as initiating disk I/O are charged less frequently when there are fewer, larger segments. However, the total time required to complete the checkpoint also decreases since larger segments can be flushed more efficiently to the backup disks. In most cases this latter effect, which increases the *per transaction* cost of checkpointing, dominates and transactions pay a higher overhead when segments are large. Of course, faster checkpoints also manifest themselves in the form of quicker recovery from system failures, so the higher overhead of larger segments is not without advantage.

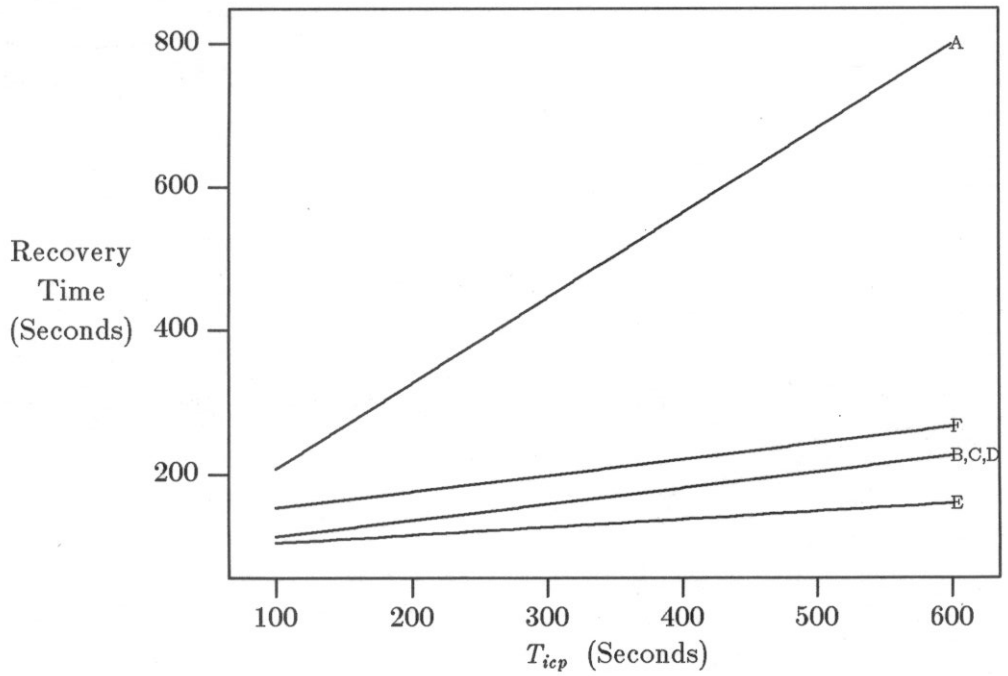


Figure 4.10 - Recovery Time Variation with Intercheckpoint Interval

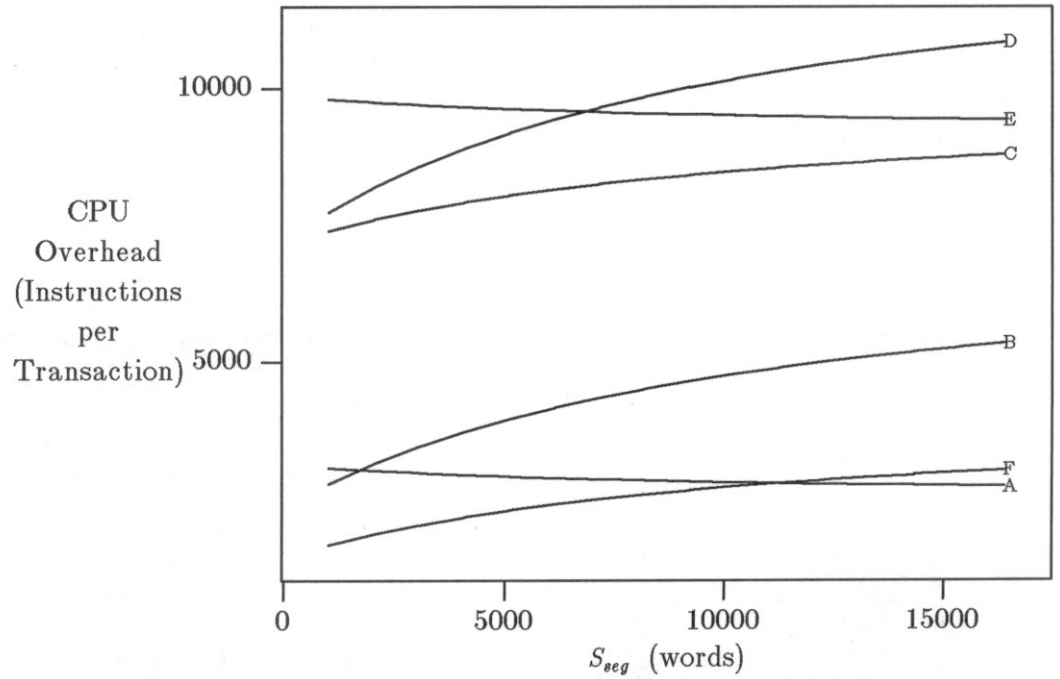


Figure 4.11 - CPU Overhead Variation with Segment Size

## 5. Conclusions

We have presented models of a number of recovery policies for memory-resident database managers and compared them using two performance metrics, CPU overhead and recovery time. CPU overhead is the amount of processor resources (instructions) required for recovery management. Recovery time is the time required to restore an up-to-date consistent primary database after a system failure (e.g., a power outage).

Based on the performance comparisons we have presented, we can draw some general conclusions about the recovery policies. All of these conclusions are meant to be taken in the spirit of rules of thumb and not as hard judgements for or against particular policies.

Update Policy

- Shadow pages, as used in dynamic policy, are too costly to use if they are significantly larger than the record size. If there is memory available for a large indirection table, the cost can be reduced by making the shadows smaller.
- The update policy has little effect on system recovery time unless FUZZYCOPY checkpoints are being used. In the case of FUZZYCOPY checkpoints, a deferred update policy can shorten recovery time my eliminating the need to flush UNDO information to the log disks.

Log Propagation Policy

- At 1000 transactions per second, it is clear that SINGLE commits are not a practical alternative for log propagation. The only advantage to a SINGLE commit policy is that it permits the use of FASTFUZZY checkpoints.

Log Policy

- Switching from VALUE to OPERATION logging has little effect on either CPU overhead or recovery time, thus the choice between these policies is probably best left to be made for reasons not considered in our model, such as ease of implementation. Of course, this could be expected to change for applications with significantly larger record sizes or with larger transactions (more updates) than those considered here. In those situations, OPERATION logging can be expected to reduce both CPU overhead and recovery time through a reduction in log volume.

Backup Policy

- The backup policy makes little difference in CPU overhead unless FUZZYCOPY checkpoints are used. In the case of FUZZYCOPY checkpoints, the "slower" backup policies, such as FMONO and SHADOWS, spread the CPU overhead of segment copying over a longer time, thus reducing the per-transaction overhead. (This would be true of other "copy"-type checkpoint policies, except that they cannot be used in conjunction with the slower backups, which are not duplex.)

- The "slow" monoplex backups FMONO and (to a lesser extent) SHADOWS, significantly increase recovery time by reducing the maximum checkpoint rate. These backup policies are therefore not appealing unless the application's log volume is low, i.e., transactions update only few small records and group commits are used.
- We say that the database is *saturated* if most of the database segments are dirty at each checkpoint. Our default parameter settings produce a saturated database. In the case of saturated databases, PINGPONG and SMONO backups perform equally well.
- TWIST backups perform as well as PINGPONG backups during normal operation, but increase recovery time significantly due to the greater volume of data that must be read into primary memory from the backup disks. TWIST backups may permit faster checkpoints than PINGPONG when the database is not saturated, since in that situation TWIST backups will require fewer segment flushes while transactions are being processed.

Checkpoint Policies

- As we have already noted, the checkpoint policies can be totally ordered according to CPU overhead given our default parameters. However, the differences among the two FUZZY policies and the two COU policies are fairly small. The black/white locking policies, on the other hand, are too costly because of the high cost of restarting aborted transactions. Though the CPU overheads of the FUZZY and COU policies are similar, each has distinct disadvantages. The COU policies require primary memory to hold the checkpointer's database snapshot. FASTFUZZY checkpoints preclude GROUP log propagation, and thus are not really useful when the transaction load is high. FUZZYCOPY checkpoints pay a relatively high overhead cost for copying data, and may suffer from long recovery times if used without DELAYED updates.
- The distinction between "copy"-style and "flush"-style checkpointing has little effect on recovery time.
- COU and LOCKING checkpoints have similarly recovery times unless the as long as GROUP log propagation is used. In the case SINGLE log propagation, LOCKING checkpoints recover more slowly than COU because of the increase in log bulk due to transactions aborted because of the black/white restrictions.

What is clear from these rules is that there is no single best recovery policy or policy combination. We have presented some simple rules for guidance, however the selection of a recovery policy must be made in the context of the parameters of a particular application.

As we have already mentioned, there are other possible performance metrics that we have not studied quantitatively in this paper. Recovery mechanisms consume resources other than CPU time, such as shared data ("consumed" by locking), I/O

bandwidth, and primary and secondary storage. Some of these metrics, such as storage consumption, are relatively simple to compute; we have only not presented them here for lack of space. Others, like the delays caused by locking data objects, are more difficult to model analytically.

We are currently implementing a main memory recovery testbed with which will be able to experimentally evaluate many of the policies described here. We hope to be able to measure locking and other delays using the testbed, as well as to verify the overhead and recovery time models presented here. We also plan to use the testbed to study issues other than those considered here, such as the scheduling of transactions and other system processes (e.g., the checkpointer) in a main-memory environment.

**References**

Agra85a.

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

Borg83a.

Borg, Anita, Jim Baumbach, and Sam Glazer, "A Message System Supporting Fault Tolerance," *Operating Systems Review*, vol. 17, no. 5, pp. 90-99, Oct., 1983.

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.

Eich87a.

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

Garc83a.

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

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.

Gray85a.

Gray, Jim, Bob Good, Dieter Gawlick, Pete Homan, and Harald Sammer, "One Thousand Transactions Per Second," *Proceedings of IEEE COMPCON*, San Francisco, CA, February, 1985.

Gray76a.

Gray, J. N., R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modeling in Data Base Management Systems*, ed. G. M. Nijssen, pp. 365-394, North Holland Publishing Company, 1976.

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.

Lehm86a.

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

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.

Reut80a.

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

Reut84a.

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

Sale86b.

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.

Sale86a.

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

Ston87a.

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

## Appendix A - Model Parameters

| 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 |
| $C_{trans}$ | transaction cost | 25000 | instructions |
| $T_{seek}$ | I/O delay time | 0.03 | seconds |
| $T_{trans}$ | transfer time constant | 3 | $\mu$seconds/word |
| $N_{shadows}$ | disk shadow blocks | 10 | blocks |
| $N_{disks}$ | number of disks | 60 | disks |
| $f_{\log}$ | fraction of disks for logging | 0.67 | |
| $S_{db}$ | database size | 256 | Mwords |
| $S_{rec}$ | record size | 32 | words |
| $S_{spg}$ | shadow size | 1024 | words |
| $S_{lpg}$ | log page size | 1024 | words |
| $S_{seg}$ | segment size | 8192 | words |
| $S_{init}$ | transaction log overhead | 32 | words |
| $S_{op}$ | operation log entry size | 32 | words |
| $\lambda$ | arrival rate | 1000 | transactions/second |
| $N_{ru}$ | number of updates | 5 | records/transaction |
| $M$ | degree of multiprogramming | 10 | transactions |
| $p_{fail}$ | transaction failure probability | 0.05 | |

## Appendix B - Cost Model

In this section we present the cost models for the recovery policies we have outlined. There are two metrics for which we ultimately want to arrive at expressions: CPU overhead and recovery time (from a system failure). Before we present the models, we first derive an expression for the length of the intercheckpoint interval, a parameter that will be used in the determination of both recovery time and CPU overhead.

### Intercheckpoint Interval

The intercheckpoint interval is the length of time between the beginnings of checkpoints. The minimum possible intercheckpoint interval is a function of the recovery policies that have been selected since these determine how much data must be flushed to the backup disks and how quickly that data can be flushed. It is this minimum intercheckpoint interval, $T_{icpmin}$, that will be calculated in this section.

Two assumptions are used to determine the intercheckpoint interval. We assume that the length of the interval can be approximated by the time necessary to write out the necessary pages to the disk. Although the checkpointer may result in a great deal of CPU overhead (affecting transaction throughput), the intercheckpoint *time* is dominated by I/O delays. Secondly, we assume that partial checkpoints are used whenever possible (i.e., as long as sliding monoplex backups are not being used.)

According to our model of the disks, the number of database segments that can be written out to the backup disks in time $t$ is given by

$$N_{io}(t) = N_{disks}(1 - f_{\log})\frac{t}{T_{seek} + T_{trans}S_{seg}}$$

Transactions update $N_{ru}$ records each, with the update probability being distributed uniformly over all records in the database. As long as the number of segments in the database is much larger than $N_{ru}$, then the number of segments updated by a transaction, $N_{su}$, will be approximately $N_{ru}$. If

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

is the number of segments in the database, the probability than any particular transaction $T$ will not update that segment is given by

$$\left[1 - \frac{N_{su}}{N_{seg}}\right]$$

Over a time interval of length $t$ we expect $\lambda t$ transactions. The probability that at least one of these will update a particular database segment is one minus the probability that all of them will miss that segment, or

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

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

$$N_{dirty}(t) = N_{seg}\left[1 - \left[1 - \frac{N_{su}}{N_{seg}}\right]^{\lambda t}\right]$$

Let $t_{icp}$ represent the intercheckpoint interval. The number of pages the checkpointer must flush to the backup disks during the intercheckpoint interval, written $N_{flush}(t_{icp})$, is a function of the number of dirty pages and of the backup policy, as described in the following:

- FMONO backups flush each dirty page to the backup disks twice.
- SHADOW backups require only one flush per dirty page, but the indirection table must be updated carefully (i.e., twice) for every $N_{shadow}$ pages that are flushed.
- TWIST backups flush each dirty page once yet have no indirection table to update.
- Each of the two backup copies under a PINGPONG backup policy sees updates only during every other checkpoint. Thus the currently active copy sees as many dirty pages as a monoplex backup would see during a checkpoint interval of twice the duration. Each dirty page is flushed only once to the active backup.
- SMONO backups require full checkpoints, thus all database segments are flushed during every checkpoint, no matter what the value of $N_{dirty}$.

These relationships between $N_{flush}$ and $N_{dirty}$ are summed up in Table B1. The two extra segment flushes charged to SMONO and PINGPONG backups are for the one-time (per checkpoint) careful updating of the base (SMONO) or current backup (PINGPONG) pointers on disk.

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

We can find the minimum possible value of $t_{icp}$, namely $t_{icpmin}$, by setting $N_{flush}(t_{icpmin}) = N_{io}(t_{icpmin})$ and solving for $t_{icpmin}$. When this equality holds, the system is flushing dirty segments at the same rate they are being created by the transactions. The equality results in an expression with the general form $at_{icpmin} + log(t_{icpmin}) + b = 0$, where $a$ and $b$ are constants. We solve this expression numerically to arrive at a value for $t_{icpmin}$.

### CPU Overhead

We will model the CPU recovery overhead in two parts: *synchronous* and *asynchronous* overhead. The synchronous overhead, $C_{synch}$ is the per transaction cost of recovery operations occurring synchronously with transaction execution, such as the creation of

the log. The asynchronous overhead, $C_{asynch}$ is the cost of completing a database checkpoint. This cost is divided evenly over all transactions occurring during the intercheckpoint interval to get a per-transaction measure of the asynchronous recovery overhead. Since there are $\lambda t_{icp}$ transactions during an intercheckpoint interval, the total CPU recovery overhead, $C_{tot}$, is given by

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

**Synchronous Recovery Costs**

In this section we develop and expression for $C_{synch}$, the per transaction synchronous CPU recovery overhead. In fact we will develop two such expressions, one for successful and one for unsuccessful transaction. These are then combined into a single expression using $p_{fail}$, the probability that a transaction will be unsuccessful.

An "unsuccessful transaction" is a transaction that fails as a result of something other than the actions of the recovery manager, e.g. a data input error. $p_{fail}$, the probability that any transaction is unsuccessful, is a model parameter. There is also a probability that a transaction, whether ultimately successful or not, may have to be aborted and restarted as a result of actions of the recovery manager. The probability of this, $p_{restart}$, is a function of the recovery policy combination We will develop expressions for $p_{restart}$ shortly.

As noted in [Agra85a], the *entire cost*, up to the point of restart, of executing a transaction aborted by the recovery manager must be considered part of the overhead of the recovery mechanism. If not for recovery management, only the cost of the restarted transaction would have been incurred. Since we assume that failed transactions fail halfway through their execution, an appropriate expression for $C_{synch}$, the total synchronous CPU overhead of the recovery mechanism is

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

where $C_{succ}$ and $C_{fail}$ are the synchronous recovery overheads of successful and unsuccessful transactions, respectively.

The restart probability is a function of the checkpoint policy. Fuzzy checkpoints never cause transactions to restart, thus $p_{restart} = 0$ for both types of fuzzy checkpoints. If copy-on-update checkpoints are used, those transactions that are active when a checkpoint begins are aborted to bring the database into a transaction-consistent state.† During an intercheckpoint interval there are $\lambda T_{icp}$ transactions, thus for either kind of copy-on-update checkpoint we have

$$p_{restart} = \frac{M}{\lambda T_{icp}}$$

---

† Alternatively, the checkpointer can wait for active transactions to complete before starting. During this period, new transactions cannot be initiated.

Two-color checkpoints are a little more complicated. Let $W$ be the fraction of the database that is colored white, and let $P[OK \mid W = \omega]$ 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 transactions small.) This occurs when all segments touched by the transaction are the same color, so

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

If checkpoints are always occurring (i.e., $t_{icp} = t_{icpmin}$) we can assume that $W$ is uniformly distributed from zero to one. Using this and the symmetry of the previous expression, we get

$$P[OK] = \int_0^1 2\omega^{N_{su}} d\omega = \frac{2}{N_{su} + 1}$$

The restart probability, $p_{restart}$, is simply $1 - P[OK]$.

If $t_{icp} > t_{icpmin}$, and the checkpoint proceeds as quickly as possible, then there is a period of duration $t_{icp} - t_{icpmin}$ following every checkpoint during which the checkpointer is idle. During this period, no transactions will have to be restarted, since the entire database will be the same color. We weight our expression to take this into account and arrive at an expression for $p_{restart}$ when two-color checkpoint policies are used:

$$p_{restart} = \frac{t_{icpmin}}{t_{icp}}(1 - P[OK]) = \frac{(N_{su} - 1)t_{icpmin}}{(N_{su} + 1)t_{icp}}$$

**Successful Transactions**

We are now in a position to develop expressions for $C_{succ}$, the synchronous recovery overhead of successful transactions. $C_{succ}$ depends on the amount of log data generated by transactions and flushed by the system to the log disks. We will consider $D_{redo}$, the amount of REDO log data, and $D_{undo}$, the amount of UNDO log data, separately. The various recovery policies treat the two types of log data differently.

If value REDO logging is used, each updated record must be copied to the log, along with the begin- and end-transaction records. Thus

$$D_{redo} = S_{rec}N_{ru} + S_{init}$$

($S_{init}$ is a model parameter representing the log data overhead of each transaction, i.e., the size of the begin- and end-transaction records in the log.) If operation REDO logging is used, the volume of REDO log data is simply

$$D_{redo} = S_{op} + S_{init}$$

where $S_{op}$, also a model parameter, is the size of the operation log entry.

UNDO logging is needed only for immediate update policies. If a delayed update policy is employed, $D_{undo} = 0$. UNDO logging is always done by value, thus it involves copying the old values of modified records to the log. If FUZZYCOPY checkpoints are used, it is necessary to flush UNDO information to the log disks. In this case the

UNDO and REDO logs are combined into a single log, and the amount of UNDO information is

$$D_{undo} = S_{rec} N_{ru}$$

Otherwise, UNDO information is maintained in a separate log. In this case, begin- and end-transaction records must be entered in the UNDO log as well as the REDO log. Thus we have

$$D_{undo} = S_{rec} N_{ru} + S_{init}$$

The number of log pages that a particular transaction is responsible for depends on the amount of log data it generates and on the log propagation policy. Two kinds of log pages are distinguished according to which log they are part of. The *primary* log is flushed to the log disks; it contains REDO information and may contain UNDO information as well. The *secondary* log contains only UNDO information and is not flushed to the log disks. Depending on the recovery policy combination there may be no secondary log at all.

In the case of transaction-consistent checkpoints, UNDO information, if any, is placed in the secondary log. The number of secondary log pages per transaction is

$$N_{lsec} = \frac{D_{undo}}{S_{lpg}}$$

The number of primary pages depends on the log propagation policy. If immediate propagation is used, any primary log page partially filled by a (successful) transaction counts as a full page, since it is immediately flushed to the log disks. (The empty portion of the page is wasted.) Thus the number of primary log pages per transaction is

$$N_{lprime} = \left\lceil \frac{D_{redo}}{S_{lpg}} \right\rceil$$

With group propagation, the empty fraction of a partially-filled log page can be used by other transactions, so

$$N_{lprime} = \frac{D_{redo}}{S_{lpg}}$$

With fuzzy checkpoints, the situation is much the same, except that UNDO data, if any, must be flushed to the log disks. Thus all log data goes to the primary log and we have $N_{lsec} = 0$. For immediate log propagation

$$N_{lprime} = \left\lceil \frac{D_{redo} + D_{undo}}{S_{lpg}} \right\rceil$$

For group propagation:

$$N_{lprime} = \frac{D_{redo} + D_{undo}}{S_{lpg}}$$

The cost model for the synchronous recovery overhead of successful transactions is presented in Figure B1. The model is not intended to be an algorithmic description of a

transaction. The ordering of the operations shown is for purposes of presentation only.

**FOREACH** update **DO** $\{N_{ru} = N_{pu}$ times$\}$
  **IF** COU or TWO-COLOR updates **THEN**
    lock and unlock segment $\{2C_{lock}\}$
  **IF** FUZZYCOPY or TWOCOLOR checkpoints **THEN**
    update segment's LSN $\{C_{lsn}\}$
  **IF** delayed updates **THEN**
    **IF** dynamic updates **THEN**
      allocate and deallocate shadow $\{2C_{alloc}\}$
      copy page to shadow $\{S_{spg}\}$
    **ELSE** (static updates)
      allocate and deallocate shadow $\{2C_{alloc}\}$
      copy updated record back to its location $\{S_{rec}\}$
**END_FOR**
**IF** *COU* checkpointing **THEN**
  **FOREACH** segment that needs to be copied **DO** $\{N_{cou}/\lambda t_{icp}$ times$\}$
    allocate buffer for segment $\{C_{alloc}\}$
    copy segment to buffer $\{S_{seg}\}$
allocate and de-allocate pages in primary log buffer $\{2N_{lprime}C_{alloc}\}$
allocate and de-allocate pages in secondary log buffer $\{2N_{lsec}C_{alloc}\}$
**IF** FUZZYCOPY or TWOCOLOR checkpoints **THEN**
  assign LSN to log pages $\{N_{lprime}C_{lsn}\}$
copy REDO data to log buffer(s) $\{D_{redo}\}$
copy UNDO data to log buffer(s) $\{D_{undo}\}$
propagate primary log pages to log disks $\{N_{lprime}C_{io}\}$

Figure B1 - CPU Costs for Successful Transactions

In the figure, the cost of each operation is shown in brackets following its description. The figure makes use of the log parameters just calculated, and the parameter $N_{cou}$, calculated below. When dynamic updates are used, we have assumed that the number of shadow pages updated, $N_{pu}$, equals $N_{ru}$, the number of records updated. This follows from our earlier assumption that $N_{ru} = N_{su}$ provided that $S_{spg} \leq S_{seg}$.

Given a set of recovery policies, an expression for $C_{succ}$ can be determined from Figure B1 by adding the appropriate costs within loops and multiplying by the loop multipliers. For example, if TWOCOLOR checkpoints and DLST updates are used, we get

$$C_{succ} = N_{ru}(2C_{alloc} + S_{rec} + 2C_{lock} + C_{lsn}) + N_{lprime}(2C_{alloc} + C_{lsn} + C_{io}) + 2N_{lsec}C_{alloc} + D_{redo} + D_{undo}$$

$N_{cou}$ represents the number of segments copied-on-update by transactions when copy-on-update checkpoints are used. During an intercheckpoint interval, the checkpointer will be active for a time $t_{icpmin}$. During this period, transactions will dirty $N_{dirty}(t_{icpmin})$ database segments. If we assume that the checkpointer is, on the average,

halfway through the database while it is active, then

$$N_{cou} = \frac{N_{dirty}(t_{icpmin})}{2}$$

segments must be copied by transactions. Only half of the dirtied segments must be copied, since the copy only occurs if the checkpointer has not yet reached the dirtied page. In Figure B1, $N_{cou}$ is divided by $\lambda t_{icp}$, the number of transactions during an inter-checkpoint inteval, to get the expected number of copies per transaction.

**Unsuccessful Transactions**

The cost expression for unsuccessful transactions is similar to that for successful transactions. As already noted, we assume that transactions fail halfway through their execution. Thus the main loop in Figure B1 is only traversed half as many times, i.e., $N_{ru}/2$, for unsuccessful transactions.

Another difference between successful and unsuccessful transactions is the volume of log data they generate. If immediate updates are used, transactions make updates (and thus log entries) throughout their lifetime. Since they fail halfway through, value logging immediate update transactions generates

$$D_{redo} = \frac{S_{rec}N_{ru}}{2} + S_{init}$$

words of REDO data. If operation logging is used, or if delayed updates are used, then the transaction normally would write nothing (except its initialization record) to the log until it finished. Thus in these cases the REDO log volume is simply

$$D_{redo} = S_{init}$$

The situation is similar for UNDO data. If UNDO is needed, the amount is

$$D_{undo} = \frac{S_{rec}N_{ru}}{2} + S_{init}$$

if both primary and secondary logs are used, and

$$D_{undo} = \frac{S_{rec}N_{ru}}{2}$$

if the REDO and UNDO information is kept in a single log.

Finally, unsuccessful transactions that use UNDO logging pay an additional cost for rollback, i.e., to copy the UNDO information back into the database. This cost is

$$\frac{S_{rec}N_{ru}}{2}$$

instructions per transaction.

**Asynchronous Recovery Costs**

Cost models for copy-on-update, two-color, and fuzzy checkpoints are given in Figure B2. Like the synchronous overhead model shown in Figure B1, the models presented in Figure B2 are not algorithmic and are not intended to define checkpointing protocols.

The checkpointer puts a begin-checkpoint record in the transaction log each time a new checkpoint begins. We assume this is the same size as a begin/end transaction record, $S_{init}$. In addition, fuzzy and two-color checkpoints must log the active transaction list each time a checkpoint begins. We assume that the active transaction list takes $MS_{init}$ words of log. Thus, $S_{chklog}$, the size of the checkpoint log entry, is

$$S_{chklog} = (M + 1)S_{init}$$

for fuzzy and two-color checkpoints and

$$S_{chklog} = S_{init}$$

for the COU checkpoints.

A critical parameter for the checkpoint cost model is $N_{chk}$, the number of segments that need to be checkpointed. If sliding monoplex backups are used, then $N_{chk} = N_{seg}$ because SM backups require full checkpoints. If PINGPONG backups are used, any segments dirtied over the last two checkpoint intervals must be checkpointed, since each backup copy is updated only during every second checkpoint. Thus $N_{chk} = N_{dirty}(2t_{icp})$ for ping-pong backups. For other backup styles $N_{chk}$ is simply $N_{dirty}(t_{icp})$, the number of pages dirtied in a checkpoint interval.

(de)allocate space for checkpoint log entry $\{\frac{S_{chklog}}{S_{lpg}} 2C_{alloc}\}$
copy checkpoint log entry to log $\{S_{chklog}\}$
flush checkpoint log entry to log disks $\{C_{io}\}$
**FOREACH** database segment **DO** $\{N_{seg}$ times$\}$
    **IF** COU or TWOCOLOR checkpoint **THEN**
        lock and unlock segment $\{2C_{lock}\}$
**END_FOR**
**FOREACH** segment that must be checkpointed **DO** $\{N_{chk}$ times$\}$
    **IF** 2CCOPY or FUZZYCOPY checkpoints **THEN**
        copy segment to special buffer $\{S_{seg}\}$
        check segment's log sequence number $\{C_{lsn}\}$
    flush segment to backup disks $\{\frac{N_{flush}(t_{icp})}{N_{chk}} C_{io}\}$
**END_FOR**
**IF** COUCOPY checkpoints **THEN**
    **FOREACH** segment outside the buffer that must be checkpointed **DO** $\{\frac{N_{seg} - N_{cou}}{N_{seg}} N_{chk}$ times $\}$
        copy segment to special buffer $\{S_{seg}\}$
    **END_FOR**

Figure B2 - CPU Costs for Checkpoints

As we have already noted, the total asynchronous recovery cost is divided by the number of transactions per intercheckpoint interval to determine $C_{asynch}$, the contribution of the asynchronous recovery component to the total transaction overhead.

**Recovery Time**

As already noted, we assume that the recovery time is dominated by the sum of the time to read in the backup database copy plus the time to read in the log. We also assume that the system failure occurs halfway through an intercheckpoint interval.

$T_{back}$, the time to read in the backup database copy, is straightforward to compute. For fixed monoplex backups, we simply read in the backup copy from its fixed location, taking time

$$T_{back} = N_{seg}(T_{seek} + T_{trans}S_{seg})(1 - f_{\log})N_{disks}$$

Sliding monoplex, shadow, and ping-pong backups require that an additional segment be read in to determine the location of the database on secondary storage. Thus for these backup styles we have

$$T_{back} = (N_{seg} + 1)(T_{seek} + T_{trans}S_{seg})(1 - f_{\log})N_{disks}$$

For twist backups, the segments to be read in are twice the size of the main memory segment size, so

$$T_{back} = N_{seg}(T_{seek} + 2T_{trans}S_{seg})(1 - f_{\log})N_{disks}$$

The time to read in the log is determined by a number of factors. The logging, log propagation, and update policies determine the rate at which the log is created, i.e. the number of log pages produced per transaction. The checkpoint policy determines how far back the log must be read. For example, COU checkpoints require that the log be read as far back as the "begin checkpoint" entry of the most recent completed checkpoint. The the backup and checkpoint policies also have an effect on the log reading time because they determine the checkpointing rate. Faster checkpoints mean less log to read at recovery time.

As already mentioned, COU checkpoints require that the log be read from the beginning of the most recently completed checkpoint. The TWOCOLOR checkpoint policies have a similar requirement. FUZZY checkpoints require that the log be read past this point, to the beginning of the oldest transaction that was active at the time the checkpoint began. (We assume that the time for this excursion past the beginning of the most recent complete checkpoint is short and ignore the extra time involved.) FUZZYCOPY checkpoints require that the log be read twice, once forward and once backward, because both UNDO and REDO records must be processed.

We have already determined the rate, $N_{lprime}$, at which transactions create log pages. Let $n_{lps}$ and $n_{lpf}$ be the values of $N_{lprime}$ for successful and unsuccessful transactions, respectively. We can express the total log page creation rate, $n_{lptot}$ by

$$n_{lptot} = (1 - p_{fail})n_{lps} + (p_{fail} + p_{restart})n_{lpf}$$

By our assumption, the most recent complete checkpoint began $3t_{icp}/2$ seconds before

the failure. Thus the total number of log pages produced since the beginning of that checkpoint is

$$\frac{3\lambda t_{icp} n_{lptot}}{2}$$

Log pages can be read from the disk at the rate of

$$\frac{N_{disks} f_{\log}}{T_{seek} + T_{trans} S_{lpg}}$$

log pages per unit time. So the log replay time under any checkpoint policy except FUZZYCOPY is given by

$$T_{\log} = \frac{3\lambda t_{icp} n_{lptot} (T_{seek} + T_{trans} S_{lpg})}{2 N_{disks} f_{\log}}$$

The time for FUZZYCOPY checkpoints is twice this.