SYSTEM M: A TRANSACTION PROCESSING SYSTEM FOR MEMORY RESIDENT DATA

Hector Garcia-Molina
Kenneth Salem

CS-TR-195-88

December 1988

System M: A Transaction Processing System for Memory Resident Data

Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, N.J. 08544

Kenneth Salem

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

System M is an experimental transaction processing system that runs on top of the Mach operating system. Its database is stored in primary memory. This paper describes the structure and algorithms used in System M. The checkpointer is the component that periodically sweeps memory and propagates updates to a backup database copy on disk. Several different checkpointing (and logging) algorithms were implemented, and their performance was experimentally evaluated.

System M: A Transaction Processing System for Memory Resident Data

Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, N.J. 08544

Kenneth Salem

Department of Computer Science
University of Maryland
College Park, MD 20742

1. Introduction

System M is an experimental transaction processing system with a memory-resident database. It has been implemented at Princeton as part of the Massive Memory Machine Project. System M was built with three goals in mind:

(1) *To evaluate the potential gains of memory resident databases.* Year by year, semiconductor memory is becoming cheaper and memory chip densities are increasing. It is now feasible to store in main memory realistically large databases. The improvements in performance can be great: I/O is substantially reduced, transaction context switches (and associated CPU cache flushes) are cut, lock contention is decreased, more efficient memory search structures [Lehm85a] and query processing [Bitt87a, Lehm86a] can be used, and so on.

(2) *To explore the architecture and algorithms best suited for memory-resident databases.* A memory-resident transaction processing system (MTPS) could be implemented simply as a disk-based system (DTPS) with a buffer that happens to be large enough to hold the entire database. This approach fails to capitalize on many of the potential advantages that memory-residence offers. System M, on the other hand, has been implemented from scratch with memory-residence in mind. As we will see in this paper, this leads to a novel internal process architecture and recovery strategies.

(3) *To provide an experimental testbed for comparing algorithms.* In particular, System M implements several checkpointing and logging strategies. (The checkpointer is the component that periodically sweeps memory, propagating changes to the backup

database on disk.) These have been experimentally compared and the results are given in this paper.

The logging and checkpointing components are probably the most critical components of a MTPS. Unlike the rest of the system, they still have to execute expensive I/O operations. Thus, it is essential to decouple these components as much as possible from the rest of the system, so that transaction processing rarely has to wait for logging and checkpointing. How well the various checkpointing and logging strategies achieve this goal will be illustrated by our performance results. MTPS checkpointing has other interesting features as well. For example, the checkpointer's updates of the backup can follow a sweep pattern and are not random, i.e., disk access patterns are not driven directly by the data requirements of transactions. This reduces disk seek times and hence the time required to update the backup.

A far as we know, there is only one other running, documented MTPS designed explicitly for memory–resident data: IMS/VS Fastpath [Gawl85a]. The commercial success of Fastpath attests to the advantages of memory–resident data. However, public information on the internals and performance of Fastpath is limited, and as far as we know, its algorithms have not been compared to alternatives. Thus we feel that System M can contribute to our knowledge of MTPSs.

System M (like Fastpath) runs on conventional hardware; it does not rely on the existence of special purpose or functionally segregated processors, or on non–volatile primary memory. Additional performance improvements may be obtainable via such hardware aids [Eich87a, Lehm87a, Ston88a]. The cost effectiveness of such hardware is discussed in [Garc87a]. System M assumes that the entire database is memory resident. This does not rule out the existence of slow archival storage. One can think of a system as having two databases (as in Fastpath): one memory–resident that accounts for the vast majority of accesses, and a second on archival storage [Ston87a].

We begin by describing the major components of System M. In Section 3 we present the checkpointing and logging strategies studied, while Section 4 contains the experimental results.

2. System M

In this section we describe the overall design and implementation of System M. The system is implemented in C [Kern78a] on a VAX 11/785 with a 128 Mbyte main memory and running the Mach [Acce86a] operating system. All I/O (to the log and the backup database copies) is done through Mach's raw device interface (avoiding the file system overhead). The system contains very little machine–specific code.

System M consists of a collection of processes† operating on shared data structures, including the database itself. Each process acts as a server, accepting work requests and returning results. There are four types of servers that make up the system.

1) The transaction server (TS) accepts transaction requests from the transaction request queue and runs the requested transactions against the primary (main–memory) database.

2) The log server (LS) accepts log requests and flushes the appropriate log buffers to disk.

3) Message servers (MS) place transaction requests into the transaction request queue and take responses from a transaction response queue. There can be any number of message servers. All deposit requests into the same transaction request queue, but each has its own response queue.

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

Figure 2.1 shows the servers and their shared data structures. The "critical path" of a transaction through the system is illustrated by the double lines. In the following sections we describe each of the servers in more detail. Following that we describe the organization of the disks and main memory.

----

† In Mach, a process is implemented as a *task* (collection of resources) plus one or more *threads* of control.

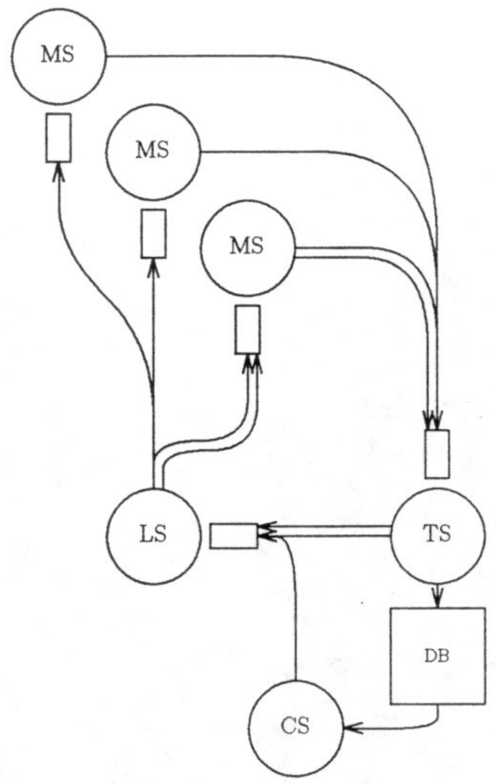

Figure 2.1 – System M Servers and Queues

2.1 The Log Server

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

The LS and TS interact through two shared data structures: the memory–resident log buffer and the log request queue. The log buffer consists of a collection of log pages which are filled with log data by the TS. In addition there is a list associated with each log page. These lists are used at transaction commit time as follows. When the TS is ready to commit a transaction, it does not send the response message directly to the corresponding message server. Instead, the TS places the response message in the list corresponding to the log page that holds the commit record for the transaction. The LS will then forward the message to the MS *after* the commit record has been flushed to disk.

At certain times, such as when a page is full, the TS determines that the log page should be flushed to disk. To accomplish this it places a request in the log request queue, indicating which page is to be flushed. The LS takes requests from the queue and flushes the specified pages to disk. Normally, the I/O is done asynchronously with respect to the TS, i.e., the TS does not wait for the LS to flush the pages it has requested. Once the I/O is complete, the LS examines the response list associated with the page that was just flushed. Pending response messages are forwarded to the corresponding MS and the buffer page is marked as clean so that it can be reused by the TS. Note that this implements *group commit* [DeWi84a, Gawl85a]. (A group of transactions is committed by each log page flush.)

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

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

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

2.2 Message Servers

In a real transaction processing system, the telecommunications monitor receives requests to run transactions. These requests originate from users (e.g., travel agent, teller) via a network. The requests do not contain code; instead they have the necessary parameters (e.g., desired flight) to run a canned program (e.g., to reserve a flight).

In System M, message servers emulate the function of the telecommunications monitor. They are responsible for feeding transaction requests to the TS and for handling the response messages from those transactions. Two different types of message servers have been implemented. Generative message servers (GMS) use random number generators to continuously create requests, driving the performance experiments that we will describe in Section 4. Interactive message servers (IMS) prompt a user for transaction requests. Both types of servers have facilities for displaying response messages and other debugging information.

2.3 Checkpoint Server

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

The exact method used to update the backup, and the method of synchronizing copying with database updates made by the TS, are determined by the checkpoint algorithm selected for the CS (Section 3). However, all of the algorithms have a few points in common. The CS transfers data to (and from) the backup in fixed–size blocks called *segments*. Except for modifications to dirty bits (see below), the checkpointer's access to database segments is read–only. Most of the checkpointing algorithms require that the CS lock segments before accessing them. These locks can conflict with database accesses by transactions running in the TS.

To keep track of what segments have been visited during a sweep, the checkpointer keeps two ***dirty bits*** per segment. Two bits are needed because of our dual backup database strategy. (See Section 2.5.) When a transaction modifies a segment, it sets both bits. When the checkpointer flushes a modified segment to one of the backups, it resets one of the bits. When it flushes it to the second copy, it resets the second bit. Thus, the checkpointer will only ignore segments that have been flushed to both backup copies.

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

2.4 Transaction Server

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

Since the database is memory resident, in most cases it is possible to execute transactions sequentially (transactions never have to be suspended waiting for I/O). However, in some hopefully rare cases (long lived transactions, conflicts with the checkpointer) transactions do have to be suspended. Thus, the TS must have a transaction multiprocessing capability, but it should only be used when it is impossible to run transactions serially. Serial execution will always be more efficient since it avoids context switching and reduces lock contention.

To achieve this, the TS (a single Mach task) consists of a number of transaction servers, each of which is capable of serial transaction execution. We will term each sub–server a ***transaction executor***, or TE. Transaction executors are implemented as coroutines within the TS task. TEs are created in two flavors, ***priority*** and ***standby***. Standby TEs are normally inactive. In case all active TE's are blocked, a single standby TE is activated to process a transaction. The standby TE is deactivated once its transaction has completed.

Normally, the TS consists of a single priority TE and one or more standby TEs. Transactions are processed serially by the priority TE until a lock (in most cases, a lock set by the CS) is encountered, at which point a standby is activated.

Currently, System M uses a modified version of the CThreads† [Coopa] coroutine library to implement TEs. The library has been modified in several ways. In particular, new primitives have been added to allow synchronization of coroutines with Mach tasks. For example, if a TE (coroutine) attempts to lock a segment locked by the checkpoint server (a task), only the locking TE is blocked, and not the whole TS task.

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

2.5 Backup Management

System M keeps two complete backup databases on disk and a *ping–pong* update scheme is used. Only one of the two copies is updated during a single checkpoint, and successive checkpoints alternate between copies. The CS uses its dirty segment bits to keep track of what segments to flush.

When a checkpoint completes, the current checkpoint copy is noted at a known location on disk called *home*. The home block also contains a pointer to the begin–checkpoint log entry made by this completing checkpoint. At recovery time, the home block is read to determine the current backup database and the portion of the log that must be analyzed.

----

† When the implementation of System M began, threads were not yet fully supported by the Mach kernel.

System M keeps a mapping of main memory segments to their corresponding location on the backup copies. Each time a segment is created or deleted, the mapping is updated. The backup databases contain enough information to reconstruct this mapping at recovery time.

A number of alternative backup strategies were considered before ping–pong was implemented. Very briefly, ping–pong is easy to implement and makes recovery simple. (The copy used for recovery reflects a completed checkpoint.) A drawback is that updates must be propagated twice. (In a heavily loaded system segments are always dirtied between checkpoints, so this drawback is not serious.) The tradeoffs between backup strategies are discussed further in [Sale88a].

2.6 Memory Management

The System M memory resident database is organized as records and sets. A record is a collection of fields, which may be of fixed or variable length. Each record has a type, which determines the number of fields in the record and their lengths. A set is a collection of records, each of which has a unique identifier. All of the records in a set are of the same type. Sets and records are defined by an application–specified catalogue.

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

Record updates are done using *shadows* to eliminate the need for UNDO logging. A record that is to be updated is not overwritten. Instead, space is allocated for a new copy. A pointer to the old copy is saved in a *shadow table* maintained for the set. The shadow is not freed until the updating transaction places its commit record in the log tail. In case of an abort, the new copy, rather than the shadow, is freed.

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

Incidentally, note that System M does *not* log changes to the search structures (e.g., hash tables used to find records). The search structures are recreated from scratch at recovery time as the records are loaded into memory.

System M provides a subroutine interface for the application programmer writing transactions. Procedures are provided for retrieving, updating, inserting, and deleting records. It also supports retrievals and updates of individual (fixed–length) record fields. For example, to modify a field, the field–update procedure is called by the programmer. This procedure gets the necessary locks for the update, takes the new value from the application record buffer, stores it in the new record version in the database, and makes the appropriate log entries.

2.7 Lock Management

The lock manager provides shared and exclusive lock modes (for records and segments), deadlock detection, and lock escalation (shared to exclusive). Locks can be requested in non–blocking mode. This mode is used by the CS: if a segment it wants to checkpoint is locked, it will attempt to checkpoint others in the meantime. Currently, lock acquisition time is reduced by pre–allocating lock data structures (i.e., "lock table" entries) for lockable objects.

3. Recovery Strategies

System M provides different types of checkpointing and logging strategies (selectable when System M is compiled). The checkpointing strategies can produce fuzzy, action–consistent (AC), and transaction–consistent (TC) backup databases. Logging can be by value, by action, or by transaction.

Consider a transaction that updates records R_1 and R_2 with two update actions. A TC backup will reflect transaction activities atomically, i.e., the backup will contain either the old versions of R_1 and R_2 or their new versions, but not one old and one new.

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

A value logging strategy records in the log the location and new value of any part of the database modified by transactions. On the other hand, transaction logging records a description of the transactions themselves, so that changes they made can be reconstructed by re–executing the transactions. A variation of transaction logging is action logging where the record level actions that make up a transaction (e.g., insert record) are stored. Transaction and action logging are types of logical, or operation, logging.

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

The log strategy is closely tied to the checkpointing strategy. As a general rule, transaction logging requires TC checkpoints and action logging requires at least AC checkpoints. This has important performance implications. For instance, as we will see, fuzzy checkpoints are less costly to produce than say action consistent ones. However, fuzzy checkpoints require value logging which in most cases produces more log bulk. Thus, writing the log is more costly. At recovery time, reading the log will also be more time consuming because of its larger size. We will return to these tradeoffs in Section 4.

In the rest of this section we will outline the three checkpointing strategies available in System M. These strategies are based in varying degrees on ideas in [DeWi84a, Hagm86a, Pu86a, Rose78a, Sale87a].

3.1 Fuzzy Checkpoints

We call the System M fuzzy checkpointing strategy FUZZY. (Fuzzy checkpoints are suggested for recovery in main memory databases in [Hagm86a].). It begins a checkpoint by entering its begin–checkpoint marker in the log. Once the marker is in place, the checkpointer processes database segments. A segment is processed by carefully examining and clearing the appropriate dirty bit, and flushing the segment to secondary storage if it was dirty. Locks and other transaction activity are ignored. Once all segments have been processed, the (in–memory) log tail is flushed to disk and the new current backup is noted in the home block.

If one is not careful, fuzzy checkpointing may in general lead to violations of the *log write–ahead protocol* [Gray78a]. (Such a violation occurs if a transaction's updates are reflected in a secondary database but not in the log.) However, because we are using two ping–pong secondary databases, the problem does not arise. While a checkpoint is in progress, a transaction's updates may indeed appear in the current secondary database before they do in the log. Nevertheless, since the checkpoint is incomplete, all such updates will be ignored at recovery time. It is only when the checkpoint completes that the updates in it become valid.

3.2 Black/White Checkpoints

One way to produce a consistent backup is to treat the checkpoint operation as a (long–lived) transaction. The checkpointer acquires a lock on each segment before flushing and holds the locks until the checkpoint is complete. Clearly, this method will result in unacceptably frequent and long lock delays for other transactions. (At some point during each checkpoint the checkpointer will have all of the dirty database segments locked simultaneously.) An alternative, which produces consistent backup copies but requires that locks be held on only one segment at a time, is presented in [Pu86a]. (It can also be viewed as a special case of the "altruistic" locking protocol described in [Sale87a].) The strategies we will describe next are variants of the mechanism proposed in that paper.

The basic strategy described in [Pu86a] proceeds as follows. There is a "paint bit" for each segment which is used to indicate whether or not that segment has already been included in the current checkpoint. Assuming that all segments are initially colored white (i.e., paint bit = 0), checkpointing is accomplished by the strategy shown in Figure 3.1.

Once all of the segments have been processed, the log tail is flushed to disk and the new current backup is noted in the home block. The strategy can be used to produce either a TC or an AC backup. To ensure that the checkpointer produces a TC backup, no transaction† is allowed to access both white and black segments. Any transaction that attempts to do so is aborted and restarted. Similarly, an AC backup can be

```
WHILE there are white segments DO
        lock any white segment
        process the segment
        paint the segment black (set paint bit = 1)
        unlock the segment
END-WHILE
```

Figure 3.1 – Black/White Checkpoint

produced by ensuring that no action accesses both black and white segments. (Note that a transaction may contain a mix of black–accessing and white–accessing actions.) A transaction is aborted if any of its actions attempt to access both white and black segments.

A segment can be "processed" in one of two ways. One possibility is to flush the segment immediately to secondary storage. An alternative is to spool the segment by first copying it to a special buffer before flushing it. The advantage of spooling is that the checkpointer's lock can be released as soon as the segment is spooled. Without spooling, the lock must be retained through an I/O operation. Spooling and non–spooling checkpointers will be compared in Section 4.2.

3.3 Copy–on–Update Checkpoints

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

To begin a COU checkpoint, the database must first brought into a state of the desired consistency (either action–consistent or transaction–consistent). One way to achieve a TC database state is to quiesce the system: currently executing transactions are completed, while no new transactions are begun. To achieve an AC state, all update

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

actions (e.g., install record) are completed while new actions are disallowed.

When the database is quiescent a begin–checkpoint record is written to the log. The consistent database state that exists in primary memory is the "snapshot" that will be flushed to secondary storage by the checkpointer. Once the begin–checkpoint entry is in the log, transaction activity can resume.

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

Checkpointing is accomplished by the algorithm shown in Figure 3.2. (As before, we assume that all segments are initially colored white.) As with the fuzzy and black/white algorithms, the log tail is flushed and the new current backup is noted in the home block once all segments have been processed.

```
WHILE there are white segments DO
   lock any white segment ($S$)
   IF $S$ has a pointer to a "snapshot" copy $S'$ THEN
       paint $S$ black
       save pointer to $S'$
       erase pointer in $S$
       unlock $S$
       IF $S'$ is dirty THEN
           flush $S'$ to the backup
       free $S'$
   ELSE
       process $S$
       unlock $S$
END_WHILE
```

Figure 3.2 – COU Checkpointing

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

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

4. Performance

In this section we present the results of several experiments designed to evaluate System M and compare the performance of the various checkpointing and logging strategies. All of the experiments were performed on a a VAX–11/785 equipped with 128 Mbytes of memory and running Release 1 of the Mach operating system. The disk–resident portion of the log and the backup database were kept (1 partition each) on a pair of RA81 disk drives. The log and backup partitions were accessed using the "raw" I/O facilities provided by Mach, i.e., the file system is bypassed.

The application used to drive System M simulates a credit card data processing environment. It is based loosely on an application used to drive a recent set of benchmarks of IMS/VS FastPath [Vigu87a]. The credit card application defines four sets of records and eight different transactions. The sets include:

- *Account Set*: one record per account. Record fields include account number, credit limit, used credit, expiration date. 40,000 accounts; expected record size is 52[†] bytes.
- *Customer Set*: one record per customer. Records hold customer information such as name, address, social security number, and account number; 40,000 customers, expected record size is 200 bytes.
- *Hot Card Set*: one record for each stolen card reported. Records hold the account number, number of attempted uses of the stolen card, and the report date. Initially, 100 cards reported. Expected record size is 80 bytes.
- *Store Set*: one record for each point of sale (retail store). Record fields include store name and number and counters for various types of credit card activity at the store; 5000 stores, expected record size is 80 bytes.

----

† This and all record sizes include sixteen bytes of system header information.

The five most frequent transactions are

* *BAL (Balance Check)*: Returns information about an account, including customer name and current balance (17% of submitted transactions).
* *CCCK (Credit Card Check)*: Checks the validity of an account and increments a counter at the store from which the transaction originated (20% of transactions).
* *CLCK (Credit Limit Check)*: Checks that there is sufficient credit available for a purchase. Also increments a counter in the originating store's record (20% of transactions).
* *DEBIT (Account Debit)*: Modify account information to reflect a purchase (20% of transactions). Also increments a counter in the seller's record.
* *PAY (Make Payment)*: Make a payment on an account (20% percent of transactions).

The remaining transactions change the customer description and handle lost cards.

4.1 Throughput

The first set of experiments measured the throughput of transactions when various combinations of logging and checkpointing strategies were used. Checkpoints run as quickly is possible, i.e., no delay between completion of one checkpoint and initiation of the next. Non–spooling checkpointers were used. Except for checkpointing and logging strategies, System M was identically configured for each of the experiments.

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

The results of these experiments are presented in Table 4.1. Server times are given as μseconds of processor time per true transaction. Server times are computed by dividing the total processor time used by the server (over the 30 minute experiment) by the total true transaction count. Processor time was measured using operating system process task timing facilities (via calls to the system routine "getrusage").

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

Table 4.1 – True Throughput and Per–Transaction Process Times

As can be seen from the table, the recovery strategy can have a significant effect on performance. Transactions run almost twice as fast under the best logging/checkpointing combinations as under the worst. The total per–transaction processor time can be obtained by summing the per–process times in a particular row of the table. For the better recovery strategies, this yields 12 to 13 milliseconds of processor time per transaction, including all recovery costs. With all logging and checkpointing turned off, the transaction mix used for these experiments was measured at an average of slightly more than eight milliseconds per transaction (i.e., recovery costs 4 to 5 milliseconds per transaction). Given that the testbed's VAX platform is roughly a one MIP machine, these times compare very favorably with transaction execution times in disk–based systems [Ston88a, Scru85a].

The experiments indicate that TCBW checkpoints, even when combined with transaction logging, do not perform well. This is because of the high cost of rerunning transactions that are aborted for accessing both black and white segments. TCCOU checkpoints, when combined with transaction logging, perform about as well as FUZZY checkpoints. The savings from transaction logging cancel the additional expense of producing the TC checkpoint in this case. However, transaction logging (and TC checkpoints) becomes more difficult when there are long or complex (e.g., conversational or multistep) transactions. Furthermore, recovery times may be longer with a transaction log because logged transactions must be re–run [Sale88a]. Therefore, FUZZY/VALUE or

ACBW/AOPER would appear to be the best recovery choices for flexibility and performance.

4.2 Spooling

None of the checkpointing strategies evaluated in the last section used spooling. Recall that spooling checkpoints copy segments before flushing them to disk in order to reduce the amount of time they must lock the segment. The price paid is the processor time required to copy the segments.

Experiments with spooling checkpointers indicate that, as expected, they have slightly lower throughputs than their non-spooling counterparts (typically 1–3 transactions per second less). Using facilities built into the testbed's lock manager, we evaluated lock contention using spooling and non-spooling checkpointers to determine whether spooling produced less contention in return for its higher overhead. Table 4.2 shows the results of these experiments. Lock contention is expressed as the percentage of lock requests that block.

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

Table 4.2 – Lock Contention with Spooling and Non-Spooling Checkpointers

While spooling does reduce contention slightly, lock conflicts are rare enough with non-spooling checkpointers that the reduction is not significant. Note that lock contention is low because the TS runs transactions serially as much as possible. Spooling may be a useful technique in a multiprocessor system with multiple priority TE's. However, in our environment it is unnecessary.

4.3 Response Times

Table 4.3 shows the mean transaction response time for different combinations of checkpointing and logging strategies. To measure response time, message servers in System M timestamp each transaction request message when it is placed on the transaction request queue. Another stamp is generated when the response is taken from the message server's response queue, and the response time is taken to be the difference between the stamps. Thus, response time includes the entire time spent in the system, including queueing delays.

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

Table 4.3 – Transaction Response Times

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

The response time of the TCBW strategy can be improved by varying the retry time. (Recall that the retry interval is delay between the abortion of a transaction for black/white violations and it's restart.) We investigated this, but the improvement was small [Sale88a]. Thus, we rule out TCBW as a viable recovery strategy on the basis of its poor response time.

Note that the response times for the remaining recovery strategies are still relatively high. This is because the message servers used to drive the experiments create messages much more quickly than they can be processed. The request rate is limited only by the size of the request queue, which is much larger than necessary. Thus, transaction requests wait a long time in the request queue before being accepted by the TS. We expect that response times can be lowered significantly (with little or no loss of throughput) by reducing the size of the request queue.

4.4 Basic Measurements

In order to understand the System M performance and to facilitate comparison with other systems, we evaluated some of the basic system operations. Some of these measurements are shown in Table 4.4. Times are given in μseconds.

operation	time
segment allocation	267
log page allocation	1050
log sequence number	81–148
lock request	285
I/O request	5200

Table 4.4 – Some Basic Operations

Segment allocation refers to the cost of allocating and freeing a memory–resident database segment, an operation performed frequently when COU or spooling checkpoints are used. The log page allocation time is incurred each time a log page is filled and a new current log page must be prepared. It includes the time to place a request in the LS queue to have the full page flushed to the log disks. The log sequence number is the cost of checking whether a page with a particular log page is safely on disk. This operation is used by some checkpointers to implement the log page "clearance" facility. The costs vary depending on whether the log page in question is still memory–resident or not. The lock request time includes both the lock and unlock times. It is for the most common case, namely the case when no other server holds a lock on the requested object. The locking operation is more costly when other locks are held on the object. Finally, the I/O request operation is used by the checkpointer and logger to move data from memory to the log and backup disks. Note that 5200 microseconds is only the CPU time involved in handling the I/O request. Also note that much of this time is accounted for by Mach's I/O routines and not by System M itself.

5. Conclusions

We have described System M, a transaction processing system for memory–resident databases. System M has been designed from scratch for memory–resident databases. As a result, it uses a unique process architecture and recovery techniques not normally employed in other systems. System M permits stable backup databases and the transaction log to be maintained on disk with minimal interference with transactions accessing the memory–resident database. Because the primary database is in memory, transaction execution in System M is relatively simple and efficient. Transactions run serially if possible, and they require no expensive I/O operations to access data.

Disk–based transaction processing systems may require 100,000 instructions to process a transaction [Ston88a]. Thus, the CPU will limit such systems to 10 transactions per MIP. Currently, System M requires roughly 12,000 instructions per transaction, and we believe that this figure can be brought below 10,000. Thus, memory–resident data may represent an order of magnitude increase in transaction processing capability. Of course, it is difficult to compare System M to commercial systems, since such systems certainly provide greater functionality than is available in the testbed. On the other hand, System M's performance is the result of only two man–years of work.

Another relevant issue in comparing System M performance to that of a conventional system is the communications overhead. In System M the message server (MS) emulates the component that would communicate with terminals. In System M the MS consumes roughly 27% of the CPU cycles (see Table 4.1). It was actually implemented in an inefficient fashion so that its load would be comparable to that of a component that actually communicated over a local area network with a set of communications processors. These communications processors would in turn connect to the long haul network and the terminals. We think that this two level structure is the most appropriate for a MTPS; the main processor is a critical resource and should not be burdened with all the communications. Thus, the 27% overhead for the MS seems reasonable. However, if all the telecommunications load were to be placed at the main processor, then the MS overhead should be higher and the overall System M performance would be lower.

The testbed is intended to be a vehicle for comparison of algorithms in the memory–resident environment. We have used it to compare a variety of checkpointing and logging strategies. Our results suggest that FUZZY (inconsistent) checkpoints perform as well as or better than consistent checkpoints, even when the consistent checkpoints are combined with efficient operation logging.

Many other experiments are possible using System M. We have begun to study other aspects of recovery and memory management. Experiments with other aspects of the system, such as concurrency control, are also possible. System M's processor architecture is also ideally suited to shared memory multiprocessors. Its message, logging, and execution processes effectively pipeline transaction execution, while checkpointing proceeds in parallel. Thus, a number of processors can easily be brought to bear without the need for large numbers of concurrently executing transactions.

References

Acce86a.
Accetta, Mike, Robert Baron, Willian Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernal Foundation For UNIX Development," *Proceedings of the Usenix Association Summer Conference*, pp. 93–111, Atlanta, GA, June, 1986.

Bitt87a.
Bitton, Dina, Maria Butrico Hanrahan, and Carolyn Turbyfill, "Performance of Complex Queries in Main Memory Database Systems," *Proceedings of the Third Int'l. Conference on Database Engineering*, pp. 72–81, Los Angeles, CA, February, 1987.

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

DeWi84a.
DeWitt, David J., Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood, *Implementation Techniques for Main Memory Database Systems*, ACM, 1984.

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.

Garc87a.
Garcia–Molina, Hector and Kenneth Salem, "High Performance Transaction Processing with Memory Resident Data," *Proc. Int'l. Workshop on High Performance Computer Systems*, Paris, December, 1987.

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.

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.

Kern78a.
Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice–Hall,

Inc., Englewood Cliffs, NJ, 1978.

Lehm85a.
Lehman, Tobin J. and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc. Int'l Workshop on High Performance Transaction Systems,* Asilomar, CA, September, 1985.

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

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

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

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

Sale88a.
Salem, Kenneth, "Failure Recovery in Memory–Resident Transaction Processing Systems," PhD Thesis, Department of Computer Science, Princeton University, Princeton, NJ, November, 1988.

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

Scru85a.
Scrutchin, Tom, "TPF: Performance, Capacity, Availability," foils and minutes from a presentation at the Int'l Workshop on High Performance Transaction Systems, Asilomar, CA, Sept., 1985.

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

Ston88a.
Stonebraker, Michael, Randy Katz, David Patterson, and John Ousterhout, "The Design of XPRS," *Proc. 14th VLDB Conference,* pp. 318–330, Los Angeles, CA, 1988.

Vigu87a.
Viguers, Dave, "IMS/VS Version 2 Release 2 Fast Path Benchmark (ONEKAY)," *Proc. of the Second Int'l Workshop on High Performance Transaction Systems,* Asilomar, CA, September, 1987.