CRASH RECOVERY MECHANISMS FOR MAIN STORAGE DATABASE SYSTEMS

Kenneth Salem
Hector Garcia-Molina

CS-TR-034-86

April 1986

CRASH RECOVERY MECHANISMS FOR MAIN STORAGE DATABASE SYSTEMS

Kenneth Salem
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

In a main storage database system the primary copy of all data resides permanently in primary (semiconductor) memory. A major problem for such systems is crash recovery, i.e. ensuring that transactions are atomic and durable in spite of main memory's volatility. In this paper we study several possible crash recovery mechanisms and analyze their impact on performance.

1.0 INTRODUCTION

Current trends for semiconductor memories include increasing chip densities and decreasing cost per bit. One result of these trends is that main storage databases are becoming feasible. For the purposes of this paper, we will consider a main storage database (MSDB) to be one in which the primary copies of all data reside permanently in primary (semiconductor) memory.

The migration of data from secondary to primary storage leads us to reexamine the components of traditional database management or transaction processing systems. In general it will not be a good idea from a performance viewpoint to simply move a transaction manager (TM) from a disk-based database to a memory-resident one. Any component of the transaction manager whose operation is premised on disk-based data should first be modified to reflect the new environment. For many, the modification will result in a simplification of the original code. Thus, the memory-resident system can realize performance improvements through reduced CPU overhead as well as through the elimination of disk access times.

One component of a TM which might be troublesome is the crash recovery mechanism, which is charged with guaranteeing transaction atomicity and durability in the face of system failures and user-initiated aborts. A crash recovery mechanism is usually implemented as software which *logs* database changes to a stable medium such as a disk. The disk activity and CPU overhead associated with maintaining the log can be a bottleneck for a disk-based system; for a memory-resident system with no other need for disk traffic this problem may be even more pronounced, possibly nullifying the advantages of having the data in main memory. Thus, crash recovery appears to be one of the most critical issues for a MSDB.

The objective of this paper is to study various MSDB crash recovery mechanisms and their impact on performance. The mechanisms include several that have been proposed or already implemented. Two others involve special hardware, for we believe that dedicated hardware may significantly alleviate the recovery problem at relatively low cost. The first of the hardware strategies supposes that a portion of primary memory can be made non-volatile for use as a buffer for log disks. The second uses a new hardware logging device we propose, HALO, that would assume many of the crash recovery duties previously handled by the database processor.

We have also modeled a hypothetical MSDB in which all of primary memory is made reliable, perhaps through the use of replication and a backup power supply. While such a system might not be practical for most applications, it does provide a good base case for comparisons.

Our studies concentrate on performance measures like transaction response time and throughput. They do not attempt to compare the costs (in dollars) of the mechanisms. This is not because cost is an unimportant measure. Rather, we want to determine the benefits of each scheme so that their costs can be seen in the proper light. For example, we would want to know what performance improvements we could expect from migrating logging operations to hardware before we invest the effort to build that hardware.

This paper is organized as follows. The next section gives a general description of a TM for a MSDB, and its recovery mechanism. Section 3 discusses the particular mechanisms we have modeled, including a description of HALO, the proposed hardware logging device. Section 4 covers the model we have used to compare these mechanisms. Response time and throughput data from the model are presented in Section 5, where we compare the various recovery alternatives using several sets of parameter values.

2.0 MAIN STORAGE DATABASES

Main storage databases have been receiving increased exposure in print over the last several years. Papers of particular interest include [3] which describes IMS/VS Fast Path, a commercial product that has supported MSDBs for a number of years (it is from this paper that we have borrowed the term MSDB), and [2], which discusses MSDB crash recovery. This latter paper suggests the use of non-volatile RAM as a log buffer. Group commits, another mechanism we have modeled, are described in both papers and are implemented in IMS/VS Fast Path.

Fuzzy checkpoints, potentially inconsistent checkpoints which can be done in parallel with normal transaction processing, are described in [6], which discusses some of the crash recovery issues raised by MSDBs. [5] describes applications for very fast transaction processing systems; these are some of the applications for which MSDBs could be useful.

Several authors have described analyses of database crash recovery techniques. Most recently, [10] presents a taxonomy of such techniques and a set of models for describing them. The models differ from those presented here in that the cost metric used as the basis for comparison in the former is the number of disk IO's required by the various techniques. [1] focuses on modeling recovery time rather than normal operation. The model presented there assumes that action-consistent (non-fuzzy) checkpoints are used, and it considers time-varying transaction loads. Finally, [13] presents a comprehensive if slightly outdated survey of recovery techniques in general.

2.1 OUR MODEL

We assume that the TM is running on a single processor with enough volatile RAM to hold the entire database.* In such a system the processor is the critical resource for determining transaction response times and throughputs.

We expect to see both *direct* and *indirect* performance benefits from memory-residence. Direct benefits come from the elimination of disk accesses. A memory-resident database is an access time faster than a disk-based one for every time the latter would have had to go to its disks. Thus we could expect to move an existing TM onto a large-memory system and see significant performance improvements as a direct result of the increased memory size.

Intuitively, we should be able to do better than just picking up the TM and moving it to a large memory system. Those parts of the TM that were premised on disk-resident data can be rewritten and simplified to take advantage of the larger memory. The improvements we see from these changes can be thought of as an indirect benefit of memory-residence. Because the CPU is critical to the performance of the new system, the benefits from the simpler, lower-overhead system could be substantial in themselves.

Specifically, we assume that transactions are executed serially, with no locking.** In typical transaction processing applications (e.g. banking, reservation systems) transactions are not large.*** The only reason they are not executed serially in conventional systems is that long delays might result from disk accesses. Clearly, a TM with memory-resident data does not suffer from this problem. Transactions can be executed one at a time, without the complexity and overhead of concurrency control mechanisms like locking.

Another component that can be eliminated from memory-resident systems is the buffer manager. With data permanently in core we can eliminate the CPU overhead involved in copying buffers, initiating disk IO, and implementing a replacement policy (recovery mechanisms, discussed below, may have to copy data and manage buffers in certain cases).

MSDB systems can also benefit by moving system call overhead costs to compile-time. To illustrate, consider a transaction T that wishes to increment the balance of an account by an amount *AMT*. In a conventional system, the transaction would make a system call requesting that a record (identified with a pointer retrieved earlier) be brought into a buffer local to T. The system would verify that the pointer points to the right type of record, that the transaction is authorized to read, that a read lock is set, and would then initiate the IO operation and suspend the transaction until the data was in. When the transaction was restarted, it would increment the balance field in the record and perform a second system call. The system would be told what field was changed (else it would have to log the entire record), and this information would have to be verified too.

----

* There may be, of course, applications where the "entire database" may not fit into RAM or where we may want to have multiple processors. In these cases, we can view our single processor MSDB system as holding a high-traffic portion of the database, and connected to other (possibly conventional) systems containing the rest.

In a MSDB, on the other hand, it will be desirable and perhaps necessary to reduce this overhead by replacing the system call interface with an environment in which transactions and system structures are integrated at compile-time. Application programmers will use a conventional programming language, modified only in that certain data structures can be defined to be "system" structures, and certain procedures or functions can be defined to be "transactions".

In such a system, the transaction T would be a procedure (in the application programming language) that included a statement like

$$BAL(i) \leftarrow BAL(i) + AMT$$

The *BAL* data structure would be declared a "system" structure. At compile-time, statements would be inserted directly into the transaction code to check whether the user is authorized to read and update *BAL* and whether *i* is a valid index for *BAL*. Reads and updates of system data structures are thus treated like system defined macros. At execution-time, the expanded macros are executed with no system call overhead.

Replacing system calls with compiler macros will be particularly useful in a MSDB for several reasons. First, because of the simplified nature of transaction processing when data is in core, the amount of code that needs to be added will be reasonable. For example, no locking and unlocking need be considered since transactions are executed serially. Second, because transactions suffer no disk delays and their execution times are CPU-bound, reductions in CPU overhead will translate directly into decreased response times and increased throughputs. In a disk-based system, such a savings would be overshadowed by the long delays associated with disk IO.

There are other areas where we can expect indirect benefits from a MSDB system. For instance, access methods can be designed for rapid retrieval from RAM rather than to minimize disks accesses. These improvements do not affect crash recovery and are not discussed further.

2.2 CRASH RECOVERY

One potential stumbling block for this kind of low-overhead, high performance transaction manager is crash recovery. Like a disk-based system, a MSDB should guarantee atomicity and durability of transactions even though the entire database resides in volatile storage. Furthermore, this guarantee must be made without sacrificing the performance gains we can get from memory residence. Because of these requirements, non-volatile storage such as disks must be used to hold recovery information and a backup copy of the data. Thus with recovery considered we have a system as shown in Figure 1.

The crash recovery mechanisms we have studied consist of two components, a *logger* and a *checkpointer*. The logger maintains records of transaction activities (particularly updates) in a stable storage area. The checkpointer periodically copies modified segments of primary memory to a backup on stable storage.

There are a number of different ways to implement each of these recovery components. However, in order to achieve the desired properties, a number of constraints must be enforced. These constraints are discussed in general terms in the following paragraphs. The descriptions of the

----

** We assume that there is no locking even in the scenario where our system is just one node of a larger system. Inter-node atomic transactions would require that locks be held until a transaction committed. Our hypothesis is that our MSDB system requires high throughput, and it is too expensive to hold resources for a remote transaction that we have no control over. We assume that inter-node transactions are executed as a sequence of local transactions. Compensating transactions (e.g. "cancel the reservation I just made") are run if the global transaction has to be rolled-back.

*** As commonly done in transaction processing systems, we assume that long-lived transactions are broken into sequences of smaller transactions. As with inter-node transactions, a compensating transaction can be executed to roll back a partially executed long-lived transaction.

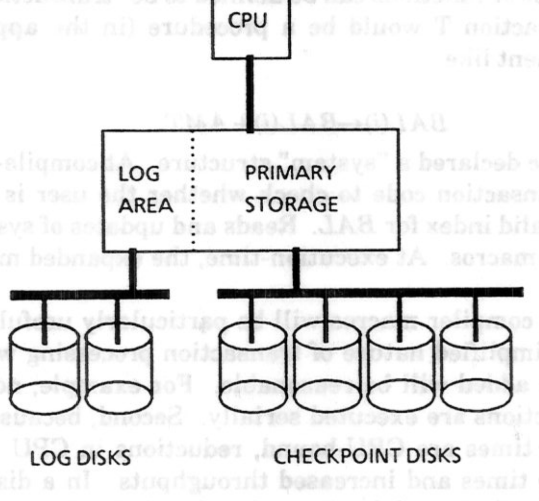


Figure 1

recovery mechanisms given in the next section discuss more specifically how each mechanism satisfies these constraints.

To guarantee atomicity we must ensure that before an update is migrated (checkpointed) to the backup database copy, either 1) undo information for that update is logged in a stable storage area, or 2) the updating transaction has committed and has its log record in a stable storage area (undo information is not necessary in this case). There are a number of ways to enforce these conditions. One possible scheme, involving time-stamps and multiple copies of database pages to produce a consistent backup, is described in [2]. Our mechanisms use schemes that are based instead on fuzzy checkpoints. Either approach requires that updates be recorded in the log area (whether it is stable or not) before they are applied to the primary database.

To guarantee that aborted transactions do not affect the database, we also need to ensure that the transaction dependencies determined by the execution schedule are observed by the logging mechanism. This may become a problem in systems with multiple parallel log disks. When necessary we assume, as suggested in [2], that explicit dependencies are maintained between transaction log records. Thus if transaction T_1 executes before T_2, T_1's log record arrives on stable storage at the same time or earlier than T_2's. Transactions can thus freely update the database in primary storage when they execute with the assurance that any dependent transactions will not be logged before them.

To make transactions durable, we ensure that before the transaction commits externally (sends its output message), either 1) all pages dirtied by a transaction are flushed to the backup copy, or 2) redo information for the updating transaction is in stable storage. The latter scheme is generally preferable to the former when the stable storage is high-latency media such as disks, since the former may involve more IO activity. Our recovery mechanisms all ensure the latter, although doing so is simpler with some mechanisms than with others.

In addition to these constraints, there are also implementation issues which affect all of the recovery mechanisms we will describe. They are related more to performance than to correct maintenance of transaction durability and atomicity. We now briefly discuss some of these issues that affect the logger and the checkpointer.

If logging is to be done by the database CPU, we assume that it is done by the individual transactions without recourse to system calls. This will be handled by the mechanism described in the last section, i.e. logging instructions are generated at compile-time and added to the transaction. For example, consider once again the account balance example. The statement updating *BAL* is supplemented with recovery code like:

$$LOGOLDVALUE(j) \leftarrow BAL(i);$$

$$LOGNEWVALUE(j) \leftarrow BAL(i)+AMT;$$

$$LOGADDRESS(j) \leftarrow address[BAL(i)];$$

$$j \leftarrow j+1;$$

$$BAL(i) \leftarrow BAL(i)+AMT;$$

Thus, when transactions run, they do their own logging in a very simple and efficient way. As shown above, both redo and undo information is assumed to be logged for each update. The redo information is useful for guaranteeing transaction durability, as discussed above. Undo information is useful for backing out aborted transactions, however it is not necessary to flush this information to the log disks once a transaction has committed. We have not investigated the trade-offs involved in compressing the log to remove undo information (at the cost of additional CPU overhead) vs. the cost of flushing the extra undo information to the disks since the trade-off can be applied to any of the mechanisms we consider. All of our mechanisms flush undo as well as redo information to their logs.

We assume that the checkpointer is active only between transactions. (We are referring here to the execution of checkpointing instructions by the database CPU. DMA devices, such as disks, activated by the checkpointer will certainly operate asynchronously and concurrently with transactions.) When the checkpointer is active, it scans through the database from the most recently checkpointed page until a dirty page is found (or until some maximum number of pages have been examined). Depending on the particular mechanism, the dirty page may be copied to a buffer area. A flush to the checkpoint disks is then initiated before control of the database CPU is relinquished for execution of the next transaction.

3.0 RECOVERY MECHANISMS

The recovery mechanisms we have modeled differ in the way their components are implemented. Two of the mechanisms are completely software-based. The remaining four involve modifications to the basic system of Figure 1, such as the availability of a non-volatile log area. The following subsections describe each of the mechanisms in turn.

3.1 IMMEDIATE COMMIT

The term *immediate commit* (IC) was chosen to contrast with *group commit*, the mechanism we will describe next. Immediate commit is the base mechanism; the others we describe are modifications designed to improve its performance.

Immediate commit means that each transaction flushes its own transaction record to the log before completing. Each transaction goes through a procedure like the following:

① The body of the transaction is executed. Modifications (if any) are first noted in the log buffer area, and are then made directly to the primary data copy.
② When the transaction reaches its commit point, the CPU initiates a flush of a copy of the transaction's log record to the log disks. No further action occurs until the flush is completed successfully (although the CPU is free to work on other transactions).
③ When the flush is completed, the transaction commits externally by sending its response message.

Thus each transaction results in a flush to the log disks, and that flush time is part of the response time of the transaction. This procedure guarantees transaction durability by ensuring that the log record is flushed to the disk before external commit. (This is known as a *log write-ahead protocol* [4].)

Dependencies between log pages are maintained explicitly by the system. A log page is considered successfully flushed when it has been written to the disk and the previously queued log page has been successfully flushed. With each write to the log disks, the system marks the next page in the log queue (whose flush may have already begun) to indicate the completion of the previous page's write. These operations result in CPU overhead for the maintenance of the dependency marks.

Checkpointing is handled by the database CPU. New checkpoint processes are initiated at regular intervals specified by a model parameter. Frequent checkpoints, though more costly, reduce recovery time by reducing the amount of log data that must be scanned.

The checkpointer copies pages to a special checkpointing buffer area before flushing them. There are several reasons for this. First, it is a way of ensuring that transaction updates do not inadvertently get "caught" by the checkpointer, possibly violating the transaction atomicity requirement. Since checkpointing is done asynchronously, updates from uncommitted transactions might otherwise be flushed to the checkpoint disks. In the case of a system crash this could leave the database in an inconsistent state, even after the log is applied to the checkpointed data. Secondly, copying a page before flushing ensures that the set of transactions on which that page depends does not change once the copy has occurred. Dependencies between database pages and transaction log records are discussed next.

Explicit dependencies are maintained between database pages and transaction log records. The checkpointer is prohibited from flushing any database page until transactions that have updated that page have committed and successfully flushed their log records to disk, or have aborted and undone their updates. We assume that the extra overhead costs entailed by these dependencies are borne entirely by the checkpointer, and not by individual transactions. Before flushing a page to disk, the checkpointer will scan the in-core log queue for any entries that affect that page. If any are found, checkpointing is delayed and a special marker is placed at the end of the log queue. The page can be checkpointed when all log records that affect the page and that are before the marker have been flushed.

3.2 GROUP COMMIT

The *group commit* mechanism (GC) is identical to immediate commits except for the manner in which the transaction logs are flushed (step 2 in the IC logging procedure). The group commit scheme attempts to reduce traffic to the log disks by delaying flushes of transaction log records so that several can be combined into a single flush.

Transaction log records remain in the log area in primary storage until a log page is full. At this point the log page is flushed to disk. Transactions whose records are contained on that log page are termed a *commit group*. All of the transactions in the commit group are simultaneously internally committed and can proceed to step 3 in the procedure when the log page is safely on disk.

Grouping commits reduces the number of flushes to the log disks and therefore queueing delays and CPU overhead. However, transactions must spend additional time waiting for their commit groups to assemble. This becomes particularly important when transaction throughput is low.

Checkpointing under the group commit scheme is handled in the same manner as IC checkpointing.

3.3 BATTERY BACKUP

The remaining mechanisms involve modifications to the hardware of the basic system we have considered thus far. The *battery backup* mechanism (BB) is the simplest of these. We assume that the log area of primary storage can be made non-volatile (perhaps through the use of batteries as a backup power source) as depicted in Figure 2. Transaction processing would proceed exactly as in the group commit case. This time, however, transactions do not have to wait for commit groups to assemble or for log records to be flushed to disk. Once transaction records have been placed in the log

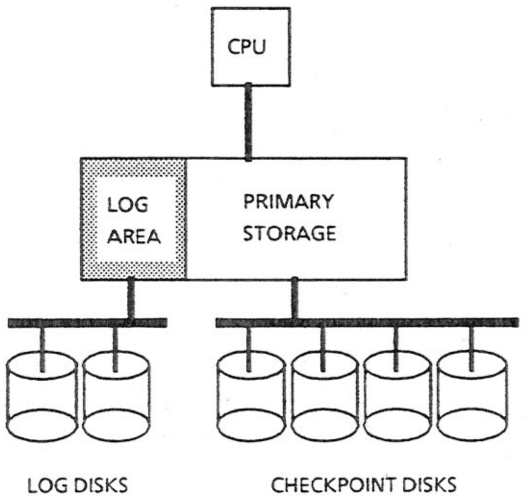


Figure 2

area they are safe and the transaction is free to commit externally. The transaction processing procedure would go as follows:

① The body of the transaction is executed. Modifications (if any) are first noted in the log buffer area, and are then made directly to the primary data copy.
② The transaction reaches its commit point.
③ The transaction commits externally by sending its response message.

The CPU must still flush pages from the log area to the log disks, but the time involved is not directly a part of the response time of any transaction. We assume that there is sufficient non-volatile buffer space so that the log area never overflows as a result of a discrepancy between the flush rate and the rate at which transaction records are being placed into the buffer. The amount of such memory we would need is not great; some simple estimates are made in appendix A.

Since log records of updates are in stable storage before the update is actually made, it is not necessary for the checkpointer to worry about dependencies between database pages and transaction log entries. Any update inadvertently "caught" by the checkpointer will be corrected when the log is played back after a system crash. Also, dependencies need not be enforced between log pages, since all log data enters (in-core) stable storage in a correct order. In fact, log pages can be flushed to the disks in any order without consequence as long as the in-core version of a page is not disposed of until its write is completed successfully. Note that aborting a transaction may be somewhat more complex using this scheme than when using the scheme proposed for IC and GC. This is because it is possible that updates from the aborted transaction will have migrated to disk, and will have to be corrected there as well as in-core. We do not consider the cost of aborting transactions in the model.

Because we do not have to worry about which updates get checkpointed with a database page, and because dependencies are not maintained between database pages and transaction log records, there is no need for the checkpointer to copy pages to a buffer area before flushing them. In other respects, checkpointing under BB proceeds as it did under previous mechanisms.

3.4 HALO

Recovery-related operations appear to be costly in terms of CPU time. The database processor must expend valuable cycles on tasks such as initiating IO to the log disks and copying to buffers. We have considered a hardware logging mechanism which could assume many of the recovery duties normally handled by the main processor as a way to eliminate this problem.

The salient features of the *HALO* (HArdware LOgging) mechanism are:

1) Logging functions are implemented in hardware.
2) Log entries are made at the word level.
3) Internal buffers are non-volatile.
4) The logging is transparent to the database CPU.
5) Disk delays are avoided by initially storing the log on non-volatile memory (as is used in the BB mechanism, above). As time permits, the log is then written out to disk.

Figure 4 shows a block diagram of a system incorporating a HALO device. Figure 5 describes

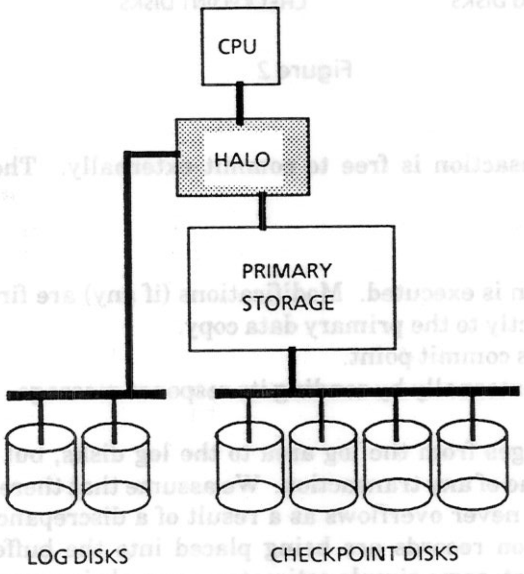


Figure 4

HALO in more detail, showing the internal registers and data and command paths to the CPU and primary memory. As shown in the figure, HALO intercepts communications between the processor and the memory to produce a log. The log is stored within HALO in a circular queue implemented in hardware. Registers IN and OUT point to the head and tail of the queue. The queue and other registers are non-volatile.

Each queue entry represents an update made to memory, and contains the address of the word, the old contents of the word, and the new contents. A new entry is made each time the processor writes a database word. Only modifications to the database need be logged; HALO can ignore changes to data in other areas, such as system tables. The entries at the tail of the queue are constantly being written out to disk to make room for new entries*.

When HALO detects a write command, it must make a new log entry. To begin, it stores the address and write value in a temporary register queue. Once this is done, the database CPU is allowed to continue. HALO does not immediately forward the write request to the memory unit, for it

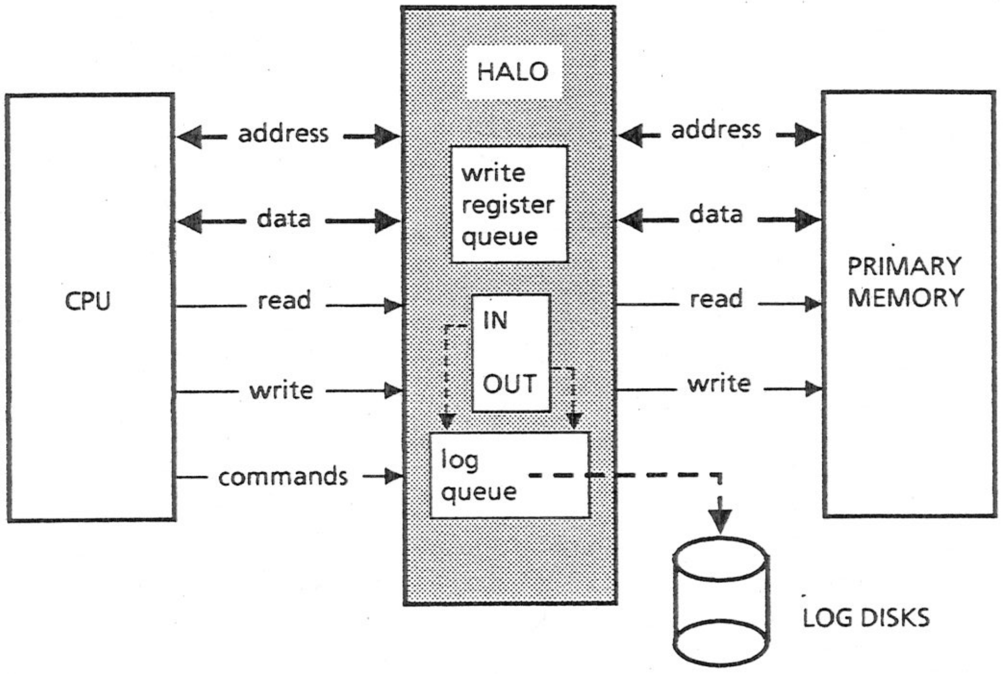


Figure 5

must first obtain the old value stored at the given address. Thus, the write request is converted into a read request. When the old value is obtained, a new log entry is made. Once the log entry is safe in the queue, HALO uses a spare main memory cycle to perform the write.

Read commands issued by the CPU are inspected by HALO. If the address corresponds to a pending write, HALO returns the new value stored in the register bank. Otherwise, HALO forwards the request to the memory unit. (This same strategy is used to handle pending writes in some caches, e.g. in the IBM 3033. A small number of write registers is usually sufficient [12].)

There are several commands the CPU issues directly to HALO. One is a "begin/end transaction" command. It causes a special BET entry to be made in the log, indicating the end of the previous transaction and the beginning of a new one. (As soon as the BET entry is made, the ending transaction is committed. At this point, the old data values in the log, i.e. the undo information, can be discarded.) A second command is the "abort transaction" command. When HALO receives this, it goes through each log entry until the last BET record. For each entry, it stores the old data value in the corresponding memory location.

Thus HALO manages logging transparently to the CPU. HALO would also cooperate with the CPU in restoring the primary copy of the data in the event of a system crash.

----

* As with the BB mechanism, we assume that HALO's log queue is sufficiently large to handle bursts of writes from the CPU.

The CPU is still responsible for checkpointing in this scheme, however, and continues to pay those overhead costs. Checkpointing under HALO is like checkpointing under BB, i.e. pages are not copied to a buffer area before being flushed and no dependencies need be maintained between database pages and transaction log records.

3.5 RELIABLE MEMORY

We have considered two additional recovery schemes which we have dubbed the *reliable memory* mechanisms. In these we assume that all of primary memory is made non-volatile and error-free, perhaps through the use of battery power, replicated data, and a voting mechanism or an algorithm for reliable stable storage such as the one described in [9]. While this kind of reliable memory would be costly, it would offer a number of performance benefits. Unlike their unreliable counterparts, reliable memory systems need not perform any checkpoint operations since the primary data copy is never lost. Also, log disks (and their associated overhead costs) are not necessary since log records need only be maintained for the life of the transaction. Thus, a reliable memory system might be useful when system restart time or transaction response times are critical. The results we obtain for these systems are useful in themselves as a base against which to compare results from the unreliable memory systems.

We model reliable memory systems with and without HALO devices. The latter case (RM) is similar to the BB case discussed previously. The former case is called RMHALO. Block diagrams for RM and RMHALO are shown in Figures 6 and 7, respectively. Note that a hardware logger for a reliable memory system could be designed somewhat differently than we have described since it is only necessary to maintain log records for the life of their transaction. Thus HALO could keep log records temporarily buffered instead of flushing them to disk.

4.0 PERFORMANCE MODEL

There are two important situations in which one can compare the performance of recovery mechanisms. The first is during normal system operation, when transactions are being processed. In this situation we can compare degradations in the system's performance caused by the various mechanisms. The second is during recovery itself, when no transactions are running and the system is trying to attain a consistent state from which to restart.

Recovery consists of reloading primary storage from the checkpoint disks and processing the log to redo transactions that have been committed since the most recent checkpoint. Except for reliable memory*, none of the recovery mechanisms we have considered will differ significantly in the size or content of their logs. Each handles checkpointing in a similar fashion. Factors which do affect recovery, such as parallel paths from disks to primary memory, and log compression, will have their effects on any of the mechanisms. (See, for example, [6] or [11] for more complete discussions.) For these reasons, we expect recovery times to vary little from mechanism to mechanism, and have concentrated out efforts on the normal processing situation.

Degradations of transaction throughput and response time are the metrics we use to compare the mechanisms' effects on the normal operation of the transaction manager. Throughput is defined as the maximum number of transactions per second we can run while maintaining an average response time below some prescribed value. Response time is the sum of the time a transaction spends at the database CPU (CPU time) and the time, if any, spent at the log disks (disk time). The remainder of

----

* The reliable memory mechanisms do not suffer failures in the sense that the unreliable mechanisms do. A reliable memory mechanisms has no storage reload time since its storage is never destroyed. It has no log processing time since a log is not maintained. Transaction recovery information is only kept until commit time, in case the of a user-initiated abort. Thus we can consider recovery time for the reliable memory mechanisms to be zero.

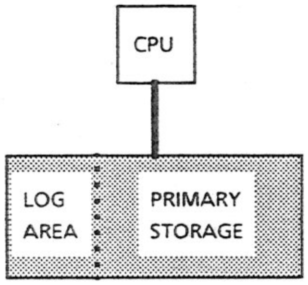


Figure 6

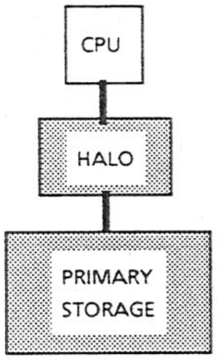


Figure 7

this section describes our performance model. The model consists of a simple queueing system with servers to represent the CPU and the log disks.

Section 4.1 describes the computation of CPU time, the response time of a transaction at the CPU server. We express the service time of the CPU server as a function of the amount of instruction overhead incurred as a result of using a particular recovery mechanism. We break this overhead down into two types, general and transaction specific, and develop equations to express each of them in terms of the model parameters. Once the service time has been determined we can determine CPU time using our queueing system.

Section 4.2 deals with disk time. Only two mechanisms (IC and GC) have disk time since the others do not require transactions to wait for log disks as long as there is sufficient bandwidth and buffer space available to handle the expected transaction rate.

4.1 CPU TIME

The database CPU is modeled as a single FCFS server with λ-dependent (λ is the transaction arrival rate) service time t_{cpu}. We determine t_{cpu} using

$$t_{cpu} = \frac{S}{A} \tag{4.1}$$

$$A = r_{cpu} - \textit{general recovery overhead}$$

$$S = T + \textit{transaction specific overhead}$$

where T is the size of the transaction body and r_{cpu} is the processor instruction rate. The units of A, r_{cpu}, and the general recovery overhead are instructions per second, while S, T, and the transaction specific overhead are measured in instructions per transaction.

A can be thought of as the net processing power available for transaction processing after *general recovery overhead* has been accounted for. General recovery overhead is overhead that is not attributable to any particular transaction, for example the overhead costs of the checkpointer. Although this overhead is not transaction-specific, its magnitude can depend strongly on the transaction rate (e.g. checkpointing). Therefore the general recovery overhead is a function of λ as well as the model parameters.

On the other hand, some types of overhead can be attributed to individual transactions. We term those costs the *transaction specific overhead.* Such overhead effectively increases the size of the raw transaction. Thus we add transaction specific overhead to T to get S, which can be thought of as the actual size of a transaction after crash recovery is considered. The transaction specific overhead is a function of model parameters only.

We assume that T is an exponentially distributed random variable with mean T'. For a given set of values for the model parameters and a given input rate λ, the recovery overheads, r_{cpu}, and A are constants. t_{cpu} is then exponentially distributed with mean given by

$$t'_{cpu} = \frac{S'}{A}$$

Assuming that transaction inter-arrival times are exponentially distributed allows us to calculate the mean response time for transactions at the CPU using the standard equations for M/M/1 servers [7], namely

$$CPUtime = \frac{1}{\mu_{cpu} - \lambda} \tag{4.2}$$

where the mean service rate μ_{cpu} is simply

$$\mu_{cpu} = \frac{1}{t'_{cpu}}$$

The following subsections describe in more detail how we arrive at values for A and S'. Table 1 lists model parameters which are used in these calculations, along with their default values.

4.1.1 TRANSACTION OVERHEAD

The net transaction size, S, is the sum of the raw transaction size (in CPU instructions) plus any instruction overhead related to crash recovery. This overhead may include the log copying cost, flushing transaction records to the log disks, maintaining dependencies between log pages, and context switching, depending on which recovery mechanism we are considering.

Transactions incur a cost when they note modifications in the log area before actually making changes to the database; we call this expense the *log copy cost.* The log copy cost is taken to be

name	description	default value	units
r_{cpu}	processor speed	5	MIPS
c_d	CPU disk IO constant	1000	instrs.
c_{cp}	CPU checkpoint initialization constant	5000	instrs.
c_{bc}	CPU buffer copy constant	10	instrs.
c_{cs}	CPU context switch constant	10	instrs.
c_{cpd}	CPU checkpoint-log dependency constant	10	instrs.
c_{lpd}	CPU log-log dependency constant	100	instrs.
T^s	mean transaction size	5000	instrs.
w_p	database pages written by one transaction	4	
w_r	database records written by one transaction	4	
w_w	database words written by one transaction	4	
d_l	per transaction log data constant	32	bytes
d_e	per entry log data constant	8	bytes
s_{log}	log page size	8K	bytes
s_{db}	DB page size	4K	bytes
s_r	record size	512	bytes
s_w	word size	4	bytes
n	total number of pages	250000	
α	locality parameter	0.01	
t_{cp}	inter-checkpoint interval	300	seconds
λ	mean transaction arrival rate		transactions/sec
gran	logging granularity	WORD	

Table 1

$$\frac{d_l}{s_w} + (\textit{\# of data items})\left[\frac{(\textit{size of data item})}{s_w} + \frac{2(\textit{size of data item})}{s_w} + \frac{d_e}{s_w} + c_{bc}\right] \qquad (4.3)$$

instructions per transaction. The first term represents the cost of writing the transaction's log header information (e.g. transaction ID) to the buffer. The terms in brackets represent the costs of reading undo data, writing redo and undo data, writing header information (e.g. address), and initialization overhead, for a single data item. (Reading or writing one word of data is assumed to cost a single instruction.) A data item may be a page, a record, or a word depending on the logging granularity chosen (this is specified by the parameter *gran*). The size of a data item is thus given by one of s_{db}, s_r, or s_w. A single transaction handles either w_p, w_r, or w_w data items, again depending on the logging granularity.

The log copy cost is paid under any mechanism in which the database CPU manages the log. This includes IC, GC, BB, and RM. It does not include the HALO schemes since logging there is accomplished by special purpose hardware transparent to the database CPU.

Flushing log pages to disk also incurs a cost. Transactions are charged

$$c_d + c_{cs} + c_{lpd} \tag{4.4}$$

instructions for each page flushed. The first two terms represent the costs of initiating the IO and handling the interrupt that occurs when it completes. The last term represents the overhead associated with maintaining dependencies among the queued log pages. IC is the only mechanism under which these charges are incurred by transactions. The database CPU is responsible for log flushes in other mechanisms (GC, BB) as well, however in those cases we will treat the flushing costs as part of the general recovery overhead. This is because the log flushes are shared among transactions under the GC and BB mechanisms. Note that the c_{lpd} term is not part of the flush cost for BB since the BB mechanism does not need to maintain log record dependencies.

Each IC transaction only incurs the log flush cost once as long as the total size of its log data is less than the log page size s_{log}. A transaction's log data is

$$\textit{transaction data} = d_l + (\textit{\# of data items})(2(\textit{size of data item}) + d_e) \tag{4.5}$$

bytes in size, where the number and size of units depend on the logging granularity, as described above.

4.1.2 GENERAL OVERHEAD

Determining A, the number of CPU instructions per second available for transaction processing, involves subtracting from r_{cpu} any recovery-related overhead costs which are not specifically attributable to any single transaction. These are principally checkpoint costs although log flushing and context switching overheads are sometimes included, as described below.

The cost of checkpointing is a function of the number of database pages dirtied since the last checkpoint. Each dirty page must be flushed to the checkpoint disks. Specifically, the model takes the total cost of a checkpoint to be

$$c_{cp} + (\textit{\# of dirty pages})\Big[2(c_d + c_{cs})\Big] \qquad \textit{BB, HALO} \tag{4.6}$$

$$c_{cp} + (\textit{\# of dirty pages})\left[2(c_d + c_{cs}) + \frac{2s_{db}}{s_w} + (\textit{total queue size})c_{cpd}\right] \qquad \textit{IC, GC} \tag{4.7}$$

An expression for the number of dirty pages is given by equation 4.8. In equations 4.6 and 4.7 the bracketed term in the expression is the cost of checkpointing a single dirty page. This includes the costs of copying the page to a temporary buffer (IC and GC only) and flushing it to the backup copy. For the IC and GC mechanisms only, the per-page cost also includes checking for dependencies between the page and transaction log records. This takes time proportional to the number of such records in the log queue, which will be determined in the next section (equation 4.12 for IC, 4.15 for GC) when the log disks are considered.

We assume that each dirty page must be flushed twice for the sake of "careful" updating. Each flush incorporates the cost of a context switch (c_{cs}) since the checkpoint process must block while waiting for the disks each time a page is flushed.

This total checkpoint cost is assumed to be spread by the database CPU over the entire interval between checkpoints, t_{cp}. We therefore divide the above expressions by this constant to arrive at the checkpoint overhead per unit time.

The number of dirty pages needing to be checkpointed is a function of the number of page writes that have occurred since the last checkpoint. We assume that the number of dirty pages approaches the total number of pages in the database (n) according to the function

$$(\#of\ dirty\ pages) = D(w) = n\left[1 - e^{-\beta w}\right] \tag{4.8}$$

where w is the number of writes and $D(w)$ the number of dirty pages.

The parameter β is used to set the rate of approach. This rate depends on the locality of the database references, since highly local updates will dirty new pages more slowly than less local updates. Therefore, β depends on the locality parameter α. Specifically, β is given by

$$\beta = \frac{-\log_e(1-\alpha)}{n}$$

This relation guarantees that

$$D(n) = \alpha n$$

which is intended to provide a convenient guide for choosing a value for α, the locality parameter. α should have a value between zero and one, and it can be interpreted as the fraction of the n database pages that will be dirtied when n updates have been made since the last checkpoint. Choosing a value of α exactly specifies the function $D(w)$.

We use w', the expected value of w, to determine the number of dirty pages. w' is given by

$$w' = \lambda w_p t_{cp}$$

which is just the expected number of transactions during the checkpoint interval times the number of pages modified by each transaction.

Checkpoint overhead costs are not paid by the reliable memory mechanisms (RM and RMHALO). These mechanisms do not require checkpoints since their primary storage is never lost.

The GC and BB mechanisms have additional overhead costs for flushing log pages to the log disks. These costs are treated as general overhead and not as part of the transaction size because they apply to groups of transactions when using group commits. As was the case with per-transaction commits, the cost per log page flushed is

$$c_d + c_{cs} + c_{lpd} \tag{4.4}$$

instructions. The expected number of flushes per unit time is given by

$$\left[\frac{(transaction\ data)}{s_{log}}\right]\lambda \tag{4.9}$$

An expression for transaction data was given in equation 4.5. The total overhead, in instructions per second, is then the product of the expressions 4.4 and 4.9.

4.2 DISK TIME

For the IC and GC mechanisms it is necessary to determine the response time of the log disks, since it is a part of a transaction's response time. This is not true of the other mechanisms because their log entries are maintained in stable storage. Table 2 lists some parameters we will use in our calculations. We will also make use of the rotation time of the log disks and the transfer time for a log page, which we determine using

name	description	default value	units
d	number of log disks	12	
v	log disk rotation speed	60	r.p.s.
s_{track}	log disk track capacity	32K	bytes

Table 2

$$t_{rot} = \frac{1}{v} \qquad t_{trans} = t_{rot}\left[\frac{s_{log}}{s_{track}}\right]$$

and values from Tables 1 and 2.

We assume that each of the log disks is a FCFS server with service time distributed as shown in Figure 8. There are d such disks, and record is flushed to a given disk with probability $1/d$, as shown

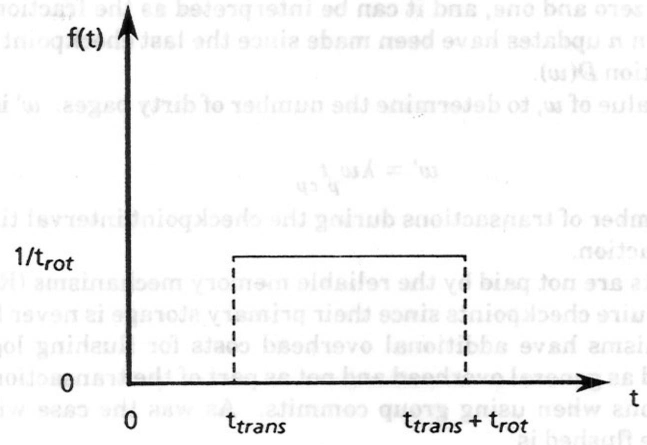


Figure 8

in Figure 9. We now discuss the response time for each of the mechanisms in turn.

4.2.1 IMMEDIATE COMMIT

For the IC mechanism, the inter-arrival times for the log disk system are distributed in the same way as the inter-departure times for the database CPU, since each transaction creates one log flush (provided it does not write more than a log page full of data). Thus the disk servers are each M/G/1, the average total input rate is $\lambda_{dtot} = \lambda$, and the input rate for each disk is

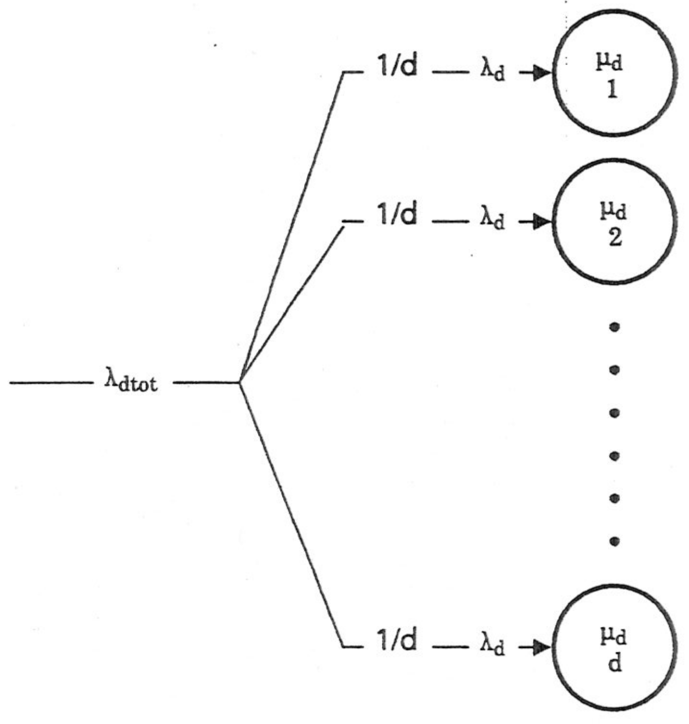


Figure 9

$$\lambda_d = \frac{\lambda}{d}$$

Response time is then given by [7]

$$IC\ response\ time = \frac{1}{\mu_d} + \frac{\rho(1+C_b^2)}{2\mu_d(1-\rho)} \tag{4.10}$$

$$\mu_d = \frac{1}{t_{trans} + \frac{t_{rot}}{2}} \qquad C_b^2 = \frac{t_{rot}^2 \mu_d^2}{12} \qquad \rho = \frac{\lambda_d}{\mu_d}$$

To determine the CPU overhead for checkpointing, as discussed in the previous section (equation 4.7), we need to determine the *total queue size* for the IC log disks. The total queue size is the total number of data items queued or being serviced at the log disks. The checkpointer needs to scan this queue to check for data dependencies before flushing database pages.

We use Little's result [7] and the response time just determined to get the average number of log pages queued at an individual disk. We then multiply by the number of data items per log page and the number of log disks to get the mean total queue size, which is used in the checkpointing overhead calculation. Thus

$$\textit{pages in IC disk queue} = \lambda_d(\textit{IC response time}) \tag{4.11}$$

by Little's result, and

$$\textit{total IC queue size} = (\textit{pages in IC disk queue})(\textit{\# of data items})d \tag{4.12}$$

The number of data items per log page is simply the number of data items per transaction, since each transaction writes a single log page. As before, this is given by one of w_p, w_r, or w_w, depending on the logging granularity.

4.2.2 GROUP COMMIT

The situation is slightly more complex for the GC mechanism because arrivals to the log disk system are no longer Poisson. Departures from the database CPU are Poisson, but they are then grouped into commit groups of size f transaction log records. f, the number of transaction log records per log page, is given by

$$f = \frac{s_{log}}{(\textit{transaction data})}$$

Inter-departure times of full log pages are thus distributed as the sum of f transaction record inter-departure times, i.e. with mean f/λ and variance f/λ^2. We can then determine the mean arrival rate and variance of the inter-arrival times at each of the d log disks (see appendix B):

$$\lambda_d = \frac{\lambda}{df} \qquad \sigma_d^2 = \frac{d^2 f}{\lambda^2(2d-1)} \tag{4.13}$$

Since the log disk queues are G/G/1 it is difficult to get an exact value for response time. The service rate μ_d is the same as for the IC case. C_b and ρ are also calculated as before, using the new values for μ_d and λ_d. We then use the following approximation [8]

$$\textit{GC response time} \approx \left[\frac{1+C_b^2}{\frac{1}{\rho^2}+C_b^2}\right]\left[\frac{\sigma_d^2+\sigma_b^2}{2(1-\rho)}\lambda_d\right] \tag{4.14}$$

$$\sigma_b = \frac{t_{rot}^2}{12}$$

The approximation gets better as ρ-->1. This is reasonable since for small values of ρ response time is often dominated by *grouping time*, which we discuss next.

To the response time just determined for the log disks we add another term called the *grouping time*. This represents the time a transaction's log record sits in memory waiting for its commit group to form. Grouping time is unusual in that it decreases as the transaction input rate λ increases. To account for grouping time we note that the ith transaction record to become part of a commit group must wait for $f-i$ additional transactions to arrive. If x is the inter-arrival time, the average grouping time across transactions is given by

$$\textit{grouping time} = \frac{\sum_{i=1}^{f}(f-i)x}{f} = \frac{\frac{f(f-1)x}{2}}{f} = \frac{(f-1)}{2}x$$

The model uses the expected value of the average grouping time, which is just

$$\frac{(f-1)}{2}\left[\frac{1}{\lambda}\right]$$

This value is added to the response time of all GC transactions.

Total queue size needs to be determined for the GC mechanisms as well. The number of pages in the disk queue is determined in much the same way as it was for the IC mechanism (equation 4.11), namely

$$\textit{pages in GC disk queue} = \lambda_d(\textit{GC response time}) + 0.5 \tag{4.15}$$

The extra half-page arises because on the average half of a log page will be in-core waiting to be filled before being placed in a queue. Total queue size (compare equation 4.12) is then determined by

$$\textit{total GC queue size} = (\textit{pages in GC disk queue})(\textit{\# of data items})fd \tag{4.16}$$

The extra factor f results from grouping multiple transaction log records onto a single log page.

5.0 RESULTS

In this section we compare the six recovery mechanisms according to transaction response time and throughput. In addition we examine the sensitivity of the results to variations in parameter values. Figure 10 shows average transaction response time as transaction input rate is varied, using the default parameter values.

Most noticeable are the relatively long response times for GC transactions and the unusual shape of their curve. Both of these are the result of grouping time, a delay which increases with decreasing transaction rates. This delay can be reduced by introducing time-outs to the log buffers, so that partially empty log pages are flushed if they remain in core for too long. IMS/VS Fast Path implements such a time-out mechanism [3]. Of course, time-outs increase the number of log flushes, which was troublesome for the IC mechanism, so the choice of an appropriate limit is very important.

We define the maximum throughput for each mechanism as the greatest λ such that the expected response time is less than t_{rlim}, a parameter. Maximum throughput can be determined graphically from the response time curves by drawing a horizontal line corresponding to the desired response time and looking at the crossing points of the curves for the various mechanisms. These throughputs are given in Table 3 for the default set of parameter values and for several other sets whose response time curves are not shown. The table briefly describes how these other sets differ from the default. In all cases, t_{rlim} is 1 second.

For the default values there is little difference among the mechanisms except for the inferior performance of IC. The primary cause of this gap is the high CPU overhead paid by IC for log flushes (one flush per transaction). We ruled out queueing time at the log disks as an important factor by observing only a slight change in IC's throughput when we increased the number of log disks. Similarly, we eliminated checkpointing overhead as an important factor since it is identical for the IC and GC mechanisms.

None of the other parameter sets change the performance ranking observed for the default set (except between HALO and RM), but the gaps between mechanisms vary significantly. As expected, the relative benefits of special-purpose recovery mechanisms, e.g. HALO, increase with increasing logging overhead costs or volume of log data.

To determine the sensitivity of our numbers to changes in parameter values, we varied each parameter over a wide range and plotted throughputs for each of the mechanisms. Graphs for three of the parameters are shown in Figures 11-13. Not all of the mechanisms are represented in every graph; only those which are affected by the parameter are plotted.

The first graph describes throughput variations with $\overline{T}$, the mean transaction size. Although the actual throughputs vary, the relationship among the various mechanisms changes little. The flattening of the IC curve for small transaction sizes is due to a log disk bottleneck.

Figure 12 shows the effect of variation of the disk IO constant, c_d. The fact that the HALO, BB, and GC values vary at a similar rate implies that most of the IO is a result of checkpointing;

parameter set	recovery mechanism					
	IC	GC	BB	HALO	RM	RM HALO
defaults	777	934	968	983	985	999
logging granularity = RECORD	628	674	695	983	757	999
high overhead (10x $c_d, c_{cp}, c_{bc}, c_{cs}$)	284	734	787	861	920	999
low overhead (0.5x $c_d, c_{cp}, c_{bc}, c_{cs}$)	860	948	980	991	989	999

Table 3

otherwise BB and GC would have steeper curves than HALO. The IC mechanism suffers for its log flush rate.

The last figure describes throughput changes when we vary the number of updates done by each transaction in the least local way, i.e. every word updated by a transaction resides on a different page. This results in increases in checkpointing costs as well as logging costs as we increase the number of updates. The drop-off of throughput described by the graph is different for the various mechanisms, introducing performance discrepancies where there were none. By comparison, varying the number of words updated without changing the number of pages touched (not shown) results in nearly identical rates of decline for BB, and GC, while HALO's throughput is not affected at all.

5.0 CONCLUSIONS

We have studied transaction processing in systems where the database resides in main memory. Because of their simplicity, MSDB systems have the potential for very high performance. That is, as long as crash recovery is performed efficiently.

We have presented a family of crash recovery mechanisms for MSDB systems and have developed a simple model to compare their impact on performance. The model identifies three sorts of recovery-related CPU overhead, namely copying to and dumping the log, and checkpointing. In addition we model the log disks for the two mechanisms in which the disks figure in a transaction's response time.

We have compared six mechanisms, two of them based on a hypothetical reliable primary memory. The others include a proposed hardware logging device, group commits, and the use of non-volatile memory to maintain log buffers. By comparing these mechanisms, we have also determined the relative importance of the various types of recovery overhead.

Our results indicate that if transaction throughput is the major consideration, then a group commit (GC) strategy may be indicated. As long as the log disks do not represent a serious bottleneck, throughputs for GC are almost indistinguishable from those of the more expensive hardware-based approaches.

This is not true of GC's transaction response times, which suffer because it is necessary to wait for the log disks. If response times are critical, then the battery-backup (BB) strategy may be the best. For our default parameter set the BB mechanisms proved to be nearly the equal of HALO in terms of both throughput and response time. The former mechanism would be significantly easier to implement; all that is required is static RAM or a battery backup and reserve power supply on a small portion of the system's memory. Therefore it appears that BB is a very practical alternative as long as transactions are not extremely verbose and recovery software overheads (e.g. for disk IO) are reasonable.

If transactions update substantial amounts of data, if the disk IO overhead is high, or if we wish to push the system to its limits, then a device like HALO may be indicated. However, in all the cases considered, the improvement in throughput was less than 10%. Similarly, the reliable memory mechanisms (RM) provide a minimal performance improvement over HALO. (In fact, HALO outperforms RM when transactions make a lot of clustered updates.) Comparing BB to RM shows that most of the performance gains we can expect from converting to non-volatile RAM come as soon as we have converted enough to maintain the log buffer.

Although our results for HALO and RM are not encouraging (given their current costs), they are interesting because they show that clever crash recovery mechanisms (like GC) on conventional hardware do fairly well. Furthermore, HALO and RM have advantages that are not observed directly in the performance numbers. As discussed earlier, RM recovers from a system failure in negligible time. HALO simplifies the software substantially. Since it lets one "play back" CPU activity, it could be useful for debugging and recovery from operating system errors (when an inconsistent state is detected, we roll back to a previous consistent state).

6.0 REFERENCES

[1] Chandy, K.M., et. al., "Analytic Models for Rollback and Recovery Strategies in Data Base Systems", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975.

[2] DeWitt, D.J., et. al., "Implementation Techniques for Main Memory Database Systems", *SIGMOD Record*, Vol. 14, No. 2, 1984.

[3] Gawlick, D., and Kinkade, D., "Varieties of Concurrency Control in IMS/VS Fast Path", *Database Engineering Bulletin*, Vol. 8, No. 2, June 1985.

[4] Gray, J., "Notes on Database Operating Systems", in "*Operating Systems: and Advanced Course*", R. Bayer, et. al., Ed., Springer-Verlag, 1979.

[5] Gray, J., et. al., "One Thousand Transactions Per Second", Tandem TR 85.1, Tandem Computers, Cupertino, CA, 1985.

[6] Hagmann, R., "A Crash Recovery Scheme for a Memory Resident Database System", Xerox Corp. Palo Alto Research Center, Palo Alto, CA, 1985.

[7] Kleinrock, L., "*Queueing Systems, Vol. 1: Theory*", John Wiley & Sons, NY, 1975.

[8] Kleinrock, L., "*Queueing Systems, Vol. 2: Computer Applications*", John Wiley & Sons, NY, 1975.

[9] Lampson, B.W., and Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System", Xerox Corp. Palo Alto Research Center, Palo Alto, CA, April 1979.

[10] Reuter, A., "Performance Analysis of Recovery Techniques", *ACM TODS*, Vol. 9, No. 4, December 1984.

[11] Salem, K.M., and Garcia-Molina, H., "Disk Striping", *Proc. Int'l Conference on Data Engineering*, Los Angeles, CA, February, 1985.

[12] Smith, A.J., "Cache Memories", *ACM Computing Surveys*, Vol. 14, No. 3, September 1982.

[13] Verhofstad, J.S.M., "Recovery Techniques for Database Systems", *ACM Computing Surveys*, Vol. 10, No. 2, June 1978.

APPENDIX A - LOG QUEUE SIZES

In section 4.2.2 we determined the total number of data items (words, records, or pages) in the log queue for the GC mechanisms. We can use the same procedure to estimate the mean size of the log queue for the mechanisms with non-volatile memory (BB and HALO). Recall that the mean total number of queue data items was given by

$$(\lambda_d(\textit{response time}) + 0.5)(\textit{\# of units})fd$$

Log disk response time using BB or HALO will be the same as the response time determined for GC since all three mechanisms group their commits. λ_d and f can also be determined as was done for GC. We multiply the number of data items in the queue by the size of a data item (s_{db}, s_r, or s_w) to get the mean total size of the queue in bytes. Figure 14 shows a plot of the mean total queue size as λ is varied (λ_d is a function of λ). The default parameter set is used, except that curves for several values of s_{log}, the log page size, are shown.

The actual size of non-volatile storage will have to be greater than is indicated in the figure if we expect overflows to be rare. However, having even three of four times this much buffer space available will not result in a huge non-volatile store. For example, 32K bytes of stable storage should be more than adequate for the buffer area using the parameters given, even at very high transaction rates. By using more log disks, we can reduce the necessary buffer area still further.

APPENDIX B - INTER-ARRIVAL TIMES OF GROUP COMMITS

We wish to determine the mean and variance of inter-arrival time at the log disks for the group commit mechanism. The situation is illustrated in Figure 15, where the Y_i is the time between the departures of the (i-1)th and ith groups from the grouping mechanism. These groups go to one of the d disks at random, with probability $1/d$ of going to any particular disk.

Without loss of generality, we can assume that the (i-1)th commit group was dispatched to disk j. The next inter-arrival time for disk j, call it X_j, is the amount of time that will elapse before another commit group is sent to that disk. If group i goes to disk j, then $X_j = Y_i$. This event occurs with probability $1/d$. If group i goes to some other disk, but group $i+1$ goes to disk j, then $X_j = Y_i + Y_{i+1}$. This occurs with probability $(1/d)((d-1)/d)$. Continuing in this fashion, we arrive at an expression for X_j:

$$X_j = \sum_{k=0}^{\infty} \frac{1}{d} \left[\frac{(d-1)}{d} \right]^k \left[\sum_{l=0}^{k} Y_{i+l} \right]$$

We then rearrange the terms to put the outer summation in terms of the variables Y_i and arrive at the new expression:

$$X_j = \sum_{l=0}^{\infty} \frac{Y_{i+l}}{d} \left[\sum_{k=l}^{\infty} r^k \right] \qquad r = \left[\frac{d-1}{d} \right]$$

Rewriting this equation as

$$X_j = \frac{Y_i}{d} \sum_{k=0}^{\infty} r^k + \sum_{l=1}^{\infty} \frac{Y_{i+l}}{d} \left[\sum_{k=0}^{\infty} r^k - \sum_{k=0}^{l-1} r^k \right]$$

and making use of the equalities

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r} \qquad \sum_{k=0}^{l-1} r^k = \frac{1-r^l}{1-r}$$

we get a simplified expression for the disk inter-arrival time

$$X_j = \sum_{l=0}^{\infty} Y_{i+l} r^l$$

The grouping mechanism clusters transaction log records into groups of f records each, where f is determined by the size of a log page and the amount of log data written by a transaction. Arrival of transaction log records at the grouping mechanism is Poisson since transaction arrivals to the system and processing time are both Poisson. Therefore all of the Y_i are distributed as the sum of f identical exponential random variables, each with parameter λ, i.e. with mean and variance given by:

$$\overline{Y} = \frac{f}{\lambda} \qquad \sigma_Y^2 = \frac{f}{\lambda^2}$$

(We drop the subscript i since all of the times Y_i are identically distributed.) With this knowledge we can determine the mean and variance of the disk inter-arrival time X (we again drop the subscript since all of the disks are identical). The mean is given by

$$\overline{X} = \sum_{l=0}^{\infty} \overline{Y} r^l = \overline{Y}\left[\frac{1}{1-r}\right] = d\,\overline{Y} = \frac{df}{\lambda}$$

The Y_i are independent, so we can write a similar equation for variance

$$\sigma_x^2 = \sum_{l=0}^{\infty} \sigma_Y^2 r^{2l} = \sigma_Y^2\left[\frac{1}{1-r^2}\right] = \left[\frac{d^2}{2d-1}\right]\sigma_Y^2 = \frac{fd^2}{\lambda^2(2d-1)}$$

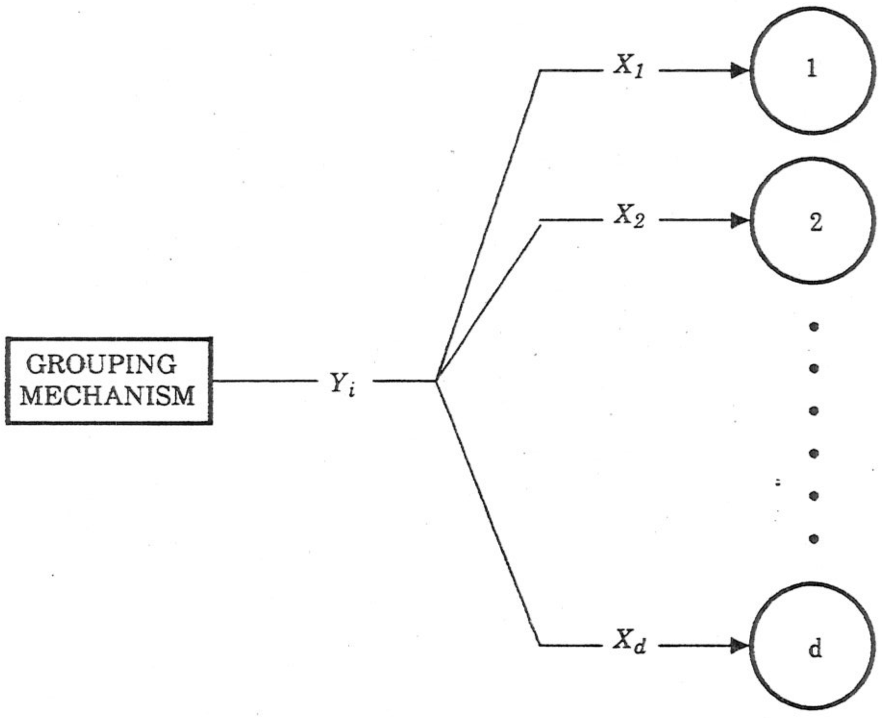
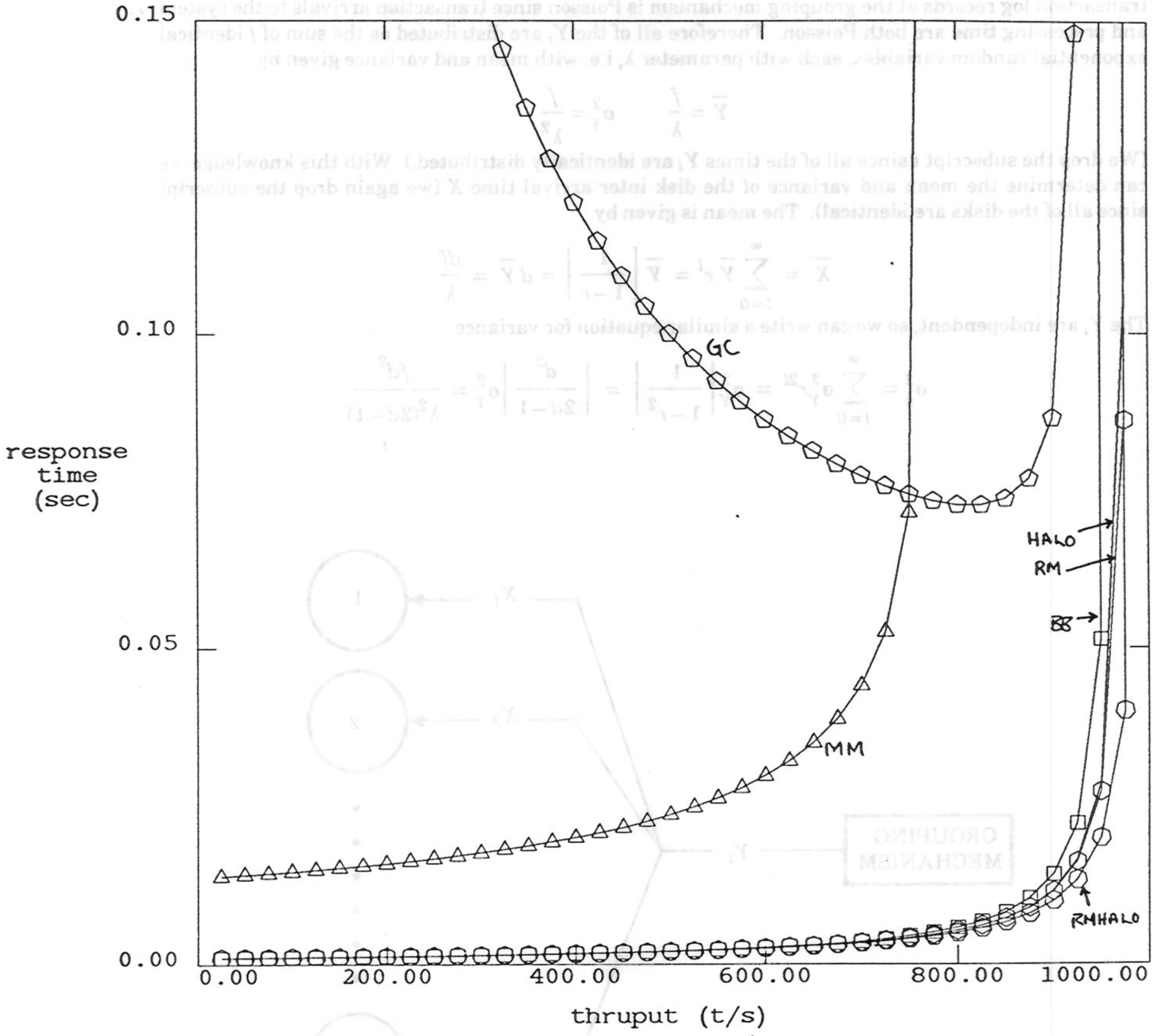
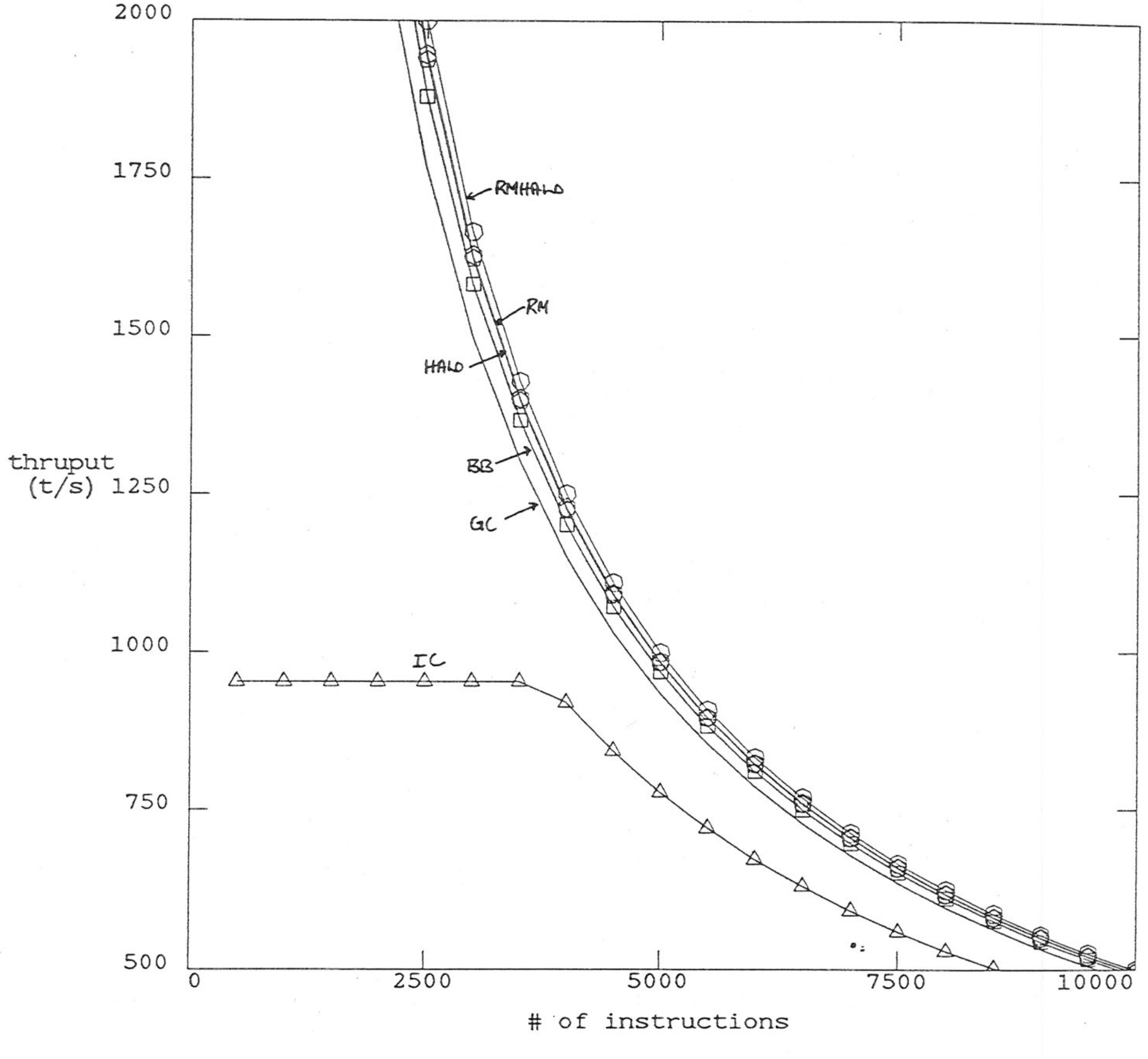


Figure 15



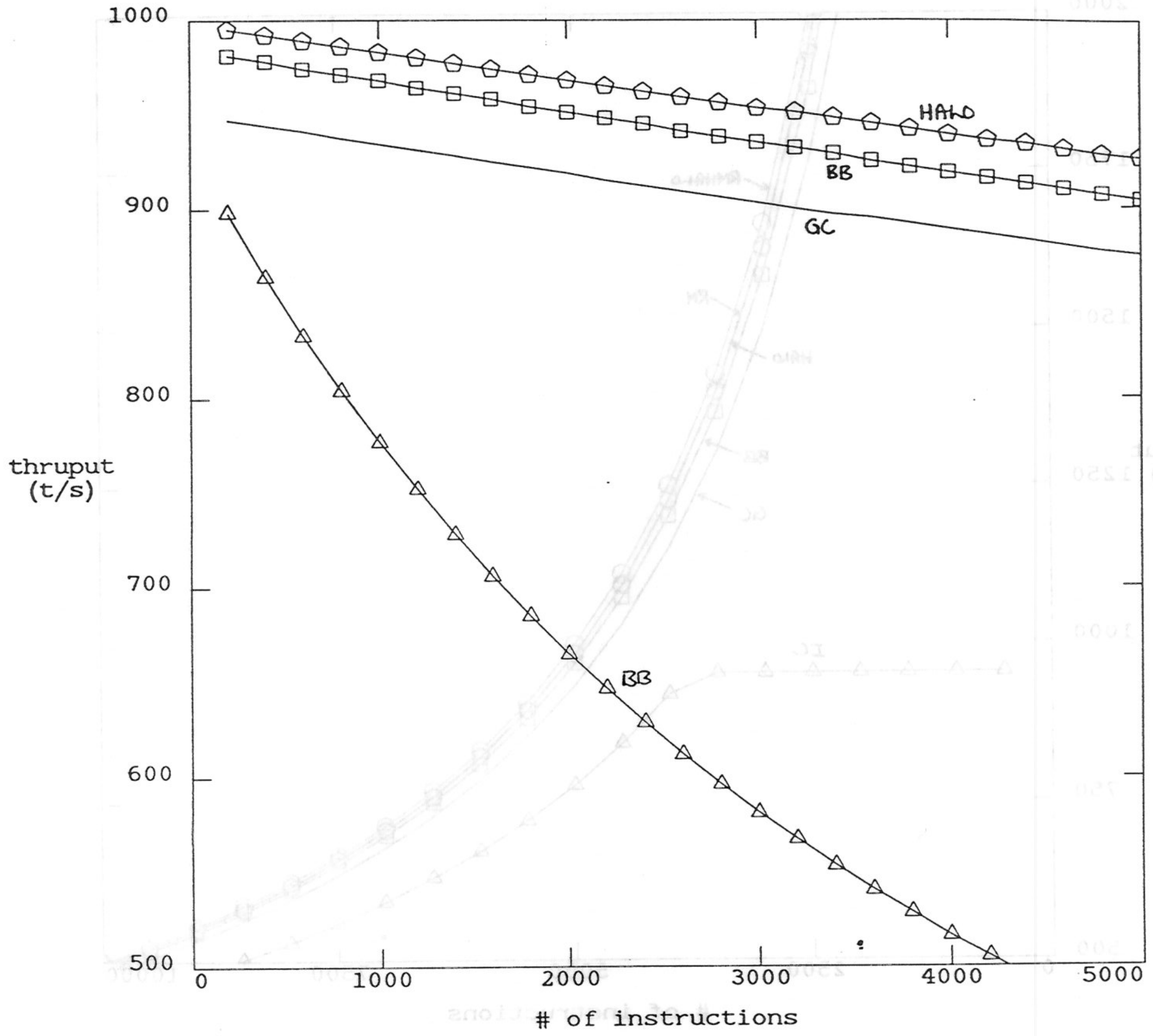
RESPONSE TIME

FIGURE 10



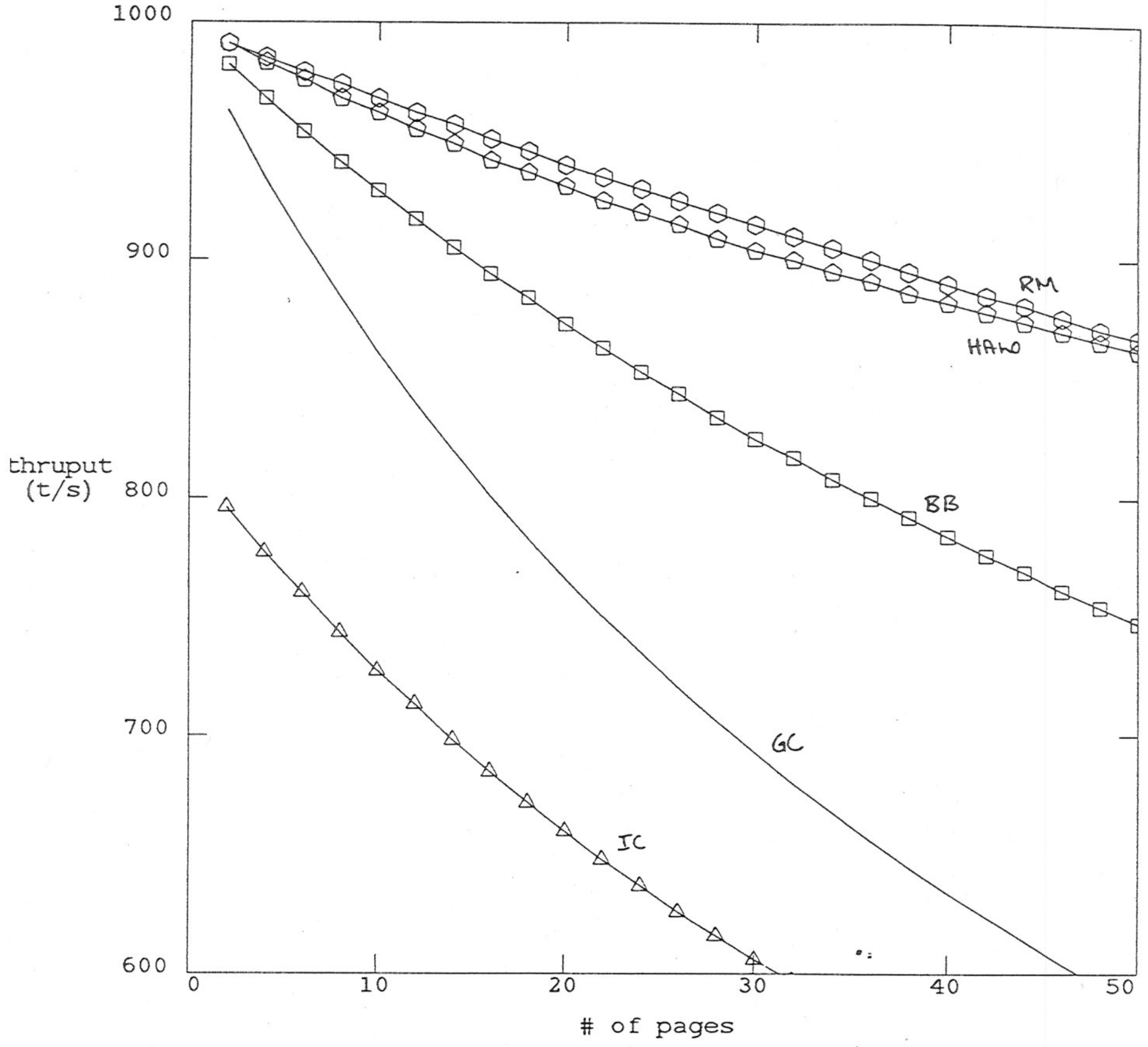
TRANSACTION SIZE

FIGURE 11



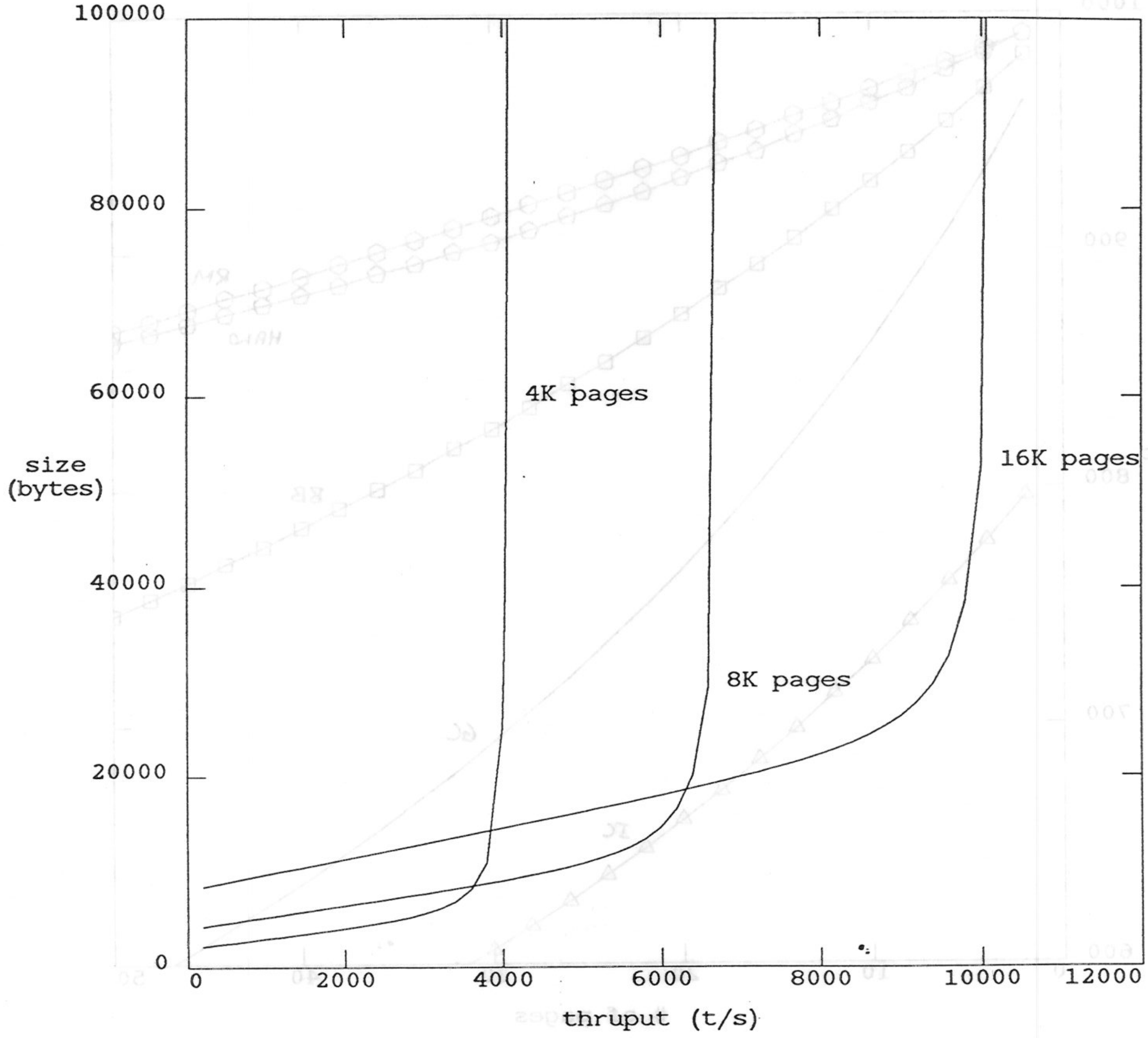
IO OVERHEAD CONSTANT

FIGURE 12



PAGES WRITTEN PER TRANSACTION

FIGURE 13



LOG QUEUE SIZE

FIGURE 14