PERFORMANCE EVALUATION OF AN OPERATING SYSTEM TRANSACTION MANAGER

Akhil Kumar and Michael Stonebraker

University of California
Berkeley, Ca., 94720

Abstract

A conventional transaction manager implemented by a database management system (DBMS) was compared against one implemented within an operating system (OS) in a variety of simulated situations. Models of concurrency control and crash recovery were constructed for both environments, and the results of a collection of experiments are presented in this paper. The results indicate that an OS transaction manager incurs a severe performance disadvantage and appears to be feasible only in special circumstances.

1. INTRODUCTION

In recent years there has been considerable debate concerning moving transaction management services to the operating system. This would allow concurrency control and crash recovery services to be available to any clients of a computing service and not just to clients of a data manager. Moreover, this would allow such services to be written once, rather than implemented within several different subsystems individually. Early proposals for operating system-based transaction managers are discussed in [MITC82, SPEC83, BROW81]. More recently, additional proposals have surfaced, e.g: [CHAN86, MUEL83, PU86].

On the other hand, there is some skepticism concerning the viability of an OS transaction manager for use in a database management system. Problems associated with such an approach have been described in [TRAI82, STON81, STON84, STON85]. and revolve around the expected performance of an OS transaction manager. In particular, most commercial data managers implement concurrency control using two-phase locking [GRAY78]. A data manager has substantial semantic knowledge concerning its processing environment. Hence, it can distinguish index records from data records and implements a two-phase locking protocol only on the latter objects. Special protocols for locking index records are used which do not require holding index locks until the end of a transaction. On the other hand, an OS transaction manager cannot implement such special tactics unless considerable semantic information can be given to it.

Crash recovery is usually implemented by writing before and after images of all modified data objects to a log file. To ensure correct operation, such log records must be written to disk before the corresponding data records, and the name write ahead log (WAL) has been used to

This research was sponsored by a grant from the IBM Corporation

descibe this protocol [GRAY81, REUT84]. Crash recovery also benefits from a specialized semantic environment. For instance, data managers again distinguish between data and index objects and apply the WAL protocol only to data objects. Changes to indexes are usually not logged at all since they can be reconstructed at recovery time by the data manager using only the information in the log record for the corresponding data object and information on the existence of indexes found in the system catalogs. An OS transaction manager will not have this sort of knowledge and will typically rely on implementing a WAL protocol for all physical objects.

As a result, a data manager can optimize both concurrency control and crash recovery using specialized knowledge of the DBMS environment. The purpose of this paper is to quantify the expected performance difference that would be incurred between a DBMS and an OS transaction manager. Consequently, we discuss in Section 2.1 the assumptions made about the simulation of a conventional DBMS transaction manager In Section 2.2 we turn to discussing the environment assumed in an OS transaction environment and then discuss intuitively the differences that we would expect between the two environments. Section 3 presents the design of our simulator for both environments, while Section 4 closes with a collection of experiments using our simulator.

2. TRANSACTION MANAGEMENT APPROACHES

In this section, we briefly review schemes for implementing concurrency control and crash recovery within a conventional data manager and an operating system transaction manager and highlight the main differences between the two alternatives.

2.1. DBMS Transaction Management

Conventional data managers implement concurrency control using one of the following algorithms: dynamic (or two-phase) locking [GRAY78], time stamp techniques [REED78, THOM79], and optimistic methods [KUNG81].

Several studies have evaluated the relative performance of these algorithms. This work is reported in [GALL82, AGRA85b, LIN83, CARE84, FRAN83, TAY84]. In [AGRA85a] it has been pointed out that the conclusions of these studies were contradictory and the differences have been explained as resulting from differing assumptions that were made about the availability of resources. It has been shown that dynamic locking works best in a situation of limited resources, while optimistic methods perform better in an infinite-resource situation. Dynamic locking has been chosen as the concurrency control mechanism in our study because a limited-resource situation seems more realistic. The simulator we used assumes that page level locks are set on 2048 byte pages on behalf of transactions which are held until the transaction commits. Moreover, index level locks are held at the page level and are released when the transaction is finished with the corresponding page.

Crash recovery mechanisms that have been implemented in data managers include write-ahead logging (WAL) and shadow page techniques. These techniques have been discussed in [HAER83, REUT84]. From their experience with implementing crash recovery in System R, the designers concluded that a WAL approach would have worked better than the shadow page scheme they used [GRAY81]. In a another recent comparison study of various integrated concurrency control and crash recovery techniques [AGRA85b], it has been shown that two-phase locking and write-ahead logging methods work better than several other schemes which were considered. In view of this a WAL technique was simulated in our study. We assume that the before and after images of each changed record are written to a log. Changes to index records are not logged, but are assumed to be reconstructed by recovery code.

2.2. OS Transaction Management

We assume an OS transaction manager which provides **transparent** support for transactions. Hence, a user specifies the beginning and end of a transaction, and all objects which he reads or writes in between must be locks in the appropriate mode and held until the end of the transaction. Clearly, if page level locking is selected, then performance disasters will result on index and system catalog pages. Hence, we assume that locking is done at the subpage level, and assume that each page is divided into 128 byte subpages which are individually locked. Coonsequently, when a DBMS record is accessed, the appropriate subpages must be identified abd locked in the correct mode.

Furthermore, the OS must maintain a log of every object written by a transaction so that in the event of a crash or a transaction abort, its effect on the database may be undone or redone. We assume that the before and after images of each 100 byte subpage are placed in a log by the OS transaction manager. These entries will have to be moved to disk before the corresponding dirty pages to obey the WAL protocol.

The reason for choosing this level of locking and logging granularity is because larger granularities seem clearly unworkable, and this particular granule size is close to the one proposed in an OS transaction manager for the 801 [CHAN86].

2.3. Main Differences

The main differences between the two approaches are:

the DBMS transaction manager will acquire fewer locks
the DBMS transaction manager will hold locks for shorter times
the DBMS will have a much smaller log

The data manager locks 2048 byte pages while the OS manager locks 100 byte subpages. Moroever, the DBMS sets only short-term locks on index pages while the OS managr holds index level locks until the end of a transaction. The larger granule size in the DBMS solution will inhibit parallelism; however the shorter lock duration in the indexes will have the opposite effect. Moreover, the larger number of OS locks will increase CPU time spent in locking.

The third difference is that the log is much larger for the OS alternative. The data manager only logs changes made to the data records. Corresponding updates made to the index are not logged because the index can be reconstructed at recovery time from a knowledge of the data updates. For example, when a new record is inserted, the data manager does not enter the changes made to the index into the log. It merely writes an image of the new record into the log along with a 20-byte message indicating the name of the operation performed, in this case an insert. On the other hand, the OS transaction manager will log the index insertion. In this case half of an index page must be rearranged, and the before and after images for about 10 subpages must be logged. and after-images of all these sub-pages.

These differences are captured in the simulation models for the data manager and the OS transaction manager described in the next section.

3. SIMULATION MODEL

A 100 Mb database consisting of 1 million 100-byte records was simulated. Since sequential access to such a large database will clearly be very slow, it was assumed that all access to the database takes place via secondary indexes maintained on up to 5 fields. Each secondary index was a 3-level B-tree. To simplify the models it was assumed that only the leaf level pages in the

index will be updated. Consequently, the higher level pages are not write-locked. The effect of this assumption is that the cost associated with splitting of nodes at higher levels of the B-tree index is neglected. Since node-splitting occurs only occasionally, this will not change the results significantly.

The simulation is based on a closed queuing model of a single-site database system. The number of transactions in such a system at any time is kept fixed and is equal to the multiprogramming level, MPL, which is a parameter of the study. Each transaction consists of several read, rewrite, insert and delete actions, and its workload is generated according to a stochastic model described below. Modules within the simulator handle lock acquisition and release, buffer management, disk I/O management, CPU processing, writing of log information, and commit processing. Each job is assigned CPU time in a round-robin manner. CPU and disk costs involved in traversing the index and locating and manipulating the desired record are simulated.

First, appropriate locks are acquired on pages or sub-pages to be accessed. In case a lock request is not granted because another transaction holds a conflicting lock, the transaction has to wait until the conflicting transaction releases its lock. Next a check is made to determine whether the requested page exists in the buffer pool. If the page is not in the buffer, a disk I/O is initiated, and the job is made "not ready". When the requested pages become available, the CPU cost for processing it is simulated. This cycle of lock acquisition, disk I/O (if necessary), and processing is repeated until all the actions for a given transaction are completed. The amount of log information that will be written to disk is computed from the workload of the transaction and the time for this task is accounted for. When a transaction completes, a commit record is written to the log in memory and I/O for this log page is initiated. As soon as this commit record is moved to disk the transaction is considered to be over and a new transaction is accepted into the system. Checkpoints are simulated at 5 minute intervals. Deadlock detection is done by a timeout mechanism. The maximum duration for which a transaction is allowed to run is determined adaptively.

Figure 1 lists the major parameters of the simulation. The parameters that were varied along with the range of variation are listed in Figure 2. Figure 3 gives the values assigned to the fixed parameters. The number of disks available, *numdisks,* was varied between 2 and 10. *cpu_mips,* the processing power of the cpu in mips, was kept at 2.0. The cpu cost of various actions was defined in terms of the number of cpu instructions they would consume. For example, *cpu_lock* the cost of executing a lock-unlock pair, was initially kept at 2000 instructions and reduced in intervals to 200 instructions.

In order to simulate a real-life interactive situation, two types of transactions, short and long, were generated with equal probability. The number of actions in a short transaction was uniformly distributed between 10 and 20. Long transactions were defined as a series of two short transactions separated by a think time which varied uniformly between 10 and 20 seconds. A certain fraction, *frac1,* of the actions were updates and the rest were reads. Another fraction, *frac2,* of the updates were inserts or deletes. These two fractions were drawn from uniform distributions with mean values equal to *modify1* and *modify2,* respectively, which were parameters of the experiments.

Rewrite actions are distinguished from inserts and deletes because the cost of processing these actions is different. A read or a rewrite action affects only one index while an insert or a delete action would affect all indexes. The index and data pages to be accessed by each action are generated at random. Assuming 100 entries per page in a perfectly balanced 3-level B-tree index, it follows that the second-level index page is chosen at random from 100 pages, while the third-level index page is chosen at random from 10,000 pages. The data page is chosen at random from 71,000 pages. (Since the data record size is 100 bytes and the fill factor of each data page is 70%,

buf_size: size of buffer in pages
cpu_ins_del: cpu cost of insert or delete action
cpu_lock: cost of acquiring lock
cpu_IO: cpu cost of disk IO
cpu_mips: processing power of cpu in mips
cpu_present: cpu overhead of presentation services
cpu_read: cpu cost of read action
cpu_write: cpu cost of rewrite action
disk_IO: time for one disk I/O in mili sec
modify1: average fraction of update actions in a transaction
modify2: number of inserts, deletes as a fraction of all updates
MPL: Multiprogramming Level
numdisks: number of disks
numindex: number of indexes
page_size: size of a page
sub_page_size: size of a sub-page in bytes

Figure 1: Major parameters of the simulation

there are 71,000 data pages.)

The main criterion for performance evaluation was the overall average transaction processing time, *av_proc_time.* This is defined as:

$$\frac{Total\ number\ of\ transactions\ completed}{Total\ time\ taken}$$

Notice that *av_proc_time* is the inverse of throughput. Another criterion, *performance gap,* was

buf_size: 250,.....,1000 pages
cpu_lock: 200,.....2000 instructions
cpu_mips: 2.0
modify1: 5,....,50
MPL: 5,.....,20
numdisks: 2,.....,10
numindex: 1,2,.....,5

Figure 2: Range of variation of the parameters

used to express the relative difference between the performance of the two alternatives. *Performance gap* is defined as:

$$\frac{(av_proc_time_{os} - av_proc_time_{data})x100}{av_proc_time_{data}}$$

where
$av_proc_time_{os}$: transaction processing time for the OS alternative
$av_proc_time_{dm}$: transaction processing time for the data manager alternative

4. RESULTS OF THE EXPERIMENTS

In this section we discuss the results of various experiments which were conducted to compare the performance of the two alternatives.

4.1. Varying Multiprogramming Level

In the first set of experiments, the multiprogramming level was varied between 5 and 20. The number of disks, *numdisks* was 2 and the cost of executing a lock-unlock pair, *cpu_lock* was 2000 instructions. *Modify1* was kept at 25 which means that on the average, 25% of the actions were updates and 75% actions were reads. *Modify2* was made 50 indicating that on the average about half the updates were rewrites and the remainder were inserts or deletes. The average transaction processing times for various multiprogramming levels are shown in Figure 4.

The figure shows that the average transaction processing time, *av_proc_time* falls sharply when the multiprogramming level increases from 5 to 8 because the utilization of disk and cpu resources increases. The improvement in *av_proc_time,* however, tapers off as MPL increases beyond 15 because the utilization of one of the resources saturates. The figure also shows that the data manager performs consistently better by more than 20%. When MPL is 15 or 20, the *performance gap* is 27%. This gap is due to the increased level of contention in the indexes and the extra cost of writing more information into the log. The OS transaction manager writes a log which is approximately 30 times larger than the data manager log.

---

cpu_IO: 3000 instructions
cpu_present: 10000 instructions
cpu_read: 7000 instructions
cpu_write: 12000 instructions
disk_IO: 30 ms
page_size: 2048 bytes
sub_page_size: 100 bytes

Figure 3: Values assigned to fixed parameters

---

2 10
3 10
file graph4

Figure 4: Average processing time as a function of multiprogramming level

4.2. Varying Transaction Mix

In order to examine how the transaction mix affects the performance of the two alternatives, *modify1,* the average fraction of modify actions (i.e., the sum of rewrite, delete and insert actions) as a percentage of the total number of actions was varied and the average transaction processing time was determined. The value of *modify1* affects the logging activity in the system, and, consequently, it was also expected to alter the relative performance of the two alternatives.

Modify1 was kept variously between 5 and 50. The multiprogramming level was kept at 15, while the cost of setting a lock was 2000 instructions. The average transaction processing time as a function of *modify1* is shown in Figure 5. The figure shows that *av_proc_time* grows linearly with increasing *modify1* in both cases, although the slope of the line is much greater for the operating system alternative. When the average fraction of modify operations is 5, the performance gap between the data manager and the OS transaction manager is small (7%). However, the gap widens as *modify1* increases and becomes 45% when *modify1* is 50.

There are two reasons for this behavior. First, contention is less when *modify1* is small. Contention occurs when one transaction tries to write-lock an object which is already read-locked by another transaction or when an attempt is made to lock an object which is write-locked by another transaction. When the fraction of modify actions is small, fewer write-locks are applied, and, hence, contention is reduced. Secondly, since fewer objects are write-locked, the amount of data logged for crash recovery purposes is also reduced. Both these factors benefit the OS alternative more than they do the data manager. Therefore, the relative performance of the OS transaction manager improves.

These experiments show that the transaction mix has a drastic effect on the relative performance of the two alternatives being considered. It appears that the OS transaction manager would be viable when updates are few (say, less than 20%). However, when the fraction of update actions in a transaction is high, the extra overhead incurred in performing transaction management within the OS is severe.

4.3. High Conflict Situation

The next set of experiments was conducted to see how the two alternatives would behave when the level of conflict is increased. Reducing the size of the database increases the conflict level because the probability that two concurrent transactions will access the same object becomes greater. Therefore, in order to compare the two alternatives, the size of the database was used as a surrogate for the level of conflict, and *av_proc_time* was determined for various values of database size. The transaction size was kept constant while the size of the database was

Figure 5: Average processing time as a function of transaction mix

reduced in intervals from 100 Mb to 6.4 Mb. The number of entries in each index page was reduced correspondingly in such a way that the B-tree remained balanced. For example, if the number of entries on an index page of a 3-level B-tree is reduced from 100 to 50, and the B-tree is kept perfectly balanced, there would be 125,000 entries in the leaf-level pages of the B-tree index. Since a record in our model is 100 bytes wide, this corresponds to a 12.5 Mb database.

In each case, the simulator was modified for the new size of the database. The multiprogramming level was kept at 10 and *modify1* was 50. Figure 6 shows the behavior of the two alternatives for various database sizes. The database size is plotted on the X-axis on a logarithmic scale. Note that a smaller value for the database size indicates a higher level of conflict. The *av_proc_time* is plotted on the Y-axis.

In both cases, *av_proc_time* increases as the database becomes smaller. Furthermore, the *performance gap* widens from 28% for a 100 Mb database to 51% for a 6.4 Mb database. This means that the performance of the OS transaction manager drops more sharply than that of the data manager. This happens because contention increases faster in the OS transaction manager than in the data manager since the former holds locks on the index pages for a longer duration. This factor overshadows any advantages that the OS alternative gets from applying finer granularity locks. This experiment illustrates that in high-conflict situations the OS alternative becomes clearly unacceptable.

Figure 6: Transaction processing time for various database sizes

4.4. Adding More Disks

With 2 disks and a 2 mips cpu the system became I/O-bound. To make it less I/O-bound, the number of disks, *numdisks* was increased in intervals from 2 to 10, and *av_proc_time* was determined for both alternatives. *MPL* was kept at 20 and *cpu_lock* was made equal to 2000 instructions. The average transaction processing time as a function of number of disks is plotted in Figure 7.

Two observations should be made. First, when *numdisks* is increased from 8 to 10 the improvement in performance is negligible. Therefore, with 8 disks the system becomes cpu-bound. Secondly, with 2 disks the *performance gap* is 27% while with 10 disks it widens to 60%. This means that the *performance gap* in a cpu-bound system is two times as large as in an I/O-bound system. When the system is I/O-bound the gap is mainly because the OS transaction manager has to write a larger log and, therefore, it consumes greater I/O resources. On the other hand, when the system is cpu-bound, the gap is explained by the greater cpu cycles that the OS transaction manager consumes in applying finer granularity locks.

4.5. Lower Cost of Locking

The experiments described above show that the OS transaction manager consumes far more cpu resources than the data manager. This occurs because, as explained earlier, the OS transaction manager must acquire more locks than the data manager. In this section we have varied the cost of lock acquisition in order to examine its effect on the performance of the two alternatives. Basically, the cost of executing a lock-unlock pair which was originally 2000 cpu instructions was reduced in intervals to 200 instructions. The purpose of these experiments was to evaluate what benefits were possible if *cpu_lock* could be lowered through hardware assistance.

It is obvious that a reduced cost of locking would improve system throughput only if the system were cpu-bound. This was done by increasing the number of disks to 8. The multiprogramming level was kept at 20. Figure 8 shows the *av_proc_time* of the two alternatives for various values of *cpu_lock.* The performance of the OS transaction manager improves as *cpu_lock* is reduced while the data manager performance does not change. Consequently, the *performance gap* reduces from 54% to 30% as *cpu_lock* falls from 2000 instructions to 200 instructions. In the case of the data manager, the cost of acquiring locks is a very small fraction of the total cpu cost of processing a transaction, and therefore, a lower *cpu_lock* does not make it faster. On the other hand, since the OS transaction manager acquires approximately five times as many locks as the data manager this cost is a significant component of the total cpu cost of processing a transaction

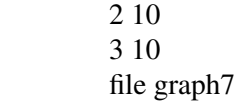

Figure 7: Effect of increasing disks on transaction processing time

2 10
3 10
file graph8

Figure 8: Effect of cost of locking on average transaction processing time

and reducing it has an appreciable impact on its performance.

These experiments show that a lower *cpu_lock* would improve the relative performance of the OS transaction manager considerably in a cpu-bound situation. However, inspite of this improvement, the data manager is still 30% faster.

4.6. Buffer Size and Number of Indexes

Two more sets of experiments were done to examine how the buffer size and the number of indexes affect the relative performance of the two alternatives. In both sets, *MPL* was 15, and *modify1* and *modify2* were 25 and 50, respectively. The buffer size which was 500 pages in all of the above experiments was kept variously at 250, 750, and 1000 pages. Table 1 shows the average transaction processing time as a function of buffer size for the two situations. The relative difference between the performance of the two alternatives is approximately 28% in all cases. Therefore, the buffer size does not seem to affect the relative performance of the OS transaction manager as compared to the data manager.

In all of the experiments above, the number of indexes was kept at 5. In the next set of experiments the parameter *numindex* was varied to see how it affects the *performance gap.* Table 2 shows the average transaction processing times and the *performance gap* for the two alternatives when *numindex* is varied from 1 to 5. When *numindex* is 5 the *performance gap* between the two alternatives is 27% whereas with only one index it reduces to 9%. This occurs because as

	Buffer Size			
	250	500	750	1000
Data Manager	1.64	1.57	1.50	1.46
OS Manager	2.10	2.00	1.92	1.88
Performance Gap	28%	27%	28%	29%

Table 1: Average processing time for various buffer sizes

described above, all the indexes have to be updated for insert and delete actions. With fewer indexes the amount of updating activity is reduced and fewer locks have to be acquired. Hence the performance gap is reduced. This shows that if the number of indexes on the database is fewer, the relative performance of the OS transaction manager improves.

5. Conclusion

5.1. Implications for Feasibility

The performance of an OS transaction manager was compaed with that of a conventional data manager in a variety of situations. With few exceptions, the OS transaction manager uniformly performed more than 20% worse than the data manager which, in our opinion, is a substantial performance penalty. The effect of several important parameters on the relative performance of the two alternatives was studied and analyzed. It was found that the OS transaction manager is viable when:

the fraction of modify actions is low
number of indexes on the database is low
conflict level is low

If the above conditions do not hold then the performance of the OS transaction manager becomes unacceptable. Such restricted viability does not seem to justify the OS alternative. The effect of a lower cost of setting locks within the OS transaction manager was also examined. However, even when this cost was made very small, the OS alternative continued to be more than 20% inferior to the data manager.

5.2. Future Directions

It is evident from our experiments that in order to make the operating system solution really viable it is necessary to provide a greater level of semantics into the OS. Such semantics will take the form of an ability to distinguish between data and index, and an algorithm for updating an index. Additionally, a capability has to be provided for the user to define the structure of the index and the data pages. All this will certainly make the operating system considerably more complex and whether it is worthwhile is an open question.

	Number of Indexes				
	1	2	3	4	5
Data Manager	0.95	1.12	1.27	1.42	1.57
OS Manager	1.04	1.37	1.58	1.80	2.00
Performance Gap	9%	22%	24%	27%	27%

Table 2: Average processing time for varying number of indexes

REFERENCES

[AGRA85a] Agrawal, R., et. al., "Models for Studying Concurrency Control Performance : Alternatives and Implications," Proc. 1985 ACM-SIGMOD Conference on Management of Data, June 1981.

[AGRA85b] Agrawal, R., and Dewitt, D., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," ACM TODS, 10, 4, December 1985.

[BROW81] Brown, M. et. al., "The Cedar Database Management System," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., June 1981.

[CARE84] Carey, M. and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," Proc. 1984 VLDB Conference, Singapore, Sept. 1984.

[FRAN83] Franaszek, P., and Robinson, J., "Limitations of Concurrency in Transaction Processing," Report No. RC10151, IBM Thomas J. Watson Research Center, August 1983.

[GALL82] Galler, B., "Concurrency Control Performance Issues," Ph.D. Thesis, Computer Science Department, University of Toronto, September 1982.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1978, pp393-481.

[GRAY81] Gray, J. et. al., "The Recovery Manager of the System R Database Manager," ACM Computing Surveys, June 1981.

[HAER83] Haerder, T. and Reuter, A., "Principles of Transaction-Oriented Database Recovery," ACM Computing Surveys, December 1983.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," TODS, June 1981, pp 213-226.

[LIN83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp and Two-Phase Locking," Proceedings of the Ninth International Conference on Very Large Databases, Florence, Italy, November 1983.

[MITC82] Mitchell, J. and Dion, J., "A Comparison of Two Network-Based File Servers," CACM, April 1982.

[REED78] Reed, D., "Naming and Synchronization in a Decentralized Computer System," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., 1978.

[REUT84] Reuter, A., "Performance Analysis of Recovery Techniques," ACM TODS, 9, 4, Dec. 84.

[SPEC83] Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing," Operating Systems Review, Vol 17, No 2, April 1983. TODS 2, 3, September 1976.

[STON81] Stonebraker, M., "Operating System Support for Data Managers", CACM, April 1981.

[STON84] Stonebraker, M., "Virtual Memory Transaction Management," Operating System Review, April 1984.

[STON85] Stonebraker, M., et. al., "Problems in Supporting Data Base Transactions in an Operating System Transaction Manager," Operating System Review, January, 1985.

[TRAI82] Traiger, I., "Virtual Memory Management for Data Base Systems," Operating Systems Review, Vol 16, No 4, October 1982.

[TAY84] Tay, Y., and Suri, R., "Choice and Performance in Locking for Databases," Proceedings of the Tenth International Conference on Very Large Data Bases, Singapore, August 1984.

[THOM79] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control," TODS, June 1979.