PERFORMANCE EVALUATION OF AN OPERATING SYSTEM TRANSACTION MANAGER

Akhil Kumar[1] *and Michael Stonebraker*[2]

University of California
Berkeley, Ca., 94720

Abstract

A conventional transaction manager implemented by a database management system (DBMS) is 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 client 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 individually implemented within several subsystems. 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 implement 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 it can be given considerable semantic information.

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 describe 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 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 manager 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 turns to a collection of experiments using our simulator. Section 5 discusses one approach

[1] Graduate School of Business, 350 Barrows Hall

[2] EECS Department, 549 Evans Hall

This research was sponsored by a grant from the IBM Corporation

for making the OS semantically "smarter", and Section 6 concludes the paper.

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 if an infinite-resource environment is assumed. 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 and are held until the transaction commits. Moreover, locks on indexes 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 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 locked in the appropriate mode and the locks 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 100 byte subpages which are individually locked. Consequently, when a DBMS record is accessed, the appropriate subpages must be identified and locked in the correct mode.

This particular granule size was chosen because it is close to the one proposed in an OS transaction manager for the 801 [CHAN86]. The suitability of this granule size was further confirmed by an experiment comparing the performance of the OS transaction simulator at several different granularities. This experiment is discussed in Section 3.

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.

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 some locks for shorter times

the DBMS will write a much smaller log

The data manager locks 2048 byte pages while the OS manager locks 100 byte subpages; hence, the DBMS solution will acquire far fewer locks and spend less CPU resources in lock acquisition. Moreover, the DBMS sets only short-term locks on index pages while the OS manager 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 log is smaller for the DBMS transaction manager because it only logs changes made to the data records. Corresponding updates made to indexes are not logged because each 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 any index into the log. It merely writes an image of the new record into the log along with a header, assumed to be 20 bytes long, indicating the name of the operation performed. On the other hand, the OS transaction manager will log the index insertions. In this case half of one index page must be rearranged for each index that exists, and the before and after images of about 10 subpages must be logged.

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 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 model 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, we believe 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; the exact number 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. CPU and disk costs involved in traversing the index and locating and manipulating the desired record are also simulated.

In order to simulate an interactive transaction mix, two types of transactions 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.

Every action identifies a single record through one secondary index and then reads, rewrites, deletes, or inserts it. Rewrite actions are distinguished from inserts and deletes because the cost of processing them is different. It is assumed that a rewrite action affects only one key. However, an insert or a delete action would cause all indexes to be updated. 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 randomly 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%, there are 71,000 data pages.)

For each action, a collection of pages must be accessed. For each page the first step is to acquire appropriate locks on the page or subpages. If a lock request is not granted because another transaction holds a conflicting lock, the requesting transaction must wait until the conflicting transaction releases its lock. Deadlock detection is implemented through a timeout

Parameter Name	Description	Default Value
buf_size	size of buffer in pages	500
cpu_ins_del	CPU cost of insert or delete action	18000 instructions
cpu_lock	cost of acquiring lock	2000 instructions
cpu_IO	CPU cost of disk I/O	3000 instructions
cpu_mips	processing power of CPU in MIPS	2.0
cpu_present	CPU overhead of presentation services	10000 instructions
cpu_read	CPU cost of read action	7000 instructions
cpu_write	CPU cost of rewrite action	12000 instructions
disk_IO	time for one disk I/O in mili sec	30
fill-factor	percentage of bytes occupied on a page	70
modify 1	average fraction of update actions in a transaction	25
modify 2	number of inserts, deletes as a fraction of all updates	50
MPL	Multiprogramming Level	15
numdisks	number of disks	2
numindex	number of indexes	5
page_size	size of a page	2048 bytes
sub_page_size	size of a subpage in bytes	100

Table 1: Major parameters of the simulation

mechanism.[†] Next a check is made to determine whether the requested page is in the buffer pool. If not, a disk I/O is initiated and the job is made "not ready". When the requested page becomes available, the CPU cost for processing the page is simulated. This cycle of lock acquisition, disk I/O (if necessary), and processing is repeated until all pages in a given action have been processed. The amount of log information that will be written to disk is computed for the action and the time taken for this task is accounted for. When all actions for a transaction have been performed, a commit record is written into the log in memory and I/O for this log page is initiated. As soon as this commit record is moved to disk the current transaction is complete and a new one is started. Checkpoints [HAER83] are simulated every 5 minutes.

Table 1 lists the major parameters of the simulation and their default values. The parameters that were varied are listed in Table 2. Here the default value of each parameter is indicated as well as the range of variation simulated. For example, the number of disks available, ***numdisks***, was varied between 2 and 10 with a default value of 2. The CPU cost of each action was defined in terms of the number of CPU instructions it would consume. For example, ***cpu_lock***, the cost of executing a lock-unlock pair, was initially set at 2000 instructions and reduced in intervals down to 200 instructions.

Parameter	Range	Default
buf_size	250,.......,1000 pages	500
cpu_lock	200,......2000 instructions	2000
modify 1	5,....,50	25
MPL	5,......,20	15
numdisks	2,........,10	2
numindex	1,2,.......5	5

Table 2: Range of variation of the parameters

The main criterion for performance evaluation was the overall average transaction throughput rate, ***throughput*** defined as:

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

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

$$\frac{(\textit{throughput}_{DBMS} - \textit{throughput}_{OS})\ X\ 100}{\textit{throughput}_{OS}}$$

[†]The maximum time allocated to a transaction is a function of its number of actions and the maximum time for an action denoted by the variable ***max_action_len***. The best value for ***max_action_len*** is determined adaptively by varying it over a range of values and choosing the one which maximizes transaction throughput.

where
$\textit{throughput}_{OS}$: throughput for the OS transaction simulator
$\textit{throughput}_{DBMS}$: throughput for the DBMS transaction simulator

In order to determine the best locking granularity for the OS transaction manager, its throughput was determined for 4 different granule sizes with the other parameters fixed at their default values given in Table 1. The granule size was set at 1 record (100 bytes), 2 records (200 bytes), half-page (1024 bytes) and full-page (2048 bytes). The corresponding throughput rates are shown in Table 3, and it is evident that the OS transaction manager performs best with a granule size of 100 bytes. Notice that a granule is the basic unit for both locking and logging. When the granule size is increased, the cost of locking declines because fewer locks are acquired while the cost of writing the log increases since the before and after images become larger. This experiment shows that the net effect of having a coarser granularity is an increase in transaction processing cost. Hence, the best granule size (100 bytes) was used in the subsequent experiments.

In the DBMS transaction manager performance is not very sensitive to granule size. Moving to record level locking would allow the DBMS to lock smaller data objects; however, index locking would be unaffected. Hence, some improvement would be expected. We chose page level locking because it is popular in current commercial systems (e.g. DB2 [DATE84]). Repeating the experiments for record level granularity is left as a future exercise.

	Granule Size (bytes)			
	100	200	1024	2048
Throughput	0.50	0.49	0.45	0.34

Table 3: Throughput of the OS transaction manager for various locking granularities

4. EXPERIMENTAL RESULTS

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 set at 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% were reads. ***Modify2*** was set to 50 indicating that on the average half the updates were rewrites and the remainder inserts or deletes. The throughput rate for various multiprogramming levels is shown in Figure 1.

The figure shows that the *throughput* rises sharply when the multiprogramming level increases from 5 to 8 due to the increase in disk and CPU resource utilization. The improvement in *throughput*, however, tapers off as MPL increases beyond 15 because the utilization of the I/O system saturates. The figure also shows that the data manager consistently outperforms the OS alternative by more than 20%. When MPL is between 15 and 20, the *performance gap* is 27%. This gap results from increased contention for 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 that of the data manager.

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 corresponding throughput 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.

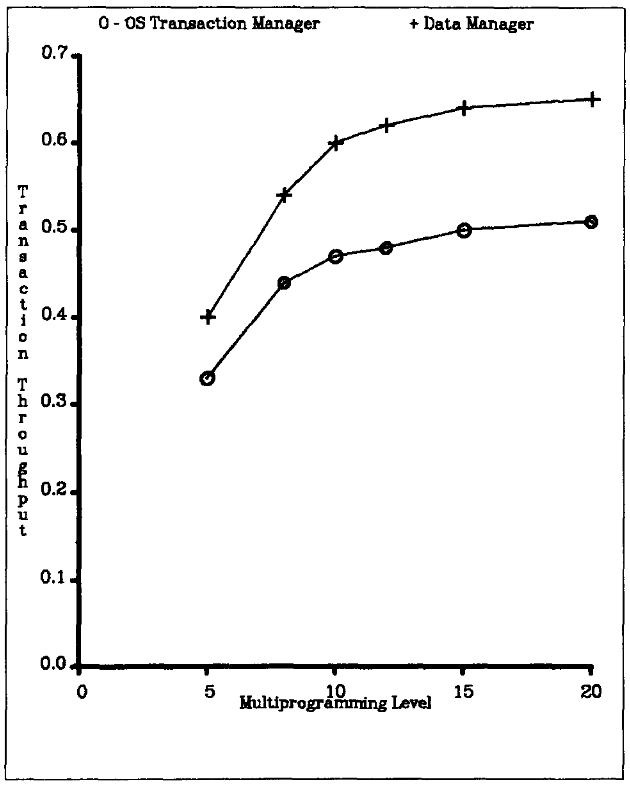

Figure 1: Throughput as a function of multiprogramming level

Modify1 was varied between 5 and 50, while the multiprogramming level was kept at 15, and the cost of setting a lock was 2000 instructions. The average throughput as a function of *modify1* is shown in Figure 2. The figure shows that *throughput* drops almost linearly with increasing *modify1* in both cases, although the magnitude of the slope is much greater for the operating system alternative. When the percentage 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 reaches 47% when *modify1* is 50 percent.

There are two reasons for this behavior. First, lock contention is lower when *modify1* is small. Such 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 percentage 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 lower. 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 could be viable when the proportion of updates is low (say, around 10%). Conversely, when the fraction of updates is high, a severe penalty is incurred by performing transaction management within the OS.

4.3. High Conflict Situation

The next set of experiments was conducted to see how the two alternatives behave when the level of conflict increases. Reducing the size of the database increases conflict since the probability that two concurrent transactions will access the same object rises. Therefore, database size was used as a surrogate for conflict level, and the corresponding *throughput* measured accordingly. The transaction size remained as before while the size of the database was reduced in intervals from 100 Mb to 6.4 Mb. As the database shrinks, the index continues to occupy three levels but with a reduced number of entries per page.

The multiprogramming level was kept at 10 and *modify1* was 25 percent. Figure 3 shows the behavior of the two alternatives for various database sizes. The database size is plotted on the X-axis in a logarithmic scale while the *throughput* appears on the Y-axis.

In both cases, *throughput* increases as the database becomes larger. Furthermore, the

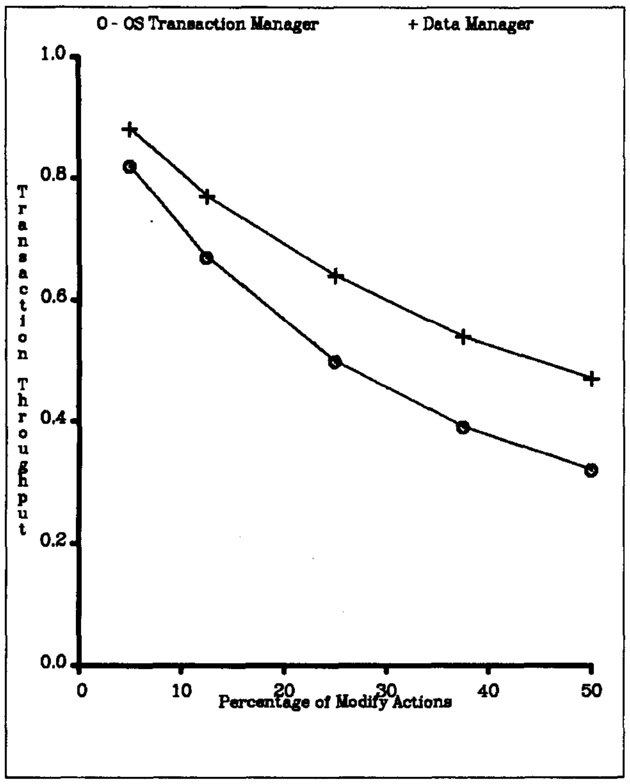

Figure 2: Throughput as a function of transaction mix

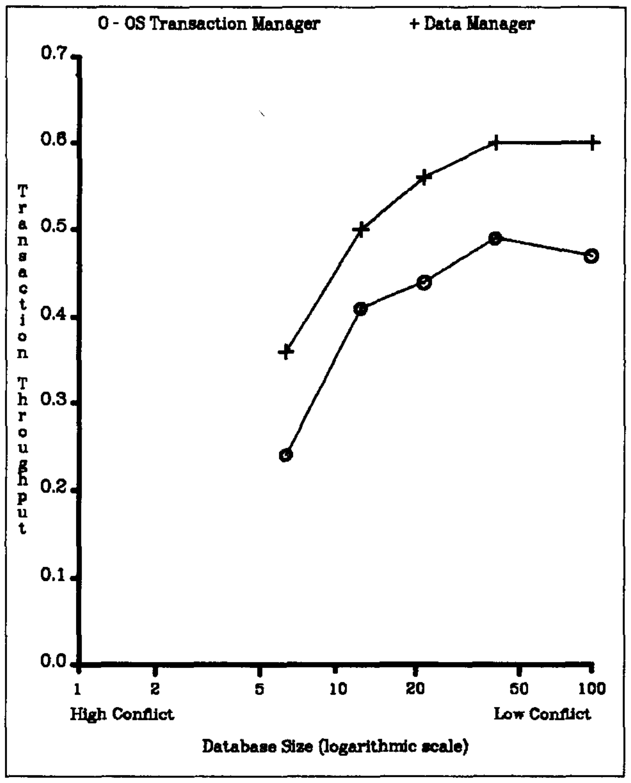

Figure 3: Transaction throughput for various database sizes

performance gap widens from 28% for a 100 Mb database to 50% for a 6.4 Mb database. This means that the performance of the OS transaction manager drops more rapidly than that of the data manager. The widening performance gap is due to a faster increase in contention within the OS since locks on the index pages are held 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 performance of the OS alternative becomes unacceptable.

4.4. Adding More Disks

With 2 disks and a 2 MIPS CPU the system is I/O-bound. To make it less I/O-bound, the number of disks, *numdisks* was increased in intervals from 2 to 10, and the *throughput* determined. *MPL* was kept at 20 and *cpu_lock* was made equal to 2000 instructions. The average throughput as a function of the number of disks is plotted in Figure 4.

Two observations should be made. First, as *numdisks* is increased, the throughput curve gradually flattens out as 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 twice as large as in an I/O-bound system. When the system is I/O-bound the gap is mainly due to the OS transaction manager having to write a larger log, thus consuming 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 greater CPU resources than the data manager in setting locks because it has to acquire more locks. In this section we have varied the cost of lock acquisition in order to examine its impact on the *performance gap*. Basically, the cost of executing a lock-unlock pair, originally 2000 CPU instructions, was reduced in intervals to 200 instructions. The purpose of this experiment was to evaluate what benefits were possible if *cpu_lock* could be lowered, say by hardware assistance.

It is obvious that a lower 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, and the multiprogramming level was kept at 20. Figure 5 shows the *throughput* of the two alternatives for various values of *cpu_lock*. The performance of the OS

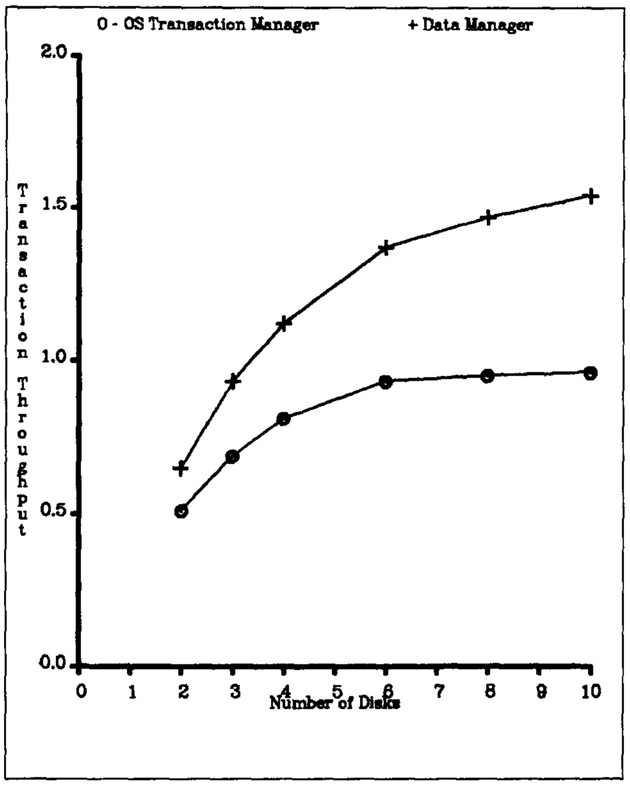

Figure 4: Effect of increasing disks on throughput

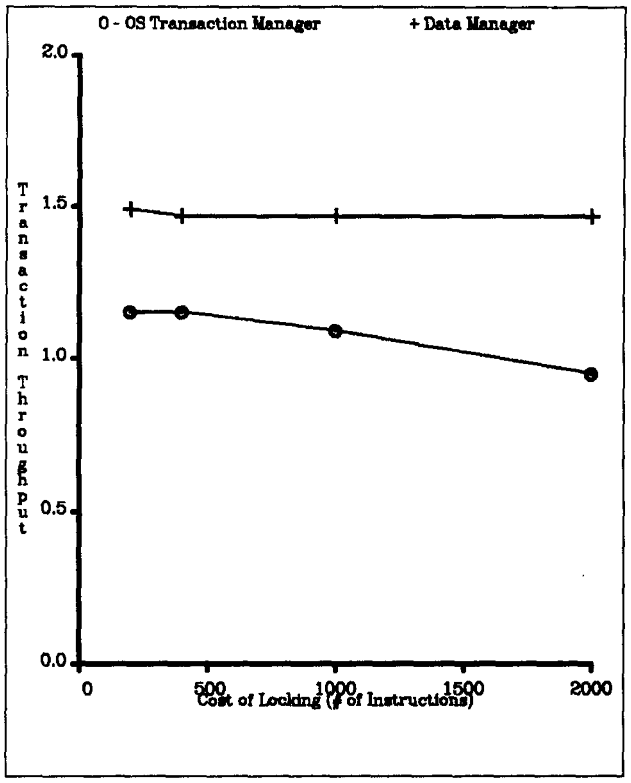

Figure 5: Effect of cost of locking on throughput

transaction manager improves as *cpu_lock* is reduced while the data manager performance changes very marginally. Consequently, the *performance gap* declines 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; hence, a lower *cpu_lock* does not make it significantly 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 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 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 the previous experiments was changed to 250, 750, and 1000 pages. Table 4 shows the average transaction throughput 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 change the relative performance of the two transaction managers.

	Buffer Size			
	250	500	750	1000
Data Manager	0.61	0.64	0.67	0.68
OS Manager	0.48	0.50	0.52	0.53
Performance Gap	27%	28%	29%	28%

Table 4: Throughput for various buffer sizes

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 5 shows the average transaction throughputs 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 28% whereas with only one index it reduces to 9%. As described earlier this occurs because all the indexes have to be updated for insert and delete actions. With fewer indexes the updating activity drops and fewer locks have to be acquired; therefore, the performance gap narrows. This shows that if the number of indexes on the database is reduced, the relative performance of the OS transaction manager improves.

	Number of Indexes				
	1	2	3	4	5
Data Manager	1.05	0.89	0.79	0.70	0.64
OS Manager	0.96	0.73	0.63	0.56	0.50
Performance Gap	9%	22%	25%	25%	28%

Table 5: Throughput for varying number of indexes

5. ADDING SEMANTICS TO THE OS

In section 2 a qualitative comparison of the main differences between the two approaches was given. The experiments of Section 4 have illustrated that these factors translate into a *performance gap* between the two alternatives and the magnitude of this gap changes in various situations. Clearly, if enough semantics could be built into the OS transaction manager the performance gap would disappear . Additional semantics may be provided to varying degrees and we would obviously expect to see the performance gap shrink as the semantics becomes more complex.

Although a detailed treatment of the effect of semantics on the performance of the OS solution is beyond the scope of this paper, we have simulated a logical first step in this direction. In particular, the OS could provide a facility to distinguish data objects from indexes; thereby allowing it to hold only short-term locks in the index. Consequently, the OS transaction simulator was modified so that it would release index locks immediately upon finishing the processing of an index page. Experiments were then performed to see how this would affect the *performance gap* and the results of these experiments are discussed below.

First, the performance of the modified OS transaction simulator was compared against the DBMS system at a multiprogramming level of 20 with *numdisks* being set alternately to 2 and 8. This was done to examine the behavior in both an I/O-bound and a CPU-bound situation respectively. In the former case, the performance gap between the OS and the data manager decreased from 27% to 25%. In the latter case it fell from 55% to 48%. This shows that the improvement was more significant in the CPU-bound environment; however, a wide performance gap still remains.

Second, two situations of still higher contention in the indexes were considered to see how this would affect the performance gap. In both cases *MPL* and *numdisks* were set to 15 and 2 respectively. In one experiment *modify1* was changed to 37.5% from its default value of 25%. The *performance gap* reduced from 38% to 35%. Finally, in another experiment the number of indexes, *numindex* was set to 1. In this case the *performance gap* declined from 9% to 4%. Again, only a fraction of the performance gap is the result of inferior lock management on the part of the OS.

6. CONCLUSION

6.1. Implications for Feasibility

The performance of an OS transaction manager was compared with that of a conventional data manager in a variety of situations. With few exceptions, the data manager uniformly outperformed the OS transaction manager by more than 20%. 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 none of the above conditions hold, then the performance of the OS transaction manager is substantially worse than its DBMS counterpart, typically by 30%. Hence, this is the performance penalty a user of an OS transaction manager must be prepared to pay for the convenience of not having to write the tedious transaction management code.

6.2. Future Directions

It is evident from our experiments that in order to make the operating system solution really viable it is necessary to provide the OS transaction manager with increased semantic knowledge pertaining to the processing environment. Such semantics must enable the OS transaction manager to distinguish between data and index records so that more efficient locking can be performed. Additionally, a facility has to be provided for the user to define the structure of the index pages so that OS logging can be optimized. Even so, index records would still be logged and a performance gap would be anticipated. Future work is anticipated to quantify its magnitude. Another possibility is to assume hardware support for locking that is available to the OS but not the DBMS.

Lastly, yet another optimization is to modify the B-tree index and implement it using forward pointers to chain all the entries on a page [BLAS87]. An insert operation into such a structure would be performed by adding a new entry into an empty slot on the correct page and including it in the pointer chain. Similarly, a delete operation would require either modifying the chain to exclude the entry to be deleted or merely invalidating the entry. This would reduce the amount of data to be shifted every time such an operation takes place and, consequently, reduce the amount of data that has to be write-locked and logged by the system, albeit at a higher cost of maintaining the structure. We expect to consider these alternatives in detail in future work.

REFERENCES

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

[AGRA85b] Agrawal, R., and Dewitt, D., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", *ACM Trans. on Database Systems* 10(4), December 1985.

[BLAS87] Blasgen, M., (private communication)

[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. Tenth VLDB Conference*, Singapore, Sept. 1984.

[CHAN86] Chang, A., (private communication)

[DATE84] Date, C.J., *A Guide to DB2*, Addison-Wesley Publishing Co., 1986.

[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, Lecture Notes in Computer Science: 60*, Springer-Verlag, 1978.

[GRAY81] Gray, J. et. al., "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys* 13(2), June 1981.

[HAER83] Haerder, T. and Reuter, A., "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys* 15(4), December 1983.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems* 6(2), June 1981.

[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", *Comm. of ACM* 25(4), April 1982.

[MUEL83] Mueller, E. et. al., "A Nested Transaction Mechanism for LOCUS", *Proc. 9th Symposium on Operating System Principles*, October 1983.

[PU86] Pu, C. and Noe, J., *Design of Nested Transactions in Eden*, Technical Report 85-12-03, Dept. of Computer Science, Univ. of Washington, Seattle, Wash., Feb. 1986.

[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 Trans. on Database Systems* 9(4), Dec. 84.

[SPEC83] Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing", *Operating Systems Review* 17(2), April 1983.

[STON81] Stonebraker, M., "Operating System Support for Data Managers", *Comm. of ACM* 24(7), July 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* 16(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 for multiple copy databases", *ACM Trans. on Database Systems* 4(2), June 1979.