ALTERNATIVE STRATEGIES FOR A SEMANTICS-BASED OPERATING SYSTEM TRANSACTION MANAGER

by

Akhil Kumar and Michael Stonebraker

Memorandum No. UCB/ERL M87/19

7 April 1987

ALTERNATIVE STRATEGIES FOR A SEMANTICS-BASED OPERATING SYSTEM TRANSACTION MANAGER

by

Akhil Kumar and Michael Stonebraker

Memorandum No. UCB/ERL M87/19

7 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALTERNATIVE STRATEGIES FOR A SEMANTICS-BASED OPERATING SYSTEM TRANSACTION MANAGER

by

Akhil Kumar and Michael Stonebraker

Memorandum No. UCB/ERL M87/19

7 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALTERNATIVE STRATEGIES FOR A SEMANTICS-BASED OPERATING SYSTEM TRANSACTION MANAGER

Akhil Kumar and Michael Stonebraker

University of California
Berkeley, Ca., 94720

Abstract

Results of a previous comparison study [KUMA87] between a conventional transaction manager and an operating system (OS) transaction manager have indicated that the OS transaction manager incurs a severe performance penalty and appears to be feasible only in special circumstances. The present study considers two approaches for enhancing the OS transaction manager performance. The first strategy is to enhance OS performance by reducing the cost of lock acquisition and by compressing the log. The second strategy explores the possibility of still further improvements by using additional semantics. The results of this study show that the OS will have to implement essentially all of the specialized tactics for transaction management that are currently used by a database management system (DBMS) in order to match DBMS performance.

1. INTRODUCTION

In recent years there has been considerable debate concerning moving transaction management services into 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, such services could 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.

This research was sponsored by a grant from the IBM Corporation

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 must 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. In a previous study [KUMA87], we have quantified the effect of these factors on the performance of the OS transaction manager and have shown that the OS transaction manager performs substantially worse than the DBMS in most situations. This paper examines various approaches to overcoming this drawback and enhancing OS transaction manager performance. First, we consider the possibility of performance improvement by setting locks more cheaply and from compressing the log. These could be easily provided by some sort of hardware assist, and we will term these tactics as **hardware assistance.** Next, we simulate the effect of providing **semantic assistance** in the OS transaction manager, similar to the semantic features used by a DBMS transaction manager. Since the complexity of the OS will increase with increasing degrees of semantics, we consider introducing semantics in stages and determine how the performance of an OS transaction manager improves at each higher semantic level.

In section 2 we highlight the salient features of an OS transaction manager and compare it with one implemented within a DBMS. The basic simulation model, borrowed from the companion study [KUMA87], is summarized in section 3. Section 4 describes two performance improvements using hardware assistance and gives the results of simulation experiments to quantify their effect. Section 5 gives a framework for introducing alternative levels of semantics into an OS transaction manager and presents the results of additional experiments which show how the gap between the OS and the DBMS performance narrows as the OS becomes "smarter".

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 was pointed out that the conclusions of previous studies were contradictory and the differences were explained as resulting from differing assumptions that were made about the availability of resources. It was shown that dynamic locking works best in a situation of limited resources, while optimistic methods perform better in an infinite-resource environment. 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 these results 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 must be 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 will be 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 have a much smaller log

The data manager locks 2048 byte pages while the OS manager locks 100 byte subpages; hence, the DBMS transaction manager 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 entry 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 each index insertion. 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 managers 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 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; 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 assumed to be 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 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.

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:

[†]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.

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
modify1	average fraction of update actions in a transaction	25
modify2	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

Parameter	Range	Default Value
buf_size	250,.....,1000 pages	500
cpu_lock	200,.....2000 instructions	2000
modify1	5,....,50	25
MPL	5,.....,20	15
numdisks	2,.....,10	2
numindex	1,2,.....,5	5

Table 2: Range of variation of the parameters

$$\frac{(throughput_{DBMS} - throughput_{OS})\ X\ 100}{throughput_{OS}}$$

where
$throughput_{OS}$: throughput for the OS transaction simulator

$throughput_{DBMS}$: 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 all 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.

4. HARDWARE ASSISTANCE

4.1. Introduction

In an earlier study [KUMA87], we have compared the OS transaction manager against a DBMS system in a variety of situations. For instance we varied the multiprogramming level, transaction mix, conflict level, number of disks, buffer size, and the number of indexes. In the first of these experiments, the multiprogramming level was varied between 5 and 20 while the number of disks, *numdisks* was set at 2 and the cost of executing a lock-unlock pair, *cpu_lock* was 2000 instructions. All other parameters were set at their default values of Table 1. The throughput rates for various multiprogramming levels are given in Figure 1.

This figure shows that the *throughput* rises sharply when the multiprogramming level increases from 5 to 8 due to increases in disk and CPU resource utilization. The

	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

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. The results of the other experiments were similar and the details are in [KUMA87]. It was found that the DBMS consistently outperformed the OS transaction manager with the *performance gap* approximately 30%.

In an effort to reduce this gap we simulated the effect of two hardware-oriented possibilities: **log compression** and **cheaper locking.** These two factors were chosen because the previous study showed that they contributed significantly to the inferior performance of the OS transaction manager. Compressing the log would reduce the amount of data that must be written out to disk at transaction commit time; thus decreasing the I/O activity and raising throughput. The extent of improvement would depend upon the compression ratio, $comp_{ratio}$. Consequently, in section 4.2 we describe experiments to measure throughput at various values of $comp_{ratio}$. Section 4.3 turns to a study of the effect of a lower locking cost on the performance of the OS transaction manager.

4.2. Log Compression

As stated above, the OS transaction manager writes a much larger log than the DBMS and therefore suffers a performance setback. The rationale for log compression is that, typically, before and after images have bytes that are identical. Therefore, the after image can be differentially encoded against the before image and the identical bytes stored just once. The degree of compression depends, among other factors, on the efficiency of the data compression algorithm. A standard algorithm would simply suppress identical bytes in the before and after images. However, a smarter algorithm would also be able to detect situations where an after image is derived by inserting or deleting a field into a before image. This is a common occurrence in the case of insertions and deletions into an index. Such encoding will be beneficial to an OS transaction manager which has a very large log. Although a DBMS can also apply log compression, it would have much less impact because of the smaller log size. Consequently, we varied the compression factor, $comp_{ratio}$ by which the OS transaction manager compresses its log leaving the DBMS log at its full size.

The compression factor, $comp_{ratio}$ was increased in intervals from 0 to 0.8 in order to study how it affects the performance of the OS transaction manager relative to the DBMS. A $comp_{ratio}$ of 0 corresponds to no compression while a $comp_{ratio}$ of 0.8 means that the log can be reduced to 20% of its original size by compression techniques. Experiments were then conducted to measure the throughput rate of the OS transaction manager compared to that of the DBMS system for various values of $comp_{ratio}$ over this range. The multiprogramming level was set at 20, while all other parameters were set to their default values given in Table 1.

The results of these experiments are shown in Figure 2. The throughput of the OS transaction manager rises steadily as $comp_{ratio}$ increases. On the other hand, the throughput of the data manager stays constant because there is no change in its log size. The *performance gap* which is 27% when $comp_{ratio}$ is 0 decreases to 10% when $comp_{ratio}$ is 0.8. These experiments show that compressing the log assists the OS transaction manager

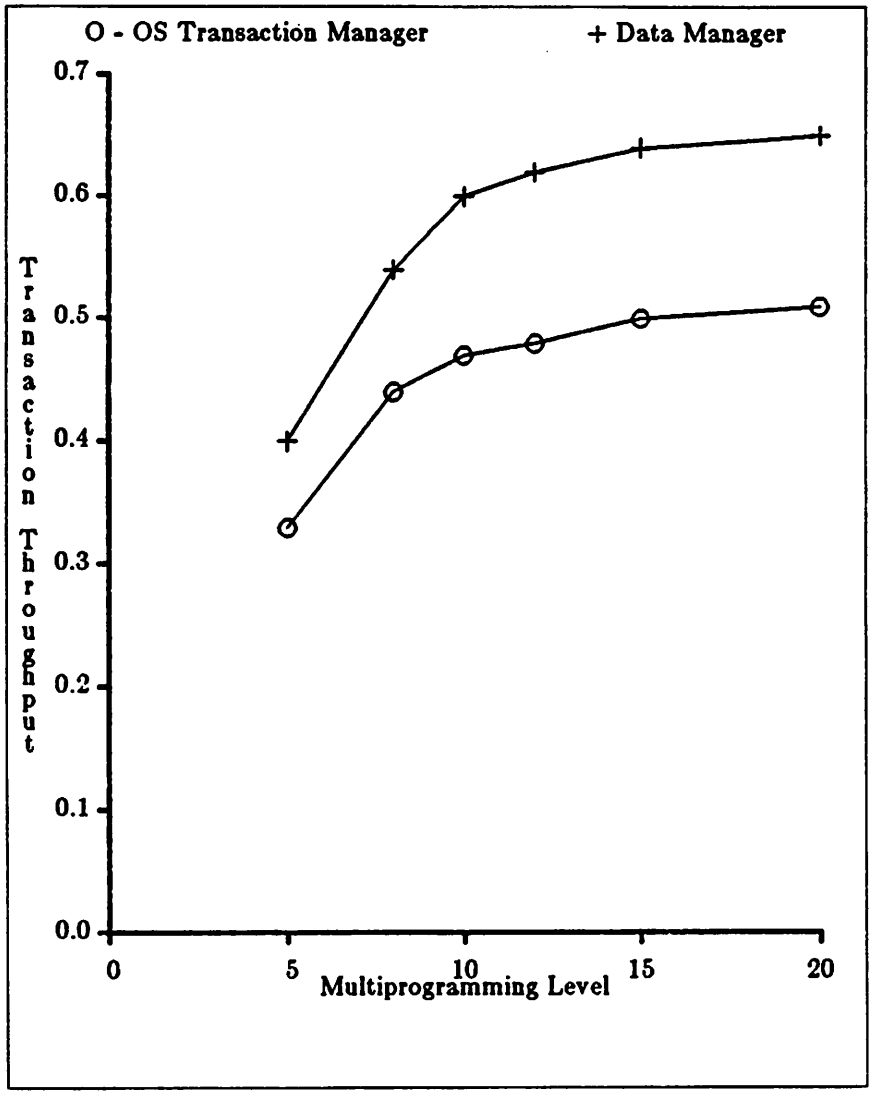

Figure 1: Throughput as a function of multiprogramming level

dramatically.

4.3. Cheaper Locking

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 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

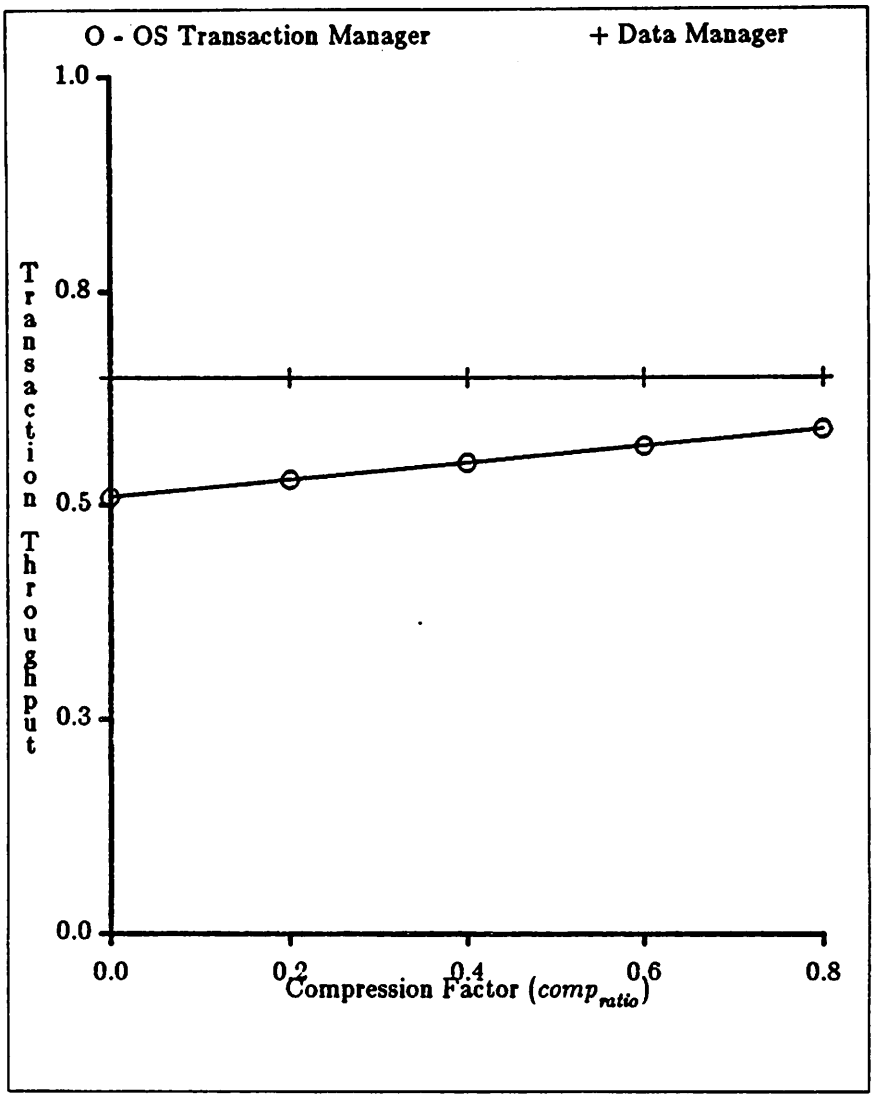

Figure 2: Throughput as a function of compression factor

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 3 shows the *throughput* 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 changes

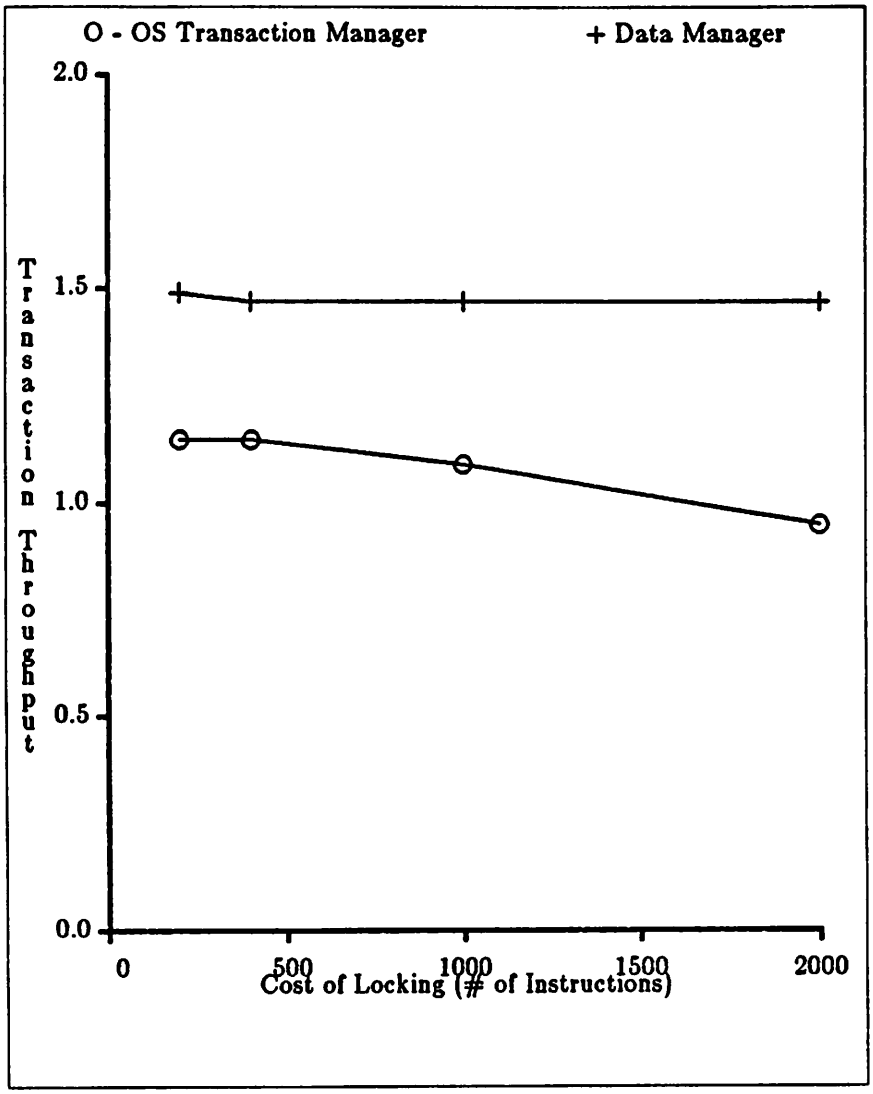

Figure 3: Effect of cost of locking on throughput

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.

4.4. Analysis

The above experiments demonstrate that noticeable improvements in the OS performance are possible from cheaper locking and log compression. An interesting observation is that both factors affect the data manager performance only slightly. Moreover, log compression decreases the burden on the I/O resources and is, therefore most effective in an I/O-bound environment while cheaper locking speeds up a CPU-bound system by reducing the CPU cost. This means that the benefits from the two performance improvements are greatest in different environments. Lastly, it should be pointed out that a processing cost should be associated with implementing a data compression algorithm. Because this cost has been neglected the results of experiments in section 4.2 are optimistic. In the next section we couple the advantages from these improvements with increased semantic knowledge by the OS in order to further improve OS performance.

5. ADDING SEMANTICS TO THE OS

5.1. Introduction

The experiments reported above show that inspite of the simulated tune-up, a performance gap remains between the OS transaction manager and the DBMS. Clearly, if enough semantic knowledge could be built into the OS transaction manager this 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 became more complex. In this section we examine semantic alternatives for further improving the OS performance. Consequently, a framework for introducing semantics into the OS in stages was first developed. Experiments were then conducted to evaluate the improvements that are attainable at each higher step on this semantic ladder. The framework is discussed in section 5.2 and the experiments are described in section 5.3.

Level	Description
0	locking and logging on 100 byte physical subpages
1	short-term index locking
2	short-term index locking + 40% log compression
3	short-term index locking + no index logging

Table 4: Alternative semantic levels for the OS transaction manager

5.2. Semantic Framework

Table 4 shows alternative schemes for providing semantic capabilities within the OS. Each proposal in this table is linked to a semantic level and the complexity of the semantics increases at higher levels. The OS transaction manager discussed in previous sections is placed at level 0 on this semantic scale. At level 1 an OS transaction manager would possess the additional capability to distinguish data from index pages, and perform short-term locking on index pages, thereby increasing the degree of parallelism. The performance of the OS alternative may be further improved by using data compression techniques to reduce the size of the log. Therefore, the semantics at level 2 is characterized by a combination of short-term index locking and data compression on the log. An estimated 40% compression factor has been used in our experiments. A further step beyond level 2 is to eliminate the need for index logging altogether in the OS solution. This would require the notion of "events" [STON85] in the OS transaction manager and is represented by level 3, the highest semantic level on our scale.

5.3. Experiments

The level 0 OS transaction manager of the previous section was suitably modified to simulate each of the other three levels. Then, using the above framework, experiments were conducted to study the *performance gap* between the DBMS and an OS transaction manager operating at the 4 different levels. The multiprogramming level was set at 20 in all the experiments and all other parameters were set to their default values of Table 1. Figure 4 is a plot of performance gap as a function of the semantic level for both I/O-bound (with 2 disks) and CPU-bound (with 8 disks) systems when the value of *cpu_lock* is 2000 instructions. Figure 5 is a similar plot with *cpu_lock* set at 200 instructions.

Figures 4 and 5 show that the performance gap drops at each higher semantic level in both environments and for both values of *cpu_lock*. However, this drop is sharper at levels 2 and 3. In the case when *cpu_lock* is 2000 instructions and the system is I/O-bound, the *performance gap* of the level 3 OS transaction manager is down to 7% (from 27% for the level 0 OS transaction manager). The corresponding value for a CPU-bound system is 35% (down from 55% for the level 0 OS transaction manager). This means that the performance improvement from writing a smaller log is greater in the I/O-bound system than in the cpu-bound one. On the other hand, when *cpu_lock* is reduced to 200 instructions, the *performance gap* of the level 3 OS transaction manager is down to 7% in both the cpu-bound and I/O-bound environments. Thus lowering *cpu_lock* results in a large improvement in the performance of a cpu-bound system but has no effect on an I/O-bound system.

These experiments illustrate that a combination of additional semantics and lower locking cost are necessary to make the OS proposal viable in both I/O-bound and cpu-bound systems. However, in the absence of a lower locking cost, it appears to be viable only in an I/O-bound environment.

6. CONCLUSION

A previous study has shown that the performance of an operating system transaction manager is substantially worse than its DBMS counterpart, typically by 30%. The objective of this study was to investigate the effect of performance tune-ups and semantics-based approaches to reduce this gap. Consequently, several new experiments were devised and their results were reported in sections 4 and 5. The experiments of section 5 show

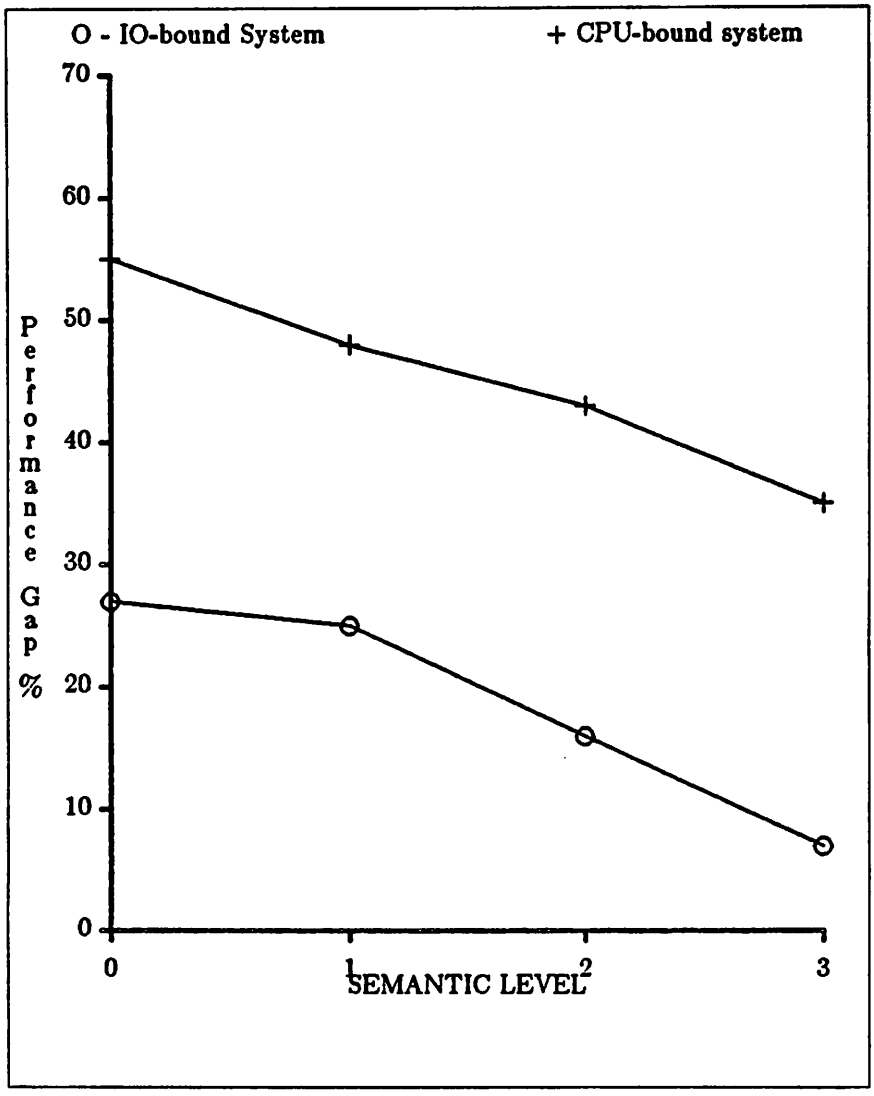

Figure 4: Performance gap at different semantic levels ($cpu\_lock = 2000$)

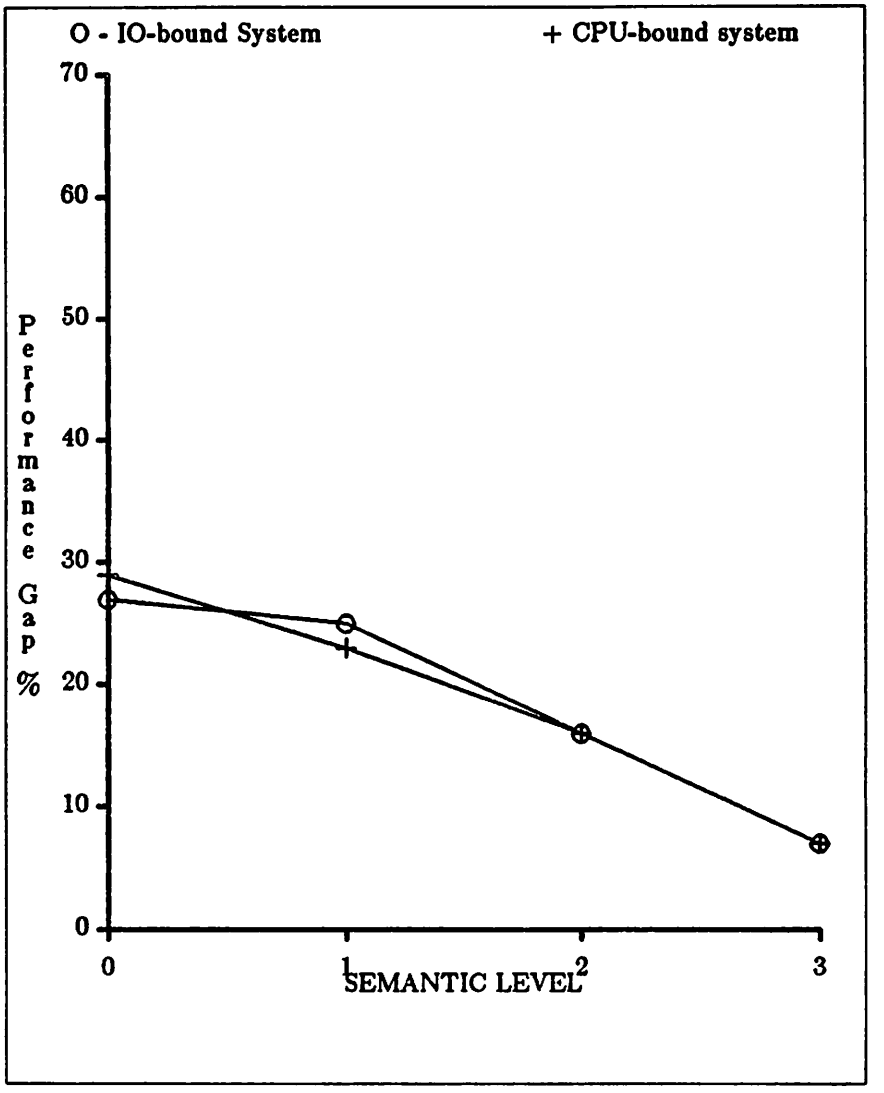

Figure 5: Performance gap at different semantic levels ($cpu\_lock = 200$)

that a combination of level 3 semantics and cheap locking are necessary to offset the OS disadvantages and bring its throughput close to that of the DBMS. The major conclusion of this study therefore, is that an operating system based transaction manager will have to implement most of the specialized features of a DBMS transaction manager in order to perform at par with the DBMS system.

Finally, it should be clearly noted that some assumptions of our model lead to estimates of *performance gap* which are too small. First, as stated earlier, the contention in the root and the second level index pages of the B-tree was ignored. This assumption

tends to favor the OS alternative because in the absence of short-term locks, conflict in the higher level index pages would tend to slow it down further. Second, the read and write-sets of each transaction were drawn from a uniform distribution. This means that the likelihood of conflict was constant over the entire database. A different distribution would divide the database into certain areas of high conflict and others of low conflict. Since, as shown in [KUMA87], the relative performance of the OS deteriorates in a higher conflict situation, a non-uniform assumption would tend to widen the *performance gap*. Furthermore, the CPU cost of compressing the log was ignored in section 4.2. Lastly, the assumption that the log can be easily compressed is an optimistic one. Hence, all of these assumptions tend to bias the results in favor of the OS alternative, and one would expect peformance gaps in real systems higher than the ones reported here.

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 1985.

[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.

[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, 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.

[KUMA87] Kumar, A., and Stonebraker, M., "Performance Evaluation of an Operating System Transaction Manager", ERL Memo M87/15, University of California, Berkeley, March 1987.

[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.

[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 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, 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, 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.