National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

LEAF LOCKING DATABASE CONCURRENCY CONTROL

by

Thomas Walter Steiner

B.Sc. (Hons. Physics) University of Waterloo 1981

PhD. (Physics) Simon Fraser University 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1994

ISBN 0-612-06821-8

APPROVAL

Name: Thomas Walter Steiner

Degree: Master of Science

Title of thesis: Leaf Locking Database Concurrency Control

Examining Committee: Dr. Bob Hadley
Chair

Dr. Tiko Kameda
Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Wo-Shun Luk
Professor, Computing Science
Simon Fraser University

Dr. Stella Atkins
Assoc. Professor, Computing Science
Simon Fraser University
External Examiner

Date Approved: December 7, 1994

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENSE

Title of Thesis/Project/Extended Essay

Leaf Locking Database Concurrency Control.

Author: _____

(signature)

Thomas Walter Steiner

(name)

December 14, 1994

(date)

Abstract

A new database concurrency control mechanism based on locking the leaves of a binary tree is proposed. This is a modification of tree locking with all the data items in leaf nodes and the interior nodes of the tree used only for concurrency control. It is shown that this technique has greater possible concurrency than ordinary tree locking. Furthermore, concurrency is greater than that exhibited by the two-phase locking technique. Simulation results on the least favourable workload indicate that leaf locking results in a 30% increase in transaction throughput as compared to two-phase locking.

Acknowledgements

I would like to thank Tiko Kameda for providing me with the opportunity to do this work and pointing me in the direction of reference material.

Contents

List of Figures

Chapter 1

Introduction

A database server must process a large number of transactions. During the servicing of a transaction the processor must often wait for disk I/O or else wait for a response from a client. It is desirable to use this idle time of the processor to handle another transaction concurrently in order to increase database throughput and decrease response time. This however cannot be done indiscriminately. A database transaction scheduler must schedule incoming transactions so that the database is not left in an inconsistent state after transaction execution. Consistency is guaranteed provided the schedule generated is equivalent to a serial schedule, i.e., one where the transactions are carried out sequentially. This is called serializability [1, 2] and forms the cornerstone of concurrency control.

One way to ensure that the resulting schedule is serializable is to lock data items in a manner that ensures serial equivalence. All data items accessed by a transaction must first be locked in accordance with a locking policy. The dominant such method is called two-phase locking (TPL) [1, 2, 3]. While a data item is locked no other transaction can lock the same item in a conflicting mode. Another transaction wishing to access the same data item in a conflicting manner must wait until the lock is released. The name two-phase refers to the requirement that no locks can be released by a transaction before the last one is acquired. Thus, during the growing phase locks are acquired and then during the shrinking phase locks are released. The two-phase locking policy guarantees serializability since transactions are effectively serialized in

the order in which they reach the lock point (the end of the growth phase).

Non-locking concurrency control schemes are also possible. The simplest one is time-stamp ordering (TSO) [2, 3]. In this scheme database reads and writes proceed at will but the time stamp of an operation is compared to read and write stamps stored with the data item. If, for example, a transaction tries to read a data item last written by a younger transaction it will have to be aborted. TSO relies on aborting transactions that would otherwise cause the resulting schedule to be non-serializable. An aborted transaction then restarts after some time delay.

It has been shown both theoretically [4] and with simulations [8] that TPL has the greatest throughput for unstructured data. Recent results indicate that within TPL the performance under high data contention can be improved by limiting the length of the chain of transactions waiting for a data item [5, 6]. Alternatively, one can consider structuring the data in order to improve performance. One example of a concurrency control scheme which uses structured data is tree locking (TL) [7, 2, 3]. For TL the data items are structured in a binary tree. In this scheme a transaction requiring a number of locks starts at the common ancestor node of all the required data items and propagates locks downwards using lock coupling. Lock coupling means that in order to get a lock on a child node a lock must be held on the parent. Furthermore, the lock on the parent may not be released until the desired lock on the child node has been acquired. This ensures that lock requests on common data items from a transaction initiated later cannot overtake the current transaction's requests anywhere in the tree. Such a sequence of events is thus serializable. The transactions are serialized in the order in which they were initiated (strictly true only if all lock requests start at the root node). A major disadvantage of this method is that when locks are held on several data items, one of which is at a node in the tree close to the root, a large part of the tree below this node is inaccessible to other transactions and thus a loss of concurrency results. TL is not optimal and an alternative structured locking policy will be proposed in the next chapters.

Chapter 2

Leaf Locking

2.1 Leaf Locking Policy

In this thesis a leaf locking (LL) scheme will be proposed and compared to TL, TPL and TSO. LL is a modification of TL with all the data items put in leaf nodes. The interior nodes of the tree are used only for concurrency control. This means that the LL tree (if balanced) will contain twice as many nodes and be one level deeper with the concomitant increased storage demands. The advantage is that now the interior nodes need not be locked for long periods of time. They need only be locked long enough to propagate a lock to the next level down. Thus the problem of unnecessarily locking up parts of a tree can be circumvented at the price of increased storage. That this method has increased concurrency as compared to standard TL can be seen from figure 2.1. Transaction T_1 accesses both data items a and d while T_2 accesses elements e and f. Using TL T_2 must wait until T_1 releases its lock on a since in order for T_2 to acquire locks on e and f it must first acquire a lock on the least common ancestor which is a. Using LL however, T_2 can proceed almost immediately even if all transactions are forced to start at the root node since in that case it need only wait until T_1 has propagated its lock one level down.

LL is thus clearly better than TL if the additional storage is available. The extra storage required is actually quite modest since the interior nodes do not store any data and are consequently very small. In order for the extra lock coupling overhead to be

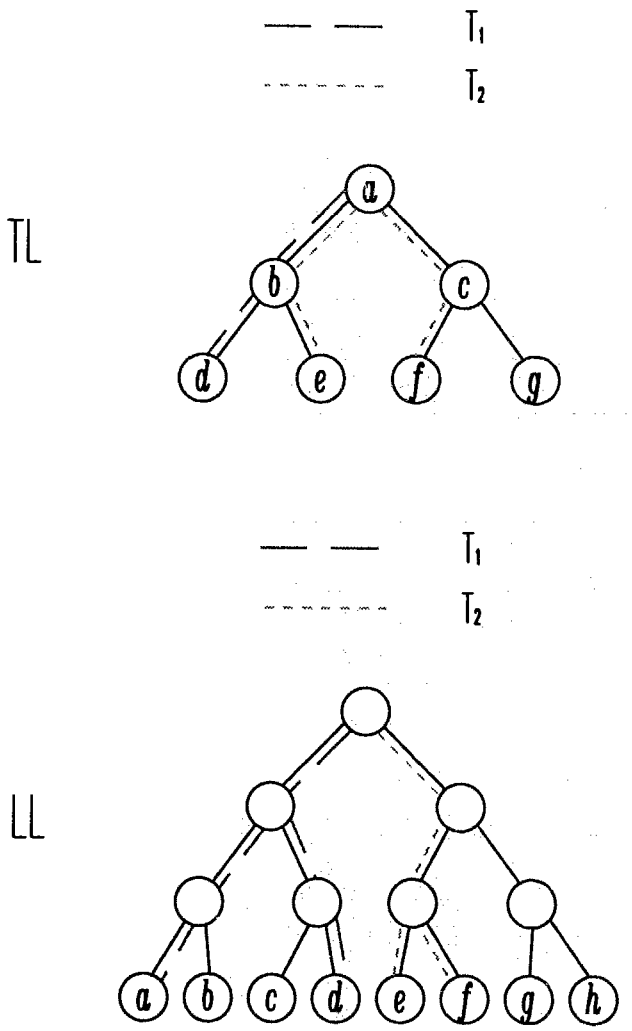

Figure 2.1: Treelocking and Leaflocking

negligible as many of the interior nodes as possible should reside in main memory. In the LL scheme outlined above it is still possible for part of the tree to become unnecessarily inaccessible if multiple transactions attempt to lock the same data item. Since the lock on the parent cannot be released until the child lock is held, waiting for a previous transaction to finish with a data item causes part of the tree to be unavailable. Consider the case of both transactions wanting to write data item b in the LL case of figure 2.1. The second transaction would have to wait for the first to finish and during this time must hold a lock on b's parent. This has the side effect of causing data item a to be effectively unavailable to other transactions requiring a along with some other data item. To overcome this problem each data item (the leaf nodes) should have a lock queue. The lock coupling rule can then be relaxed to the requirement that the lock on a parent of a leaf node can be released once all the lock requests on child data items have been queued. Since the queue is FIFO the order in which transactions access a data item is unchanged and the continued serializability of all schedules is guaranteed. These queues can be either dynamically allocated as needed or be of fixed length with a natural overflow handling mechanism.

A last optional requirement of LL is the insistence that all transactions start at the root node. This optional requirement has some advantages with only a slight penalty (at least for centralized databases). For TL such a requirement would drastically reduce concurrency since locking the root data item for a transaction would then effectively lock the whole tree. For LL the root node does not contain data and is thus never held for long. The increased lock coupling overhead is small and the effect on performance is minimal. If this rule is enforced all transactions are strictly serialized in the order of their starting times. Furthermore, locking granularity can then be naturally achieved. For example, as long as the tree root is locked all data items are effectively locked. If the tree is organized so that related data items are all grouped together in the same sub-tree, then all these items can be simultaneously locked by locking the sub-tree root node. However, for very large databases this rule might be undesirable at the top level due to "bottle-necking".

For TL, organization of the data in the tree can have a dramatic effect on performance. If most transactions require access to a node close to the root, then large

numbers of data items are unavailable during the time a lock is held on such a node. Thus, heavily used items should be placed in leaf nodes. For LL there are no such concerns since all the data items are in leaf nodes. No special tree organization is required from a performance point of view. It might still be desirable to group related items together in the same branch so that they can all be locked together for special purposes such as re-balancing the tree after inserting new items.

2.2 Lock Propagation Algorithm

In order to implement LL a lock propagation algorithm is needed. The starting point is a predeclared read and write set of data items and a database of data items organized in a binary tree with the actual data items in leaf nodes. We require that locks be propagated down to the appropriate leaf nodes using lock coupling starting from the root of the binary tree. It is important to minimize the time any interior node is locked in order to make available the maximum amount of concurrency. It is thus undesirable to propagate the locks one at a time since then the root node would have to be locked for the duration in order to prevent lock requests from other transactions from getting in between. Fortunately it is possible to proceed in a breadth first manner in which all lock requests collectively move down one level at a time locking the required children and then unlocking the parents once all requests have moved down one level.

In the implementation used here the lock requests corresponding to a transaction's data set are structures chained together in a singly linked list. The request structure contains a pointer to its right neighbour as well as a leaf pointer which at the end of the algorithm will point to the leaf node containing the required data item. Initially the leaf pointer of the request at the head of the list points to the root of the database tree and all other leaf pointers are null. Also note that the requests are ordered by data-item key value so that the request at the head of the list has the lowest key value and the one at the end the largest. This key must be the same as was used to build the binary database tree. The algorithm starts by locking the root node and setting the leaf pointer of the first request to either the left or right child node of the root depending on whether the request key value is less than or greater than the root key

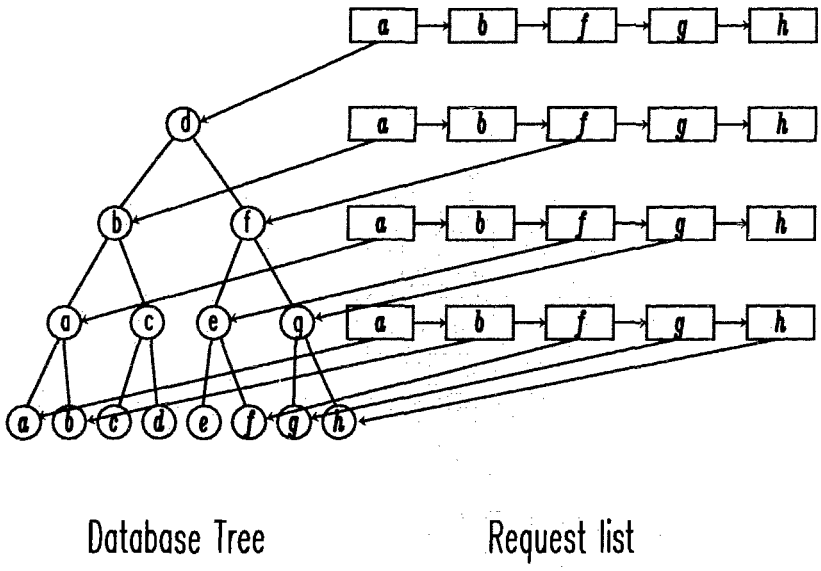

Figure 2.2: Lock Propagation Algorithm
The labels on the interior nodes are the key values used for comparisons.

value (figure 2.2). If the right branch is chosen then all other requests also follow the right branch since the list is ordered by key value. Hence the right child node is locked and the root node released. If the left branch is followed then the head request leaf pointer is set to point to the left child and the left child is locked. For the left branch the request list must be traversed from left to right comparing the request key value with the root key value. If a request key in this list to the right of the head request is greater than the root key its pointer is set to point to the right child and the right child is locked as well. In this case list traversal can be terminated and the root node unlocked. If the end of the list is found without any key being greater than the root key then all requests follow the left branch and the root is unlocked without locking the right child. At this point all lock requests have moved down one level in the tree, are holding only the required locks and the root is once again unlocked and ready for the next set of requests. The algorithm proceeds down level by level in this fashion with each subset of requests starting with a non-null leaf pointer treated separately.

Note that in order to move down one level the request list need, at most, be

traversed once in this algorithm. Thus if there are k requests and N data items in the tree ($\log N$ levels) this algorithm will be of order $O(k \log N)$. Since k is in general very small and $\log N$ is also small for all but the largest databases this lock propagation algorithm is very efficient. In the case of TPL and TSO the data items can in principle be found using a hash table with efficiency $O(k)$ which is clearly more efficient. However, provided the LL concurrency control nodes can be kept in main memory which should be possible for all but the largest databases, the lock propagation overhead is expected to be minimal since transactions will in general have to wait much longer for resource or data contention.

Another aspect which could cause concern is bottle necking at the root node since all transactions must access the root node. The key to high performance here is keeping the root node locked for the minimum amount of time. It can be shown that at very high multi-programming levels (mpl) the throughput is limited to $1/k$ where k is the fraction of the average transaction time spent accessing the root [10]. As stated above the transaction is expected to spend the bulk of its time waiting for resource or data contention so k will be a very small fraction. Consequently, the expected maximum mpl allowed by root node contention will be very large at least in a centralized system. Root node contention would be a much more serious issue in a distributed LL system where communications delays will add considerably to the time necessary to lock and unlock a node. Here we will restrict our attention to a centralized LL system. It is not strictly necessary for every transaction to lock the root node for a version of LL to work. It is only required that the highest common ancestor of all the requested leaves be locked. In a large database that point will often be several levels down in the tree. In the context of the above lock propagation algorithm it means that a lock need not be set until the request list must be split in two in order to follow both the left and right branches. With this modification the transactions are no longer guaranteed to execute in time-stamp order.

2.3 Essentials of LL

In section 2.1 LL was presented as inspired by the genesis of the idea. LL was developed as an evolution of TL in order to try and improve concurrency. In order to compare LL to other concurrency control schemes presented in the literature it is useful to now distill the essential elements of the LL scheme.

1. All locks a transaction may need are acquired or queued atomically at the start of a transaction. In other words transactions must set locks in an equivalent to serial order. It must not be possible for a second transaction to set locks on common data items until the first transaction has finished acquiring locks.

2. All data items must have FIFO lock queues.

The first property insures that there are no deadlocks and that transactions subsequently execute in the same serial order. Choosing time-stamp ordering for the serial order insures fair scheduling. The second property insures that conflicting accesses to a data item are serialized in an order that can be established well before transaction execution. The first and second properties together ensure that the serial orders at each data item are consistent with each other thereby allow the lock setting algorithm to terminate and transactions to start while some locks are still held by other transactions.

Using a tree to couple locks from the root to all the data items satisfies property 1 as well as imposing time-stamp ordering. A tree is however not strictly required as property 1 can be satisfied with other mechanisms. For example making a single process responsible for setting all locks would also satisfy property 1. In this scheme all transactions would submit their data sets to the scheduling process which would execute these serially in the order received. Transactions must delay starting execution until receiving a locks set/queued return message from the scheduler. Alternatively instead of having a scheduling process each transaction could be required to get ownership of a shared semaphore before setting any locks to ensure locks are acquired in a serial order.

Both of the above schemes, while insuring property 1, are inferior to the binary tree in terms of throughput due to contention for the serial scheduling process or semaphore. As discussed in section 2.2 at high mpl the throughput is $1/k$ where now k is the fraction of the total transaction time spent waiting for the locks to be set and acknowledged. k is much smaller with a tree since there k is the fraction of time the root node is locked. The root node is only held long enough to couple to the next level down which is a much smaller interval than the time required to set all the locks.

If the data items are structured it is possible to use lock coupling to ensure property 1. For example if all data items were linked together in a linked list then locks could be acquired by starting at the first item and traversing the list to all the required items. In this way a second transaction could start setting locks as soon as the first had moved on to the second data item. As long as transactions cannot pass each other in the lock setting process property 1 will be satisfied. Structuring the data allows one to satisfy property 1 while minimizing k and hence maximizing throughput. Structuring the data allows the acquisition of locks to be pipelined with many transactions propagating locks concurrently.

Structuring the data items as a tree, while not increasing throughput as compared to a linear list, minimizes the number of lock couples and hence reduces the total time required to set all locks. This reduces a transaction's response time. A tree also has some other advantages such as a built-in efficient index on the primary key and the possibility of locking whole sub-trees without locking the entire data structure. This then is the primary motivation for retaining the tree structure for insuring property 1.

2.4 Concurrency

While LL is clearly superior to TL it is more interesting to compare LL to TPL. In order to investigate which method has greater concurrency some concrete example transactions will be used. Consider the following two transactions:

$$\begin{aligned} T_1 &= L_W[s]W[s]L_W[v]W[v]L_W[x]W[x]U[s,v,x] \\ T_2 &= \qquad\qquad\qquad\qquad\qquad\qquad L_R[s]R[s]L_R[z]R[z]U[s,z] \end{aligned} \tag{2.1}$$

Here $L_W[x]$, $L_R[x]$ and $U[x]$ respectively stand for write locking, read locking and unlocking data item x. $R[x]$ and $W[x]$ represent reads and writes of data item x. Time increases linearly to the right and two transactions are concurrent if they can execute during the same time interval. Using TPL the above transactions cannot overlap until T_1 releases the lock on s at the end of T_1 thus there can be no significant overlap. However, with LL, the equivalent transactions

$$\begin{aligned} T_{1'} &= L_W[s,v,x]W[s]U[s]W[v]U[v]W[x]U[x] \\ T_{2'} &= \qquad\qquad\qquad\qquad L_R[s,x]R[s]U[s]R[z]U[z] \end{aligned} \tag{2.2}$$

can execute their reads and writes concurrently. Are there some schedules which can execute concurrently using TPL but not using LL? Consider the following schedule:

$$\begin{aligned} T_3 &= L_W[x]W[x]L_W[y]W[y]L_W[z]W[z]U[x,y,z] \\ T_4 &= L_R[z]R[z]U[z] \end{aligned} \tag{2.3}$$

These can execute concurrently using TPL as illustrated above. But using LL the equivalent schedule

$$\begin{aligned} T_{3'} &= L_W[x,y,z]W[x]U[x]W[y]U[y]W[z]U[z] \\ T_{4'} &= \qquad\qquad\qquad\qquad\qquad\qquad L_R[z]R[z]U[z] \end{aligned} \tag{2.4}$$

cannot execute concurrently.

The above example illustrates an important difference between TPL and LL. In TPL locks are acquired as needed but must be held until the lock point (end of the growth phase). For LL all locks are effectively acquired (or queued) atomically at the start of a transaction but can be released immediately after the corresponding data item is used. Thus in the first example the two transactions were able to execute concurrently using LL, but not using TPL, because the LL method was able to release

the lock on data item s much earlier. In the second example the two transactions can run concurrently using TPL because the acquisition of the lock on data item z could be delayed until just before writing z, whereas the LL method had to acquire a lock on z at the very beginning.

Thus, at first glance, there does not seem to be a clear concurrency advantage of one scheme over the other. However, the presence of the lock queues in the LL scheme provides an additional concurrency advantage as well as preventing the unnecessary branch lock up discussed earlier. Consider a modification of transaction T_4:

$$\begin{aligned} T_3 &= L_W[x]W[x]L_W[y]W[y]L_W[z]W[z]U[x,y,z] \\ T_5 &= L_R[s]R[s]L_R[z]R[z]U[s,z] \end{aligned} \tag{2.5}$$

These can execute concurrently using TPL as before. What about the equivalent LL transactions?

$$\begin{aligned} T_{3'} &= L_W[x,y,z]W[x]U[x]W[y]U[y]W[z]U[z] \\ T_{5'} &= L_R[s,z]R[s]U[s]R[z]U[z] \end{aligned} \tag{2.6}$$

These can execute concurrently under LL as well since $T_{3'}$'s write-lock request on data item z, which at the start of transaction $T_{3'}$ is locked by $T_{5'}$, can be queued in data item z's lock queue. Thus $T_{3'}$ can start and as long as $T_{3'}$ does not actually try to access data item z until after $T_{5'}$ releases its lock on z they can run concurrently. Of course if $T_{3'}$ did try to access z before $T_{5'}$ released its lock $T_{3'}$ would have to wait until its queued lock request was converted to an actual lock.

The crucial difference between $T_{3'}$, $T_{4'}$ and $T_{3'}$, $T_{5'}$ is the starting order of the two transactions. Using LL transactions are strictly serialized in the order of their starting times. Two transactions accessing common data items acquire locks on all common data items in the same order as their respective starting orders. A nice side effect of this is that LL (and TL) unlike TPL, is deadlock free. The addition of lock queues in LL increases the number of possible transactions which can run concurrently.

It thus appears that LL has a concurrency advantage over TPL. However, an estimate of the magnitude of this advantage, if any, requires a simulation which is the topic of Chapter 4.

2.5 Comparison with other Methods

Concurrency control algorithms can be roughly divided into two types aggressive and conservative schedulers [2]. Conservative schedulers attempt to avoid serialization problems by delaying (blocking) transactions while aggressive schedulers don't delay transactions much, relying instead on restarts when problems appear down the line. There are often conservative and aggressive versions of the same algorithm. For example, conservative TPL requires that all locks be held before a transaction begins. This ensures that deadlocks will be avoided and hence a transaction need never be restarted. The price for this is reduced concurrency since locks are held longer and the possibility of starvation of a transaction requiring a large number of data items [2]. Ordinary TPL is somewhere in the middle requiring restarts only on deadlocks. TSO on the other hand is very aggressive never delaying a transaction at all, but consequently suffers the penalty of many restarts when data contention increases. In general as an algorithm becomes more conservative the number of restarts decreases but also the concurrency decreases. Conservative schedulers also require that the read and write sets be predeclared so that a decision on serializability can be made in advance.

LL is unique since it is ultimately conservative. Read and write sets must be predeclared and restarts are never required for concurrency control reasons. Nevertheless, from section 2.1 LL appears to have greater concurrency than TPL the current dominant concurrency control scheme. This makes LL a very promising candidate from performance considerations.

LL also compares very favourably to TSO. As for LL, the TSO scheduler attempts to execute transactions in time-stamp order. The difference is that TSO must abort a transaction if a serialization problem appears. Thus, if there is heavy contention for data items many transactions will need to be aborted, often more than once. This means that a lot of useless work is being done. By useless it is meant read and write operations that will eventually be undone by aborting the transaction. While doing useless work is not a problem in an infinite resource system, it rapidly hinders performance in more realistic systems [8]. LL on the other hand never aborts a

transaction for concurrency control reasons and thus does no useless work.

A variation of TSO is conservative TSO [2, 9]. Here the scheduler waits and queues requests until it has received an operation request from every transaction manager before issuing the oldest request. It is required that each transaction manager send requests to the scheduler in serial order. In that case the scheduler will output operations in time-stamp order and no transaction need ever be aborted. At first glance this thus appears equivalent to LL. However conservative TSO has only one queue (at the scheduler) rather than queues at each data item and this forces it to output schedules that execute all operations in serial order not just conflicting ones. This is a consequence of not satisfying property 2 of section 2.3. This means that conservative TSO is much more restrictive than LL which only forces conflicting operations to execute in time-stamp order. LL thus has much greater possible concurrency.

LL also has some similarities to conservative TPL. In both schemes all locks must be acquired at the start of transaction execution. Transaction execution and the release of locks in LL appears to be equivalent to the transaction execution and lock release of the shrinking phase of conservative TPL. Furthermore, both LL and conservative TPL are deadlock free and never abort a transaction. The difference lies entirely in the lock acquisition phase before the start of execution. In conservative TPL [2] a transaction tries to obtain all required locks at the start. If there is a conflicting lock held then all the locks must be released and the transaction must wait until the conflicting transaction terminates and then tries again to acquire all the locks. This means a transaction can not start execution until it actually holds all required locks. LL on the other hand can often start execution as soon as all lock requests are queued and the locking/queuing algorithm never fails. For conservative TPL an equivalent queuing strategy cannot be used as queueing requests is not sufficient to guarantee a serial order of all conflicting database operations. In other words the greedy locking algorithm used by conservative TPL does not satisfy property 1 of section 2.3. The inability to start transactions as early as LL results in greatly reduced concurrency without any offsetting benefit. Instead conservative TPL still suffers from possible transaction starvation which is also not possible in LL.

2.6 Advantages and Disadvantages of LL

For LL it is necessary for a transaction to know in advance all the data items it will need so that locks can be propagated down to the corresponding leaf nodes when the transaction starts. If a data item might be needed in a transaction a lock must be propagated down to the corresponding leaf. This must be considered a disadvantage of LL as compared to TPL. As discussed in section 2.1 a previous transaction may still be holding a lock required by the current transaction, but the current transaction can queue its lock request and start executing provided it does not immediately require this data item.

LL naturally allows database check pointing unobtrusively and simply. Since transactions are serialized in their starting order and can start without holding firm locks on all data items, a checkpoint can be done by simply issuing a transaction that reads every data item. The resulting checkpoint is guaranteed to be of a consistent state since the checkpoint read is effectively inserted between two transactions. Data items locked by other transactions at the time of the checkpoint read will simply have read lock requests queued. Thus, checkpoints can be easily done without interfering with normal transaction processing. In contrast, a consistent read of all data items using TPL requires that all the data items be locked simultaneously.

For LL the data items must be organized as the leaves of a binary tree. It thus makes sense to order the items according to some primary key value so that data items can be efficiently found and locks propagated to them. The concurrency control tree then also provides a permanent binary tree index on the primary key. Addition and deletion of data items can be accomplished in a straightforward and inexpensive manner by adding and deleting elements from the tree. A tree built in such a haphazard manner is unlikely to be a binary tree of minimum height but this is not necessary for correct operation of the LL scheme. An unbalanced tree will just cause some slight increase in lock propagation overhead. The tree could be re-balanced periodically during times of light transaction loads.

While an index on the primary key naturally exist in LL secondary indicies require additional processing overhead. In TPL for example, multiple indicies may exist with

each index pointing directly at data items which can be locked at will if available. For LL, secondary indicies may point to data items but these items cannot be locked directly since coupling from the root of the primary tree is required for concurrency control. Thus using a secondary index is a two step process. First, the secondary index is used to obtain the primary keys required and then the primary index or tree is used to set the locks on the desired data items. This additional overhead is not a serious problem provided the LL tree (excluding leaf nodes) is in memory and the database system is I/O bound. Alternatively a hybrid of conservative TPL and LL could be used. In this scheme a transaction first attemps to get all its locks immediately without using lock coupling as in conservative TPL. In this case data items found using secondary indices can be immediately locked. If all data items are available (i.e., locks obtained not queued) then the transaction can start. If not, the transaction releases all locks and uses LL to queue or obtain locks on all data items whose primary keys are now known. With this scheme the optional LL requirement of all transactions starting at the root node has been given up with the corresponding consequences discussed earlier. This hybrid scheme produces serializable schedules since if all locks are immediately obtained then there is no conflict with LL set locks and both LL and conservative TPL produce serializable schedules. This hybrid does thus not have the extra lock propagation overhead in low data contention situations (conservative TPL lock setting almost always succeeds) but gracefully switches to LL and its corresponding advantages when the data contention increases. However in high data contention situations this hybrid has more overhead than pure LL since it always attempts to directly set all the locks which almost always fails.

On the negative side LL releases its locks much earlier than TPL and hence cascading aborts [2, 3] are, comparatively, more likely to occur. This is because there is a greater chance that a later transaction will have read a value written by the current transaction at the time that the current transaction is aborted. Fortunately, LL never aborts transactions for concurrency control reasons and hence abortions are relatively rare. Consequently, relatively expensive abortion processing can be tolerated. If a transaction is aborted, all later transactions must be tested for intersection of their read sets with the aborted transaction's write set. If a non-empty intersection is found

the corresponding later transaction must also be aborted. The data set intersection test must be done recursively until no additional cascading aborts are found.

A strict [2] implementation of LL, i.e., one that avoids cascading aborts, can be achieved by holding all write locks until after a transaction has been committed. This would reduce LL's concurrency comparatively much more than a similar strict implementation of TPL since LL achieves most of its concurrency by being able to release its locks early.

It appears that the LL concurrency control scheme is a good candidate for real databases since it is deadlock free, starvation free, restart free, and has high concurrency. The disadvantages are lock propagation overhead, slightly larger storage requirements, required predeclaration of the data set and greater propensity for cascading aborts. In the following chapters LL, TPL and TSO will all be implemented and a simulation study conducted in order to estimate their relative performance.

Chapter 3

Implementation

Using a multi-threaded operating system (OS/2) [11] modelling transactions is very straightforward. Each active transaction runs as a separate thread independently from all the other transactions. It first reads a sequence of reads, writes and delays that constitutes the simulated transaction from a file and then builds an ordered linked list of lock requests. The lock propagation algorithm of Chapter 2 is then run in order to acquire or queue all the necessary locks. Each individual node in the tree contains a mutex semaphore [11] and a lock on an interior node is held when a thread has acquired ownership of that particular semaphore. Unlike the interior node, leaf node locks are not held simply by semaphore ownership because this would preclude simultaneous read locks by several transactions on one data item. Leaf node lock status is determined by comparing the thread id stored in the leaf node's lock structures with the thread id of the thread requesting a read or write. The leaf node's mutex semaphore is used only to control access to the leaf node's shared variables such as the held and queued lock lists and is never held for long. Once all locks have been acquired or queued the thread proceeds to read and write data values. Between each read or write request there is a random delay corresponding to the time needed to retrieve the item from a disk in a real database. For the duration of the delay the thread sleeps and relinquishes the remainder of its time slice allowing other transaction threads to run, thereby interleaving many transactions. The number of simultaneously active threads is equivalent to the multi-program level (mpl). Active

here means that a transaction has been started but there are still some operations outstanding. An active transaction may be sleeping waiting for simulated I/O. During a read or write request the leaf node pointed to by the request structure is examined to see if the current thread in fact holds a lock. If so, the read or write proceeds normally and the thread continues on to the next delay. If the current thread's lock request is only queued rather than held, the thread sets an event semaphore [11] and blocks until the lock request is granted. When a transaction has written a particular data item the lock is downgraded to a read lock. The lock is immediately released once a transaction is done reading and writing a data item.

Each leaf node in the database tree has associated with it two lock pointers. One points to a list of currently held locks, which if there are more than one, must all be read locks. The other points to a list of queued locks. If the held lock(s) are read locks the first queued lock, if any, must be a write lock, because a read lock could just be added to the held locks. If the held lock is a write there can only be one and the first queued lock can be either a read or a write. In this implementation the lock structures are actually part of the lock request objects built by the transactions. Since each request requires one and only one lock this avoids any unnecessary memory allocation. Furthermore, there is then no limit to the length of the queues. The held locks and queued locks are maintained as doubly linked lists with two pointers in the leaf node pointing to the head of the held and queued lists respectively.

When a transaction unlocks a leaf node the corresponding lock is removed from the held list. If the held list is now empty but the queue list is not, then the first queued request is moved to the held list and the corresponding transaction's event semaphore is posted so that if the transaction is blocked waiting for this lock it will be reactivated. If the newly held lock is a read then any further read requests at the head of the queued requests are also moved to the held list and their corresponding transactions restarted. Lock downgrading from write to read is very similar except that instead of removing the held lock it is changed to a read lock. Any read locks at the head of the queue are also granted.

Since TSO and TPL are standard methods the implementation of these will not be discussed in any great detail. In order to be able to make meaningful performance

comparisons fairly sophisticated implementations are required. TSO is inherently simpler than LL and consequently required less programming effort than LL. A suitable version of TPL on the other hand is considerably more complicated than LL given the need to deal efficiently with deadlocks.

TSO is intrinsically very easy to implement. A basic feature of TSO is the restart-delay time. Such a delay is necessary in order to prevent two transactions from repeatedly interfering with each other. In order to insure that TSO would run at all in conditions of heavy data contention, it was necessary to use an adaptive restart delay [8]. When a transaction is aborted it is delayed by an exponentially distributed delay of average value equal to the running average of transaction run times before being restarted. Thus as contention increases and transaction run times increase due to restarts, the running average increases and the restart delays become longer thereby reducing the effective mpl and reducing data contention so that active transactions have a better chance of finishing.

TPL was first implemented using a simple timeout strategy to detect deadlocks. This proved to be unsatisfactory since there was then no way to control which transaction was aborted. Since the algorithm was tested under conditions of heavy contention where multiple restarts were common throughput rapidly dropped to zero. Also, setting suitable timeout values is difficult since if they are chosen to be large then too heavy a price is paid for restarts in low contention situations. If they are too small then transactions which are not deadlocked can needlessly time out in high contention situations. An adaptive timeout strategy where the timeout value is adjusted depending on the current load could have been used. Instead a wait-for graph testing approach was used [12]. Whenever a transaction is forced to block waiting for a lock, a node is added to the wait-for graph with a directed arc to a node representing the transaction holding the lock. Whenever a lock is requested or released the wait-for graph is tested for cycles indicating a deadlock. Thus deadlocks can be detected immediately. When a deadlock is found the time stamps of all transactions on the cycle are compared and the one with the largest time stamp (youngest transaction) is aborted. In this way it is possible to insure that the oldest transaction will make progress and will eventually finish (perhaps after restarting due to a later deadlock with an even older

transaction). Restarted transactions are not given new time stamps unlike in TSO. While it is not strictly necessary to use a restart delay for TPL [8] it was found to be very beneficial in high contention situations. The same adaptive restart delay used for the TSO algorithm was used for TPL as well. Again this works as a negative feedback mechanism limiting the effective mpl. This actually increases throughput at high contention since the extra transactions which are in their restart delay state would more than likely cause further deadlocks and restarts if allowed to run.

Two common simplifying assumptions which have been shown to adversely affect results were deliberately not used in these simulations. These are fake restarts and no lock upgrading [8]. Fake restarts are relevant to both TPL and TSO. A fake restart is simply using a new transaction rather than repeating the same aborted one. Lock upgrading is only relevant to TPL. If a data item is first read and then later written, it is commonly immediately given a write lock. This however needlessly limits concurrency.

Another concurrency issue for TPL is what to do when a write lock is released and there are several read and write request waiting [2]. For maximum concurrency it is desirable to allow other read requests to jump ahead of write requests if the next granted lock is a read. This however can prevent a write request from ever getting granted if there is a steady stream of read requests arriving. In the TPL implementation used here read locks are not allowed to jump ahead of write locks in order to minimize response time variations. An even better strategy would be to give every transaction a time stamp and then before granting the next lock sorting the queued lock request by time stamp. The next lock granted would then be the one belonging to the transaction with the smallest time stamp. Now read locks would only be allowed to jump ahead of write locks with larger time stamps. Note that this strategy in the limit of high contention, where every data item always has some lock requests queued becomes identical to LL. This is, however, not in the spirit of the TPL policy and was thus not used. It is interesting to note that these refinements to basic TPL i.e., time-stamp ordering of queued lock requests and deciding which transaction to abort in a deadlock based on time stamps brings elements of LL into TPL. In LL transactions are serialized in time-stamp order and all queued lock requests are

automatically in time-stamp order. The performance comparisons in the next chapter indicate that existing TPL implementations could be brought closer in performance to LL by implementing the above proposed sorting of queued lock requests.

Chapter 4

Simulation Results

4.1 Workload

The simulation of LL, TPL and TSO was carried out on two distinct workloads. In the first, each transaction consists of a purely random set of read and writes with each transaction accessing 5 data items. The mean percentage of write accesses was fixed at 33%. In the second workload each transaction consists of 4 random data item reads followed by zero or more writes of the same set of data items, with the probability of a data item being written set at 33%. This second workload is expected to more closely resemble real world transactions as well as being the least favourable for LL as compared to TPL. LL must request all locks at the beginning and thus with all the writes at the end, the exclusive write locks must be held or queued for the bulk of the transaction, unlike in TPL where they can be acquired just before writing the values. The latter work load is also expected to be very detrimental to TSO since the writes at the end would invalidate values read by concurrent transactions of larger time stamp thus forcing the writing transaction to restart near the end of the transaction. This is the worst time for a restart to happen as all the work already done by a transaction is thus wasted. Between each data item access and at the beginning of a transaction there was an exponentially distributed [13] delay of average value t_d. The value of t_d was adjusted so that even at the maximum mpl level used the processor was idle most of the time. This ensures that the following results are essentially in the infinite

resource limit [8] since the only resource used in this simulation is the processor.

Each simulation was run for 10,000 transactions after running an initial 100 transactions. The results from the first 100 transactions were ignored in order to allow the system to "warm up" [13]. The remaining 10,000 transactions were split into 10 batches of 1000 transactions and the batch means method [13, 14] used to analyze the results. Each concurrency control algorithm operated on the identical set of transactions.

4.2 Throughput

The primary performance metric of a concurrency control scheme is the throughput in transactions per second (TPS). In figure 4.1 the simulation results are presented for LL, TPL and TSO operating on the completely random work load. The throughput in TPS is plotted as a function of the multi-programming level for conditions of light, medium, and heavy data contention corresponding to 1024, 128 and 16 data items in the database. For the light contention results throughput increases almost linearly with mpl, as expected for all algorithms, but LL is clearly the best. For the medium contention results the throughput saturates and further increases in mpl do not increase the throughput. This is due to data contention thrashing for LL, restarts for TSO and a combination of the two for TPL. Again it is important to realize that the throughputs for both TSO and TPL would drastically drop at high mpl were it not for the adaptive restart delays used. As can be seen from figure 4.1 in situations of high contention and effectively infinite resources LL is more than twice as good as either TPL or TSO.

What about for the less favourable but probably more realistic second workload? Figure 4.2 is identical to figure 4.1 except that this time LL, TPL and TSO operated on the second workload which has all the data writes overlapping the read set and at the end of the transactions. The results are qualitatively similar to those for the first workload with LL significantly better than TPL or TSO. As expected, and discussed above, TPL is the least affected by the change in workload whereas both LL and TSO experience diminished throughput. However while here TPL performs better

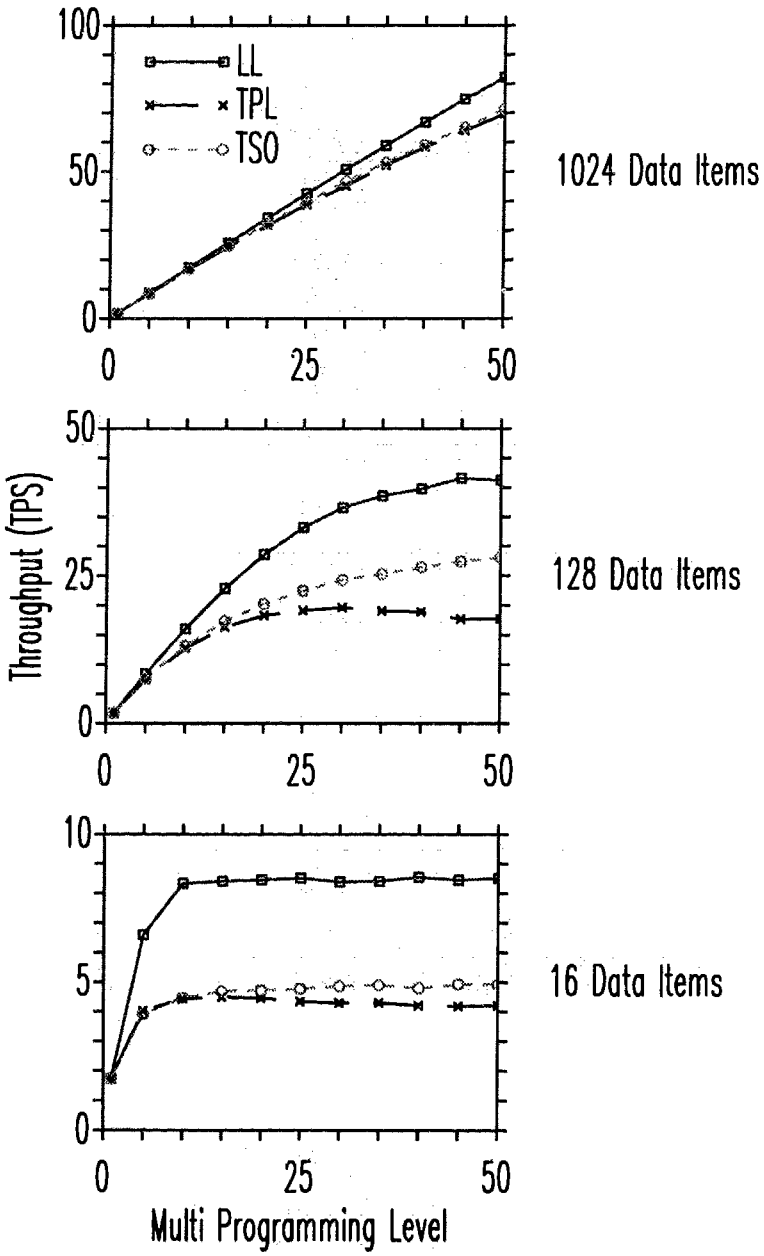

Figure 4.1: Throughput with random workload

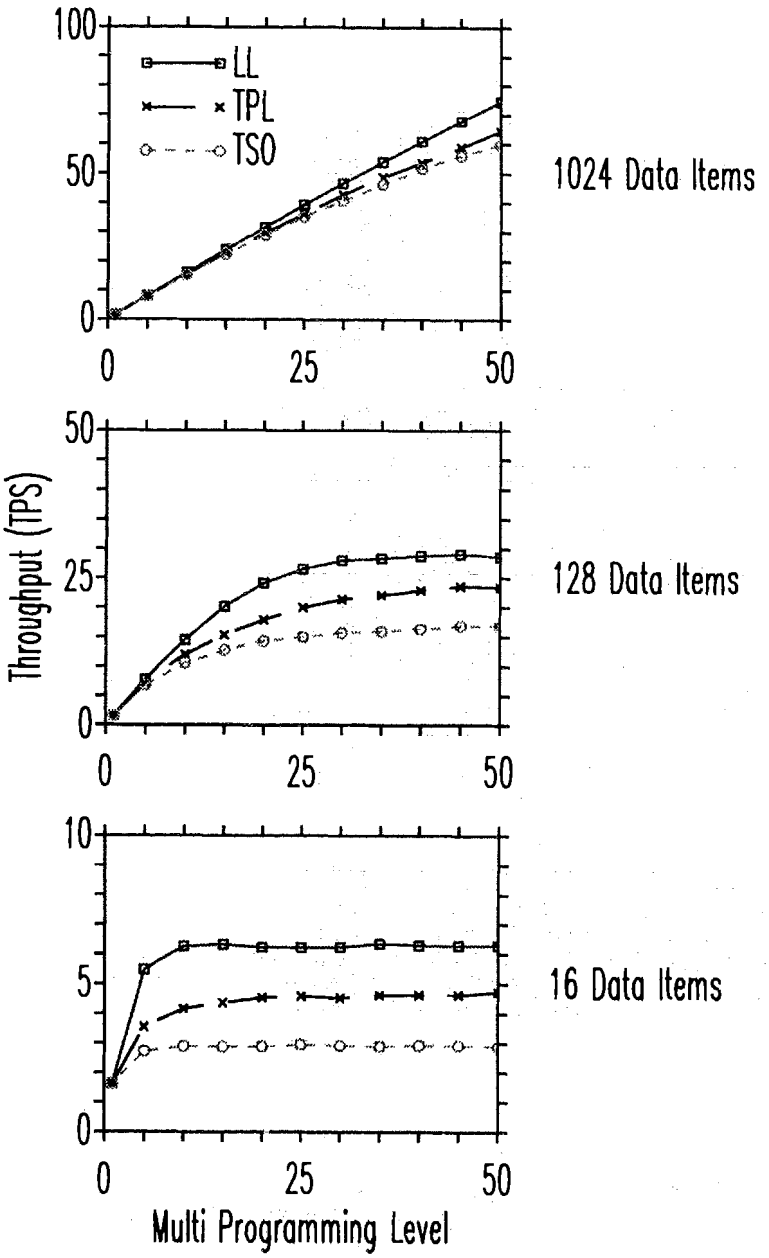

Figure 4.2: Throughput with all writes at end

than TSO, LL still has a 30% greater throughput than TPL. The remaining results presented here were all generated using this second, less LL favourable workload unless otherwise stated.

4.3 Response Time

Alternatively the simulation results can be presented in terms of the mean response time of a transaction. Here the response time is the time elapsed between the start and end of a transaction. It is essentially the inverse of the throughput results and LL has the lowest mean response time (figure 4.3) as expected from the throughput results. Not much additional information can be gleaned from these results. What is more enlightening are the standard deviations of the mean response times presented in figure 4.4. The response time variations are much larger for both TSO and TPL than for LL. This is because under LL transactions need never be restarted and they proceed in time-stamp order. Thus, each transaction finishes in a very predictable time with only a small variance, as is confirmed by the simulation results. Concurrency control algorithms which require restarts have large variances since if a transaction must be restarted the response time for that transaction will suddenly become much longer. A small response-time variance is a desirable property from a database user's point of view and in this regard LL is vastly superior to both TSO and TPL.

4.4 Blocks and Restarts

It is also informative to look at two other results which enable us to determine why LL is better than TPL even for the second workload. In figure 4.5 the mean number of blocks per transaction and in figure 4.6 the mean number of restarts per transaction are presented. First note that the mean number of blocks per transaction is zero for TSO since it is a pure restart concurrency control method. Similarly, the number of restarts per transaction for LL is zero since it is a pure blocking method. Furthermore, note that the number of blocks per transaction are almost identical for TPL and LL indicating that data contention for these two methods is about the same. Thus

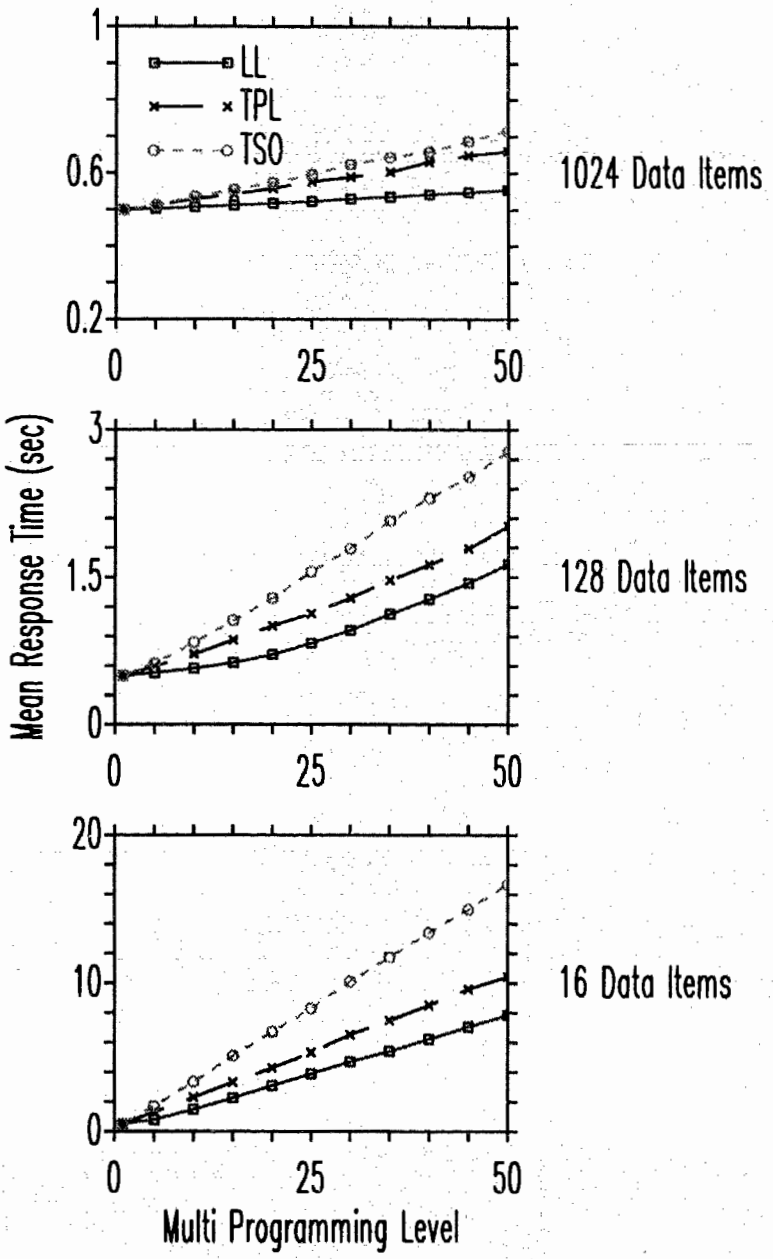

Figure 4.3: Response time

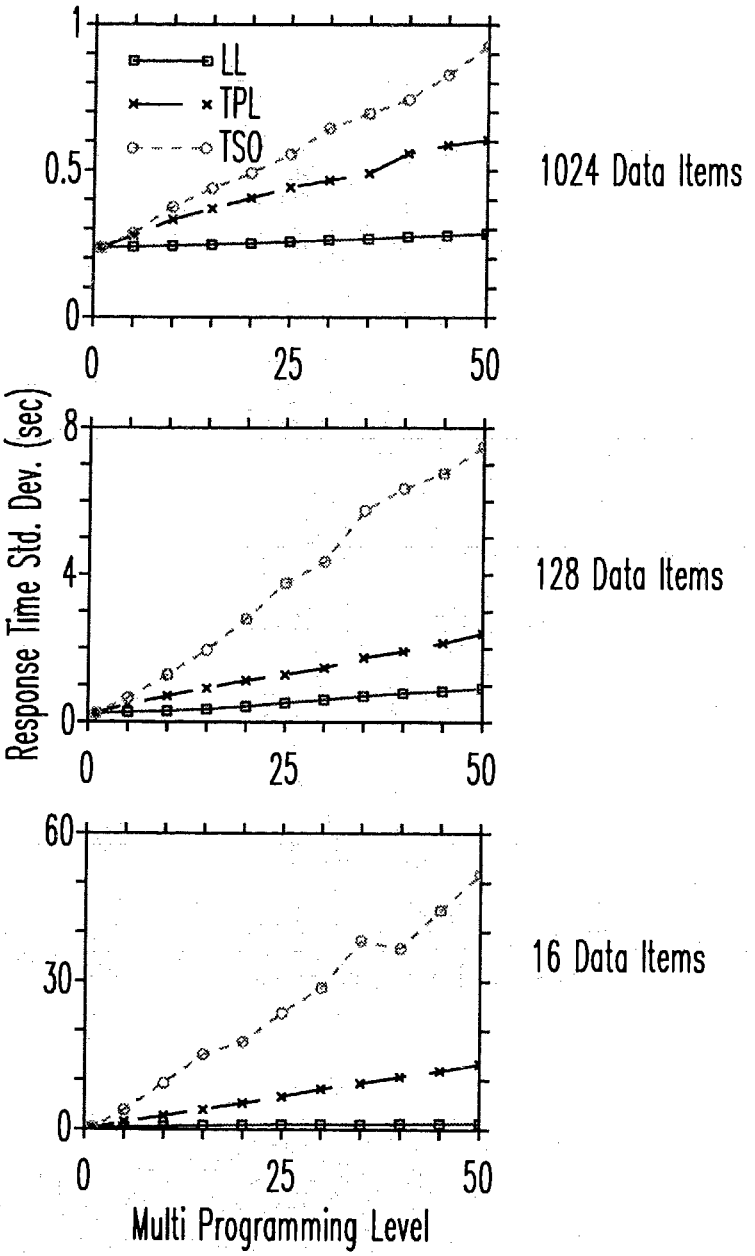

Figure 4.4: Response time standard deviation

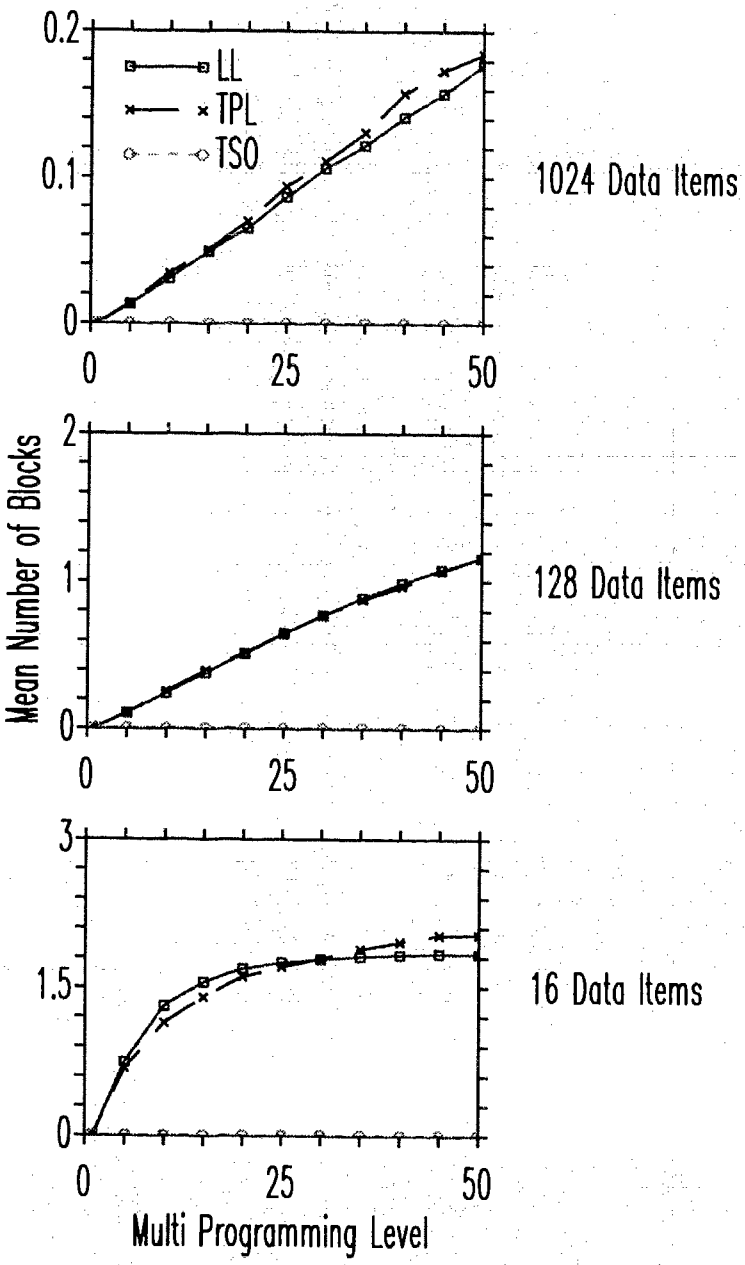

Figure 4.5: Mean number of blocks per transaction

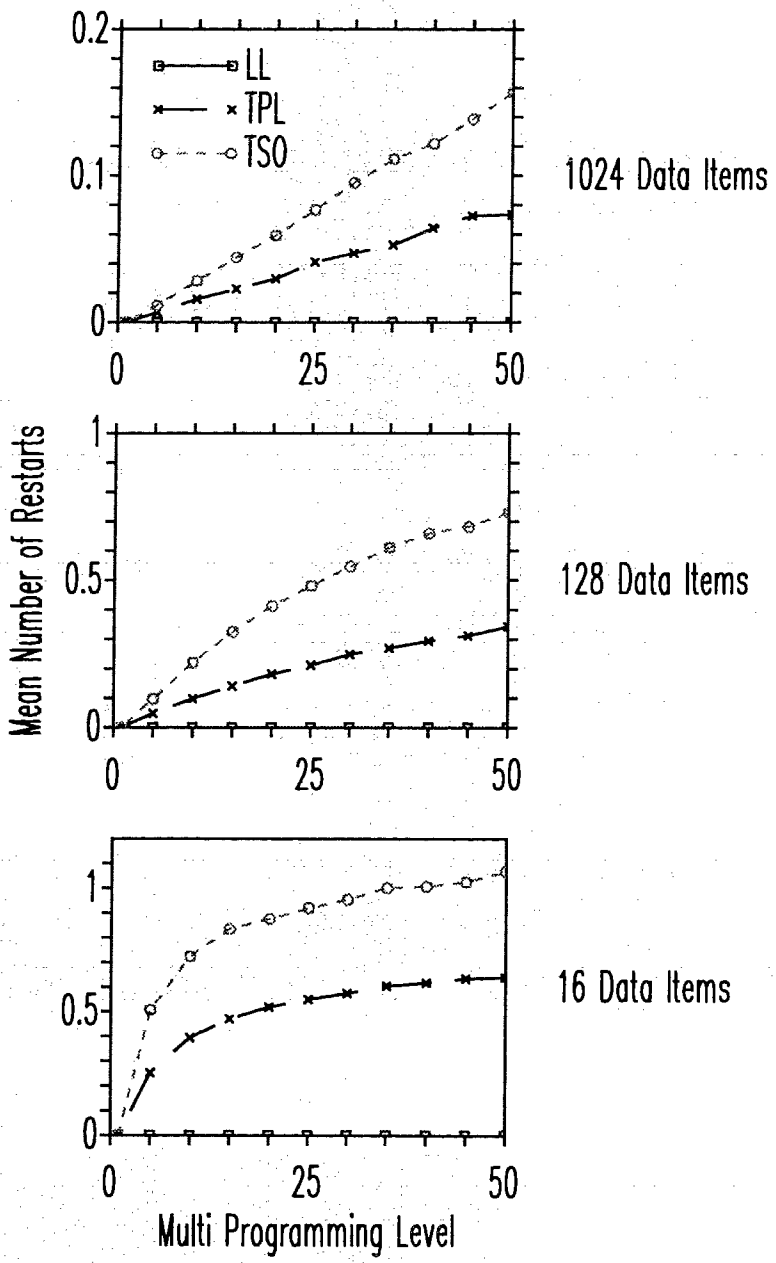

Figure 4.6: Mean number of restarts per transaction

the fact that TPL has a non-zero mean number of restarts per transaction must be to blame for its poorer performance. As extensively discussed by Agrawal *et al.,* [8] the performance of a concurrency control scheme is critically dependent on the assumptions made about the database system. It was shown that published discrepancies [15, 16, 17, 18] in the relative performance of competing concurrency control schemes were often due to different underlying assumptions. In particular, concurrency control schemes which rely heavily on restarts were shown to perform much worse when the throughput is resource contention limited rather than in the infinite resource limit. This is because all the resources already used by a transaction about to restart were wasted and thus reduced the amount of resources available to other transactions doing useful work. Consequently it is possible to infer that LL will perform even better relative to TPL and especially TSO in the more realistic resource limited case since LL never needs to restart a transaction for concurrency control. As in the case of the workload, the system assumption used for the results presented here (infinite resources) is the least favourable to LL.

4.5 Dummy Locks

It is possible to look at the effect of one of LL's disadvantages and its impact on performance. As mentioned in Chapter 2 LL's read and write sets need to be predeclared since all data items that may be needed must be locked at the beginning of the transaction. Thus if the transaction contains an if clause where one of two data items is to be accessed, then both of these data items must be locked. This will decrease concurrency since more locks are held. Predeclaration is not required for standard TPL and thus TPL's performance is not affected by transactions containing if clauses. The question is how many additional locks LL can afford to lock before performance drops down to the level of TPL? To answer this LL was modified so that for the set of lock request there was a probability of setting a dummy lock of the same type on the neighbouring data item in the tree (if not already used in the transaction). The probability values used were 0, 0.25, 0.50, 0.75 and 1. These dummy locks were held or queued until its associated neighbour was released. The results are presented in

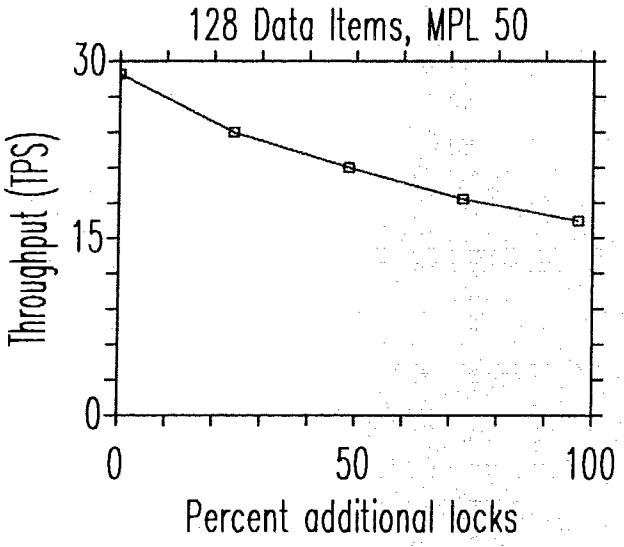

Figure 4.7: The effect of additional locks

figure 4.7 and indicate that approximately 50% more locks can be requested before LL's performance drops by the 30% advantage it held over TPL. The performance penalty due to these additional locks is somewhat less than might be expected because a dummy lock might never be held since it is not read or written. A dummy lock might just be queued for the duration of the transaction while a concurrent, smaller time-stamp transaction holds a lock on the dummy data item.

4.6 Strictness

As discussed in Chapter 2, LL is much more susceptible to cascading aborts than TPL. In applications with many user driven aborts this could make LL in its present form unworkable. In order to avoid cascading aborts LL can be made strict. In a strict implementation of LL (SLL) all write locks must be held until the end of the transaction. This is expected to be much more detrimental to LL than to TPL. Nevertheless, for the second, more realistic workload with all the writes at the end, all the write locks are already held for the bulk of a transaction and a strict version

of LL might still be better than strict TPL. The throughput of SLL as compared to strict TPL is shown in figure 4.8. These results show that SLL compares favourably with strict TPL for the second workload with its throughput still marginally larger. The effect of enforcing strictness on the first workload is illustrated as an aside in figure 4.9 and as expected, is much more detrimental.

4.7 Multi-version Leaf Locking

Instead of allowing user driven abortions one could insist that the effects of a given transaction on a database can only be reversed by issuing a transaction that cancels the effect of the previous one. If no abortions are allowed then even greater concurrency can be achieved by performing some in-queue processing. Under conditions of high data contention leaf nodes will have long queues of lock requests. Since these queues are FIFO the order of data item accesses is fixed. It is then possible for a transaction writing a value to write the new value to the lock request, in effect writing a new version, rather than blocking. When the lock request carrying the new value reaches the data leaf node the value is simply transferred. Read transactions in the queue ahead of this in-queue write still read the old value as required. A read transaction, behind the in-queue write but before the next write, may read the in-queue value once the in-queue writing transaction is done with it, without waiting until the lock request reaches the leaf node. With this multi-version leaf locking (MVLL) write operations need never block and read operations block only if the nearest write request ahead of it in the queue has not yet been satisfied. A transaction which reads a given data value may finish processing before its lock request ever reaches the leaf node in which case the request can simply be removed from the queue. Satisfied write requests cannot be removed since the new value must be written to the leaf node when the request reaches the leaf. A satisfied write request is simply flagged as done so that once it reaches the leaf node a lock for the transaction already finished with this data item is not applied. Depending on the workload a significant amount of work may be done while the requests are in the queue. Note also that this is a very space efficient multi version protocol since extra versions are only created if really needed

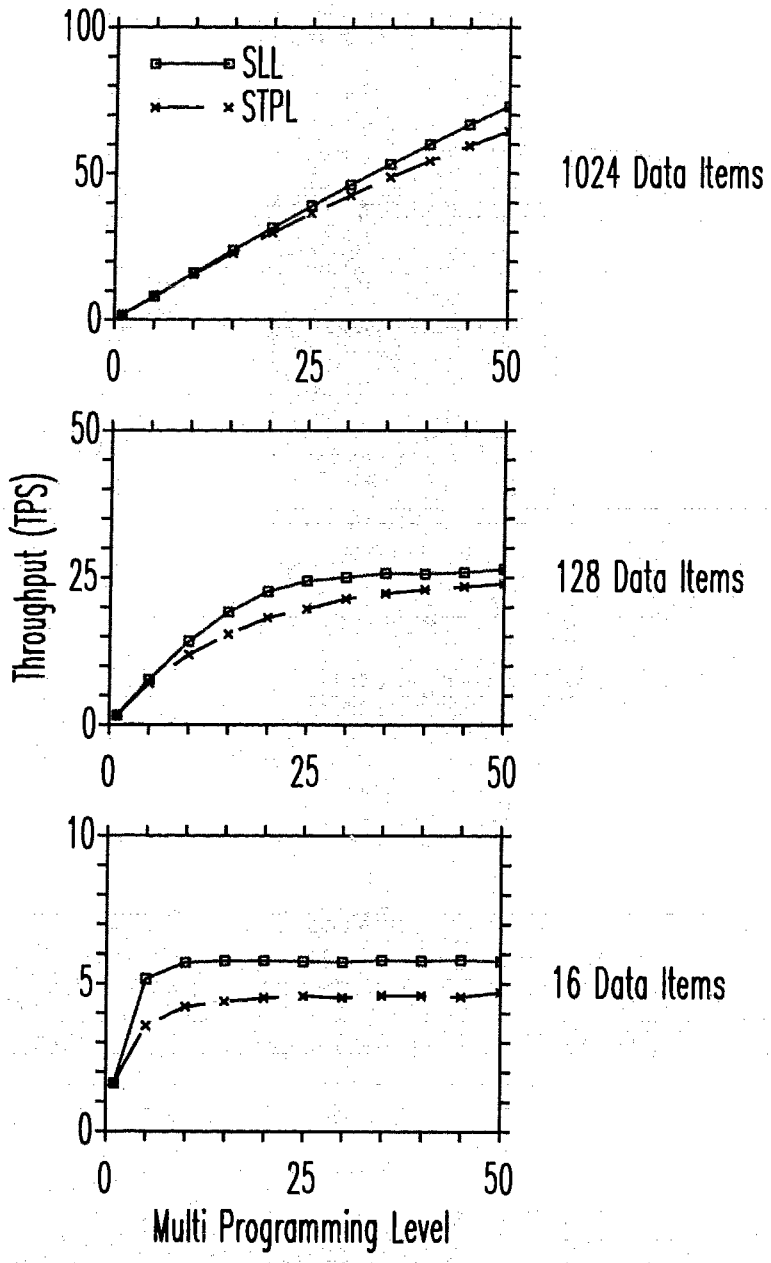

Figure 4.8: Throughput of strict versions of LL and TPL using "writes-at-end" workload.

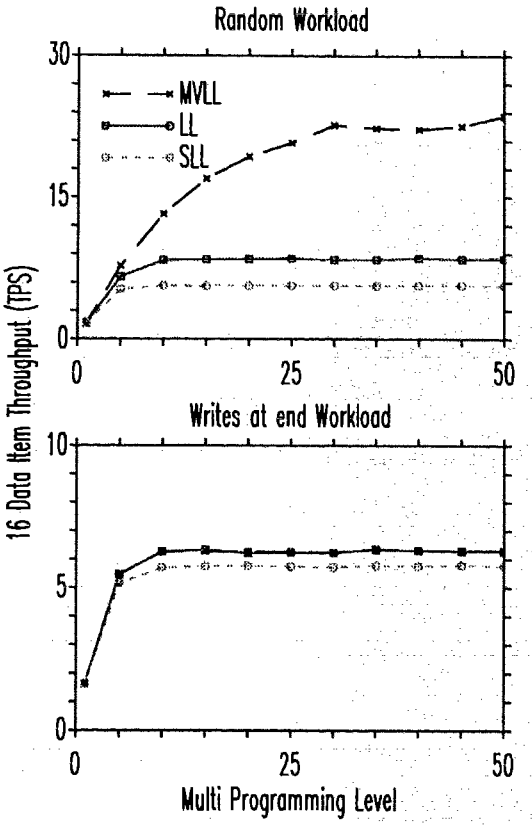

Figure 4.9: Throughput comparisons of multi-version, ordinary and strict LL

and kept only for the minimum amount of time.

The throughput results under heavy data contention for MVLL are shown in figure 4.9. These results indicate that for the random workload throughput increases by almost a factor of 3 with in-queue processing but for the writes-at-end workload there is no change in throughput. This is as expected since for the second workload each data item written must first be read. Hence no in-queue writes are done since transactions must block waiting for the read. Whether the extra complexity and restrictions of MVLL are worthwhile is thus extremely workload dependent.

Chapter 5

Conclusions

The simulation results confirm the predictions of increased concurrency for LL as compared to TPL even for workload and system assumptions least favourable to LL. For workloads where the number of additional data items to be locked due to if clauses in the transactions is less than 50%, LL is superior to TPL. The increased throughput as compared to TPL appears to be due to the complete lack of concurrency-control forced restarts for LL. No useless work is done by LL provided that there are no user driven abortions. The scarcity of abortions implies that LL will perform even better as compared to TPL in resource limited environments. All forms of LL are very fair in the sense of having a very small response time standard deviation. All forms of LL are also naturally deadlock and starvation free. Lastly, LL is comparatively simple to implement. The above statements apply to database systems which are I/O bound (implicitly assumed in the simulations) and have the LL concurrency control tree completely in main memory such that LL's lock coupling overhead (as well as TPL's deadlock detection) is a completely negligable part of the transaction processing time. No attempt was made to compare the concurrency control schemes in a CPU bound system.

Several modifications of the basic LL policy are possible such as strict LL and multi-version LL. The desirability of these modifications depends strongly on the type of workload and the probability of user driven abortions.

As yet unanswered questions include LL's performance in distributed database

systems and its applicability in parallel machines.

The results of the previous chapters clearly show that LL is a viable and competitive concurrency control mechanism at least for centralized databases. Furthermore, given LL's performance and other advantages it should find wide spread use in actual database systems.

Bibliography

[1] "The Notions of Consistency and Predicate Locks in a Database System", K.P. Eswaren, J.N. Gray, R.A Lorie and I.L. Traiger, *Comm. ACM*, Vol. 19 (1976) pp.624-633

[2] "Concurrency Control and Recovery in Database Systems", P.A. Bernstein, V. Hadzilacos, and N. Goodman, Addison-Wesley, Reading Mass. 1987 and references therein.

[3] See for example "Database System Concepts", H.F. Korth and A. Silberschatz, McGraw-Hill, New York 1991 and references therein.

[4] "Concurrency Control by Locking", C.H. Papadimitriou, *SIAM J. Comput.*, Vol. 12, (1983) pp.215-226

[5] "Concurrency Control for High Contention Environments", P.A. Franaszek, J.T. Robinson and A. Thomasian, *ACM TODS*, Vol 17 (1992) pp.304-345

[6] "Performance Evaluation of Cautious Waiting", M. Hsu and B. Zhang, *ACM TODS*, Vol 17 (1992) pp.477-512

[7] "Consistency in Hierarchical Database Systems", A. Silbershatz and Z. Kedem, *J. Assoc. Comput. Mach.*, Vol. 27 (1980) pp.72-80

[8] "Concurrency Control Performance Modeling: Alternatives and Implications", R. Agrawal, M.J. Carey and M. Livny, *ACM TODS*, Vol. 12, No. 4, (1987) pp.609-654

[9] "Concurrency Control in Distributed Database Systems", P.A. Bernstein, and N. Goodman, *ACM Computing Surveys*, Vol. 13, (1981) pp.185-222

[10] "Performance of B-Tree Concurrency Control Algorithms", V. Srinivasan and M.J. Carey, *SIGMOD record* Vol. 2, (1991) pp.416-425

[11] "OS/2 2.0 Control Program Programming Guide", IBM Corp, Que Corporation, Carmel , IN, (1992)

[12] "Notes on database operating systems", J. Gray in *"Operating Systems: An Advanced Course"*, Edited by R. Bayer, R. Graham, and G. Seegmuller, Springer Verlag, New York, 1979

[13] "Simulating Computer Systems", M.H. MacDougall, MIT Press, Cambridge, Mass. 1987

[14] "Computer Performance Modeling Handbook", edited by S.S. Lavenberg, Academic Press, New York, 1983

[15] "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", R. Agrawal and D. DeWitt, *ACM TODS*, Vol. 10, (1985) pp.529-564

[16] "The Performance of Concurrency Control Algorithms for Database Management Systems", M. Carey and M. Stonebraker, In *Proceedings of the 10th International Conference on Very Large Data Bases*, (Singapore, Aug. 1984), pp.107-118

[17] "Locking Performance in Centralized Databases", Y. Tay, N. Goodman and R. Suri, *ACM TODS* Vol. 10, (1985) pp.415-462

[18] "Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management", R. Balter, P. Bernard and P. Decitre , In *Proceedings of the 1st ACM SIGACT SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Aug. 1982), ACM, New York, 1982 pp.183-193