Investigations in Tree Locking for Compiled Database Applications

Heng Yu and Grant Weddell

School of Computer Science, University of Waterloo

Abstract

We report on initial experiments in tree locking schemes for compiled database applications. Such applications have a repository style of architecture in which a collection of software modules or subsystems operate on a common database in terms of a predefined set of transaction types, and are very often at the core of embedded systems. Since the tree locking protocol is deadlock free, it becomes possible to decouple recovery mechanisms from concurrency control, a property that we believe is critical to the successful deployment of database technology to this new application area. Our experiments show that the performance of tree locking can compete with two phase locking for cases such as the above in which a great deal can be known at the time of system generation about workloads.

1 Introduction

A *compiled database application* is a software system consisting of a collection of modules or subsystems that interact with a common database through a set of predefined transaction types [11]. Compiled database applications are very often at the core of embedded systems. One example of compiled database applications is the server control program for a *private branch exchange* (PBX) [12], whose software modules access a common database of control data including subscriber information, network and device status, routing information, etc. Another example is the processing of TPC-C benchmark [1], from which we make our experimental case in this paper. TPC-C has a set of specified transaction types of a wholesale supplier that access common data tables of customer, inventory, and order information. High performances are required for both examples.

It is the nature of the application area itself and the often very high throughput requirements that dictate the need to separate recovery from concurrency control. We believe that forcing all update to any part of the common database to be logged in order to support very general abort capabilities can lead inexorably to unacceptable throughput. The need for this happens either (a) because of a very general "begin transaction/commit/abort" transaction model, or (b) because of the use of strict two phase locking (2PL) [4] to manage concurrency control (as happens in virtually all existing commercial database systems). In regard to case (a), we shall assume a much simpler "begin transaction/commit" transaction model for the remainder of the paper. As for case (b), transactions may be aborted internally by the system for deadlock resolution, which is unavoidable under the not deadlock free 2PL protocol. Therefore, we need a locking protocol other than 2PL that guarantees both serializability [2] and freedom from deadlocks.

A locking protocol that satisfied the above requirements is *tree locking* (*TL*) [9]. TL assumes a *lock tree* structure whose nodes are the data items accessed by the transactions. It allows only write operations and exclusive locks[1]. The basic TL rules are:

1. a transaction can operate on a data item only after it obtains a lock on it;
2. except for the first data item that a transaction locks, a data item can be locked by a transaction only if the transaction currently holds a lock on its parent in the tree;
3. after a transaction releases the lock on a data item, it can never get the lock on it again.

Besides serializability and freedom from deadlock, the performance of transactions with the TL protocol is also an issue of interest. The concurrency of TL, as compared with 2PL, depends on how close the transactions match the lock tree. On one hand, TL allows a transaction to release the lock on a data item earlier than obtaining locks on other data, so that it may enable better concurrency than 2PL. On the other hand, due to rule 2 in the TL protocol, a transaction may unavoidably lock extra data items, which are not involved in the transaction itself, but lie "on the way" in the tree to some data items the transaction intends to lock and access. As we have predefined transaction types and the probability of each transaction type in a compiled database application, it is possible to optimize the tree structure to alleviate the disadvantage of TL.

There are quite some research work on the theoretical aspects of the TL protocol, to name a few, [6] [9] [15] [7] [10]. Regarding to application domain, variations of TL are used in B-tree-like searching structures, e.g. in [8]. However, as far as we know, not much research has been taken on how to apply TL to a set of predefined transactions. One recent research work with TL involved in is the implementation of transactional extensions in Java programming language [5]. A tree construction algorithm from a set of transactions is given in [5]. However, [5] only considers binary lock trees, and only allows data items to be tree leafs. And the internal nodes are extra "virtual" nodes. We believe it is preferable to have arbitrary lock tree shapes and to have all tree nodes corresponding to data items.

In this paper, we consider a heuristic method to build a lock tree from a set of transaction types along with their probabilities. To fit in compiled database applications, the lock tree, as well as some other data structures, can be constructed at compile time and stored for each transaction type. Moreover, we design runtime algorithms to lock and unlock data items based on these stored data structures.

We conduct some preliminary experiments to compare the performances of TL and 2PL on a set of predefined transactions that are taken from the TPC-C benchmark [1] with some modifications. The results show that TL performs as well as 2PL.

The organization of the remaining of the paper is as follows. In Section 2, the transaction model is defined. Section 3 introduces the runtime TL locking and unlocking algorithms for our transaction model, while Section 4 covers the process to generate at compile time the data structures that are crucial for runtime locking/unlocking. Experiment setting and results are presented in Section 5. Finally Section 6 concludes the paper.

2 Transaction model

In classical transaction model, a transaction type is a linear sequence of $r(x_i)$ and $w(x_i)$ operations in which each x_i is a data item. For compiled data applications, we allow a more general form of transactions. Each transaction type is modelled as a finite state machine, in which each state represents an operation on a data item, and each arc indicates a transition to a next operation. For an operation, there may be more than one succeeding operations with probabilities. Therefore, arcs are drawn for such transitions, and each arc is assigned with a probability of the transition from the source operation to the target one. For the time being, we assume only write operations, so it suffices to label each state with the data item that is written. The formal definition of a

[1]Variations of the TL protocol with both shared locks and exclusive locks are proposed in [7]. However, we only consider the TL protocol with exclusive locks in this paper.

transaction type is:

Definition 2.1 *Let D be a* data set *that consists data items in the database, and each data item is identified with a name.*

A transaction type *T is defined as $T = \langle N, s, F, A, data, duration, prob\rangle$.*

- *N is a set of states in the finite state machine, each representing an operation in the transaction type.*
- *$s \in N$ is the starting state of the transaction type.*
- *$F \subseteq N$ is a set of terminating states of the transaction type.*
- *$A \subseteq (N - F) \times N$. A is a set of transition arcs from states to states. Terminating states have no outgoing arcs.*
- *$data$ is a mapping from N to D. For each $n \in N$, $data(n)$ is the data item accessed by state n.*
- *$duration$ is a mapping from N to non-negative real number set. For each $n \in N$. $duration(n)$ is the time cost of the operation at state n to access data item $data(n)$. For $n_1, n_2 \in N$, it is possible that $data(n_1) = data(n_2)$ while $duration(n_1) \neq duration(n_2)$.*
- *$prob$ is a mapping from A to real numbers between 0 and 1. For each arc $\langle n_1, n_2 \langle \in A$, $prob(\langle n_1, n_2\rangle)$ is the probability that a transaction goes from n_1 to n_2.*

Moreover, we require that for each $n \in (N - F)$,

- *$\sum_{\langle n,n'\rangle \in A} prob(\langle n, n'\rangle) = 1$, and*
- *there is a directed path from n to some $f \in F$.*

Intuitively, the last two requirements state that a transaction (see Definition 2.2) must go from a non-terminating state to some state and only end with a terminating state[2].

Based on the notion of transaction type, we can define an individual transaction:

[2]In general finite state machines, it is legal to allow terminating states to have outgoing arcs, i.e. $A \subseteq N \times N$. And for each $n \in F$, it is possible that $\sum_{\langle n,n'\rangle \in A} prob(\langle n, n'\rangle) < 1$, and $1 - \sum_{\langle n,n'\rangle \in A} prob(\langle n, n'\rangle)$ is the possibility that a transaction terminates at the node. However, we can always transform such transaction type to the one in Definition 2.1 by introducing extra terminating states with duration 0.

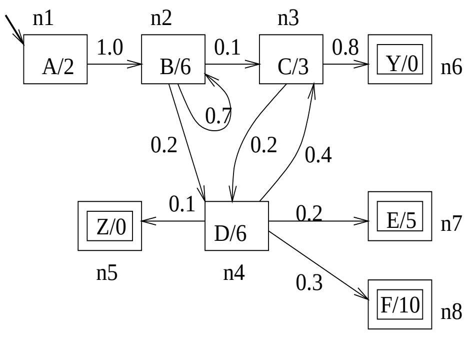


Figure 1: An example of transaction type

Definition 2.2 *A* transaction *t of transaction type $T = \langle N, s, F, A, data, duration, prob\rangle$ is a sequence $n_0 \cdots n_k$, where $n_i \in N$ for $0 \leq n \leq k$. Along the sequence, $n_0 = s$, $n_k \in F$, and for each adjacent pair n_{i-1}, n_i such that $1 \leq i \leq k$, $\langle n_{i-1}, n_i\rangle \in A$.*

Example 2.1 *An example of transaction types in given in Figure 1. The starting state is n_1, and the terminating states are n_5, n_6, n_7, n_8. The data items and durations of the states are shown in the boxes, separated by "/". The probabilities are attached to the arcs. An example of the transactions of this transaction type is $n_1, n_2, n_2, n_3, n_4, n_3, n_6$.*

In a compiled database application, there is a set of predefined transaction types, each with a probability. Such a set of transactions is defined as a *transaction system* as follow.

Definition 2.3 *A* transaction system *$\langle D, TS, prob\rangle$ is characterized by a set of transaction types, denoted as TS, on a data set D, and a mapping $prob$ from TS to real numbers between 0 and 1. For each $T \in TS$, $prob(T)$ specifies the probability that transactions of transaction type T is chosen when it is to select the next transaction to execute. $\sum_{T \in TS} prob(T) = 1$. Without loss of generality, we can always assume that D is the set that consists of all data items appearing in all transaction types, i.e. $D = \bigcup_{t \in TS}\{data_t(n) | n \in N_t\}$.*

3 Runtime locking and unlocking algorithms

Before we present the locking/unlocking algorithms under the TL protocol, we first define a *lock tree* as follow:

Definition 3.1 *A* lock tree $LT = \langle D, E \rangle$ *w.r.t. a transaction system* $\langle D, TS, prob \rangle$ *is a tree with a node set* D *and an edge set* E*. Each node of the tree corresponds to a data item involved in the transaction system. When we know the transaction system in the context, we also call such a tree a* global lock tree*.*

Each transaction, when executing, must lock/unlock data items following the TL rules on the lock tree. However, it is not necessary for a transaction to have the knowledge of the whole global lock tree. The transaction only needs to keep the part of the global lock tree that covers all data items its transaction type accesses.

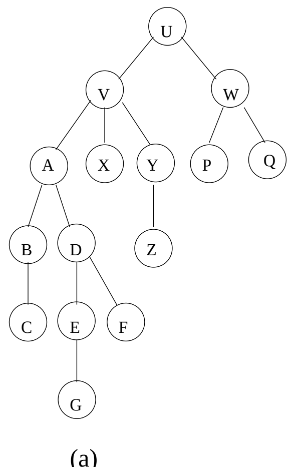
Definition 3.2 *A* cover *of a transaction type* $T = \langle N, s, F, E, data, duration, prob \rangle$ *in a global lock tree* $GLT = \langle GD, GE \rangle$*, denoted as* $CLT = \langle CD, CE \rangle$*, is a subtree of* GLT *such that for each* $n \in N$*,* $data(n) \in CD$*. Different from convention, we allow the leafs of* CLT *to correspond to internal nodes of* GLT*.*

It is possible there are some nodes in CD *whose data are never accessed by* T*. We define* in-transaction nodes *as the nodes* $\{d \in CD|$ *there exists some* $n \in N$ *and* $data(n) = d\}$*. The remaining nodes are* non-in-transaction nodes*. When* CLT *satisfies*

1. *all leafs of* CLT *are in-transaction nodes, and*
2. *there is no lock tree* $LT' = \langle D', E' \rangle$ *such that* $D' \subset CD$*,* $E' \subset CE$*, and* LT' *is also a cover of* T *in* GLT*,*

we call CLT *the* minimal cover *of* T *in* GLT*. When the context is clear, we call such a minimal cover a* local lock tree*, denoted as* $LLT = \langle LD, LE \rangle$*.*

Example 3.1 *For a global lock tree in Figure 2(a), the minimal cover of it by a transaction type in Figure 1 is shown in Figure 2(b). In the local lock tree, all nodes but v are in-transaction nodes.*



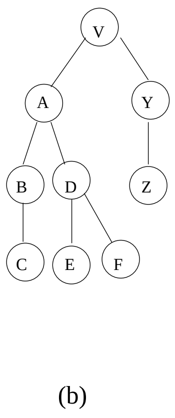


Figure 2: A global lock tree (a) and a local lock tree (b)

For each transaction type in a transaction system, we store its local lock tree. In the implementation, the nodes with the same name in all local lock trees share a common semaphore which only allows counts 0 and 1. Therefore, locking and unlocking a single node can be realized by the **p** and **v** semaphore operations. When a transaction accesses a data item at a state, it retrieves the corresponding node in the local lock tree for its transaction type and tries to lock it. To facilitate this process, in our implementation, a hash table is maintained for the retrieval. It is noteworthy that both the local lock tree and the hash table are built at compile time and kept static during the run time.

The algorithm to lock a data item at a state when executing a transaction, is shown in Algorithm 1. Each tree node has a field **visited** to indicate whether it has been locked for later unlocking choice.

Algorithm 1 *Locking a data item x at a state:*

```
Require: an empty stack
1: find the tree node $n$ corresponding to $x$ in the hash table;
2: while $n$ is not locked do
3:   push $n$ to stack;
4:   if $n$ is root of the local tree then
5:     break;
6:   else
```

```
7:       $n \leftarrow n.parent$;
8:     end if
9:   end while
10:  while the stack is not empty do
11:    pop the node $n$ on top of the stack;
12:    lock node $n$; {semaphore p operation}
13:    $n.visited \leftarrow true$;
14:  end while
```

For unlocking a data item, it is more complicated than locking. As we see from rule 2 of the TL protocol, a lock on an tree node not only protects the shared data item but also serves as a "bridge" to lock other nodes that are its descendants in the tree. Because of rule 3, we cannot relock the data item after we have unlocked it. So we have to hold the lock until we know all its children either have been locked or lead to subtrees whose data items will never be accessed in the future. Moreover, also because of rule 3, we cannot unlock a data item unless we are certain that the data item will not be accessed at any future states of the transaction type.

To formalize these conditions for unlocking, we define *unreachability* and *tree unlockability* as follow.

Definition 3.3 *Given a transaction type* $T = \langle N, s, F, A, data, duration, prob \rangle$, *for a state* $n \in N$, *if on all paths from* n *to all terminating states* $f \in F$, *each state* n' *satisfies* $data(n') \neq d$, *which means* d *will never be accessed by transactions of* T *after state* n, *we say* d *is* unreachable *from state* n. *All data at non-in-transaction nodes in the local tree are also unreachable from any state* n. *And the data set that are unreachable from state* n *is denoted as* $UR(n)$.

Definition 3.4 *Given a transaction type* $T = \langle N, s, F, A, data, duration, prob \rangle$, *when a transaction of* T *is executing and reaches a certain state* $n \in N$, *we say a data item* d *is* tree-unlockable *if for each child* c *of the node* d *in the local lock tree, either*

1. *c has been locked by the transaction before, or*

2. *all data items in the subtree rooted at c are in $UR(n)$.*

The unreachable set is related to a transaction type and can be built at compile time, while tree unlockability depends on individual transaction and can only be tested at runtime. The following theorem is straightforward.

Theorem 3.1 *Given a transaction type along with its local lock tree, for a transaction of this transaction type, at each state* n, *if we lock data items following Algorithm 1, and unlock a data item only when it is both tree-unlockable and is in* $UR(n)$, *then the transaction observes the TL protocol.*

The conditions of unreachability and tree unlockability specifies the correctness of locking and unlocking. However, in the compiled database application context, we notice that:

1. although it is possible to compute the $UR(n)$ set at each state n at compile time, storing such information takes much memory. Moreover, we only care about which of those previously locked data items can be unlocked at the state;

2. checking condition 2 of tree unlockability is very time consuming at runtime.

To reduce the storage space, for each state of a transaction type, we only keep an *unlockable set* instead of all unreachable items.

Definition 3.5 *For a state* $n \in N$ *in a transaction type* $T = \langle N, s, F, A, data, duration, prob \rangle$, *its* unlockable set, *denoted as* $UL(n)$, *is a set of data items. And each* $d \in UL(n)$ *satisfies:*

1. *there exists some state n' which lies on a path from s to n, such that $d = data(n')$;*

2. *$d \in UR(n)$ is unreachable from state n;*

3. *on each path from each n' (specified in 1) to n, there is no state n'' between n' and n such that $d \in UR(n'')$.*

Suppose a transaction of transaction type T *has gone through states* $n_1 = s, n_2, \cdots, n_{i-1}$, *and is currently at state* n_i, *the* total unlockable set *for the transaction, denoted as* TUL, *is defined as* $TUL = \bigcup_{k=1 \cdots n} UL(n_k)$.

state	UR	UL
n_1	$\{\}$	$\{\}$
n_2	$\{A\}$	$\{A\}$
n_3	$\{A, B\}$	$\{B\}$
n_4	$\{A, B\}$	$\{B\}$
n_5	$\{A, B, C, D, E, F, Y\}$	$\{C, D\}$
n_6	$\{A, B, C, D, E, F, Z\}$	$\{C, D\}$
n_7	$\{A, B, C, D, F, Y, Z\}$	$\{C, D\}$
n_8	$\{A, B, C, D, E, Y, Z\}$	$\{C, D\}$

Table 1: Unreachable and unlockable sets of each state of the transaction type in Figure 1

Intuitively, the elements in $UL(n_i)$ are those data items possibly locked by the transaction before it enters state n_i, and n_i is the earliest state where these data items can be unlocked. For each state n, $UL(n)$ can be computed at compile time. The TUL is a runtime set for an executing transaction, initialized as empty. When the transaction reaches a state n_i, it adds all elements in the unlockable set of state n to the total unlockable set, i.e. $TUL \leftarrow TUL \cup UL(n)$. It is obvious that $TUL \subseteq UR(n)$ at any state n. The inverse does not hold, because TUL may not include some data items that the transaction never accesses (See Example 3.2). However, we can still use TUL to approximate $UR(n)$ at each state. The advantage is we only need to store a smaller unlockable set for each state. This saves considerable memory space, and still helps to build TUL incrementally from static $UL(n)$ at each state n that the transaction goes through at runtime.

Example 3.2 *For the transaction type in Figure 1, the unreachable sets UR and unlockable sets UL for all states are shown in Table 1.*

If a transaction follows the sequence n_1, n_2, n_3, $n4$, n_3, n_4, n_7, the incrementally computed total unlockable set TUL at each state is shown in Table 2. We can see that TUL is always a subsets of $UR(n)$ at each state n. And at n_7 state, TUL does not include F, Y, and Z that are never accessed by the transaction.

Another modification is we use *weak tree unlockability* in stead of the original *tree unlockability.* Following is the definition of weak tree unlockability.

state	TUL up to the state
n_1	$\{\}$
n_2	$\{A\}$
n_3	$\{A, B\}$
n_4	$\{A, B\}$
n_3	$\{A, B\}$
n_4	$\{A, B\}$
n_7	$\{A, B, C, D\}$

Table 2: Total unlockable set at each state of a transaction

Definition 3.6 *When a transaction of a transaction type is executing and reaches a certain state, we say a data item is* weakly-tree-unlockable *if each child of the data item node in the local lock tree either*

1. *has been locked by the transaction before, or*

2. *is a leaf node and in TUL.*

If a node is weakly-tree-unlockable, it must be tree-unlockable. Again the inverse does not hold. Testing weak tree unlockability is much cheaper than testing the original tree unlockability, because it stops at the direct children instead of testing the whole subtrees underneath. Therefore, we believe the simplification is worthwhile.

From Theorem 3.1 and the properties of unlockablity and weak tree unlockability, we can conclude with the following corollary.

Corollary 3.1 *Given a transaction type along with its local lock tree, if a transaction locks data items following Algorithm 1, and unlock an data item only when*

1. *it is in TUL and is weakly-tree-unlockable, or*

2. *it is a non-in-transaction node and is weakly-tree-unlockable,*

then the transaction observes the TL protocol.

Based on Corollary 3.1, we can design the unlocking algorithm as shown in Algorithm 2. To improve the efficiency of weak tree unlockability testing, we attach a field **num_children_qualified** to each tree node.

This field shows how many children of the node either have been locked, or are both leaf nodes and in TUL. If **num_children_qualified** equals to number of children of the node, the node is weakly-tree-unlockable. Therefore, each time we only need to do a number comparison instead of iterating over the children of the node. The computing of **num_children_qualified** is very easy. Each time a node is locked[3], or its data item is added to TUL at some state (if it is a leaf node), we increment this field of its parent node.

Algorithm 2 *Unlocking data items, called at each state n:*

```
1:  for each $e$ in $UL(n)$ do
2:    if $e$ is not in $TUL$ then
3:      add $e$ into $TUL$;
4:      if not $e.visited$ then
5:        increment $e.parent.num\_children\_qualified$ by 1;
6:      end if
7:    end if
8:  end for
9:  for each tree node $d$ locked do
10:   if $d$ is an in-transaction node then
11:     if $d$ is in $TUL$ then
12:       if $d.num\_children\_qualified$ equals to number of children of $d$ then
13:         unlock node $d$;
14:       end if
15:     end if
16:   else
17:     if $d.num\_children\_qualified$ equals to number of children of $d$ then
18:       unlock node $d$;
19:     end if
20:   end if
21: end for
```

The processing in Algorithm 2 is invoked when a transaction reaches a state. In addition, when the transaction finishes at a terminating state, all remaining locks are released.

[3]This can be done by add an increment statement after line 13 in Algorithm 1.

4 Compile-time processing

In Section 3, the runtime locking/unlocking relies on two types of data structures: the unlockable set at each state and the local lock tree. Both can be generated at compile time.

4.1 Building unlockable set at each state

First, we address to the computation of unlockable set at each state. Given a transaction type $T = \langle N, s, F, A, data, duration, prob \rangle$, we define $G(T) = \langle N, A \rangle$ as a directed graph whose nodes are the states in T and whose arcs are transitions in T. $G(T)$ has a source node with no incoming arcs and a set of sink nodes with no outgoing arcs, which represent s and F respectively.

For two nodes $n_1, n_2 \in N$, we say n_2 is *reachable* from n_1 if there is a directed path from n_1 to n_2 in $G(T)$, and n_2 is *after* n_1 if n_2 is reachable from n_1 but not vice versa. So $d \in UL(n_i)$ only if there exists some n_j such that $d = data(n_j)$ and n_i is after n_j, i.e, d is locked before n_i and will never be accessed after n_i. For each $n \in N$, we define a set $RL(n) = \{n_k | n_k$ is after n and $data(n) \in UL(n_k)\}$. Intuitively, $RL(n)$ contains all "earliest" states where the data item accessed at state n can be unlocked. Given a data item d, it is obvious that $\bigcup_{n \in N, data(n)=d} RL(n) = \{n \in N | d \in UL(n)\}$. So once we have RL sets for all states, we can build UL sets as well. Next, we address to how to build RL sets.

As we know, the nodes of a directed graph can be partitioned based on which maximal strongly connect component that a node is in. Let $SC = \{C_1, \cdots, C_n\}$ be such a partition on N. We can see, if a data item d is accessed at state n, at each state n_i that is reachable from n, d cannot be unlocked if and only if,

1. n_i is in the same $C \in SC$ as n, or

2. there is an n_k after n and reachable from n_i in $G(T)$ such that $data(n_k) = d$.

We denote the set consisting of all n_i that satisfy the two conditions as $L(n)$. Given the above definitions, the following theorem holds.

Theorem 4.1 *For each $n_j \notin L(n)$, $n_j \in RL(n)$ if and only if there exists an $n_k \in L(n)$ and $\langle n_k, n_j \rangle \in A$.*

The algorithm to compute $UL(n)$ for all states is presented in Algorithm 3. At each iteration of the main loop at line 5, the algorithm processes a state n by calling the **dfs_mark** procedure. **dfs_mark** is from the depth-first searching algorithm, and it computes a subset of $L(n)$, denoted as $L'(n) = \{n_i | n_i \in L(n)$ and there is a directed path from n to n_i without passing any $n_k \neq n$ such that $data(n_k) = data(n)\}$. After all nodes in $L'(n)$ are marked and the procedure returns, the main algorithm continues to look for a node set $RL'(n) = \{n_i \notin L'(n) | data(n_i) \neq data(n)$ and there is an $n_k \in L'(n)$ such that $(n_k, n_i) \in A\}$. for each $n_i \notin L'(n)$. We can see that $RL(n) - \bigcup_{n' \text{ is after } n, data(n')=data(n)} RL(n) \subseteq RL'(n) \subseteq RL(n)$. For each $n_i \in RL'(n)$, the main program puts $data(n)$ to $UL(n_i)$. Therefore, $UL(n_i)$ only gets correct data items from the algorithm. Furthermore, we show the completeness of the algorithm. Because the algorithm loops over all state nodes, the elements in $RL(n)$ that are missing in $RL'(n)$ will be compensated when other nodes that access the same item are processed in the loop. For each data item d, $\bigcup_{n \in N, data(n)=d} RL(n) = \bigcup_{n \in N, data(n)=d} RL'(n)$ holds. By the end of the algorithm, d is added into the unlockable sets at all nodes in $\bigcup_{n \in N, d=data(n)} R(n)$. As a conclusion, Algorithm 3 correctly computes $UL(n)$ for each $n \in N$.

Algorithm 3 *Compute $UL(n)$ for each state n:*

```
1:  compute the strongly connected components of $G(T)$ [3]
2:  for each state $n \in N$ do
3:      $UL(n) \leftarrow \{\}$
4:  end for
5:  for each state $n \in N$ do
6:      for each state $n' \in N$ do
7:          $n'.marked \leftarrow false$
8:          $n'.visited \leftarrow false$
9:      end for
10:     $n.marked \leftarrow true$
11:     call procedure $dfs\_mark(data(n), n)$
12:     for each marked $n'$ do
13:         for each successor $n''$ of $n'$ do
14:             if $n''$ is not marked and $data(n'') \neq data(n)$ then
15:                 $UL(n'') \leftarrow UL(n'') \cup \{data(n)\}$
16:             end if
17:         end for
18:     end for
19: end for
```

Function dfs_mark does a depth-first searching, and marks nodes to build $RL'(n)$:

```
1:  function $dfs\_mark(d, n)$ returns boolean
2:  $n.visited \leftarrow true$
3:  $flag \leftarrow false$
4:  if $n \in F$ then
5:      return(false)
6:  else
7:      for each success $n'$ of $n$ do
8:          if $n'.visited$ then
9:              if $n'.marked$ then
10:                 $flag \leftarrow true$
11:             end if
12:         else
13:             if $data(n') = d$ then
14:                 $flag \leftarrow true$
15:             else
16:                 $flag1 \leftarrow dfs\_mark(d, n')$
17:                 $flag \leftarrow flag \vee flag1$
18:             end if
19:         end if
20:     end for
21: end if
22: if flag then
23:     $n.marked \leftarrow true$
24:     if $n$ is in a strongly connected component $C$ then
25:         for each $n'' \in C$ do
26:             $n''.marked \leftarrow true$
27:         end for
28:     end if
29: end if
30: return(flag)
```

4.2 Building the global lock tree

Building a global lock tree is the most difficult part for an efficient application of the TL protocol. It is still unclear what quantitative measurement is the most suitable to judge the "fitness" of a tree. In our transaction model, we find some categories:

1. a lock tree should favor those more frequent transaction types in a transaction system.
2. a lock tree should favor transactions with higher probabilities under the same transaction type.
3. for those data items accessed at states that are close in some transaction type, it is preferable that their corresponding tree nodes are also close, e.g. as parent and child, or children under the same parent.
4. a transaction should lock as few non-in-transaction nodes as possible in the lock tree.

Our idea is first building individual trees from each transaction type, then merging them into a global lock tree. To build individual trees, we use a modified depth-first searching algorithm as shown in Algorithm 4. The algorithm always expands at first the unvisited successor with the highest probability. This follows the 2nd category above. It builds a tree covering all data items accessed in the transaction type.

Algorithm 4 *Given a transaction type* $T = \langle N, s, F, A, data, duration, prob\rangle$*, build a tree* t*:*

```
1: add $data(s)$ to $t$ as root
2: call procedure $build\_tree(s, t)$

1: procedure $build\_tree(n)$
2: $n.visited \leftarrow true$
3: while there is unvisited successor of $n$ in $G(T)$ do
4:    choose the unvisited successor $c$ with highest $prob(n, c)$
5:    if $data(c)$ is not a node in $t$ then
6:       add $data(c)$ as child of $data(n)$ in $t$
7:    end if
8:    call procedure $build\_tree(c, t)$
9: end while
```

For a transaction system with transaction type set $TS = \{trans_1, \cdots, trans_n\}$, after we build the trees from all $trans_i \in TS$, we merge them into a global lock tree. We sort the trees in the descending order of the probabilities of the transaction types, and get an array of trees $[tree_1, \cdots, tree_n]$. Then we build the global lock tree with a greedy algorithm, shown as Algorithm 5.

Algorithm 5 *Given an ordered array of trees* $[tree_1, \cdots, tree_n]$*, build a global lock tree* t*:*

```
1: initialize $t$ as a copy of $tree_1$
2: for $i \leftarrow 2$ to $n$ do
3:    for each node $d$ in $tree_i$ do
4:       if $d$ is not in $t$ then
5:          if the parent of $d$ in $tree_i$, $d'$, is in $t$ then
6:             add $d$ to $t$ as a child of $d'$
7:          else if $d$ has a child $d'$ in $tree_i$, and $d'$ is the root of $t$ then
8:             add $d$ to $t$ as the root and parent of $d'$
9:          else if $d$ has a child $d'$ in $tree_i$, and $d'$ is a non-root node in $t$ then
10:            find the parent of $d'$ in $t$, $d''$
11:            add $d$ to $t$ as a child of $d''$
12:         else
13:            add $d$ as the child of an arbitrary leaf node in $t$
14:         end if
15:      end if
16:   end for
17:   if there is a path $d$, $m_1$, $\cdots$, $m_k$, $n_1$, $\cdots$, $n_l$ in $t$ and a path $d$, $n_1$, $\cdots$, $n_l$ in $tree_i$ then
18:      change $n_1$ to be a child of $d$ {move the subtree rooted at $n_1$ to under $d$}
19:   end if
20: end for
```

Algorithm 4 initializes the global lock tree as the tree of the "most likely" transaction type (line 1), and processes other trees following the decreasing order of probabilities of transaction types. By doing this, we try to favor the transaction types with higher probability (category 1) and avoid non-in-transaction nodes in their local lock trees (category 4). The rules applied to add a new node into the global lock tree are reflected at line 5-14, which is to fulfill category 3 and makes the data items probably accessed by close states in a transaction type to be also close in the global lock tree. The final process to move the subtree (line 17-18) intends to enable the transaction type $trans_i$ to immediate lock n_i after it holds a lock on d without locking the possibly unnecessary data items like m_i in between.

It is obvious that the greedy algorithm works better if the transaction types have great difference in their probabilities and the most frequent transaction type have a dominant probability. When the probabilities of transaction types are very close, the global lock tree may be over biased to the transaction type with slightly higher probability.

After the global lock tree is built, building local lock trees for all transaction types is a trivial task.

5 Experiments

5.1 Experimental case derived from the TPC-C benchmark

In the experiments we would like to examine our tree locking on some transaction system which is close to real world applications, and compare to 2PL in terms of performance. Our experimental case are derived from transactions in the TPC-C Benchmark [1]. Similar to TPC-C, our transaction system has five transaction types, *New-Order*, *Payment*, *Order-Status*, *Delivery*, and *Stock-level*. Following the percentages of transaction mix in TPC-C with some simplification, we fix the probabilities of the above transaction types to 0.45, 0.43, 0.04, 0.04, and 0.04 respectively. There are nine tables involved, so the base data set includes these table names. As another simplification from TPC-C, we abstract all selections, updates, insertions, and deletions to one type of "write" operation on some individual table, and do not consider shared read lock.

To build each transaction type as defined in Section 2, we approximate each TPC-C transaction with a finite state machine. From the program of each transaction, we are only interested in embedded SQL statements and control statements like **if-else**, **for**. First, we transform each SQL query to a part of a transaction type graph $G(T)$. Most embedded SQL statements in TPC-C transactions operate on a single table, and only a couple are two table joins. For a SQL statement on a single table A, we add a new state n and let $data(n) = A$ (Figure

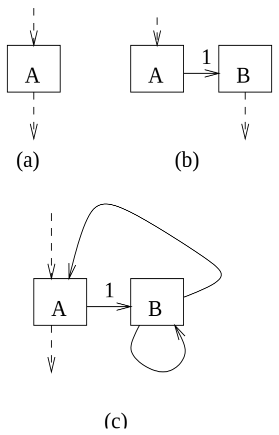


Figure 3: Transformations of queries

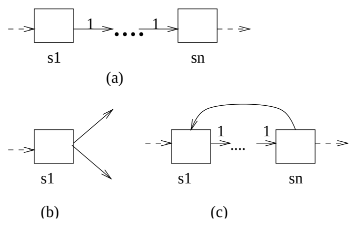


Figure 4: Transformations of control flows

3(a)). For a join on two tables A and B, if the tables are pre-selected based on their key values to have at most one row per table involved in the join, we add two nodes that access A and B respectively and let the one accessing B follow that accessing A (Figure 3(b)); otherwise, the join is transformed to Figure 3(c), and the probabilities on the unlabelled arcs in Figure 3(c) can be estimated from the table cardinality and query selectivity.

The transformation of control statements is straightforward, and the result is similar to a program flow-graph. A sequential execution of SQL statements s_1, $\cdots$, s_n is transformed to a lineal path through these nodes as in Figure 4(a). An **if** statement is transformed to a branch out in the graph (Figure 4(b)). Because the TPC-C benchmark already gives the probability for each branch, we can it apply here on each arc. And a loop statement can be transformed as Figure 4(c).

Example 5.1 *The new-order transaction in*

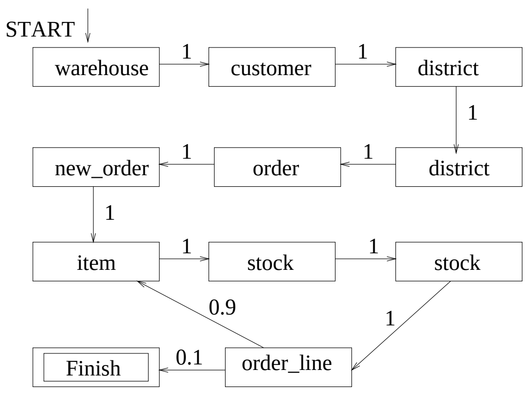


Figure 5: New-order transaction type

TPC-C can be transformed into a transaction type in Figure 5. To satisfy the requirement in Definition 2.1 that terminating state cannot have outgoing arcs, we introduce a new state, and assume it access a virtual data item "finish" with duration 0. The loop in the graph is from a ***for*** *loop in the TPC-C transaction program. So far the durations for other states have not been filled in yet.*

With those transaction types, we build a global lock tree following the procedures in Section 4 and local lock trees for all transaction types.

Next step is getting the durations of all the states. We create tables and insert rows into them in IBM DB2 as specified by the TPC-C benchmark. Because most tables have a primary key, we create unique indexes for these tables in addition. We run the SQL statements with plugged-in parameters in DB2, and get the durations on indexes and tables from query plan graphs that are provided by DB2 visualization tools.

In addition, we also consider partitioning a table to $table_1$, $\cdots$, $table_n$, with n as a parameter **partition_factor**. To accommodate indexes and table partitions, we change the graphs of transaction types and local lock trees. If the query at a state is a key-matching query, we change the state in transaction type as in Figure 6(a). If it is a full sequential table scan, we change it according to Figure 6(b). If keyed queries on the table are dominant in the transaction type, we change the tree node corresponding to the table as in Figure 6(c), otherwise we change the node as in Figure 6(d)[4].

5.2 Experiment setting and results

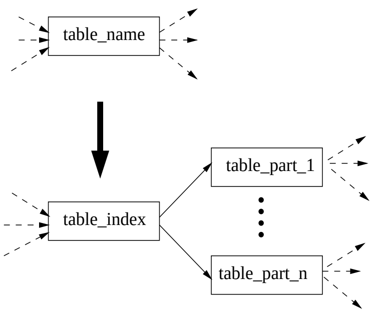
Our experiments are carried on a workstation with Intel P4 1.5GHz CPU, 256M RAM, and 20G hard disk. The operating system is Microsoft Window 2000 Professional. The programs for the experiments are written in Java language.

Similar to the TPC-C specifications, we simulate a number of user terminals with Java threads. On each user terminal, transactions run sequentially. Each transaction is a simulation on the finite state machine of the transaction type. When a transaction completes, the transaction type of the new one is chosen based on the probabilities of the transaction types. We also add some keying time before a transaction and some thinking time after it, following the TPC-C benchmark. Transactions from different user terminals are executed concurrently and compete for the locks. And the locks on tree nodes are basically semaphores whose **p** and **v** methods are implemented with Java synchronized methods. During the simulation, we use random number sequences to select the transaction type for the next transaction and to select the next state transition in a transaction respectively.

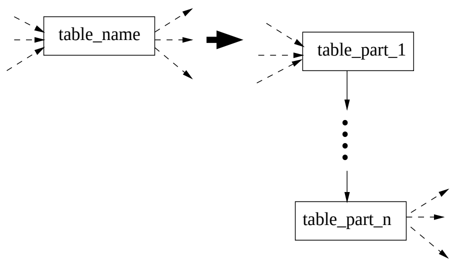
To make a comparison with 2PL, we also implement strict 2PL locking with semaphores in a similar way in Java. The code for the 2PL protocol are much simpler than the TL protocol because each transaction only needs to remember the locks it holds and release them when it reaches a terminating state. The extra work for the 2PL implementation is another thread working as deadlock detector. It checks a wait-for graph structure periodically, and aborts the youngest transaction involved in the deadlock when it finds a cycle in the wait-for graph.

Figure 7 and Figure 8 show some throughput

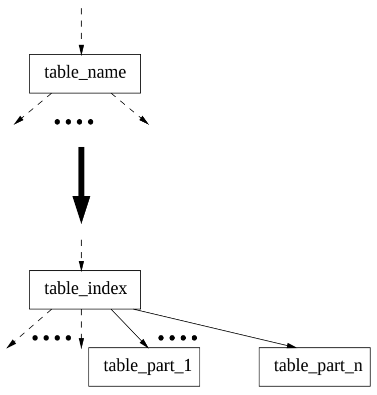
[4]We add partitions to graphs and lock trees after generating the original local lock trees from the graphs without partitions. The reason is that we believe it is necessary to have all partitions directly under their indexes in lock trees.



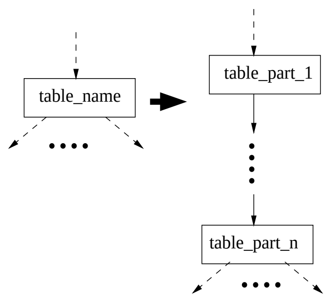
(a) transformation of state in transaction type for keyed query



(b) transformation of state in a full table scan query



(c) transformation of tree node for keyed query



(d) transformation of tree node for full table scan query

Figure 6: Adding index and partitions

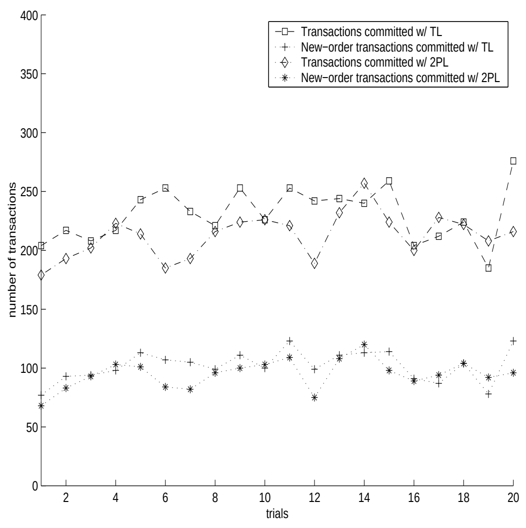


Figure 7: The throughput comparison between TL and 2PL: 10 user terminals, 20 minutes, partition_factor = 100

comparisons between TL and 2PL. Each trial is based on the same random number sequences for both 2PL and TL. The total numbers of all transactions committed on all user terminals for each trial are shown in the two figures. We also show for both locking protocols the numbers of committed transactions of New-Order transaction type, which has a high probability. Under 2PL, the abort rate is about 1%. And we do not show the numbers of aborted transaction in the figures.

From the experimental results, we can induce that the performance of TL can compete with that of 2PL in our application context. In most of our trials, TL outperforms 2PL, although the difference is not enough to convince the overall performance superiority of TL. We believe that the early "unlocking in TL" enables a better concurrency than strict 2PL, which can compensate the CPU overhead for TL locking/unlocking.

6 Conclusion and future work

Our preliminary experiments show that the TL protocol can produce good throughput and can

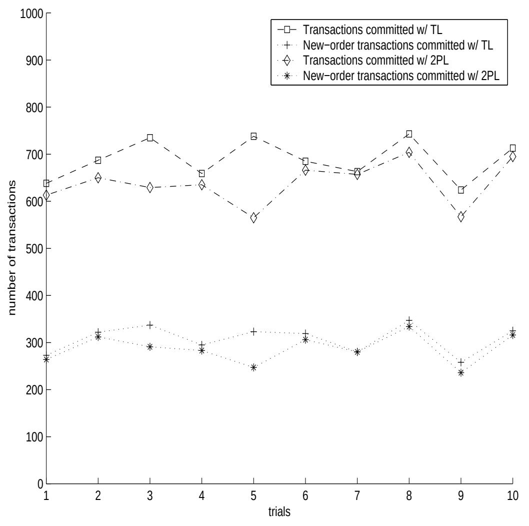


Figure 8: The throughput comparison between TL and 2PL: 10 user terminals, 60 minutes, partition_factor = 100

compete with the widely used 2PL protocol in the respect. This is very very promising for the usage of the TL protocol in compiled database applications. First, we can generate the lock trees and unlockable sets at compiled time. Second, for query processing, we can expand the compile-time query optimizer in [12] to generate the locking and unlocking code for each operation at compile time as well. Although such code is not trivial, it is still manageable, and can still produce good throughput as shown in the experiments. Moreover, because of the properties of TL, serializability is guaranteed and deadlocks are prevented. Not only are the transactions executed correctly, but also the costly recovery for transactions aborted for deadlock resolution becomes unnecessary. Therefore, we believe TL is a good choice for the "begin transaction/commit" transaction model for compiled database applications.

Future work includes an in-depth research on the measurement of lock tree fitness. We hope to find some better guidance to build the trees. Another direction of future research is how to extend our transaction model to multilevel transaction model [13] [14] to allow independent object level transaction management and physical level transaction management for compiled database applications.

References

[1] Transaction Processing Performance Council. URL http://www.tpc.org.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, 2001.

[4] K. P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in Database System. *Communications of the ACM*, 19(16):624–633, 1976.

[5] Pasal Felber and Michael K. Reiter. Advanced Concurrency Control in Java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002.

[6] Zvi Kedem and Abraham Silberschatz. Controlling Concurrency Using Locking Protocolsi (Preliminary Report). In *Proceedings of 20th Symposium on Foundations of Computer Science*, pages 274–285, October 1979.

[7] Zvi M. Kedem and Abraham Silberschatz. Locking Protocols: From Exclusive to Shared Locks. *Journal of ACM*, 30(4), October 1983.

[8] Dennis Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems (TODS)*, 13(1):53–90, March 1988.

[9] Abraham Silberschatz and Zvi Kedem. Consistency in Hierarchical Database Systems. *Journal of ACM*, 27(1), January 1980.

[10] Abraham Silberschatz and Zvi M. Kedem. A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs. *IEEE Transactions*

on Software Engineering, 8(6):558–562, November 1982.

[11] Lubomir Stanchev and Grant Weddell. Index Selection for Compiled Database Applications in Embedded Control Programs. In *Proceedings of CASCON 2002*, pages 156–170, Toronto, Canada, September - October 2002.

[12] David Toman and Grant E. Weddell. Query Processing in Embedded Control Programs. In *2nd International Workshop on Databases in Telecommunications*, number 2209 in Lecture Notes in Computer Science, pages 68–87. Springer-Verlag, 2001.

[13] Gerhard Weikum. Principles and Realization Strategies of Multi-level Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[14] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann, 2001.

[15] Mihalis Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of ACM*, 29(3):718–740, July 1982.