ALTRUISTIC LOCKING: A STRATEGY FOR COPING WITH LONG LIVED TRANSACTIONS

Kenneth Salem
Hector Garcia-Molina
Rafael Alonso

CS-TR-087-87

April 1987

ALTRUISTIC LOCKING: A STRATEGY FOR COPING WITH LONG LIVED TRANSACTIONS

Kenneth Salem
Hector Garcia-Molina
Rafael Alonso

Department of Computer Science
Princeton University
Princeton N.J. 08544

ABSTRACT

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the completion of shorter and more common transactions. To alleviate these problems we propose an extension to two-phase locking, called altruistic locking, whereby LLTs can release their locks early. Transactions that access this released but uncommitted data run in the wake of the LLT and must follow special locking and commit rules. Additional performance improvements can be obtained if the LLT predeclares its access set (this is only an option). Altruistic locking guarantees serializability and allows transactions to access the database in any order. In addition to presenting the protocol, we discuss how LLTs can be automatically analyzed in order to extract the access pattern information used by altruistic locking.

ALTRUISTIC LOCKING: A STRATEGY FOR COPING WITH LONG LIVED TRANSACTIONS

Kenneth Salem
Hector Garcia-Molina
Rafael Alonso

Department of Computer Science
Princeton University
Princeton N.J. 08544

1. Introduction

A *long lived transaction* (LLT) has a long duration compared to the majority of other transactions because it accesses many database objects, it has lengthy computations, it pauses for inputs from users, or because of a combination of these factors [Gray81a]. For example, in a shipping company, a transaction to schedule and lay out the routes for next day's deliveries (based on today's received orders) can easily take several hours. Similarly, a transaction that scans a large file, say to reorganize the index of an indexed-sequential file or to add a field to all records of a given type, will again take a substantial amount of time. Other examples of LLTs include transactions to process purchase orders, to process insurance claims, print monthly account statements at a bank, and so on.

In most cases LLTs present serious performance problems. To execute the LLT as an atomic action, the system usually locks the objects accessed until the transaction commits, and this usually occurs at the end of the transaction. Other transactions wishing to access the same objects must face substantial delays. In addition, the likelihood that a transaction (LLT or not) has to be aborted is higher than usual when LLTs are present precisely because of their long duration, i.e., LLTs are more likely to be involved in a deadlock or a failure.

In this paper we propose a strategy for ameliorating the adverse effects of LLTs. The basic idea is to allow LLTs to release their locks early, once it is determined that the data the locks protect will no longer be accessed. Unlike other approaches (which will be surveyed shortly), our strategy still guarantees serializable executions and places no restrictions on the way data must be accessed (e.g., transactions are not forced to scan the database sequentially or to traverse it down a tree).

To illustrate our approach, consider a set of transactions executing without concurrency control against a database. Figure 1a is a depiction of the access pattern of four such hypothetical transactions. Each data object is represented along the vertical axis, and the horizontal axis represents time. Each transaction is represented by a solid line. A transaction T_i accesses a data object x at time t_i if the line for T_i passes

through a black dot at the point (t_i,x). For example, transaction T_L accesses data object e at time t_i in the Figure. Note that the execution schedule represented in the Figure is serializable. By inspection we can see that the schedule is equivalent to the serial execution of transactions in the order $T_A T_L T_B T_C$.

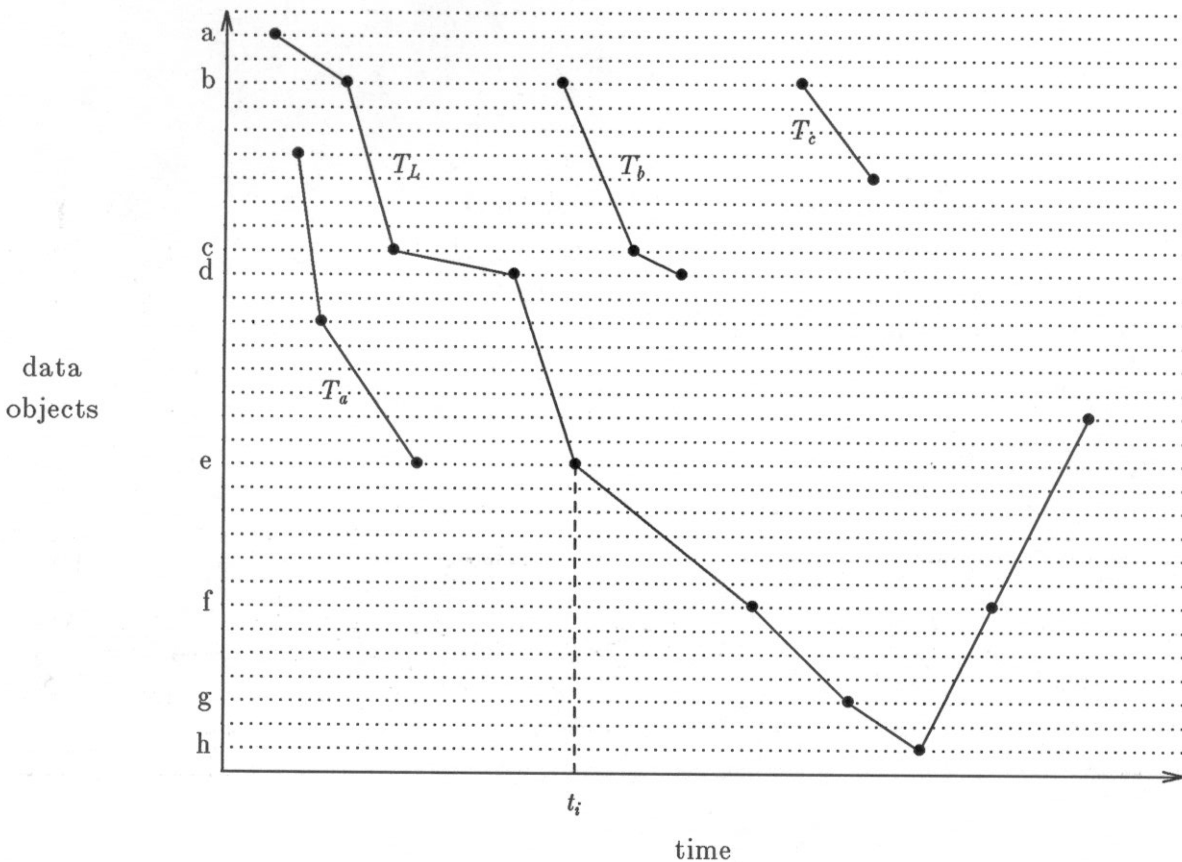

Figure 1a - Transaction Data Access Patterns

Consider what happens if the same execution schedule is attempted under a two-phase locking concurrency control manager that has no knowledge of the access patterns of the transactions. Since nothing is known in advance about the transactions' behavior, locks must be held until transactions are finished (committed). Figure 1b shows the same execution schedule, with the data objects that would be locked by T_L boxed for the duration of the lock. From the Figure we can see that, while T_L and T_A could run concurrently, T_B and T_C would be forced to wait on the locks of T_L until T_L completed. If T_L were a LLT, this delay could be lengthy.

The concurrency control manager could do better if it was provided with some information about the access pattern of T_L. Access pattern information is any information that describes a transaction's interaction with the database. For example, it may be information like: "during its lifetime, transaction T will access database objects in

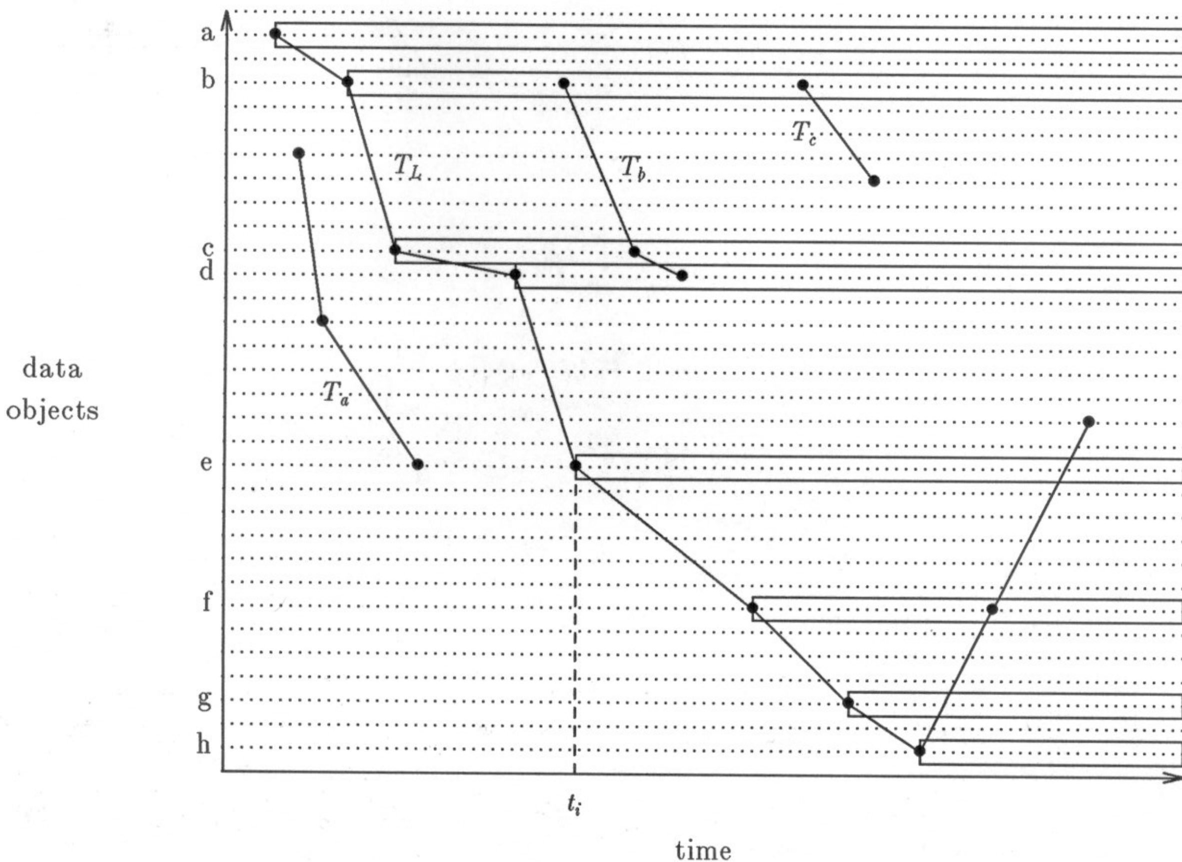

Figure 1b - Transaction Data Access Patterns Showing Locks of T_L

the set $\{a,b,c,d,e\}$." Or it might be information like: "if T accesses data object a once, it will not access a again."

We will distinguish between two general types of access pattern information, *positive* and *negative*. Negative access pattern information about a transaction T describes objects that will *not* be accessed by T. Conversely, positive information describes objects that *will* be accessed. This distinction will be useful in our discussion; we do not intend these to be rigorously defined classes of information.

If we assume that T_L is known never to access data objects more than once (negative access pattern information), the concurrency control manager can execute T_B without delay since it remains entirely within the *wake* of T_L. The wake of a transaction is the set of data objects that it has locked (and accessed) that it will not access again. Since T_B remains in the wake of T_L, it can appear after T_L in a serialization of the execution schedule.

Similarly, if the concurrency control manager knows the complete access set† of T_L (the set of database objects that will be accessed by T_L), it can allow T_C to run without delay as well. Although T_C does not remain in the wake of T_L, neither does it access

any data which T_L will later require. So T_C can appear after T_L in a serialization. Of course, we must guarantee that no other transaction sees the results of T_C and makes them available to T_L. As we shall see, a simple way of doing this is to let T_C perform certain actions *on behalf of* T_L.

With our strategy transactions can optionally predefine their access sets or allow other transactions in their wake. These actions do not benefit the transaction directly, but may reduce the delays of other transactions. Hence we call our protocol *altruistic locking*. Of course, altruistic locking is most effective if the altruistic transaction is a LLT that would otherwise cause very long delays.

Before presenting the details of altruistic locking, it is worthwhile to briefly review the alternate solutions that have been proposed. These solutions tend to fall into two broad categories. In the first are solutions that relax the *serial consistency* requirement of atomic transactions [Garc83a, Lync83a, Giff85a, Garc87a]. Intuitively, the idea here is to split up LLTs into shorter steps that are executed as individual transactions. After each step, locks can be released, and other transactions can then access the resources used by the LLT. The price paid is that transactions no longer observe data that could have been produced by a serial schedule. This may be a reasonable price to pay in some applications but not in others. Altruistic locking is different from these protocols because it does guarantee serializability.

In the second category are solutions that restrict the way transactions can access the database [Kede80a, Baye87a, Ahuj87a]. If transactions access the database in a known way, e.g., a sequential scan or down a tree, then it is possible to allow other transactions to see parts of the data accessed by a LLT before its completion, and at the same time guarantee serializability. (Some of these protocols were not explicitly developed for LLTs, but they can be helpful nonetheless.)

For example, in non-two-phase locking protocols [Kede80a] the database is modeled as a tree or as a directed acyclic graph. Transactions must access the database entities in the order imposed by the graph. Because of this, transactions can release locks before commit time without endangering serializability.

Similarly, a random-batch LLT [Baye87a] scans the entire database examining each entity once. The entities can be examined in any random order, so that the order can be specified by the system. This flexibility lets the system in essence add any object to the wake of the LLT. Hence, short transactions can all execute within the wake. (If the short transaction wants to access an entity not currently in the wake, the system forces the LLT to process the entity first, making it part of the wake.) In altruistic locking we also use the idea of running within the wake of a LLT, but we do not restrict ourselves to random-batch LLTs.

---

† Knowledge of the complete access set provides both positive and negative access pattern information since objects in the complement of the set will not be accessed.

Of the proposed concurrency control mechanisms, the one that is most similar to ours is one by Lafortune and Wong [Lafo84a]. Their protocol makes use of access pattern information which is obtained through the use of a *declare* operation. Declare operations are used in addition to the normal lock and unlock operations. Before locking a database entity, a transaction must declare it. The system decides whether to grant declare and lock requests based on a transaction precedence graph that it maintains. Transactions are allowed to begin releasing locks once they have finished declaring entities, a weaker restriction than the normal two-phase requirement. The major difference between that protocol and those presented here is that altruistic locking does not require *all* transactions to provide access pattern information to the concurrency control manager. Instead, transactions need only provide access information if it is likely to allow greater concurrency (as is the case with LLTs). In addition, altruistic locking does away with the precedence graph of [Lafo84a] and instead requires some transactions to acquire locks on behalf of other transactions.

We wish to emphasize that altruistic locking *should not be viewed as a new concurrency control mechanism*. It is simply an extension to two-phase locking. However, we feel that this is its greatest strength. Any transaction, at any time, can run under the conventional two-phase locking rules, without having to perform any special operation and with exactly the same locking overhead if would have had in a conventional system. Thus, if a transaction performed well under two-phase locking, it can continue with the same protocol. Only LLTs and short transactions that have performance problems need to concern themselves with the extended locking rules.

In Section 2 we present the details of altruistic locking. In Section 3 we study its performance and give some guidelines for its use. Crash recovery issues are discussed in Section 4. In discussing altruistic locking we will assume that the appropriate access pattern information is available to the system's concurrency manager. In Section 5 we discuss how this information might be obtained and made available.

2. Altruistic Locking

We will first describe the basic altruistic protocol, which takes advantage of negative access pattern information. In particular, it is able to take advantage of the knowledge that a transaction no longer needs access to a data object that it has locked. We say that that data object can be *released* (as opposed to "unlocked") by the transaction. A release operation can be thought of as a conditional unlock operation. Other transactions will be permitted to access the data, but by doing so they in effect agree to abide by certain restrictions that ensure that all transactions see a consistent database state.

We will then discuss three extensions to the protocol. The first allows transactions to release data objects that they have not first locked. The second extension makes use of positive access pattern information (i.e. knowledge of what objects will be accessed

during a transaction's lifetime). The last extension is for hierarchical locking. Here access pattern information can be defined at different granularities, e.g., we can release an entire relation in a single operation. We only deal with exclusive locks in this paper. The extension to shared and exclusive locks is presented in [Sale87a].

For clarity of discussion, we will consider a transaction to be a sequence of operations of one of three types, **L** (lock), **R** (release), or **A** (access). (We will not use explicit unlock operations. A transaction will be assumed to unlock all locks after its last operation.) Database entities are represented by lower case letters, e.g. a. Each operation operates on a database entity. All operations of all transactions are totally ordered by an *execution schedule* S which describes when each operation occurs.

A transaction T_i is *well-formed* in S if it accesses no database entity without first locking it, it accesses no database entities it has already released, and it releases no entities that it has not first locked. A schedule is said to be simply, or basically, altruistically locked (BAL) if it has the following properties:

1) No two transactions hold locks on the same data object simultaneously unless one of the transactions locked *and released* the object before the other locked it. In this case, we say that the later lock-holder is in the wake of the releasing transaction.

2) If a transaction is in the wake of another transaction, it must be completely in the wake of that transaction. In other words, if T_x locks a which has been released by T_y, then anything currently locked by T_x must have been released by T_y before it was locked by T_x. (This requirement is relaxed once T_y finishes.)

BAL schedules are serializable. This is proved for the more general case of both shared and exclusive locks in [Sale87a]. We next develop a simple locking protocol for well-formed transactions that results in BAL schedules. We will assume in doing so that the system maintains information about database entities and transactions. In particular, for each database entity a the system maintains:

- $L(a)$: the set of transactions that have locked a
- $R(a)$: the set of transactions that have released a (note that if transactions are well-formed, $R(a) \subseteq L(a)$)

For each active transaction T_x the system maintains:

- has-lock(T_x): a boolean set true when T_x obtains its first lock
- wake(T_x): the set of transactions in whose wake T_x is running
- $\delta(T_x)$: the set of database entities that have been locked by T_x

This information is updated when transactions begin and end, and when they release or lock database entities. When transaction T_x begins, has-lock(T_x) is set false and wake(T_x) and $\delta(T_x)$ are emptied. When T_x completes all of its operations, its unlocks and un-releases are performed (using $\delta(T_x)$), and it is removed from *wake*(T_y) of each active T_y. (The commit point may occur later. See Section 4.) To release an

entity a, T_x simply adds itself to $R(a)$. When T_x requests a lock on a, the protocol in Figure 2a is followed.

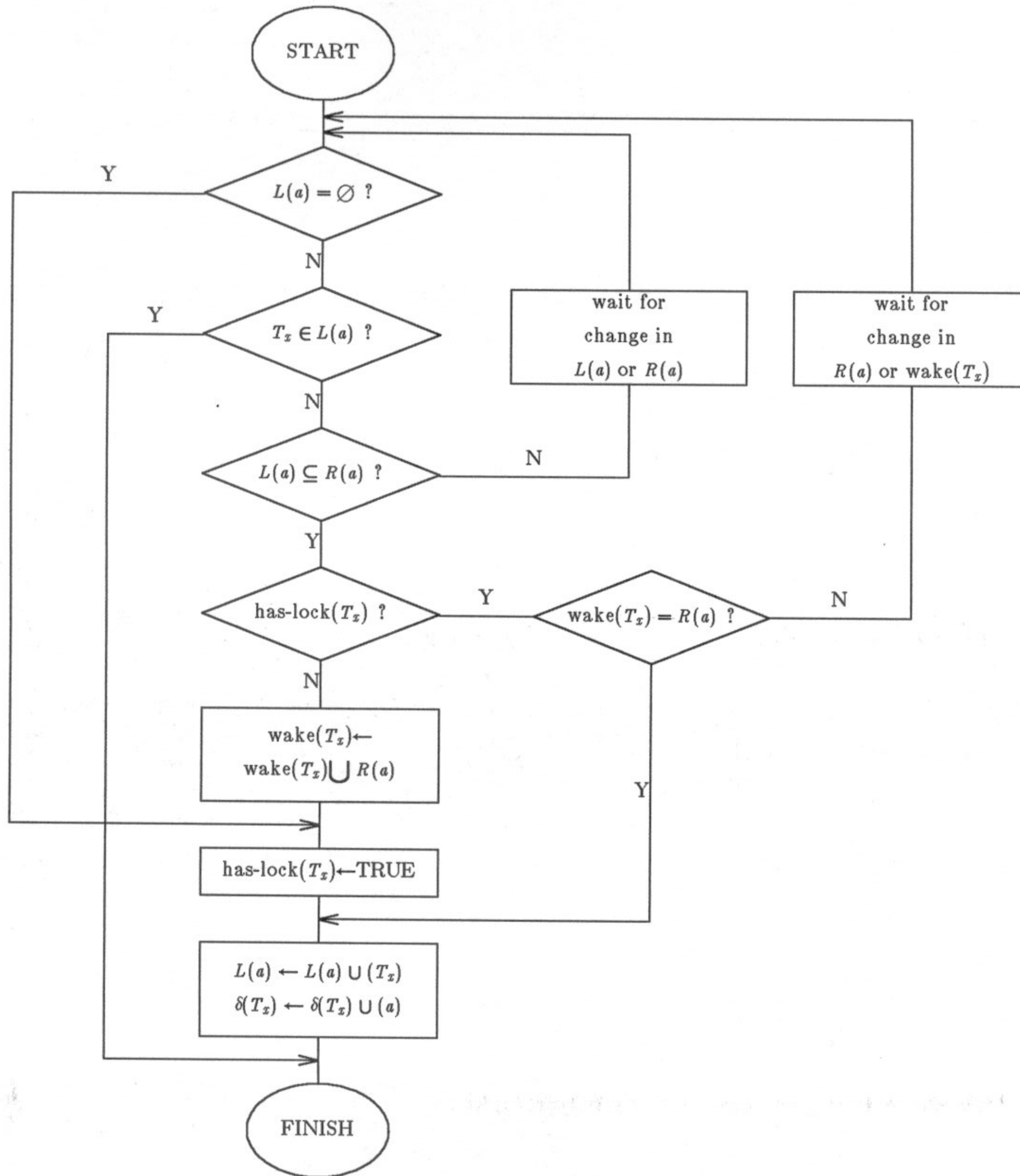

Figure 2a - BAL protocol

The test "$L(a) \subseteq R(a)$?" in Figure 2a ensures that T_x does not lock an entity that has been locked but not released by some other transaction (first property of BAL schedules). The tests "has-lock(T_x)?" and "wake(T_x) = $R(a)$?" ensure that T_x remains either completely inside of or completely outside of the wakes of other active transactions (second property of BAL schedules).

An appealing aspect of this protocol is that it is fully compatible with two-phase locking. 2PL and BAL transactions can coexist in the same system provided that 2PL

transactions do not take advantage of releases made by altruistic BAL transactions. In Figure 2a, this means that a 2PL transaction that failed the test "$T_x \in L(a)$?" when requesting a lock on a would proceed directly to a wait state without checking whether a's locks had been released. A 2PL transaction T_{2PL} need not have wake(T_{2PL}) maintained for it at all, since it will never enter the wake of another transaction. Furthermore, the cost of updating the wake list is paid only by those BAL transactions that release entities or that lock released entities. Thus short BAL transactions that do no releases of their own only pay extra concurrency control costs if they have a possibility of increased concurrency.

As an example of the use of the BAL protocol, consider a hypothetical banking database that contains a (large) relation *ACCOUNTS*. T_L is a long-lived transaction that is accessing a large number of entities (e.g. tuples, pages) from *ACCOUNTS*. Suppose that since T_L needs to access each entity only once, so that as it finishes with each entity it releases (**R**) it.

Now consider a short update transaction T_S that accesses a single database entity, an entity in *ACCOUNTS*. If a is not locked by T_L, T_S can access it without restriction. If a is locked by T_L, T_S must wait until the lock is removed or until the tuple is released by T_L (by the first BAL restriction). T_L releases a as soon as it is finished using it, possibly long before it would have released a under two-phase locking. Once a has been released, T_S may immediately access it. By doing so, T_S enters the wake of T_L. Had T_S desired to make additional database accesses, they could be made without delay, *provided that they too lie in the wake of* T_L.

Incidentally, note that altruistic locking reduces the likelihood of blocking, but it is still possible for short transactions to block for long periods of time waiting for a LLT. If these short blocked transactions hold other locks, it may be best to abort them. (A timeout mechanism on locks could be used.) This allows other short transactions that do not require data locked by the LLT but do need the data held by the short transaction to proceed. (The same is true in conventional two-phase locking.) We will not pursue this idea further in this paper.

2.1. Extending the Protocol

Allowing access to the wake of LLTs is most likely to be beneficial to simple transactions with few database accesses. Transactions with more complicated access requirements are less likely to remain in the wake of the LLT. Several extensions to BAL are possible if greater concurrency is required by the application. In the following discussion we consider three possible extensions.

2.1.1. Enlarging the Wake

Our definition of well-formed transactions requires that a transaction hold a lock on a database entity before it can release that entity. Thus a transaction's wake is limited to entities it has itself locked. In fact, this restriction is not needed to maintain the serializability of a BAL schedule.

We can relax our definition of "well-formed" so that transactions need not lock an entity before releasing it. Transactions are permitted to release any entity, even if it is currently locked (or released) by another transaction, provided that they have first locked at least one (any) database entity. We will refer to such releases without locking as *extended releases*. Note that a well-formed transaction still may not access any entity that it has released or that it has not locked.

Consider once again our hypothetical banking database. Assume that the database contains another relation *NAME_TO_NUMBER* (used to map customer names to account numbers) in addition to *ACCOUNTS*. Since our LLT T_L uses only the *ACCOUNTS* relation, it could release (**R**) (without locking) entities in *NAME_TO_NUMBER* even if those entities are locked by other transactions. By doing so T_L can expand its wake substantially. Now our short transaction T_S, after accessing a, is free† to access any other entities in T_L's wake, including the released entities in *NAME_TO_NUMBER*. We can implement such a change by making one small modification to the BAL protocol (Figure 2a). With extended releases, a transaction that passes the first test ("$L(a) = \emptyset$?") must proceed next to the test "has-lock(T_x)?" instead of immediately obtaining the lock. In addition, T_x must ensure that has-lock(T_x) is true before it makes any extended releases.

These two changes ensure that released but not locked entities are included in a transaction's wake. Unfortunately, an expanded wake may make it more difficult for other transactions to run *outside* the wake. This is not a problem when the wake is restricted to entities that have already been locked by the releasing transaction. These entities would be inaccessible to other transactions anyways (because they are locked), so releasing them can only be beneficial. In Section 3, we present a simple performance model and discuss in more detail the issue of when enlarging the wake will be beneficial.

2.1.2. Marking

A second way to boost concurrency is to make use of positive access pattern information, i.e., advance knowledge of which entities will be accessed by a transaction. With such information we can often allow short transactions in a LLT's wake to access data outside the wake without jeopardizing serializability. In addition, transactions outside of the wake can be allowed to enter it under certain conditions. These events will still involve expanding the LLT's wake, but the expansion will be done "on demand"

† As always, T_S must first obtain a lock on any entity it accesses.

from short transactions. The wake can be expanded to include only those entities that are needed by the short transaction, thus minimizing conflicts with transactions running outside the wake.

To take advantage of positive access pattern information, we define a fourth transaction operation, the mark (**M**). Like other operations, a mark operates on a single database entity. Like release (**R**) operations, marks do not conflict, i.e., different transactions can simultaneously hold marks on the same entity. Transactions are not required to use mark operations. Those that do will be termed *marking* transactions.

To be well-formed, a marking transaction must meet our original criteria for well-formed transactions. In addition, a well-formed marking transaction must satisfy two additional criteria:

1) A well-formed marking transaction may not lock (**L**) any entity without first marking (**M**) it.

2) Once a well-formed marking transaction has released (**R**) any entity, it may no longer mark.

In schedules we will term XAL (for "extended altruistic locking"), interactions among non-marking transactions are the same as in BAL schedules: a transaction must be either completely within another's wake, or completely outside of it (while both transactions are unfinished). However, transactions are allowed to be both inside and outside the wakes of marking transactions under certain conditions. These conditions can be stated informally as follows:

1) A transaction T_x in the wake of a marking transaction T_m may access an entity a outside of the wake of T_m provided that a has not been marked by T_m, and that T_x first releases (**R**) a on behalf of T_m, i.e., as if T_m had released a itself.

2) A transaction T_x outside the wake of a marking transaction T_m can access an entity a in T_m's wake provided that three conditions are met. First, T_x must not hold locks on any entities marked by T_m. Second, T_x must not have released (**R**) any entities itself. And finally, T_x must first release on behalf of T_m all entities on which it currently has locks if they are not already so released.

Thus, when a transaction attempts to leave the wake of a marking transaction, it really expands the wake of the marking transaction (by releasing on its behalf) instead. When a transaction enters a marking transaction's wake, it expands the wake to include all of its locked data.

XAL schedules are serializable [Sale87a]. As we did for BAL schedules, we will next develop a protocol that produces XAL schedules. The protocol will employ the same information used by the BAL protocol. In addition it will make use of a list $M(a)$ for each entity a and a boolean has-release(T_x) for each transaction T_x. $M(a)$ is a list of transactions that have marked a, while has-release(T_x) indicates whether T_x has made its first release.

Transactions begin, end, and release entities on their own behalf just as they did under the BAL protocol. Under XAL, $M(a)$ is cleared and has-release(T_x) is set false when T_x begins. T_x marks a by adding itself to $M(a)$. T_x can release a on behalf of T_m by adding T_m to $R(a)$. T_x follows the protocol in Figure 2b if it wishes to lock a. Note that the XAL protocol is an extension of the BAL protocol, and so Figure 2b is an extension of Figure 2a. The dashed steps in Figure 2b correspond to steps in Figure 2a.

Figure 2b makes use of a shorthand notation for two lists of transactions, $A(x,a)$ and $B(x,a)$. $A(x,a)$ represents "wake(T_x) $-$ wake(T_x) $\cap R(a)$", i.e., those transactions in whose wake T_x is running that have not released a. For T_x to lock a it must first release a on behalf of transactions in $A(x,a)$ to satisfy the first condition on XAL schedules. $B(x,a)$ represents "$R(a)$ $-$ wake(T_x) $\cap R(a)$". These are transactions that have released a but in whose wake T_x is not (yet) running, i.e., T_x will enter the wakes of transactions in $B(x,a)$ by locking a. Before locking a, T_x must first satisfy the second condition on XAL schedules.

The extended segment of the protocol calls for T_x to first check whether it is attempting to move outside any wakes ("$A(x,a) = \varnothing$?"). If so the wakes are expanded if possible to satisfy the first condition on XAL schedules. If this can be accomplished T_x then checks whether it is attempting to enter any new wakes ("$B(x,a) = \varnothing$?"). If so and T_x has not already made releases of its own, entities already locked by T_x must first be brought into the new wakes as prescribed by the second condition on XAL schedules.

In our banking example, the short update transaction T_S would have more freedom of access if T_L was a marking transaction. In our original example, T_S would not have been able to access an entity n in *NAME_TO_NUMBER* if it had already accessed an entity a released by T_L in *ACCOUNTS*, and vice versa. Under XAL, T_S is free to access both n and a by first releasing n on behalf of T_m.

2.1.3. Hierarchical Releases and Marks

For simplicity we have assumed so far that each entity is individually locked, released, or marked by a transaction. Along the lines of hierarchical locking [Gray76a], it is relatively easy to extend releasing and marking to hierarchies.

Due to space limitations, we will restrict ourselves to illustrating the basic idea on a two level hierarchy, say records and files. Even with two levels, there are a number of possible variations; we will only describe the simplest.

For lock releases, suppose that every record x has a parent file $file(x) = y$. Record x has a lock, $L(x)$, that can be in one of two states, free or locked. If $L(x)$ is locked, then $T(x)$ is the transaction that has locked it. File y also has a lock, $L(y)$, and it can be in states free or released. If $L(y)$ is released, then $S(y)$ is the set of transactions that have released it.

If $L(y)$ is free, then record x is simply locked or free, according to $L(x)$. If $L(y)$ is released, then all its records are released. That is, if $L(x)$ is locked, then record x has

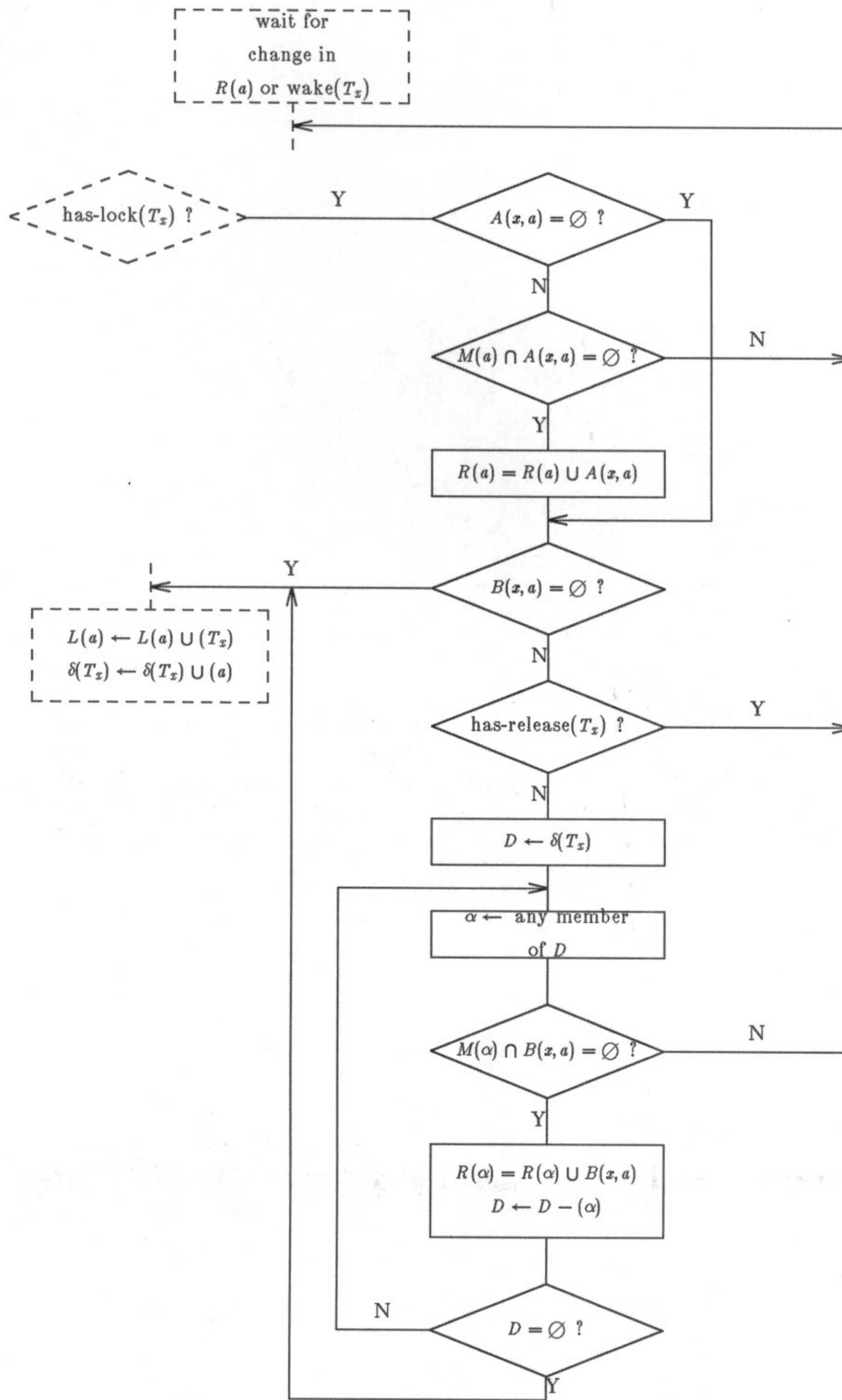

Figure 2b - XAL protocol

been locked are released. If $L(x)$ is free, then record x has been released without being locked, as discussed in Section 2.1.1.

The locking rules are exactly as before, except that an entire file can be released in a single operation ($L(y)$ is set to released). The releasing overhead can clearly be reduced substantially. Furthermore, the storage cost of the release information is also much less. For example, if a LLT has released a large portion of the database we only need to store a release set $S(y)$ for each file and not each record.

Hierarchical marking could be done in a similar fashion. A marking transaction could mark the files it will access. During the locking protocol, if another transaction wishes to see if a record is marked, it simply checks if the file it belongs to is marked. (A variation would be to mark at both levels. Then a record would be marked if it or its file are marked.) The advantages of hierarchical marking are similar to those of hierarchical releasing.

3. A Simple Performance Model

As mentioned earlier, releasing a locked entity can only be beneficial. However, extended releases and marking could be counterproductive in some cases. That is, extending a LLT wake can benefit some transactions but can slow down others. If more transactions are hurt than are helped, then extended releases and marking may not improve performance at all. In order to understand the tradeoffs involved, in this section we will explore a simple performance model. The objective is not to accurately predict the performance of altruistic locking; rather, the goal is to understand the situations in which extended releases and marking can be helpful.

We start with extended releases. Suppose that we look at the database at a given instant of time and determine the number of entities that are locked or released by a given LLT. In particular, let

k = the number of entities locked by the LLT

r = the number of entities locked and released by the LLT

s = the number of entities released but not locked by the LLT (extended releases)

n = number of objects not accessed by the LLT in any way

The situation is illustrated in Figure 3a.

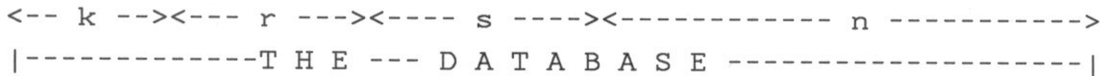

Figure 3a

Consider now a short transaction T that accesses two random entities. If T

accesses one of the s extended release entities and one of the n entities not accessed by the LLT, then T will be blocked (it cannot run partially in and out of the wake). This would not have occurred if the LLT had not done extended releases. In that case, both entities would not have been accessed by the LLT and T would proceed unaffected. The probability that this undersirable blocking occurs, P_{bad}, is

$$P_{bad} = 2\frac{s}{(k+r+s+n)}\frac{n}{(k+r+s+n)}.$$

On the other hand, if T accesses one of the r locked and released entities and one of the s extended release entities, then it will not be blocked. If the LLT had not done the extended releases, then T would have blocked. The probability of this desired event is

$$P_{good} = 2\frac{r}{(k+r+s+n)}\frac{s}{(k+r+s+n)}.$$

Note that all other types of accesses by T would have identical outcomes regardless of whether extended releases were used or not. For instance, if T accesses two of the s objects, then it will not block in either case. Similarly, if it accesses any of the k locked objects it will block. This implies that the two probabilities given above define the desirability of extended releases. If $P_{good} > P_{bad}$ then short transactions like T are more likely than not to proceed in parallel to the LLT. If the reverse is true, then extended releases are counterproductive.

From the equations, it is clear that P_{good} will be greater than P_{bad} when $r > n$. This means that a LLT should only perform extended releases after it has normally released more data than what remains untouched in the database. At that point the wake is large enough that it pays to make it even larger so that more transaction can execute in it.

Of course, since our model is very simple, this must be taken as a very general rule of thumb. In particular situations where the access pattern of short transactions is known to be non-random, it may pay off do do extended releases earlier. For example, suppose that a LLT has locked half a relation and will no longer need any portion of it. It may be better to release both halves (a simple operation with hierarchical releases) if it is likely that a short transaction that accesses one half will access the other.

A similar analysis can be performed for marking. Suppose that at a given instant the situation is as follows:

m = number of objects that are marked by a marking LLT
n = number of objects that are NOT marked
k = number of objects currently locked by the marking LLT
r = number of objects that have been relased by the marking LLT
s = number of objects that have been released on behalf
of the LLT by short transactions.

Graphically, the situation is as shown in Figure 3b. (The a and b points will be explained in a moment.)

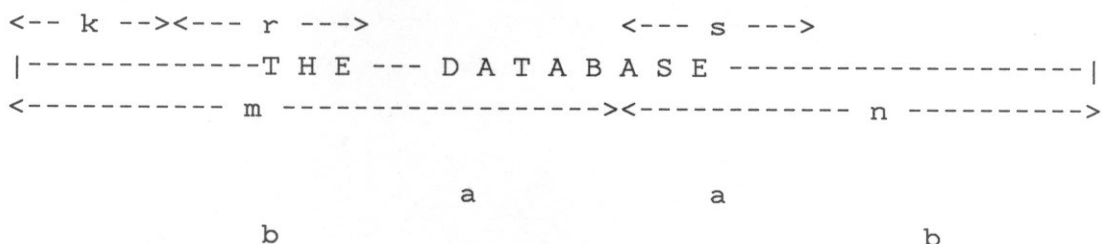

Figure 3b

Consider again the short transaction T that accesses two random entities. If T accesses entities at points a in the diagram, it will be blocked (the marked entity cannot be added to the wake). This would not have occurred if T had not been a marking transaction. The probability that this undesirable blocking occurs is

$$P_{bad} = 2\frac{m-r-k}{(m+n)}\frac{s}{(m+n)}.$$

On the other hand, if T accesses entities at points b in the diagram, T will proceed, and this would not have happened without marking. This desirable event occurs with probability

$$P_{good} = 2\frac{r}{(m+n)}\frac{n}{(m+n)}.$$

Once again, these are the only two situations where the marking protocol makes a difference. Thus, marking will be beneficial if $P_{good} > P_{bad}$.

There are several situations where $P_{good} > P_{bad}$. For example, the condition is true if $m<n$ and $r \geq s$. This occurs when the LLT marks less than half of the database and when releases on its behalf are less than its own releases. This last condition is reasonable because the likelihood that a short transaction releases on behalf of T is proportional to r.

A second, possibly more important case where $P_{good} > P_{bad}$ occurs when the LLT progresses at a uniform rate. During the first half of the LLT's life the factor $m-r-k$ in P_{bad} will be greater than the factor r in P_{good}. However, during the second half, the opposite is true. Since the additional factor n in P_{good} will always be greater than the corresponding s factor in P_{bad}, the overall contribution of marking will be positive. To see the magnitude of the positive contribution, assume that there is a time variable t that goes from 0 to 1 during the lifetime of the LLT. Say that $k = 0$ (worst case), $r = mt$ and $s = nt$ (also pessimistic). Then the top factor of P_{good} is $2mnt$ and of P_{bad} is $2mnt(1 - t)$. Clearly, P_{good} will always be greater than P_{bad} (except at $t = 0$), and as time

progresses the difference becomes more significant.

As with the extended release case, there could be additional situations where marking pays off if transactions have non-random patterns.

4. Crash Recovery Issues

If a short transaction runs in the wake of a LLT, it may see uncommitted updates. Note that if a LLT releases only objects that it has read, and no objects that it has written, this is not a problem. Thus LLTs can permit some access to their wake without jeopardizing their own atomicity. An extreme example is a *read-only* LLT, which could provide complete access to its wake without atomicity problems. However, in some cases it may be desirable for a LLT to release its updates. This means that a short transaction that sees the such an update cannot commit until the LLT commits. If the LLT aborts, then any transaction that ran in its wake must be aborted.

Achieving this is not difficult. (It may, however, be inconvenient to the end users; see below). For each transaction T the system maintains a list $commit(T)$ that includes all transactions that ran in the wake of T, finished all their actions, and will commit together with T. When a transaction T_x finishes all its actions, its $wake(T_x)$ lists all the active transactions in whose wake it has run. If $wake(T_x)$ is not empty, then T_x will commit with one of the transactions in $wake(T_x)$. That is, let T_y be *any* transaction in $wake(T_x)$. When T_x finishes, it adds itself and all transactions in its own $commit(T_x)$ to $commit(T_y)$. In addition, T_x unlocks (and un-releases) all objects it accessed and removes itself from any $wake(T_z)$ sets it may be in. On the other hand, if $wake(T_x)$ is empty, then T_x and all transactions in $commit(T_x)$ commit at once. As in the other case, T_x also unlocks (and un-releases) all its objects and removes itself from any $wake(T_z)$.

To illustrate this mechanism, consider a LLT T_L with two transactions T_1 and T_2 running in its wake. Suppose in addition that T_2 runs in the wake of T_1. That is, $wake(T_L)$ is empty, $wake(T_1)$ is $\{T_L\}$, and $wake(T_2)$ is $\{T_L, T_1\}$. If T_1 finishes its actions, it adds itself to $commit(T_L)$ and makes $wake(T_2)$ equal to $\{T_L\}$. Note that at this point T_1 unlocks all its data even though it has not committed. This is possible because all of T_1's data is entirely within the wake of T_L. Hence, any other transaction that looks at the data will still know it is uncommitted. Continuing with the example, if T_2 finishes before T_L, it also adds itself to $commit(T_L)$. When T_L finishes, it commits all three transactions. On the other hand, if T_L finishes ahead of T_2, it only commits itself and T_1. Transaction T_2 continues running with $wake(T_2)$ empty.

If a transaction T_x must be aborted, any transaction in $commit(T_x)$, any transaction T_y in the wake of T_x (i.e., $T_x \in wake(T_y)$), plus any transactions in $commit(T_y)$ must all be aborted. The database updates performed by these transactions must be undone, so that the database reflects the initial value seen by T_x.

Delayed commits have two disadvantages in the context of LLTs. First, the users who submit short transactions may not know whether their transaction was successful or not for some time. Second, if there is a LLT abort, the cascading rollbacks will make an already difficult task even more painful. Fortunately, there is a solution that we believe is feasible for many types of LLTs.

The basic idea is to insert a sequence of *save points* into the LLT such that at each of these points the partial results of the LLT are committed. At each save point, the short transactions that are waiting to commit can be committed. A similar suggestion is made for "random batch" LLTs in [Baye87a].

To be able to commit the partial LLT results at each save point, we must guarantee that the results will not be undone due to (1) a voluntary abort, (2) a deadlock, or (3) a hardware failure. The first of these guarantees can be enforced by prohibiting voluntary transaction aborts after the first save point. (It is still possible for a transaction to voluntarily undo all changes up to the last save point. In this case, other transactions in its commit set would be aborted as described above.) This does not seem like a harsh restriction because in any case a LLT should perform all checks that could indicate an inability to complete (e.g., insufficient funds for a transfer) as soon as possible to avoid waisting large amounts of effort.

For the second guarantee, we must ensure that a LLT is never aborted due to a deadlock. The simplest way to achieve this is to run a single LLT at a time. Any time a deadlock is detected, the short transactions would be aborted. A second way to achieve the guarantee is to make marks by different transactions incompatible. In this case, marking would ensure that each LLT preclaims all the data it will need, eliminating the possibility of a deadlock with another LLT.

Finally, for the third guarantee, the system must save the LLT's state information on stable storage [Lamp79a] at each save point. This makes it possible to restart the LLT at the save point after a hardware failure [Gray78a]. At the time the stable storage record for T_L is made for the save point, all transactions in $commit(T_L)$ can be committed. (Because of our deadlock avoidance rule, $wake(T_L)$ should be empty.)

Save points can reduce the problems of cascading rollbacks and transaction delays, but at the expense of sacrificing the atomicity of the LLT. Another solution, which also does away with cascading rollbacks and delays yet does not require savepoints, is to use *compensating transactions*. A compensating transaction is used to "semantically undo" the effects of an aborted LLT. The database is returned to a consistent state which may not be the same as the state that existed before the LLT started, but which is reasonable in terms of the semantics of the application.

When compensating transactions are used, short transactions that see the updates of an aborted LLT are not themselves aborted. For some types of LLT's in some applications this is not a problem. For example, if the LLT and short transactions are both making seat reservations in an airline database, it may not be critical that the LLT be

undone, as long as any reservations it made are canceled instead. The difference is that short transactions that have made reservations on the same plane in the meantime can be allowed to keep them. Compensating transactions are explored in more detail in [Garc87a, Garc83a] and we will not pursue the topic further here.

5. Obtaining Access Pattern Information

We have seen that access pattern information can be used to boost the concurrency of long-lived transactions. But how do we obtain such information, and how do we make it available for use? We will address these questions in this section.

Transactions can take many forms. Their database interactions can occur through an abstract, high level interface (e.g., the relational algebra), or through a lower level interface. Similarly, concurrency controls can be implemented at various levels of abstraction. Lockable entities might be individual records or more abstract entities such as predicates on the database.

To simplify our discussion we will postpone consideration of high level interfaces until the end of this section. For now we will consider low level interfaces. In particular, we will assume that the database interface consists of a small of collection of primitives such as **READ**(*pathID*), **WRITE**(*pathID*), **READ_NEXT**(*setID*), and **READ_FIRST**(*setID*). The primitives with *pathID* arguments are used for associative accesses. The *pathID* is the unique identifier of a path to a particular (low level) database entity. For example, *pathID* could be a key for a specific index into a database table, or it might simply be a physical or logical pointer to a record or a tuple. A *setID* argument identifies an ordered collections of data, e.g. a linked list of tuples. Operations **FIRST** and **NEXT** are used to scan such sets.

Access pattern information for the concurrency controller is supplied through various **RELEASE** and **MARK** primitives. For now we will assume that these primitives have arguments similar to those of the access primitives, i.e., either *pathID* or *setID*. For example, the actual primitives might look like **RELEASE**(*pathID*) or **MARK_CURRENT**(*setID*). The question to be discussed in this section now becomes: Given a transaction, where (if anywhere) should the new primitives be included, and who should include them?

The simplest way (from the system's point of view) to acquire access pattern information is to ask the application programmer to provide it. This means that **RELEASE** and **MARK** primitives would be put by the application programmer into the transaction when it was written. The placement of primitives would be determined by the programmers knowledge of the access behavior of the transaction. Note that incorrect placement by the programmer will *not* endanger consistency; it can only hurt performance (e.g., the system will abort any transaction that tries to access data it previously released).

Another possibility is for application programmers to categorize transactions into "access pattern classes". This may mean less work for the programmers than explicitly placing access pattern primitives. The classifications (together with the transactions themselves) could be used by the system to generate positive or negative access pattern information, or both. For example, to obtain positive information, transactions might be classified into an application-dependent set of classes according to the data they access. The system would then insert the appropriate **MARK** operations into the transaction code. This kind of scheme is used in SDD-1 [Bern80a] (although in that system the information is used to determine transaction conflicts, rather than to generate **MARK**s.) Transaction classes could also be used to produce negative access pattern information. For example, there might be a class of transactions known as "sweep transactions" that access a collection of entities in a single pass. For transactions in this class, the system could place a **RELEASE** operation after each database access.

The system might also attempt to generate access pattern information itself (without involving the application programmer) by analyzing transactions. Such an analysis would require knowledge of the control structure of the transaction, and the semantics of the database access primitives. For example, consider the transaction shown in Figure 5a, which might be run against our hypothetical banking database.

GET_FIRST(*ACCOUNTS*)
compute summary information
WHILE there are more accounts **DO**
    GET_NEXT(*ACCOUNTS*)
    compute summary information
ENDWHILE
write summary information

Figure 5a - Hypothetical Banking Transaction

We assume that *ACCOUNTS* uniquely identifies a set in the database (in this case, the table of accounts) which can be scanned in some order. This is an extremely simplified (and not terribly realistic) transaction, but it serves to illustrate our points. The WHILE loop (control structure) and the semantics of the **GET-FIRST** and **GET_NEXT** primitives dictate that each entity in *ACCOUNTS* will be accessed once by the transaction. Thus a **RELEASE_CURRENT** operation could be included after the **GET_FIRST** and **GET_NEXT** operations. If positive access pattern information was desired as well, the system could also insert code at the beginning of the transaction to mark entities in *ACCOUNTS*, using **MARK_FIRST**(*ACCOUNTS*) and **MARK_NEXT**(*ACCOUNTS*). (If the summary information is being written to the database, the summary record would also be marked using **MARK**.)

Such an analysis is more complicated if the transaction accesses database entities from more than one set. The analysis would then require information about the structure of the database to determine whether or not a particular entity was being accessed once or more than once. Structural information describes the relationships among database sets. For example, it would be necessary to determine whether any of the sets being accessed overlap (share records).

The analysis is also made more complicated if the parameters of the database access primitives are not constants, as they were in our example. For example, in **READ**(*pathID*), *pathID* might be a function of input parameters of the transaction, or of the results of previous database accesses. The latter case is particularly difficult. While it may be possible to obtain useful access pattern information, such an analysis would require some knowledge of the semantics of the data. We will not discuss such transactions further here.

If the values of parameters of access primitives are dependent on transaction input parameters, we cannot add access pattern primitives to the transaction as we did in the previous example. However, in many cases, primitives can be added *conditionally*, i.e. the primitive is to be executed only if certain conditions are satisfied when the transaction is run. For example, consider a transaction with only two access primitives **READ**(*pathA*) and **READ**(*pathB*), where *pathA* and *pathB* are supplied directly as transaction parameters. At compile-time, the conditional statement

IF $pathA \neq pathB$ **THEN RELEASE**(*pathA*)

can be inserted after **READ**(*pathA*) (along with a similar statement after **READ**(*pathB*)).

There are two difficulties with this method. First, the parameters of the access primitives may be arbitrary functions of the transaction parameters. The identity function used in our example is the simplest case. Second, there may be a large number of parameterized access primitives in the transaction. Both of these difficulties increase the complexity of the condition that must be evaluated at run time to determine whether or not the **RELEASE** or **MARK** operation should be executed. Thus, as transactions get more complex, the cost of including access pattern information increases. At some point, the concurrency benefits provided by the access pattern information will be outweighed by this additional cost. The exact point at which this occurs is highly transaction and database dependent.

One way to reduce the complexity of a transaction analysis is to use access primitives whose arguments are larger granules than records or tuples. For example, the system might have available a **RELEASE**(*setID*) primitive, which releases an entire set or relation, in addition to the **RELEASE_NEXT**(*setID*) primitive we have already considered. Using larger granules allows the analysis to ignore the details of how the

transaction accesses the various components of the large granule. The entire granule is released at once when the transaction is completely finished accessing it.

The price paid for using larger granules is that (for **RELEASE** operations) some parts of the large granule may not be released as quickly as they would have been had smaller granules been used. However, large granules reduce the overhead of locking, since fewer primitives need to be executed. They may also be able provide some access pattern information for transactions whose analysis proves to complex at a lower level.

So far in our discussion we have assumed that transactions use a fairly low-level database interface. What can be done if a higher level interface is used? For example, what if the interface supports relational algebra operations like JOIN?

If both the database interface and concurrency controls are implemented at a high level, then the situation is similar to that we have just considered. In fact, as long as the entities recognized by the concurrency control primitives are the same as those recognized by the database access primitives, all of the techniques we have already mentioned can apply. The principal difference between the high level and low level cases is the difficulty of determining whether lockable entities conflict.

In some cases the concurrency controls may be implemented at a lower level than the database interface. For example, System R [Astr76a] supports a high level relational interface for transactions, while concurrency control is implemented at the tuple/record level below the relational interface. In such systems it is not possible for application programmers to supply access pattern information unless some mechanism is provided to translate from the high level entities known to the programmers to the lower level entities used for concurrency control. Unless such a translator is provided, access pattern information must be obtained through transaction analysis. The analysis can be accomplished once the transaction has been converted to a lower level form by the query processing component of the system.

6. Conclusions

In this paper we have presented an extension to two-phase locking that allows long lived transactions to release their resources earlier, freeing up other shorter transactions. We by no means claim that our solution is the best among the proposed solutions for coping with LLTs. Each of the proposed solutions has its own strengths and weaknesses that make it appropriate in certain circumstances. However, we do claim that our proposal has certain important advantages that make it worthy of consideration:

(a) Altruistic locking does guarantee serializable executions.

(b) Transactions can access the database in any order.

(c) Declaration of access set (**MARK**ing) is optional.

(d) The protocol is fully compatible with two-phase locking; transactions that do not wish to participate in the extended protocol perform (almost) as well as they would under conventional two-phase locking with essentially no additional overhead.

(e) Altruistic locking improves performance by early release of LLT locks. It is difficult to quantify the improvements in general, but at least it is clear that in some representative scenarios it does help.

The major disadvantages are:

(1) The performance improvements are probably not as great as those achievable under strategies, such as sagas [Garc87a], that do not guarantee serializable executions for LLTs.

(2) The access pattern of the LLTs must be analyzed, either automatically or manually, in order to achieve the performance improvements.

(3) The mechanism is not as simple as two-phase locking (neither are all of the other proposed solutions).

References

Ahuj87a.
Ahuja, Mohan L. and J. C. Brown, "Concurrency Control by Pre-Ordering Entities in Databases with Multi-Versioned Entities," *Proc. Int'l Conf. on Data Engineering,* pp. 312-321, Los Angeles, CA, February, 1987.

Astr76a.
et al, Astrahan, M. M., "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems,* vol. 1, no. 2, pp. 97-137, June, 1976.

Baye87a.
Bayer, Rudolf, "Consistency of Transactions and Random Batch," *ACM Transactions on Database Systems,* Jan., 1987.

Bern80a.
Bernstein, Philip A., David W. Shipman, and James B. Rothnie, Jr., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems,* vol. 5, no. 1, pp. 18-51, March, 1980.

Garc83a.
Garcia-Molina, Hector, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems,* vol. 8, no. 2, pp. 186-213, June 1983.

Garc87a.
Garcia-Molina, Hector and Kenneth Salem, "Sagas," *Proc. ACM SIGMOD Annual Conference,* pp. 249-259, San Francisco, CA, May, 1987.

Giff85a.

Gifford, David K. and James E. Donahue, "Coordinating Independent Atomic Actions," *Proceedings of IEEE COMPCON*, San Francisco, CA, February, 1985.

Gray78a.

Gray, Jim, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. G. Seegmüller, pp. 393-481, Springer-Verlag, 1978.

Gray81a.

Gray, Jim, "The Transaction Concept: Virtues and Limitations," *Proceedings of the Seventh Int'l. Conference on Very Large Databases*, pp. 144-154, IEEE, Cannes, France, Sept., 1981.

Gray76a.

Gray, J. N., R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modeling in Data Base Management Systems*, ed. G. M. Nijssen, pp. 365-394, North Holland Publishing Company, 1976.

Kede80a.

Kedem, Zvi and Abraham Silberschatz, "Non-Two-Phase Locking Protocols With Shared and Exclusive Locks," *Proceedings of the Conference on Very Large Databases*, pp. 309-317, IEEE, Montreal, Canada, Oct., 1980.

Lafo84a.

Lafortune, S. and E. Wong, "A New Locking Protocol that Achieves All Serializable Executions," Memorandum No. UCB/ERL M84/77, Elec. Research Lab., Univ. of California, Berkeley, CA, September, 1984.

Lamp79a.

Lampson, Butler W. and Howard E. Sturgis, *Crash Recovery in a Distributed Data Storage System,* Xerox Palo Alto Research Center, Palo Alto, California, April, 1979.

Lync83a.

Lynch, Nancy, "Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control," *ACM Transactions on Database Systems,* vol. 8, no. 4, pp. 484-502, December, 1983.

Sale87a.

Salem, K. and H. Garcia-Molina, "The Correctness of Two Locking Protocols for Long-Lived Transactions," unpublished report, Dept. of Computer Science, Princeton University, July, 1987.