TOWARDS A SELF-ADAPTING
CENTRALIZED
CONCURRENCY CONTROL ALGORITHM

by
H. Boral* and I. Gold

Technical Report #317
May 1984

* Microelectronics and Computer Technology Corporation,
  Austin, TX, USA.
TOWARDS A SELF-ADAPTING CENTRALIZED CONCURRENCY CONTROL ALGORITHM

Haran Boral
Israel Gold

Computer Science Department
Technion -- Israel Institute of Technology
Haifa 32000-Israel

*Microelectronics and Computer Technology Corporation, Austin, TX, USA
ABSTRACT

The idea of concurrency control algorithms that consists of several rw synchronization techniques and several rr synchronization techniques is introduced. Some algorithms that utilize both 2PL and Certification based synchronization techniques are presented along with their proofs of correctness.

The algorithms have the capability of using any number of the techniques at their disposal concurrently; as well as allow a transaction to be synchronized by different techniques at different times. Such algorithms can form the basis of a concurrency control mechanism in a database management or transaction processing system that adapts itself to the system workload by selecting the appropriate techniques to be used.

The paper concerns itself with correctness issue of such algorithms. In addition to presenting example algorithms, a general proof method is outlined and applied. Furthermore, it is shown that mixing locking and non-locking methods can be achieved and the level of the mix may be varied dynamically.
1. Introduction

Several centralized concurrency control algorithms have been proposed during the past several years. The majority of the algorithms are based, to one degree or another, on the Two Phased Locking method (2PL) [ESWA76] in which blocking is used to synchronize conflicting transactions; and on methods that allow conflicting transactions to run concurrently but use transaction restart in cases where inconsistent updates to the database could result [KUNG81] (known as Certification methods because they certify a transaction for additional processing or commit, or cause it to abort):

At system design time a concurrency control algorithm is picked for the system (typically a 2PL variant) and incorporated into it. This design decision may be made based on some a priori knowledge of the expected use of the system or simply because the algorithm may appear to be (or actually may be) the best. Due to the complicated structure of software systems such as database management systems, it is unlikely that the original algorithm incorporated into the system will ever be changed or even be improved, despite the fact that the system may be used under a variety of workload conditions in two separate settings or even in the same setting over a long time interval.

Recently researchers have begun to compare several different algorithms in an attempt to reach some conclusion concerning their operational merit. Naturally, if a clear cut conclusion can be reached about one algorithm being "best" at almost or even at all times then that algorithm should be employed by all database management systems. The studies range from the purely abstract [PAPA79] to more "useful" where such measures as transaction throughput and cost of the concurrency control mechanism are evaluated [AGRA83], [CARE83], [GALL82]. Carey [CARE83] reports that when the overhead associated with locking, i.e. context switching, is small, transaction restarts have a more negative
effect on throughput than blocking. He thus concludes that variants of 2PL lead to higher system throughput than certification methods, as long as critical system resources (CPU and I/O in a centralized system) are well utilized. Further, of the various methods he examined (variants of 2PL, certification, and timestamp methods), dynamic 2PL performs best when no a priori knowledge about transaction processing requirements and transaction workload is known. Carey's conclusions are supported, to one degree or another, by Robinson [ROBI82A] [ROBI82B] Agrawal [AGRA83] and others.

However, other researchers (Reinl and Reuter [REIN83] for example) disagree with this final conclusion and the reason for the disagreement appears to be the use of different metrics in comparing the various methods.

Carey's metric was throughput. Although high throughput is desirable in a system, it alone cannot be used in evaluating the system's performance. Just what the "right" metric is, is not clear. In fact, there may not be a single metric to be used in evaluating all systems; the metric should be a function of the user community's needs. However, it seems reasonable to assume that any metric will incorporate into it both system throughput and response time with the emphasis on one or the other varying across installations.

Ignoring for a moment transaction restarts and transaction initiation and termination costs, we can observe that a transaction that is synchronized using a certification policy will have faster response time than a transaction synchronized by a 2PL-variant. As an example consider a database management system that allows "short readers" to run "almost unhindered", all other transactions are synchronized with a 2PL-variant. Response time of the short readers will be faster than in a system that synchronizes all transactions using a 2PL-variant. In fact, system throughput will most likely also be higher.
This suggests a different approach to concurrency control -- one in which the concurrency control mechanism would use several different synchronization techniques concurrently each for a different class of transactions. Certification techniques should be used for those transactions for which fast response time is important whereas 2PL techniques should be used for other techniques. Just what the right mix of techniques or rather number of transactions synchronized by each type of technique is unclear and would have to be determined experimentally. Furthermore, the selection of the appropriate technique to be used in case of conflict between two transactions synchronized by different techniques is an issue to be investigated and to which a satisfactory resolution can be found empirically.

In this paper we show that it is possible to construct correct concurrency control algorithms that employ both blocking and transaction restart, concurrently, to synchronize conflicting transactions. We introduce the notion of an Integrated Concurrency Control Algorithm (ICCA) which is composed of several rw and several ww synchronization techniques all used concurrently. Different techniques use different synchronization strategies. They update common data structures but differ in the interpretation of the contents of the data structures when synchronizing conflicting transactions.

We wish to stress that the purpose of this paper is to show the possibility but not feasibility\(^1\) of such a concurrency control mechanism although towards the end of the paper we will discuss some implementation problems and their solutions as well as the envisioned use of such algorithms.

The remainder of the paper is organized as follows. In Section 2 we give an overview of the database management system model used in the paper. Section 3 discusses the Transaction Manager (TM) model -- both data structures and

---

\(^1\) This is the topic of current work
the operations the TM performs on them on behalf of transactions. In Section 4 we present rw and ww synchronization techniques based on 2PL and Certification. The notion of an Integrated Concurrency Control Algorithm (ICCA) is introduced in Section 5. In Section 6 we present a formalization of the techniques and give the proof method used to show an ICCA correct. Some practical consideration are discussed in Section 7 and we conclude with a summary of present work in Section 8.

2. The Database Management System Model

In this section we discuss the database management system model. The model utilizes private workspaces allocated to active transactions to cache their previously read data items as well as those written by the transaction during its execution. Bernstein and Goodman [BERN81] and Kung and Robinson [KUNG81] used this model previously. The following description closely models that of [BERN81].

A centralized database management system can be seen as composed of two components: a Transaction Manager (TM) and a Data Manager (DM). The TM controls interaction between users and the database management system and is responsible for such functions as concurrency control. The DM is responsible for management of the database itself, i.e., accessing it. Two data manipulation operations are recognized by the DM. DM_READ(X) -- in which data item X is read, and, DM_WRITE(X,NEW_VALUE) -- in which the value NEW_VALUE is assigned to data item X in the database.

Users of the database management system interact with it by running transactions. The TM maintains a private workspace for each active transaction in which copies of records read or written by the transaction are kept. From
the database management system point of view a transaction executes four
types of operations: TRANS, READ, WRITE, and SNART. The actions taken by the
TM upon receipt of these commands are described below.

TRANS: The TM initializes a private workspace for the transaction.

READ(X): If X already exists in the private workspace then its
value is returned to the transaction by the TM. Otherwise the TM
issues a DM_READ(X) operation to the DM. The current value of X is
returned to the TM which writes it in the transaction's private
workspace and returns it to the transaction.

WRITE(X, NEW_VALUE): X is a data item. NEW_VALUE is a value to be assigned to
X. The TM executes a PRE_WRITE(X, NEW_VALUE) operation on the
transaction's private workspace. This has the effect of updating the
previous value of X in the private workspace to NEW_VALUE if a copy of
X existed in the private workspace. Otherwise, X is created in the
workspace with the value NEW_VALUE. Note that a PRE_WRITE opera-
tion does not alter any values in the database itself. From a synchroni-
ization point of view each WRITE causes a PRE_WRITE to be executed.

SNART: The TM checks whether allowing the transaction to commit (by making
its changes permanent in the database) will leave the database in a
consistent state. In the event that it does not, the transaction will be
aborted. Otherwise the TM issues a DM_WRITE command for every pre-
viously executed PRE_WRITE command. This has the effect of making
the last change to X in the private workspace a (temporarily) per-
manent value in the database. After all DM_WRITEs have been issued
the private workspace is discarded -- the transaction has completed.
From a synchronization point of view all the transaction's DM_WRITEs
are executed atomically.

A transaction execution can be seen as composed of two phases. In the
first phase the transaction reads values from the database, performs various
computations and writes results into its private workspace. In the second
phase, which takes place after the transaction finishes all computations, the TM
first goes through (a possibly empty) procedure to ensure that committing the
transaction will not cause inconsistencies, and then (a possibly empty)
sequence of writes to the database (as described above). It is important to real-
ize that this second phase is atomic.

2 The physical implementation of the commit procedure need not be atomic as long as it ap-
ppears atomic to the outside world. Kung and Robinson [KUNG81] discussed several ways of imple-
menting non-atomic commits. In this paper we will refer to atomic commits from a logical point of
view, we ignore issues of physical implementation.
The notion of a (logical) private workspace is basic to our work. All references to data items in the private workspace are made through the TM which gives it the power to control the concurrency level in the system. Thus, the notion of a private workspace presented in this paper differs from that used by Network based database management systems, where the user program "contains" its own private workspace (or user work area) which can be accessed at any time independently of the database management system.

3. The Transaction Manager Model

In this section we examine in more detail the actions taken by the TM upon receipt of a request from a transaction. We describe the data structures involved as well as the operations performed on them. These, in turn, will be used in the next sections to describe the actions taken by the various synchronization techniques.

Two data structures are required by the TM for its operation. One is a graph (known as the Serialization Graph -- SG) that represents precedence relationships among conflicting transactions. Two transaction conflict if they access the same data item and one issues a WRITE request. The second is a table of flags (FT) in which a list of transactions and their modes of access to data items is maintained.

A node in SG represents an active or committed transaction. An edge \((T_i, T_j)\) in the graph indicates that in any execution order transaction \(T_i\) precedes transaction \(T_j\). SG is used to represent all such precedence relationships regardless of whether they originated in the deadlock detection phase of 2PL or in the detection of a conflict in a Certification algorithm.

An entry in FT exists for every data item that has been accessed, and consists of several pairs \(<\text{FLAG, TRANSACTION IDENTIFIER}>\). Each pair identifies the transaction that accessed the data item and the mode of access (Read or

Technion - Computer Science Department - Technical Report CS0317 - 1984
Write). No restriction is placed on the number and/or type of pairs associated with a single data item in an entry. It is up to the concurrency control mechanism to interpret the pairs in a single entry and to decide how to use that information.

Three types of flags are recognized:

1. An r-flag indicates that a DM_READ operation was executed on this item on behalf of the transaction holding the flag.
2. A p-flag indicates that a still active transaction issued a WRITE request on this data item. At commit time of a transaction all its p-flags are converted to c-flags if the transaction is allowed to commit.
3. A c-flag indicates that a DM_WRITE operation was executed on this data item on behalf of the committed transaction holding the flag.

A TRANS operation causes the TM to add a node to SG representing the new transaction. Edges are added to SG by the synchronization techniques as described in the next section.

At execution time a READ or a WRITE request received by the TM undergoes a possibly empty waiting phase then a possibly empty synchronization phase followed by its execution.

In the waiting phase some synchronization techniques may force the requesting transaction to wait until transactions that "hold" conflicting flags on the same data item have completed execution, whereas other techniques always enable continuation of the execution to the synchronization phase. Edges are added to SG to reflect the precedence relation imposed by the waiting.

In the synchronization phase the request is synchronized with conflicting operations from other transactions. The result of this synchronization may be abortion of the issuing transaction or continuation with execution. Edges are added to SG to reflect the precedence relation imposed by the execution of the request.

Execution of the request includes appending the appropriate flag to FT and issuing the appropriate DM_READ, PRE_WRITE, or DM_WRITE operation to the
A SNART operation causes the TM to perform a possibly empty validation phase to ensure that allowing the transaction to commit will not leave the database in an inconsistent state.

The difference between the synchronization techniques presented in the next section is in the manner in which they act in each of the three phases described above. What is common to all the techniques is that they update SG during each phase they undergo. For example, a technique that undergoes the waiting phase and discovers that transaction T_i must wait for transaction T_j will add the edge \((T_j, T_i)\) to SG. Similarly, FT is also updated for every request.

Note that we have specified above various operations that add nodes and edges to SG and pairs to FT. Information about committed transactions remains in these data structures although it can be shown that it need not be kept indefinitely. All traces of aborted transactions, however, are removed from both SG and FT. That is, the node representing an aborted transaction in SG is removed along with all incoming and outgoing edges. All pairs detailing the accesses made by the aborted transaction are also removed from FT.

4. Synchronization Techniques

The Decomposition Theorem of concurrency control [BERN81] enables the designer of a concurrency control mechanism to address himself to two subproblems: synchronization of READ WRITE requests (rw-synchronization) and synchronization of WRITE WRITE requests (ww-synchronization) rather than to a single more complex problem -- that of concurrency control. An rw (ww) synchronization technique is defined to be the procedure that guarantees correct rw (ww) synchronization. The concurrency control mechanism must then ensure that the use of a given rw synchronization technique together with a given ww synchronization technique will yield serializable execution orders.
The decomposition theorem serves as the foundation for the notion of an integrated concurrency control algorithm. Rather than the use of one rw technique together with one ww technique as is the norm in working systems and the plethora of proposed algorithms, we suggest designing an algorithm that would be composed of several rw techniques and several ww techniques. The number of techniques to be used at a given instance is something that would be left up to the system designer. The decision may be made statically or dynamically as indicated in Section 1. In this section we will present three rw techniques and two ww techniques. In sections 5 and 6 we will discuss their use in ICCAs and present the proof method to show their correctness.

4.1. Characterization of Synchronization Techniques

We propose two characterization parameters for synchronization techniques: Synchronization Strategy and Synchronization Time. By synchronization strategy we mean the method used to synchronize conflicting transactions. We differentiate between methods that use locking (in particular 2PL) and methods that use rollbacks (in our case certification methods). Synchronization between two conflicting transactions may be performed at the time the conflicts occurred (i.e., at execution time -- ET) or during the commit procedure (commit time synchronization -- CT) that each transaction must undergo before its updates to the database take effect.

The meaning of ET synchronization is that the order in which READ and WRITE operations arrive is important and that the synchronization algorithm must take that order into account. In a 2PL algorithm this order is determined by the flags transaction hold and by maintaining a queue of waiting transactions for flagged data items.

The meaning of CT synchronization is that the arrival order of READ and WRITE operations is unimportant. Thus, the CT synchronization algorithm need
only collect and maintain information about each transaction's conflicts at execution time to enable it to discover inconsistencies at commit time.

Using these two characterization parameters we can obtain five synchronization techniques for each of the two types of synchronization. For example, we may use certification synchronization at transaction execution time or transaction commit time for either the rw or ww synchronizations.

We use the following notation to represent the possible techniques. Each technique's name will be composed of three components: synchronization strategy (2PL or CERT), synchronization time (ET or CT), and synchronization type (rw or ww). For example, CERT-CT-ww is a technique that uses the certification approach to achieve synchronization at transaction commit time. Figure 3 summarizes all eight possible techniques. In Section 4.3 we give brief descriptions of the actions taken by five of the techniques, where the others are of no interest. Before describing the techniques we introduce, in the next section, the notion of a transaction's WaitFor Set.

4.2. The WaitFor Set of a Transaction

From a correctness point of view a transaction that is waiting for a flag need not wait for all active transactions that hold conflicting flags to terminate. For example, if $T_i$ using 2PL synchronization requests an $r$-flag on $X$ and $T_j$ also using 2PL holds a $p$-flag on $X$ then clearly $T_i$ should wait for $T_j$ and this should be reflected by an edge in SG from $T_j$ to $T_i$. If, however, $T_k$ which uses a certification synchronization strategy holds a $p$-flag on $X$, from a correctness point of view we may choose to have $T_i$ wait for $T_k$ or not do so. Either way would be correct (provided the synchronization phase operates correctly). This decision is a policy decision and for that reason we chose not to make it at this
Rather, we allow the implementor to define a *waitfor set* of transactions in terms of the concurrency level desired.

Definition 1, below, formalizes this notion.

**Definition 1:** The *WaitFor Set* of a transaction \( T_i \) for \( rw \) (ww) synchronization, \( \text{WFS}_{rw}(ww)(T_i) \), is the set of transactions that \( T_i \) must wait for in its \( rw \) (ww) waiting phase.

The *Minimal WaitFor Set* of a transaction \( T_i \) for \( rw \) (ww) synchronization, \( \text{MWFS}_{rw}(ww)(T_i) \), consists of only those transactions for which \( T_i \) must wait to ensure correctness of \( rw \) (ww) synchronization.

We say that transaction \( T_i \) is using *strict 2PL policy* for \( rw(ww) \) synchronization when \( \text{WFS}_{rw}(ww)(T_i) = T \).

We say that transaction \( T_i \) is using *standard 2PL policy* for \( rw(ww) \) synchronization when \( \text{WFS}_{rw}(ww)(T_i) = \{ T_j \mid T_j \text{ is using 2PL for } rw(ww) \text{ synchronization} \} \).

If the same synchronization policy applies to \( rw \) and \( ww \) synchronization type designation of the type will be omitted.

For transaction \( T_i \) using CERT synchronization strategy we define \( \text{MWFS}_{rw}(T_i) = \text{MWFS}_{ww}(T_i) = \emptyset \).

The *waitfor set* of a transaction \( T_i \) is implementation dependent and may be defined by means of synchronization techniques, transaction classes, the concurrency level desired, and even as a function of \( T_i \) waiting time on a given request or all its past requests.

### 4.3. The Synchronization Techniques

To facilitate the description of the various techniques below we utilize a compatibility-action matrix (c-a matrix). An entry in the matrix indicates how a synchronization technique interprets the values of existing flags on a data item when processing a request from a transaction to access that item. In addition, the entry specifies what edges are added to SG. The c-a matrices for all five techniques described below are shown in Figure 4.

#### 4.3.1. RW Synchronization Techniques

RW synchronization techniques synchronize between DM_READ and PRE_WRITE operations as well as DM_READ and DM_WRITE operations. In the

---

3 We were influenced to a great degree in this part of our design by Robinson's notion of separation of correctness and policy in concurrency control [ROB82].
following, if adding an edge to SG causes a cycle in the graph the transaction that caused the cycle is aborted, all its flags released and all information about it removed from SG.

4.3.1.1. 2PL-ET-rw

Assume $T_i$ is using 2PL-ET-rw and let $T_j$ be an active transaction in $\text{WFS}_{rw}(T_i)$. The c-a matrix in Figure 4 for 2PL-ET-rw applies to the waiting phase. During its waiting phase $T_i$'s request can not proceed to the synchronization phase as long as $T_j$ owns a conflicting flag. The c-a matrix for CERT-ET-rw describes the actions taken in the synchronization phase.

4.3.1.2. CERT-ET-rw

Assume $T_i$ is using CERT-ET-rw and let $T_j$ hold a flag on data item $X$. $T_i$'s waiting phase is null. During its synchronization phase, $T_i$ resolves conflicts with $T_j$'s flags using c-a matrix for CERT-ET-rw.

4.3.1.3. CERT-CT-rw

In this technique synchronization is performed at commit time using a procedure similar to that used in [KUNG81]. All requests are granted. No edges are added to SG at execution time. At commit time edges are added to SG as shown in the c-a matrix for this technique. Note that during $T_i$'s commit phase if an edge, from an active reader, $T_j$ to $T_i$ is added to SG it is guaranteed that $T_j$ will be aborted during its commit phase (and as an optimization it may be aborted during $T_i$'s commit phase).

4.3.2. WW Synchronization Techniques

WW synchronization techniques synchronize between conflicting, PRE_WRITE operations of active transactions as well as PRE_WRITE operations of an active transaction and conflicting DM_WRITE operations of committed
transactions.

4.3.2.1. 2PL-ET-ww

The meaning of the c-a matrix entries in 2PL-ET-ww is as in the matrix for 2PL-ET-rw. Conflicts between two active transactions are synchronized in the waiting phase of a request whereas conflicts between an active and a committed transaction are synchronized during the request's synchronization phase.

4.3.2.2. CERT-CT-ww

In this technique synchronization is performed at commit time. All WRITE requests are allowed to run unhindered. The c-a matrix for this technique applies to the commit phase of the transaction. The committing transaction T_i follows all transactions committed earlier and indicates that active writers will follow it (synchronization with active writers is required only when they use the 2PL-ET-ww synchronization technique).

5. Integrated Concurrency Control Algorithms

In this section we give a formal definition of an ICCA. We also outline the proof method used in showing an ICCA to be correct. In the next section we give the theoretical basis and give an example of a proof of correctness.

**Definition 2:** An Integrated Concurrency Control Algorithm (ICCA) is a triple \((S_{rw}, S_{ww}, F)\), where
- \(S_{rw}\) is a non empty set of rw-synchronization techniques
- \(S_{ww}\) is a non empty set of ww-synchronization techniques, and
- \(F\) is a mapping function \(F: T \rightarrow S_{rw} \times S_{ww}\). At any given-instance each \(T_i\) in \(T\) is mapped to exactly one \(S_{rw}\) technique and one \(S_{ww}\) technique.

**Example 1:**

\[
\text{ICCA}_1 = \{ \{2PL-ET-rw, Cert-ET-rw\}, \{Cert-CT-ww\}\},
\]

\(F\) if \(T_i\) is a reader then

\(\text{Cert-ET-rw} \times \text{Cert-CT-ww}\)

else
An ICCA operates correctly if it allows only serializable execution orders and it avoids deadlock situations. How do we prove a given ICCA as correct? We need to show that each constraint placed on the total execution order by an operation is represented in SG regardless of the transactions involved, and more importantly, regardless of the synchronization technique used presently and/or techniques used to synchronize the conflicting transactions in the past. It will be shown subsequently that the five techniques introduced in the previous section indeed satisfy this condition.

6. Formalization

6.1. Serializability

Our model differs from those of others in our use of the PRE_WRITE operation to synchronize conflicting transactions. In particular, we saw that some of the techniques cause transactions to wait in the presence of a conflicting p-flag and others do not. In this section we review the basic serializability theory results, and show that our use of PRE_WRITE as a synchronization primitive does not affect known results. We also introduce several new precedence relations to be used in the proof of correctness of ICCAs.

Definition 3: Let \( T = \{T_1, T_2, ..., T_n\} \) be a set of transactions. E, the execution schedule of \( T \), is modeled by \( L^5 \), the synchronization log of \( T \), which consists of DM_READ, PRE_WRITE, and DM_WRITE operations in the order in which they were scheduled. L, the execution log of \( T \), is derived from \( L^5 \) by removing from it all PRE_WRITE operations.

In subsequent lemma and theorem statements we shall assume that \( T, E, L \), and \( L^5 \) as defined in Definition 3 are given. Furthermore, references to \( T_i \) and \( T_j \) are to any two transactions in \( T \). A DM_READ(X) operation by transaction \( T_i \) will

---

4 For the sake of brevity we eliminate the definitions of the various terms, such as computational equivalence.
be denoted by $r_i[X]$. Similarly, $p_i[X]$ and $w_i[X]$ will denote PRE_WRITE and DM_WRITE operations (respectively) on $X$ made by transaction $T_i$. Finally, $O_i[X] < O_j[X]$ means that $O_i[X]$ precedes $O_j[X]$ in $L^S$. Figure 1 illustrates the notions of the execution schedule, the synchronization log and the execution log.

**Definition 4:** $L^S$ is serializable if it is computationally equivalent to a serial synchronization log.

**Proposition 1:** $L^S$ and $L$ are computationally equivalent.

**Proof:** Both logs contain the same sequence of DM_READ and DM_WRITE operations. PRE_WRITE operations in $L^S$ do not change database state. Figure 2 illustrates the notion of serializable logs.

**Theorem 1:** $L^S$ is serializable iff $L$ is serializable.

**Proof:**

(a) First we show that if $L^S$ is serializable then $L$ is serializable.

(1) Since $L^S$ is serializable it is computationally equivalent to some serial synchronization log $L^F_1$.

(2) Let $L_1$ be the serial execution log derived from $L^F_1$ by deleting all PRE_WRITE operations from it. From proposition 1 it follows that $L^F_1$ is computationally equivalent to $L_1$.

Since $L$ is computationally equivalent to $L^S$ (proposition 1), it follows from (1) and (2) that $L$ is computationally equivalent to some serial execution log $L_1$.

(b) Next we show that if $L$ is serializable then $L^S$ is serializable.

(3) Since $L$ is serializable it is computationally equivalent to some serial execution log $L_1$.

(4) $L_1$ is computationally equivalent to a serial synchronization log $L^F_1$ derived from $L_1$, by augmenting each transaction $T_i$, before its first DM_WRITE operation, with a set of PRE_WRITE operations as follows: for every $w_i[X]$ in $L_1$ add a $p_i[X]$ operation to $L^F_1$.

Since $L^S$ is computationally equivalent to $L$ (proposition 1) it follows from (3) and (4) that $L^S$ is computationally equivalent to some serial synchronization log $L^F_1$.

Theorem 1 establishes that to obtain serializable synchronization logs it is sufficient to maintain serializable execution logs. That is, use of the PRE_WRITE operation as a synchronization primitive while maintaining serializable execution logs does not affect the basic theory.
Definition 5 sets the background for stating the Decomposition Theorem in our extended model.

**Definition 5:** For every pair of transactions \( T_i, T_j \) and a data item \( X \), define the binary relations \( \rightarrow_u \), where values for \( u \) are given below, as follows:

1. \( T_i \rightarrow_{rw} T_j \) if \( r_i[X] < w_j[X] \) in \( L^S \)
2. \( T_i \rightarrow_{wr} T_j \) if \( w_i[X] < r_j[X] \) in \( L^S \)
3. \( T_i \rightarrow_{ww} T_j \) if \( w_i[X] < w_j[X] \) in \( L^S \)
4. \( T_i \rightarrow_{rp} T_j \) if \( r_i[X] < p_j[X] \) in \( L^S \)
5. \( T_i \rightarrow_{pr} T_j \) if \( p_j[X] < r_i[X] < w_j[X] \) in \( L^S \)
6. \( T_i \rightarrow_{wp} T_j \) if \( w_i[X] < p_j[X] \) in \( L^S \)
7. \( T_i \rightarrow_{wp} T_j \) if \( p_j[X] < w_i[X] < w_j[X] \) in \( L^S \)
8. \( T_i \rightarrow_{pp} T_j \) if \( p_i[X] < p_j[X] \) and \( w_i[X] < w_j[X] \) in \( L^S \)
9. \( T_i \rightarrow_{rwr} T_j \) if \( T_i \rightarrow_{rw} T_j \) or \( T_i \rightarrow_{wr} T_j \)
10. \( T_i \rightarrow_{rpr} T_j \) if \( T_i \rightarrow_{rp} T_j \) or \( T_i \rightarrow_{pr} T_j \)
11. \( T_i \rightarrow_{pwp} T_j \) if \( T_i \rightarrow_{pw} T_j \) or \( T_i \rightarrow_{wp} T_j \)
12. \( T_i \rightarrow_{rwr} T_j \) if \( T_i \rightarrow_{rwr} T_j \) or \( T_i \rightarrow_{ww} T_j \)
13. \( T_i \rightarrow_{rwr} T_j \) if \( T_i \rightarrow_{rwr} T_j \) or \( T_i \rightarrow_{rpr} T_j \) and \( T_k \rightarrow_{rpr} T_j \)

The binary relations (1)-(3), (9),(12) and (13) are exactly those defined in [BERN81]. The remaining relations are new relations introduced in this paper based on our use of the PRE_WRITE as a synchronization primitive. Clearly all of these relations can be derived from \( L^S \).

**Theorem 2 (Decomposition):** Let \( \rightarrow_{rwr} \) and \( \rightarrow_{ww} \) be associated with an execution schedule \( E \) modeled by \( L^S \). \( E \) is serializable if

1. \( \rightarrow_{rwr} \) and \( \rightarrow_{ww} \) are acyclic, and
2. There is a total ordering of the transactions consistent with all \( \rightarrow_{rwr} \) and all \( \rightarrow_{ww} \) relationships.

**Proof:** (1) and (2) guarantee the serializability of the execution log [BERN81]. This fact and Theorem 1 guarantee also the serializability of \( L^S \).

Lemmas 1 and 2 will be used subsequently. They formally define an important fact, that by means of the PRE_WRITE operation we are able to foresee future conflicts in \( \rightarrow_{rwr} \) and \( \rightarrow_{ww} \) relations.

**Lemma 1.** \( \rightarrow_{rpr} \geq \rightarrow_{rw} \)

**Proof:** If \( T_i \rightarrow_{rw} T_j \), then by Definition 5 there exists a data item \( X \) such that \( r_i[X] < w_j[X] \) in \( L^S \).

In \( L^S \) each \( w_j[X] \) is preceded by \( p_j[X] \) i.e., \( p_j[X] < w_j[X] \).

Technion - Computer Science Department - Technical Report CS0317 - 1984
(1) if \( r_i[X] < p_j[X] < w_j[X] \) then \( T_i \rightarrow_{rp} T_j \)
(2) if \( p_j[X] < r_i[X] < w_j[X] \) then \( T_i \rightarrow_{pr} T_j \)

from (1) and (2) it follows that if \( T_i \rightarrow_{rw} T_j \) then \( T_i \rightarrow_{rpr} T_j \)

**Corollary 1:** \( \rightarrow_{rpr} \cup \rightarrow_{wr} \supset \rightarrow_{rw} \)

**Lemma 2:** \( \rightarrow_{pwp} \supset \rightarrow_{ww} \)

**Proof:** If \( T_i \rightarrow_{ww} T_j \) then by Definition 5 there exists a data item \( X \) such that \( w_i[X] < w_j[X] \) in \( L^S \).

In \( L^S \) each \( w_j[X] \) is preceded by \( p_j[X] \) i.e., \( p_j[X] < w_j[X] \).

(1) if \( w_i[X] < p_j[X] < w_j[X] \) then \( T_i \rightarrow_{wp} T_j \)
(2) if \( p_j[X] < w_i[X] < w_j[X] \) then \( T_i \rightarrow_{pw} T_j \)

from (1) and (2) it follows that if \( T_i \rightarrow_{ww} T_j \) then \( T_i \rightarrow_{pwp} T_j \)

### 6.3. Formalization of the Techniques

The following lemma characterizes the correspondence between the relations defined above and edges in \( SG \).

**Lemma 3:** The relations \( \rightarrow_{rp}, \rightarrow_{pr}, \rightarrow_{rw}, \rightarrow_{wr}, \rightarrow_{pw}, \rightarrow_{wp}, \rightarrow_{ww} \) and \( \rightarrow_{ww} \) derived from \( L^S \) and \( SG \) maintained by the synchronization techniques satisfy the following conditions:

(1) If \( T_i \) is using 2PL-ET-rw then:
   (a) if \( T_j \rightarrow_{wr} T_i \) then \( (T_j, T_i) \) is an edge in \( SG \).
   (b) if \( T_j \rightarrow_{rp} T_i \) then \( (T_j, T_i) \) is an edge in \( SG \).
   (c) if \( T_j \) in \( WFS_{rw}(T_i) \) then \( T_i \rightarrow_{pr} T_j \) is not in \( \rightarrow_{pr} \).
   (d) if \( T_j \) not in \( WFS_{rw}(T_i) \) then \( T_i \rightarrow_{pr} T_j \) then \( (T_j, T_i) \) is an edge in \( SG \).

(2) If \( T_i \) is using CERT-ET-rw then:
   (a) if \( T_j \rightarrow_{pr} T_i \) then \( (T_i, T_j) \) is an edge in \( SG \).
   (b) if \( T_j \rightarrow_{rp} T_i \) then \( (T_i, T_j) \) is an edge in \( SG \).
   (c) if \( T_j \rightarrow_{wr} T_i \) then \( (T_i, T_j) \) is an edge in \( SG \).

(3) If \( T_i \) is using 2PL-ET-ww then:
   (a) if \( T_j \rightarrow_{wp} T_i \) then \( (T_i, T_j) \) is an edge in \( SG \).
   (b) if \( T_j \) in \( WFS_{ww}(T_i) \) and \( T_j \) in \( WFS_{ww}(T_j) \) then \( T_i \rightarrow_{pw} T_j \) is not in \( \rightarrow_{pw} \).

(4) If \( T_i \) is using CERT-CT-rw then:
   (a) if \( T_j \rightarrow_{rw} T_i \) then \( T_i \rightarrow_{rj} T_j \) is an edge in \( SG \).
   (b) if \( T_j \rightarrow_{wr} T_i \) then \( T_j \rightarrow_{rj} T_i \) is an edge in \( SG \).
(5) If $T_i$ is using CERT-CT-ww then:

(a) if $T_j \rightarrow_{ww} T_i$ then $(T_j, T_i)$ is an edge in SG.
(b) if $T_i \rightarrow_{pw} T_j$ then $(T_i, T_j)$ is an edge in SG.

Proof: Follows immediately from the definition of the synchronization techniques with their corresponding c-a matrices.

### 6.3. Showing ICCAs Correct

**Definition 6:** An ICCA = $(S_{rw}, S_{ww}, F)$ is correct, if for every mapping of $F$ the following conditions hold.

1. Serializable synchronization logs are attained, and
2. No deadlock results

To illustrate the proof method introduced in the previous section consider the following ICCAs below:

$$ICCA_2 = \{ \{ 2PL-ET-rw, CERT-ET-rw \}, \{ 2PL-ET-ww, CERT-CT-ww \}, F_2 \}$$

$$ICCA_3 = \{ \{ CERT-CT-rw \}, \{ 2PL-ET-ww, CERT-CT-ww \}, F_3 \}$$

The importance of $ICCA_2$ stems from the fact that it can model a transaction system of varying concurrency level, *from strict 2PL policy to full certification.*

$ICCA_3$ with $F_3 \rightarrow CERT-CT-rw \times CERT-CT-ww$ is just another form of Kung & Robinson algorithm [KUNG81]. We next prove the correctness of $ICCA_2$. Since the proof covers all possible combinations of mappings to the synchronization techniques, any ICCA that employs a subset of the techniques used in $ICCA_2$ is also shown to be correct.

**Lemma 4:** Let $L$ be the synchronization log for an execution using $ICCA_2$. If $T_j \rightarrow T_i$ then $(T_j, T_i)$ is an edge in SG.

Proof:

(a) First we show that if $T_j \rightarrow_{rwr} T_i$ then $(T_j, T_i)$ is an edge in SG.

$F_2$ may map $T_i$ and $T_j$ to $2PL-ET-rw$ and $CERT-ET-rw$ in 4 ways:

1. Let $F_2: T_i \rightarrow 2PL-ET-rw, T_j \rightarrow 2PL-ET-rw$
   By Lemma 3.1.a if $T_j \rightarrow_{wr} T_i$ then $(T_j, T_i)$ is an edge in SG.
By Lemma 3.1.b if \( T_j \rightarrow_{rp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.1, substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we have:
If \( T_i \) in \( WFS_{rw}(T_j) \) then by Lemma 3:1.c \( T_j \rightarrow_{pr} T_i \) is not possible, hence
we can write if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.
If \( T_i \) not in \( WFS_{rw}(T_j) \) then by Lemma 3.1.d if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

(2) Let \( F_2: T_i \rightarrow 2PL-ET-rw, T_j \rightarrow CERT-ET-rw \)
By Lemma 3.1.a if \( T_j \rightarrow_{wr} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.1.b if \( T_j \rightarrow_{rp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.2.a substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we get if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

(3) Let \( F_2: T_i \rightarrow CERT-ET-rw, T_j \rightarrow 2PL-ET-rw \)
By Lemma 3.2.c if \( T_j \rightarrow_{wr} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.2.b if \( T_j \rightarrow_{rp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.1, substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we have:
if \( T_i \) in \( WFS_{rw}(T_j) \) then by Lemma 3.1.c \( T_j \rightarrow_{pr} T_i \) is not possible, hence
we can write if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.
If \( T_i \) not in \( WFS_{rw}(T_j) \) then by Lemma 3.1.d if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

(4) Let \( F_2: T_i \rightarrow CERT-ET-rw, T_j \rightarrow CERT-ET-rw \)
By Lemma 3.3.a if \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.3.b if \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.2.a substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we get if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

In all possible mappings of \( T_i \) and \( T_j \) by \( F_2 \) we have that if \( T_j \rightarrow_{rpr} T_i \) or \( T_j \rightarrow_{wr} T_i \) then \((T_j, T_i)\) is an edge in SG. By Corollary 1 we conclude (**).

(b) Next we show that if \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge in SG (**).

\( F_2 \) may map \( T_i \) and \( T_j \) to \( 2PL-ET-rw \) and \( CERT-CT-ww \) in 4 ways:

(5) Let \( F_2: T_i \rightarrow 2PL-ET-ww, T_j \rightarrow CERT-ET-ww \)
By Lemma 3.4.a if \( T_j \rightarrow_{wp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.4.b if \( T_j \rightarrow_{rp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.5.a if \( T_j \rightarrow_{pw} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.5.b substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we get if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

(6) Let \( F_2: T_i \rightarrow CERT-CT-ww, T_j \rightarrow 2PL-ET-ww \)
By Lemma 3.4.a if \( T_j \rightarrow_{wp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.5.b substituting \( T_j \) for \( T_i \) and \( T_i \) for \( T_j \) we get if \( T_j \rightarrow_{pr} T_i \) then \((T_j, T_i)\) is an edge in SG.

(7) Let \( F_2: T_i \rightarrow CERT-CT-ww, T_j \rightarrow CERT-CT-ww \)
By Lemma 3.4.a if \( T_j \rightarrow_{wp} T_i \) then \((T_j, T_i)\) is an edge in SG.
By Lemma 3.5.a if \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge in SG.

In all possible mappings of \( T_i \) and \( T_j \) by \( F_2 \) we have that if \( T_j \rightarrow_{pr} T_i \) or \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge in SG. By Lemma 2 we conclude (**).

From (a) and (b) we have if \( T_j \rightarrow_{rwr} T_i \) or \( T_j \rightarrow_{ww} T_i \) then \((T_j, T_i)\) is an edge
in SG.
Theorem 3: ICCA₂ is correct.

Proof: Let E be Execution using ICCA₂ and modeled by L₅. Let -> be derived from L₅.

(1) By Lemma 4 if Tᵢ -> Tᵢ then (Tᵢ, Tᵢ) is an edge in SG. Since SG is maintained acyclic at all time it follows that -> is acyclic. By the fact that -> is acyclic and Theorem 1 we conclude that E is serializable.

(2) Whenever Tᵢ waits for Tᵢ upon rw or ww conflict an edge (Tᵢ, Tᵢ) is added to SG. Since SG is maintained acyclic at all time deadlock is prevented. Furthermore, since we know that Tᵢ will follow Tᵢ in any execution schedule, (i) guarantees overall consistency.

Lemma 5: Let L₅ be the synchronization log for an execution using ICCA₃. If Tᵢ -> Tᵢ then (Tᵢ, Tᵢ) is an edge in SG.

Proof:
(a) First we show that if Tᵢ -> Tᵢ then (Tᵢ, Tᵢ) is an edge in SG. (*)
(b) Next by argument (b) of Lemma 4 we have: if Tᵢ -> Tᵢ then (Tᵢ, Tᵢ) is an edge in SG. (**) From (a) and (b) we have if Tᵢ -> Tᵢ or Tᵢ -> Tᵢ then (Tᵢ, Tᵢ) is an edge in SG.

Theorem 4: ICCA₃ is correct.

Proof: Follows by arguments (i) and (2) of Theorem 3 using Lemma 5.

6.4. Minimal WaitFor Set for a transaction using ICCA₂.

Lemma 6: For Tᵢ using 2PL-ET-rw, MWFS_rw(Tᵢ) = ∅.

Proof: When WFS_rw(Tᵢ) = ∅, Tᵢ's waiting phase becomes empty and Tᵢ certifies, in its synchronization phase, all conflicting transaction using the CERT-ET-rw c-a matrix.

This means that Tᵢ is using CERT-ET-rw synchronization technique. The correctness of ICCA₂ establishes the fact that the separate or concurrent use of 2PL-ET-rw and CERT-ET- rw is correct.

Lemma 7: For Tᵢ using 2PL-ET-ww, MWFS_ww(Tᵢ) = { Tⱼ | Tⱼ is using 2PL-ET-ww }.

Proof: Let Tᵢ and Tⱼ use 2PL-ET-ww. Then the minimal wait for set defined above guarantees' condition 3.b of Lemma 3. This condition states that if Tᵢ's PRE.WRITE operation conflicts with Tⱼ's previous PRE.WRITE operation then Tᵢ will not precede Tⱼ in ->_pww

This condition is mandatory for correct computation of ->_ww relation - using Lemma 2. We give a counter example in which, the minimality condition is violated and as a consequence a non serializable synchronization log is created.
Let $T = \{T_i, T_j\}$ be defined as follows:

- $T_i$: TRANS READ(Z); WRITE(Z); WRITE(X); SNART
- $T_j$: TRANS READ(X); WRITE(X); WRITE(Y), SNART

and use ICCA$_2$ with the following mapping:

- $F_2: T_i \rightarrow$ CERT-ET-rw X 2PL-ET-ww with WFS$_{ww} = \emptyset$
- $T_j \rightarrow$ CERT-ET-rw X 2PL-ET-ww with WFS$_{ww} = \{T_1\}$

Round robin execution of $T$ will create the following non serializable log.


7. Practical Considerations

There are several implementation issues that require addressing such as: efficient implementation of atomic commits, implementation of the private workspaces by the TM, management of SG and FT, management of the wait queue associated with a data item, etc. In this section we look at two of these issues which are related to efficient implementation of an ICCA: management of SG & FT and the wait queues. Discussion of the other issues is beyond the scope of this paper.

7.1. Management of SG and FT

Let $T$ be a set of transaction using some correct ICCA. SG and FT can not grow infinitely. An SG node, of a committed transaction $T_i$, may be deleted only when it becomes a root node, i.e., it has no incoming edges. In this case there is no transaction $T_j$ such that $T_j \rightarrow T_i$ since all conflicts in $\rightarrow$ are resolved through SG and $T_i$ cannot participate in a cycle in $\rightarrow$. At that point in time $T_i$'s flags (r-flags and c-flags) may be removed from FT.

**Lemma 8:** If $T_i$ is an SG root node of a committed transaction $T_i$, then $T_i$ may be deleted from SG with its corresponding FT flags without affecting database consistency.
The condition that only root nodes may be removed from SG, may cause that SG will contain, nodes or chain of nodes, of committed transaction which can not yet be removed. The process of removing those nodes is a recursive one, as the following example shows:

Let us look at Figure 5. Suppose that $T_{11}$ is a root node of an active transaction. When $T_{11}$ commits we are able to remove node $T_{11}$. As a result $T_{22}$ becomes a root node, and since $T_{22}$ has already committed it can be also removed. Later, when $T_{10}$ commits and becomes a root node we will be able to remove recursively nodes $T_{21}$ and $T_{31}$.

Algorithm 1 implements the procedure discussed above.

**Algorithm 1:** Removal of a committed transaction node from SG.

**Input:** SG and a committed transaction node $T_i$ to be removed.

**Output:** Possible removal, from SG, of $T_i$ and other root nodes of committed transactions accessible from $T_i$.

**Method:**

1. Define the following sets of nodes in SG:

   $\text{FOLLOWING}(T_i) = \{ T_j \mid (T_i, T_j) \text{ is an edge in SG} \}$

   $\text{PRECEDING}(T_i) = \{ T_j \mid (T_j, T_i) \text{ is an edge in SG} \}$

2. Execute the following recursive procedure:

   ```
   procedure DeleteTransaction(T_i)
   {
     if $T_i$ committed and $\text{PRECEDING}(T_i) = \emptyset$
     {
       for each $T_j$ in $\text{FOLLOWING}(T_i)$
       {
         Remove edge $(T_i, T_j)$
         DeleteTransaction($T_j$)
       }
       Remove $T_i$ from SG
       Remove $T_i$'s FT flags
     }
   }
   ```
7.2. Management of WAIT QUEUE for a data item

The management of WAIT QUEUE for a data item is an implementation policy decision which does not affect basic correctness of an ICCA. ICCA correctness is based on synchronization with actual executed requests, thus FIFO, PRIORITY or any other queue management policy are acceptable.

Let $T_i$ be a transaction to be added to the wait queue for some data item $X$. In case FIFO policy we synchronize $T_i$ with the other transactions on the queue immediately upon inserting $T_i$ at the end of the queue (by adding to SG edges $(T_j,T_i)$ for each $T_j$ on the queue). In this policy the order of transaction arrival is maintained. The policy is suitable for a transaction system using strict 2PL policy. However, in general, transactions may have different WFSs and may pass each other in the queue. This can be achieved by reorganizing the queue periodically; possibly upon arrival of a new request or upon removal of a flag held on $X$ from FT. In any case, the order of transactions in the wait queue must be CONSISTENT with the order of $\rightarrow$, or else unnecessary transaction restarts may occur.

For example

Let $T=\{T_1, T_2, T_3\}$ be defined as follows:

$T_1$: TRANS READ(X); WRITE(X); WRITE(Y); SNART
$T_2$: TRANS READ(Y); WRITE(Y); WRITE(W); SNART
$T_3$: TRANS READ(Z); READ(X); WRITE(Y); SNART

and use ICCA with the following mapping and standard 2PL policy:

$F_2:T_1 \rightarrow \text{2PL-ET-rw} \times \text{2PL-ET-ww}$
$T_2 \rightarrow \text{2PL-ET-rw} \times \text{2PL-ET-ww}$
$T_3 \rightarrow \text{CERT-ET-rw} \times \text{2PL-ET-ww}$
Round Robin execution of T, after T₃ executing WRITE(Y) will create the following LS and SG:


$$SG = \{(T₃,T₁),(T₂,T₁),(T₂,T₃)\}$$

where T₁ and T₃ are waiting for T₂.

If the FIFO policy is used for wait queue management then T₃ will follow T₁ in the wait queue for Y, and will be aborted when the edge (T₁,T₃) is added to SG.

However, if T₃ precedes T₁ in the wait queue for Y as dictates, execution of T will complete successfully with no aborts.

**Algorithm 2**: FIFO type QUEUE management consistent with $\rightarrow$.

**Input**: Transaction Tᵢ and a queue of transactions $\cdot Q$.

**Output**: Q augmented by Tᵢ such that for every Tⱼ in Q, if Tᵢ $\rightarrow\rightarrow$ Tⱼ then Tᵢ precedes Tⱼ in Q.

**Method**: Execute procedure AddTransactionToQ.

```plaintext
procedure AddTransactionToQ(Tᵢ)

Tᵢ <- Start transaction on Q

while Tⱼ is defined

  if Tᵢ $\rightarrow\rightarrow$ Tⱼ
    insert Tᵢ before Tⱼ
    stop
  else Tⱼ <- Next transaction on Q;

add Tᵢ to end of Q
```

**Lemma 2**: Algorithm 2 guarantees that for every pair of transactions Tᵢ,Tⱼ on Q, if Tᵢ $\rightarrow\rightarrow$ Tⱼ then Tᵢ precedes Tⱼ on Q.

**Proof**: Trivial if we note that $\rightarrow$ is maintained acyclic at all times.
B. Conclusions

B.1. Summary of Contributions

In this paper we introduced the notion of an Integrated Concurrency Control Algorithm. An ICCA consists of several rw and several ww synchronization techniques and has the capability of using any number of them concurrently. We presented some examples of ICCAs that employ 2PL and Certification based synchronization techniques along with proofs of their correctness.

The dynamic nature of the algorithm, which allows both the use of several techniques concurrently, and transition from one technique to another for a given transaction, requires a single representation of each transaction's conflict history sufficient for each technique to make correct decisions. We use PRE_WRITEs in addition to the conventional use of DM_READs and DM_WRITEs to collect this information.

Conflicts involving PRE_WRITEs essentially foresee possible conflicts between DM_READs and DM_WRITEs. The different techniques interpret and use this information in a different manner.

We use a single directed graph to represent all the precedence relations between transactions, including those that arise from the waiting forced by 2PL.

To enable a smooth transition from locking to non-locking methods we introduced the two phases of request processing (waiting and synchronization).

Finally, the notion of a transaction's wait for set was introduced as a means for ensuring correctness and as a means for controlling the concurrency level in the system.
8.2. Related Work

The primary influences on our work have been the concurrency control algorithm of Bayer et. al. [BAYE80] in which 2PL and a certification strategy were combined into a single algorithm; Bernstein and Goodman's work -- in particular their use of the decomposition theorem to derive several distributed algorithms that utilize 2PL and timestamps [BERN81], Wilkinson's centralized algorithm for a local network in which certification and locking were also combined [WILK81], and, Robinson's notion of separating policy from correctness in the design of concurrency control algorithms [ROBI82].

To the best of our knowledge only Robinson [ROBI82] has proposed a mechanism that is similar to ours. However in his proposal the concurrency control mechanism is based on serializability in validation order which simplifies the algorithm at the expense of unnecessary transaction restarts.

8.3. Present Work

Some additional theoretical work remains in characterizing the power of the model. However, at the moment we are engaged in the construction of a simulation model. In the short run we wish to arrive at a comparison of our algorithm with more conventional algorithms using an abstract measure such as effective level of concurrency. In the long run we are interested in deriving more useful measures such as effect of the algorithm on system throughput and transaction response time.
9. References


$T_1$: TRANS READ(X);WRITE(Y);WRITE(Z); SNART

$T_2$: TRANS;READ(Z);WRITE(Y); SNART

\[ T_1 \quad \quad T_2 \]

\[ \text{TRANS } T_1 \quad \quad \text{TRANS } T_2 \]

\[ \text{READ(X)} \quad \quad \text{READ(Z)} \]

\[ \text{WRITE(Y)} \quad \quad \text{WRITE(Y)} \]

\[ \text{WRITE(Z)} \quad \quad \text{SNART} \]


$L = r_1 [X] r_2 [Z] w_2 [Y] w_1 [Y] w_1 [Z]$

Figure 1: The Execution Schedule modeled by its Synchronization and Execution logs.


$L = r_1 [X] r_2 [Z] w_2 [Y] w_1 [Y] w_1 [Z]$

$L_1 = r_2 [Z] w_2 [Y] r_1 [X] w_1 [Y] w_1 [Z]$


1. $L$ is the execution log of $L^S$
2. $L_1$ is a serial execution log computationally equivalent to $L$.
3. $L_f^S$ is a serial synchronization log computationally equivalent to $L_1$
4. From (1)-(3) it follows that $L$ and $L^S$ are serializable.

Figure 2. Serializable Logs
SrW Synchronization Technique
              Srw Synchronization Technique
2PL-ET-rw        2PL-ET-ww
2PL-CT-rw        2PL-CT-ww
CERT-ET-rw       CERT-ET-ww
CERT-CT-rw       CERT-CT-ww

Figure 3: The Synchronization Techniques.

<table>
<thead>
<tr>
<th>2PL-ET-rw</th>
<th>Tj in WFSrw(Ti)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>p</td>
</tr>
<tr>
<td>Ti</td>
<td>r</td>
</tr>
<tr>
<td>p</td>
<td>Tj --&gt; Ti</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CERT-ET-rw</th>
<th>Tj</th>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ti</td>
<td></td>
<td>Ti --&gt; Tj</td>
<td>Tj --&gt; Ti</td>
</tr>
<tr>
<td>p</td>
<td>+</td>
<td>Tj --&gt; Ti</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CERT-CT-rw</th>
<th>Tj</th>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ti</td>
<td></td>
<td>Tj --&gt; Ti</td>
<td>+</td>
</tr>
<tr>
<td>c</td>
<td>Tj --&gt; Ti</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

LEGEND

+  request granted.
-  request not granted on conflicts with active transactions
*  irrelevant

Ti --> Tj  add to SG edge (Ti,Tj)
Tj --> Ti  add to SG edge (Tj,Ti)

Figure 4: Compatibility-Action Matrices for the Synchronization Techniques
Figure 5. Typical cut of SG