The Private Workspace Model Feasibility and Applications to 2PL Performance Improvements

Israel Gold, Oded Shmueli and Micha Hofri.

Computer Science Department
Technion – Israel Institute of Technology
Haifa, 32000 Israel

Abstract

In the *private workspace model* of concurrency control the transaction manager, TM, maintains a *private workspace* for each transaction. Data items accessed by a transaction, regardless of access mode, are cached in this workspace. At transaction commit time updates are made permanent in the database.

This paper addresses two basic issues. First, the *feasibility* of the model is exhibited by introducing a relatively straightforward and efficient *parallel commit phase algorithm* in which no I/O operations are associated with a critical section of the TM. Second, by simulation experiments, a concurrency control method in which readers use certification whereas writers use 2PL and do not wait for readers is shown to usually outperform the "standard" 2PL method within the private workspace context. The detailed physical model used in the simulation captures the basic properties of the private workspace idea.

1. Introduction

This paper addresses two basic issues. First, we demonstrate that concurrency control methods developed under the private workspace assumption can be matched with an efficient recovery management procedure. Second, a "non-standard" concurrency control method is shown to usually outperform the "standard" 2PL method within the private workspace context.

Many centralized concurrency control algorithms have been proposed. A large portion of these algorithms are based, to one degree or another, on the Two Phase Locking method (2PL) [ESWA76] in which blocking is used to synchronize conflicting transactions. Others rely on methods which allow conflicting transactions to run concurrently and use transaction restart in cases where inconsistent updates to the database could result [BADAL79, KUNG81]. These methods are called Certification methods because they either abort a transaction or certify that a transaction may perform additional processing or commit.

In the *private workspace model* of concurrency control the transaction management component of a database management system (DBMS) maintains a *private workspace* for each transaction. Data items accessed by a transaction, regardless of the access mode, are cached in this workspace. At transaction commit time updates are made permanent in the database. Two major advantages follow from the fact that writing new values into the private workspaces of transactions does not affect the database state. First, a transaction restart may not cause cascading restarts. Second, the Concurrency Control has complete freedom in choosing how to synchronize conflicting transactions. This facilitates the construction of correct concurrency control algorithms which *concurrently* employ both *blocking* and *transaction restart* for synchronizing conflicting transactions. This leads to the notion of an *Integrated Concurrency Control Algorithm* (ICCA) which is concurrently employs *several* rw and *several* ww synchronization techniques (the notion of a synchronization technique is introduced in [BERN81]).

ICCA algorithms based on 2PL and Certification synchronization techniques were presented in [BORAL84]. ICCA advantages depend on transactions using a private workspace and on using Serialization Graph based synchronization techniques which only detect actual conflicts (i.e., those affecting serialization) among transactions (as opposed to possible conflicts as done by the optimistic method in [KUNG81]). To complicate matters, some ICCAs use Commit Time Synchronization (CTS) to ensure that committing a transaction does not cause database inconsistencies.

The commit procedure presented in this paper relies on an *atomic write phase*. The effect is not a disk access but the placement of the Write operations on appropriate queues. Once the transaction's Write operations are

issued, another transaction trying to read an updated item will obtain the correct result, either from the disk or from a buffer. The end results of this procedure is a faster commit phase. Complications arise in integrating the idea of an atomic write phase with CTS and a recovery mechanism. On one hand, the CTS and the atomic write phase must be executed in a critical section in order to preserve the integerity of CTS. In order to perform the commit as fast as possible, no Read operation should take place during this critical section so that it is actually a critical section of the entire DBMS. On the other hand, committing a transaction (which involves I/O operations) must take place only *after* the CTS and *before* issuing the Writes. Performing I/O operations in a critical section may slow the system considerably.

The proposed parallel commit phase algorithm performs I/O operations outside a critical section while still detecting only "real" serialization conflicts. Even in those cases where no CTS is necessary, the algorithm enhances performance by queuing commit record writing requests on a commit queue. All transactions blocked by the committing transaction, which is not yet actually committed, may resume. Only one outstanding commit request of an updating transaction on the commit queue is allowed. A transaction is completed as soon as its commit record is known to reside in stable storage.

A substantial comparative simulation study was conducted in order to determine the advantages of private workspace based concurrency control algorithms. Three important aspects of the simulation need be pointed out. First, we model a CPU bound system with infinite parallel I/O capability. Second, because we are mainly interested in concurrency control aspects, the influence of any buffer and memory management issues is factored out in the simulation. Third, as the space of possible work environments (as defined by transaction workloads) is enormous, an alternative testing method was sought. We have decided to characterize work environments by their effect on transactions' blocking rate and system resource utilization. This substantially reduced the number of experiments.

The experiments conducted were designed to investigate the performance of 2PL *relative* to a concurrency control method (ICCA) in which readers (queries) use certification (Serialization Graph Checking) whereas writers (updating transactions) use 2PL and do not wait for readers. The results show that the ICCA method tested is (almost) always at least as good as 2PL. In environments with high data contention and no system resource contention (i.e., low to high CPU utilization), the ICCA method performs significantly better. For transactions using 2PL the algorithm modeled is similar to the "ordinary" 2PL with a deferred updates recovery mechanism [GRAY78].

Various concurrency control algorithms have been compared in an attempt to evaluate their operational merit. The studies range from the purely abstract [PAPA79] to the more realistic (where such measures as system throughput and the cost of the concurrency control mechanism are taken into account) [AGRA83, CARE83, GALL82, LIN82, PEIN83, ROBI82, TAY84]. Our results confirm some previously observed phenomena. In particular, certain effects of high data contention predicted in [TAY84] and of high resource utilization (by [CARE83] in the case of I/O resources), were observed.

The paper is organized thus. In Section 2 an overview of the Private Workspace Model is presented. In Section 3 we exhibit the feasibility of the model by presenting a general Parallel Commit Phase algorithm thereby integrating the concurrency control and recovery subsystems. Section 4 describes the simulation model and the performance experiments conducted. Conclusions appear in Section 5.

2. The Private Workspace Model

In the *private workspace model* of concurrency control a private workspace is allocated to each active transaction. In this workspace the transaction caches its previously read data items and those data values written by the transaction during its execution. At transaction commit time, its updates are made permanent in the database. A similar model has been previously used in [BERN81, KUNG81].

Following the system model proposed in [BERN81,BERN83], a centralized database management system may be decomposed into three components: a transaction manager (TM), a data manager (DM) and data storage. The storage consists of a *stable storage*, a *DM buffer* and a *TM workspace*. Stable storage survives system failures (models a disk). It is divided into fixed size *physical pages*, it has an (almost) unlimited capacity and is relatively slow. The DM buffer and TM workspace model a limited capacity fast main memory which does not survive system failures. The database consists of a set of logical pages stored in a por-

tion of the stable storage called the *stable database*. Other portions of stable storage, called *transactions' virtual workspaces*, are used to store copies of transactions' updates for recovery purposes. Users interact with the DBMS by executing programs called *transactions*. From a transaction's viewpoint the database consists of a collection of logical *data items* denoted {..., X,Y,Z }. Transactions issue requests to read and write data items from the database. For simplicity of the model we assume that the granularity of a data item is a page and thus any reference to a data item is a reference to some logical database page[1].

Transactions communicate with the TM and the TM communicates with the DM. The DM is responsible for managing the database (i.e., accessing and modifying data). The TM controls interactions between transactions and the database and is charged with concurrency control and recovery functions. It receives the requests issued by the transactions and controls the order in which these are received by the DM. 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 NEW_VALUE is assigned to data item X in the database.

The TM-DM communication is as follows. The DM receives a stream of operations queued by the TM. On a DM_READ operation, the DM installs the desired item in the private workspace of the appropriate transaction and notifies the TM upon completion. Similarly, on a DM_WRITE operation, the DM moves the desired item which is fetched from the appropriate transaction private workspace to stable storage; it notifies the TM upon completion. The DM may execute several operations concurrently provided it *preserves the order of conflicting operations*.

The TM maintains a private workspace for each active transaction in which copies of data items read or written by the transaction are kept. All references to data items in the private workspace are made through the TM. From the TM point of view, a transaction executes four types of requests: TRANS, READ, WRITE, and SNART. Actions taken by the TM upon receipt of these requests are detailed below.

[1] In case the granularity of a data item is smaller than a page the TM can effectively compute the page in which any given data item is stored. Thus, any request for reading or writing a data item can be effectively translated into a request for reading or updating a logical database page.

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 and no DM_READ is issued. Otherwise, the TM issues a DM_READ(X) operation to the DM. When the current value of X is installed by the DM in the transaction's private workspace, the TM is notified and the value is forwarded to the transaction.

WRITE(X,NEW_VALUE): NEW_VALUE is a value to be assigned to X. The TM executes a PRE_WRITE(X,NEW_VALUE) operation into the transaction's private workspace. If a copy of X exists in the private workspace then this has the effect of updating the previous value of X *in the private workspace* to NEW_VALUE. Otherwise, X is created in the workspace with the value NEW_VALUE. Note that a PRE_WRITE operation does not alter any value in the item database. The PRE_WRITE operation may be used as a synchronization primitive (see [BORAL84]).

SNART: The transaction is restarted by the TM if committing it (by making its updates permanent in the database) will result in an inconsistent state. Otherwise, the TM issues a DM_WRITE operation for every item X previously referenced by a PRE_WRITE operation. This has the effect of installing the last update to X in the private workspace as a permanent value in the item database. All the transaction's DM_WRITEs are executed *atomically;* after all have been issued the transaction is complete.

A transaction execution is divided into two phases. In the *Execution Phase* (similar to the read phase in [KUNG81]) the transaction reads values from the database, performs various computations and writes results into its private workspace. In the *Commit Phase*, (similar to the validation and write phases in [KUNG81]) which takes place after the transaction finishes all computations, the transaction first goes through a (possibly empty) *Commit Time Synchronization (CTS)* to ensure that committing it causes no inconsistencies, and then it issues a (possibly empty) sequence

DM_WRITE operations (which instructs the DM to update the database). The Commit Phase should be *atomic*[2]. In the next section we present a physical implementation of the commit procedure. A transaction T_i may be *restarted* by the TM any time before a DM_WRITE has been executed on its behalf. The effect of restarting T_i is to obliterate its private workspace and to treat it as a new incoming transaction.

2.1. The Transaction Manager Model

Two data structures are required by the TM for its operation. The Serialization Graph (SG) represents a precedence relation among conflicting transactions. The Indicators Table (IT) maintains the database access history. A node in SG represents an active or a committed transaction. An edge (T_i, T_j) in SG indicates that in any possible computationally equivalent serial execution order, transaction T_i precedes transaction T_j. SG is used to represent all such precedence relationships whether they originated from blocking in 2PL or from the detection of a conflict in a Certification algorithm.

There is an entry in IT for every data item that has been accessed. An entry consists of several pairs of the form <INDICATOR, TRANSACTION IDENTIFIER>; each pair identifies a transaction that accessed the data item and its access mode (Read or Write). No restriction is placed on the number and/or type of pairs in an entry associated with a data item. The concurrency control mechanism interprets the pairs and decides how to use that information.

There are three types of indicators allowed in a pair <I, TID>:

(1) An *r-indicator* indicates that a DM_READ operation was executed on this item on behalf of transaction TID.

(2) A *p-indicator* indicates that a PRE_WRITE operation was executed on this item on behalf of *active* transaction TID. (During the commit phase, all p-indicators associated with a transaction are converted into c-indicators.)

(3) A *c-indicator* indicates that a DM_WRITE operation was executed on this data item on behalf of *committed* transaction TID.

A TRANS request causes the TM to add a node to SG representing the new transaction.

A READ or a WRITE request received by the TM undergoes a (possibly empty) *waiting phase;* then a (possibly empty) *synchronization phase* followed by execution of the request. In the waiting phase, a transaction using 2PL is forced to wait until some other transactions which "hold" conflicting indicators on the same data item have completed executing. Edges are appropriately added to SG in order to reflect the precedence relation imposed by blocking. A transaction using Certification is allowed to continue immediately to the synchronization phase. In this phase, the request is synchronized with conflicting operations from other transactions. This may result in restarting the issuing transaction or in continuing execution. Edges are appropriately added to SG in order to reflect the precedence relation imposed by executing this request. Request execution includes appending the appropriate indicator to IT and issuing the appropriate DM_READ or PRE_WRITE operation.

A SNART request triggers the *commit phase* of the transaction as described in the previous section. In this phase all the transaction's p-indicators are converted into c-indicators. Information in IT and SG about a *committed* transaction remains in these data structures as long as the node representing it in SG is not a root node, i.e. it has at least one incoming edge. All traces of a *restarted* transaction are removed from both SG and IT, i.e. the node representing a restarted transaction in SG is removed together with all its incoming and outgoing edges. All pairs detailing accesses made by the restarted transaction are removed from IT.

2.2. The Data Manager Model

The DM functions similarly to a back-end database processor. It receives a *serializable* schedule of DM_READ and DM_WRITE operations which is output by the TM and it processes it on a First Come First Served (FCFS) basis. To take advantage of its parallel I/O processing capability, the DM may execute non-conflicting operations in *parallel*. However, under the serializability constraints it must execute conflicting operations serially (i.e., one must complete before the other begins) in the order output by the TM[3]. This may be implemented

[2] The actual implementation of the commit procedure need not be atomic as long as it appears atomic to the outside world.

[3] The TM may use a certification algorithm in which serializable conflicting DM operations are scheduled immediately one after the other without waiting for DM response. It is up to the TM to decide whether to postpone the scheduling of a new DM operation before an old conflicting one has been acknowledged.

by associating a queue of requests with each "active" data item.

The DM buffer is divided into page frames of size equal to that of a stable storage page. The DM handles all stable database I/O (and other portions of stable storage I/O) through its buffer. The DM may operate in two modes. It may force a page write (resp. read) from (resp. into) the buffer into (resp. from) stable storage or it may use some buffer management strategy as described below. If possible, DM_READ(X) returns data item X directly from the buffer; otherwise, an empty page frame is selected and loaded by a desired stable database page.

A DM_WRITE(X,NEW_VALUE) overwrites page X if it is in the buffer; otherwise, an empty page frame is selected and NEW_VALUE is simply moved into the buffer[4]. At some later time the DM may write the page to the stable database. A page involved in a DM_WRITE operation must be pinned (see [LIND79]) in the buffer for the duration of the update. This prevents transferring it to stable database before the update is completed. The DM is allowed to select and write to the stable database any non-empty unpinned page. The DM may use any page replacement algorithm.

3. Parallelism of the Commit Phase

The main difficulty in practically using the private workspace model lies in attaining an *efficient* implementation of the *atomic commit phase*. Complications arise in integrating the idea of an atomic write phase with CTS and a recovery mechanism. On one hand, the CTS and the atomic write phase must be executed in a critical section in order to preserve the integerity of CTS (see discussion in section 3.2). The commit should be performed as fast as possible, therefore no DM_READ operations should take place during this critical section which makes it a critical section of the entire DBMS. On the other hand, committing a transaction (which involves I/O operations) must take place only *after* the CTS and *before* issuing the DM_WRITEs. Performing I/O operations in a critical section may slow the system considerably. In this section a *parallel commit phase algorithm* which only performs I/O operations outside a critical section together with a recovery procedure is proposed.

[4] If data item granularity is smaller than a page then a DM_WRITE(X) operation fetches the stable database page containing X into the buffer. Otherwise, an empty page frame is selected and replaced by the desired stable database page.

3.1. Supporting Atomic Commit

Supporting *atomic commit* means reaching a well defined *commit point* during transaction execution. If a system failure occurs *before* that point then the transaction's effects will eventually be undone and if a system failure effects will eventually be installed in the database and its system status would be 'completed'. In the private workspace model, the installment of a transaction's updates is deferred until the TM decides to commit the transaction. Thus, achieving the property of *atomic commitment* may be done by requiring that either all or none of a transaction's DM_WRITE operations are processed.

Atomic commitment is done in *two phases*[5] [LAMP76, GRAY78]. Let T_i be a committing transaction. After T_i is *validated* by the *CTS*, the *first phase* of commit begins. In this phase the TM *does not* issue T_i's DM_WRITE operations directly. Rather, it instructs the DM to force the *post-images* of those data item values written into T_i's private workspace out to T_i's *virtual workspace* on stable storage, followed by an additional commit record. Only during the *second phase*, does the TM issue the DM_WRITE operations for data items in T_i's private workspace. These operations instruct the DM to update the database.

In the event of a system failure, all transactions' *virtual workspaces* are inspected. If a commit record is detected for a given (transaction) workspace, then its *post-images* are reinstalled in the database; otherwise, the workspace is discarded. In order to ensure that the post-images of data items will be reinstalled in transactions commit order, each transaction, upon reaching its commit point, is assigned a *Transaction Commit Number (TCN)*, which is part of the commit record. For reasons which will be discussed later, each data item X in the database is associated with both the transaction identifier (X.tid) and the TCN (X.tcn) of the last transaction which has updated it.

[5] This is similar to the *two-phase commit* algorithm used in distributed databases.

3.2. A Parallel Commit Phase Algorithm

A transaction's *Commit Phase* takes place after it has finished its *Execution Phase*. During this phase the TM must complete three tasks. First, the (possibly empty) *CTS* which ensures that committing the transaction will not cause any database inconsistencies. Second, the (possibly empty) *commit* procedure which ends by issuing the transaction's DM_WRITE operations and the conversion of its p-indicators into c-indicators. Third, a (possibly empty) *cleanup phase* in which the transaction is removed from SG and IT. The term "commit phase" is somewhat misleading since the phase includes the possibility of restarting the transaction. A transaction is *completed* as soon as its commit record is known to reside in stable storage. The basic commit phase procedure is given in Figure 1.

```
procedure BasicCommitPhase(Ti)
begin
   certify <- CTS(Ti);
   if certify then
   begin
     (* first phase of commit *)

      ∀ X ∈ Writeset(Ti) send DM a request to force out
          the post-image of X to Ti's virtual workspace;
      Wait[6] for DM "Ready to Commit" message;

      TCN <- TCN+1;  (* get new commit number *)

      send DM a request to force out
         a commit record to Ti's virtual workspace;
      wait for DM "committed" message;
      send a completion message to Ti;

     (* second phase of commit *)

      ∀ X ∈ Writeset(Ti) do
      begin
          X.tid <- Ti; X.tcn <- TCN;
          execute DM_WRITE(X);
      end
      convert Ti's p-indicators into c-indicators;
   end

     (* cleanup  phase *)

   if certify then remove Ti from SG and IT
              else RESTART(Ti);
end
```

Figure 1. Basic Commit Phase Procedure.

If the BasicCommitPhase procedure is executed in a critical section of the TM then, clearly, the database is kept consistent. In general, executing it in parallel is incorrect as the following scenarios demonstrate.

Consider a committing transaction T_i requiring CTS. The subtle point is that T_i's CTS would only be *partially* correct. At the time T_i is validated by CTS there may be another concurrently executing transaction T_j which has completed its Execution Phase but has not yet completed its first commit phase. T_i's ww conflicts with T_j may not be resolved since T_j has not yet executed its second phase of commit and does not yet own its c-indicators. If T_j subsequently commits and issues its DM_WRITE operations before T_i does, database inconsistencies may result.

In certain cases[7] the CTS for updating transactions is a priori guaranteed to be successful. It seems that in such cases it is not essential to exactly record ww conflicts. However, consider a reader executing in parallel which uses SG checking. To properly synchronize this reader the TM must have all the ww information; otherwise, some conflicts affecting serialization will go undetected.

One way to overcome these problems is by enclosing the T_i's second commit phase, preceded by an additional CTS, in a critical section of the TM. This is implemented by the ParallelCommitPhase procedure shown in Figure 2. The critical section is enclosed by " << " and " >> ". Note that this is a critical section of the DBMS during which any access to the database is blocked as opposed to the critical section in [KUNG81] which applies only to committing transactions and allows execution of DM_READ operations in parallel.

[6] A wait frees the TM to serve other transactions; T_i is eligible for TM service when the wait condition holds.

[7] e.g. the W2PL algorithm in [GOLD85].

```
procedure ParallelCommitPhase(Ti)
begin
    (* first phase of commit *)
(*      preliminary Commit Time Synchronization      *)
  certify <- CTS(Ti);
  if certify then
  begin
     ∀ X ∈ Writeset(Ti) send DM a request to force out
        the post-image of X to Ti's virtual workspace;
     wait for DM "Ready to Commit" message;
     (* second phase of commit *)
(*      second phase Commit Time Synchronization      *)
     <<  certify <- CTS(Ti);
     if certify then
     begin
       TCN <- TCN+1; (* get new commit number *)
       send DM a request to force out
          a commit record to Ti's virtual workspace;
       wait for DM "committed" message;
       send a completion message to Ti;
       ∀ X ∈ Writeset(Ti) do
       begin
          X.tid <- Ti; X.tcn <- TCN;
          execute DM_WRITE(X);
       end
       convert Ti's p-indicators into c-indicators;
     end >>
  end
    (* cleanup phase *)
  if certify then remove Ti from SG and IT
          else RESTART(Ti);
end
```

Figure 2. Parallel Commit Phase Procedure

Given the ParallelCommitPhase procedure, consider a transaction T_i having a large writeset. After T_i is validated by the preliminary CTS, the TM begins forcing T_i's post-images to disk. Throughout this phase, T_i is in "doubt" since it may be restarted later by the CTS of the second phase. In order to decrease the probability that a transaction which requires CTS would be restarted during its commit phase, it might prove beneficial to force out the transaction's post-images to disk during its Execution Phase. This eliminates the time interval in which a committing transaction is in "doubt". Naturally, the above is not required for transactions whose CTS is empty or is a priori successful.

3.3. An Improved Parallel Commit Phase

The ParallelCommitPhase procedure (Figure 2) has a severe limitation resulting from associating an I/O operation with a critical section (of the TM). To improve the situation the I/O operation associated with forcing out the commit record should be moved outside the critical section thereby speeding up commitment. This may be implemented by queuing commit record writing requests, in TCN order, on a dedicated COMMIT QUEUE. This queue is serially processed by the DM on a FCFS basis.

The latter suggested optimization may result in some certified transactions (possibly) updating the database before their commit record has reached stable storage, i.e., before they were *actually* committed. The possibility of "dirtying" the database need not worry us if *only one certified non-committed writer* (updating transaction) *at a time* is allowed. This is achieved by preventing new writers from entering their second phase of commit until the current committing writer's commit record is in stable storage. In case of a system failure the database is restored into a consistent state by reinstalling the post-images of the transactions which have affected it in commit order (i.e., in TCN order).

Procedure RECOVERY implements system restart (see Figure 3). The main difficulty lies in finding the TCN of the certified non-committed writer which has affected the stable database (if one exists). The TCN of a committed transaction is simply found by locating its commit record. The TCN of *the* certified non-committed writer T_i is found as follows. For each active transaction T_i, each data item X appearing in its virtual workspace is read from the stable database. If T_i was the last transaction to affect X then T_i has (de facto) committed; its TCN is found in X.tcn. If T_i's TCN cannot be found then its virtual workspace may be discarded (since T_i has not affected the stable database).

It may happen that readers (queries) which do not update the database and are not required (for recovery reasons) to own a commit record, may violate database consistency by reading "uncommitted" data items. For example, let T_i be the current non-committed writer whose DM_WRITE operations have already been issued by the TM. Suppose transaction T_j has managed to read a data item updated by T_i (the value may be served from the DM buffer) and has completed successfully before a system failure. If the failure occurs before T_i's commit record has reached stable storage and before T_i has affected the stable

database, then T_i's effects will eventually be undone. Thus T_j has seen a value which never existed. However, had we required that T_j be committed *after* T_i then this problem would disappear as both transactions would be restarted.

The above leads to a general solution for readers that possibly read "uncommitted" values written by a certified non-committed writer. Readers are required to queue a *null commit request* (which need not involve any I/O) onto the COMMIT QUEUE. A reader completes only after its null commit request is dequeued. Thus, a reader can commit only after the writers it has read from have committed. The ImprovedParallelCommitPhase procedure below incorporates the above ideas.

```
procedure ImprovedParallelCommitPhase(Ti)
begin
  (* first phase of commit *)

certify <- CTS(Ti);
if certify then
begin
   ∀ X ∈ Writeset(Ti) send DM a request to force out
      the post-image of X to Ti's virtual workspace;
  wait for DM "Ready to Commit" message;

  if Ti is a writer then wait until
       cur_committing_writer is known to be committed;

    (* second phase of commit *)

   <<  certify <- CTS(Ti);
  if certify then
  begin
       (* NO I/O is associated with this phase *)

      if Ti is a writer then
           cur_committing_writer <- Ti;
      TCN <- TCN+1;  (* get new commit number *)
      Enqueue Ti's commit request on the COMMIT QUEUE;
       ∀ X ∈ Writeset(Ti) do
      begin
          X.tid <- Ti; X.tcn <- TCN;
          execute DM_WRITE(X);
      end
      convert Ti's p-indicators into c-indicators;
    end  >>
  end

   (* cleanup phase *)
  if certify then
  begin
    remove Ti from SG and IT;
    wait for DM "committed" message;
    Ti is known to be committed;
    send a completion message to Ti;
  end
  else RESTART(Ti);
end
```

Figure 3. Improved Parallel Commit Phase

Currently there can be at most one pending commit request of an updating transaction on the COMMIT QUEUE. Therefore new writers cannot commit until the current committing writer's commit record is known to be in stable storage. This restriction may be removed resulting in more parallelism and increased overhead. The idea is to require each updating transaction (which might "dirty" the database) to write to stable storage (in addition to its post-images) the set of pairs (X.tid, X.tcn) for each data item X read by the transaction during its Execution Phase. Practically, this set is usually small since the TCNs of transactions which have already committed need not be included; these TCNs can be determined by inspecting the TCN of the transaction heading the COMMIT QUEUE. Procedure RECOVERY is

```
procedure RECOVERY
begin
  AL <- { Ti | Ti has a virtual workspace }; CL <- φ;

  (* add to CL all pairs (Ti,Ti's TCN)
     such that Ti has affected the database *)

  ∀ Ti∈AL do
  begin
    (* add Ti to CL if it is known to be committed *)

    if ∃commit-record(Ti) then begin
         tcn <- Ti's TCN found in the commit-record;
         CL <- CL ∪ (Ti, tcn);
    end else
    begin

      (* Add Ti to CL if it is a certified non-committed
         transaction which has affected the database;
         otherwise discard it *)

       tcn <- null;
       ∀ X ∈ Writeset( Ti ) do begin
          DM_READ(X);
          if X.tid=Ti then tcn <- X.tcn;
       end
       if tcn = null then discard Ti's virtual workspace
                     else CL <- CL ∪ (Ti, tcn);
    end (* of else *)
  end

    (* restore the database to a consistent state by
       reinstalling the post-images of transactions
       in CL in TCN order
    *)

  sort the pairs (Ti, tcn) of CL in ascending tcn order;
   ∀ (Ti, tcn) ∈ CL in the sorted order do
        ∀ X ∈ Writeset( Ti) do begin
           X.tid <- Ti; X.tcn <- tcn;
           DM_WRITE(X);
        end
end; (* of RECOVERY *)
```

Recovery Procedure at Sytem Restart

modified by adding to CL the pairs (X.tid, X.tcn) "seen" by an updating transaction. This guarantees that a transaction whose updates affected the database will eventually be *redone*. The additional benefit of the above idea is the ability to "batch" several commit record writing requests which divides the commit operation I/O cost over several transactions[8].

4. Performance Experiments

This section describes the simulation experiments comparing the performance of 2PL with that of a concurrency control method in which readers use Certification, writers use 2PL and writers do not wait for readers. For transactions using 2PL the algorithm is similar to the "ordinary" 2PL with a deferred updates recovery mechanism [GRAY78]. Central to our simulation approach is a detailed simulation model of a centralized database management system with a fixed number of transaction processors originating transactions.

4.1. The Simulation Model

In the simulation model, aspects that are not directly related to transaction management are ignored. Therefore, we did not consider the cost of process communication, nor did we concern ourselves with buffer (or memory) management issues. This was done so that observed differences in results can be directly attributed to the differences in the concurrency control mechanisms employed. Our model assumes unlimited memory and some fixed cost is associated with concurrency control and various system services.

4.1.1. The Logical Model

The logical structure of the model is illustrated by Figure 4. It is derived from a distributed database management system architecture model [BERN81]. The model consists of four logical components: Transaction Processors (TPs), a Concurrency Controller (CC), a Data Manager (DM) and a Database.

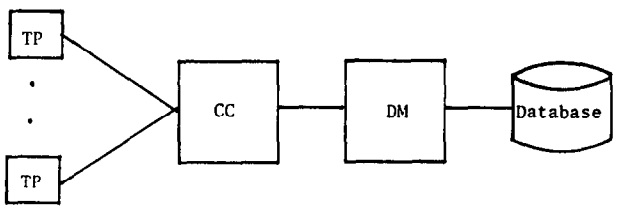

Figure 4. DBMS Logical Structure

[8] The idea of "batching" commit records is pointed out in [WILK81].

Transaction Processors (TPs)

A TP models a terminal or a user process which produces one transaction at a time. Each TP waits for some *think time*, executes a transaction and waits again before initiating another transaction. The think time controls the arrival rate of transactions.

When a transaction is initiated by a TP it is assigned a *script* consisting of the data items that it has to read and write during its execution. First, it performs *startup processing* tasks such as transaction analysis, authentication and other preliminary steps. Once this phase is complete the transaction executes a sequence of *local processing* and database *requests* bracketed by TRANS and SNART requests signaling the start and the end of the transaction, respectively. Local processing models the transaction work associated with each data item. Database requests are queued on the CC *request queue*.

Concurrency Control (CC)

The CC models a process which synchronizes the execution of transactions. It accepts requests which are queued by transactions, performs concurrency control functions and forwards service requests to the DM. We assume that the private workspaces and data structures required by the CC are maintained in primary memory and thus the CC does no I/O.

The CC continuously processes requests dequeued from its request queue. Let T_i be the transaction which is currently served by the CC. Upon T_i's TRANS request, the CC performs *transaction initialization* functions including private workspace management tasks and private workspace allocation for T_i. A TRANS request is always granted.

Upon a READ or a WRITE request, the CC performs (a possibly empty) *conflict analysis* as dictated by the concurrency control method used to synchronize T_i. If the request is granted, the CC executes a DM_READ or a PRE_WRITE operation on behalf of T_i. In an operation mode in which T_i's post-images should be forced to disk during its Execution Time, the PRE_WRITE operation is also interpreted as a request for DM processing. If the CC decides to *block* T_i, then the transaction is inserted into the *wait queue* for the data item it has requested. If at some later point in time the CC unblocks T_i, then T_i is put on the *front*

of the *request queue*[9]. If the CC decides to *restart* T_i, then a *cleanup* operation is performed and T_i is enqueued on (the *back* of) the *request queue* after a certain *restart delay period*. The purpose of this delay is to allow the transactions with which T_i has conflicted to finish before T_i is restarted.

A SNART request triggers the commit phase of transaction T_i; it is implemented by the parallel commit phase algorithm given in Figure 3. The commit phase begins with the (possibly empty) CTS which ensures that committing T_i will cause no database inconsistencies. Once T_i is validated and is ready to commit, i.e., it has already completed its first phase of the two phase commit protocol (which is always true for readers), the CC executes all the concurrency control functions required to commit T_i. These include: the issuing of T_i's DM_COMMIT operation (which instructs the DM to force T_i's commit record out to disk), issuing T_i's DM_WRITE operations, the conversion of T_i's p-indicators into c-indicators and (possibly) removing T_i from SG and IT. T_i is *committed and completed* as soon as its DM_COMMIT operation has been processed by the DM.

If T_i has not written its post-images to disk prior to issuing its SNART request, then, following the preliminary CTS, T_i is not yet ready to commit. In this case the CC must first execute DM_POST operations. These instruct the DM to force T_i's post-images out to disk. Then, the CC must wait until the DM responds with a "Ready To Commit" message and only then can it start with the second phase of the commit procedure.

Data Manager (DM)

The DM models a process which manages the data, performing functions which are similar to those performed by a back-end database processor. The DM accepts the DM_READ, DM_WRITE, DM_COMMIT, PRE_WRITE and DM_POST operations queued by the CC. DM_READ and DM_WRITE operations are queued on special dedicated queues and are processed by the *Parallelizer*[10] algorithm. Since no buffer management is modeled, each DM_READ operation implies an I/O service. If there are two consecutive pending DM_WRITE operations for a data item, the first one is discarded. DM_COMMIT operations are queued on a dedicated COMMIT QUEUE and are processed on a FCFS basis (readers' DM_COMMIT operations involve no I/O; they are processed by simply removing them from the queue). PRE_WRITE and DM_POST operations are executed immediately upon arrival, concurrently with all other DM operations. Each service by the DM involves a certain CPU processing followed possibly by an I/O processing.

4.1.2. Transaction State Diagram

Using the above description of the TP, CC and DM components, the transaction state diagram given in Figure 5 is derived. This diagram presents the sequence of logical states through which a transaction passes during its execution.

Each logical state is associated with a request for CPU service followed by a possible request for I/O service. STARTUP CPU and I/O, and LOCAL CPU and I/O represent service requests on behalf of a TP process. CCinit, CCconflict, CCcleanup, CT-Synch and CCcommit represent CPU service requests on behalf of the CC process while DM CPU and I/O represent service requests on behalf of the DM process.

DMcpu and DMio are the CPU and I/O costs associated with the DM *randomly* reading or writing a data item. For the sake of clarity we give different names to the parameters DMcpu and PRE_WRITEcpu, although they actually have identical values. This also holds for the parameters DMio and PRE_WRITEio. DMPOSTcpu and DM_POSTio model the CPU and I/O costs associated with the DM *consecutive* writing of post-images to a transaction's virtual workspace. DM_POSTio consists of the virtual workspace *access time* (associated only with the first writing in a sequence) and the post-image *transfer time* (associated with each

[9] T_i is given higher priority over other transactions in the request queue in order to minimize the possibility of starvation.

[10] To accommodate the DM parallel I/O processing capability a new system component, the Parallelizer, is introduced. The *Parallelizer* converts the *serialized* schedule output by the CC into a *parallelized* schedule in which no conflicting operations are scheduled concurrently, i.e., it enables the execution of non-conflicting operations in parallel while maintaining the serial execution order of conflicting operations. If the Parallelizer is not a standard system component, then it may be straightforwardly constructed.

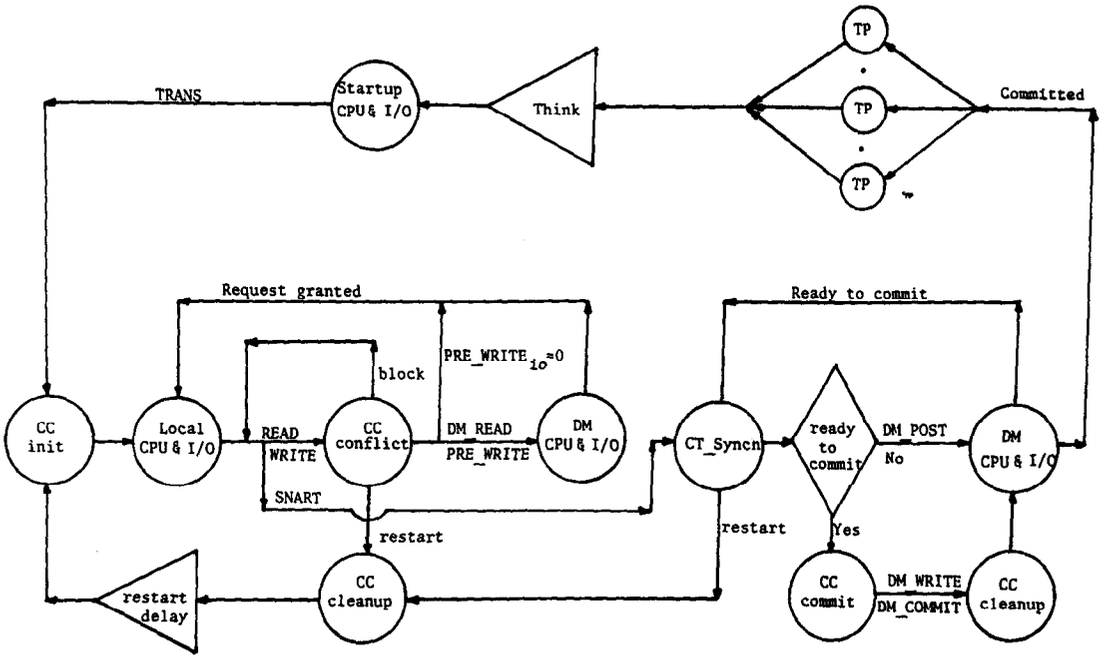

Figure 5. Transaction State Diagram

post-image). The parameter RESTARTdelay determines the period of time for which the CC delays a transaction before restarting it. For simplicity all these parameters represent constant values rather than stochastic ones. Finally, the THINKtime parameter is the mean of an exponential time distribution which models TP thinking time.

A summary of the parameters used to determine the delay time or request service time at each logical state is given in Table 3. All parameter values are specified in milliseconds.

4.1.3. The Physical Model

The logical model described in the previous section utilizes two physical resources, CPU and I/O devices (disks). Some use of these resources is associated with each CPU or I/O service in the transaction logical state diagram. The physical setting is a collection of terminals, a CPU server and an I/O server as shown in Figure 6. The CPU server has three queues servicing requests for the CC, the DM and the TPs.

The I/O server is assumed to have an *infinite parallel processing capability* and hence does not block transaction execution. This critical assumption is made in order to

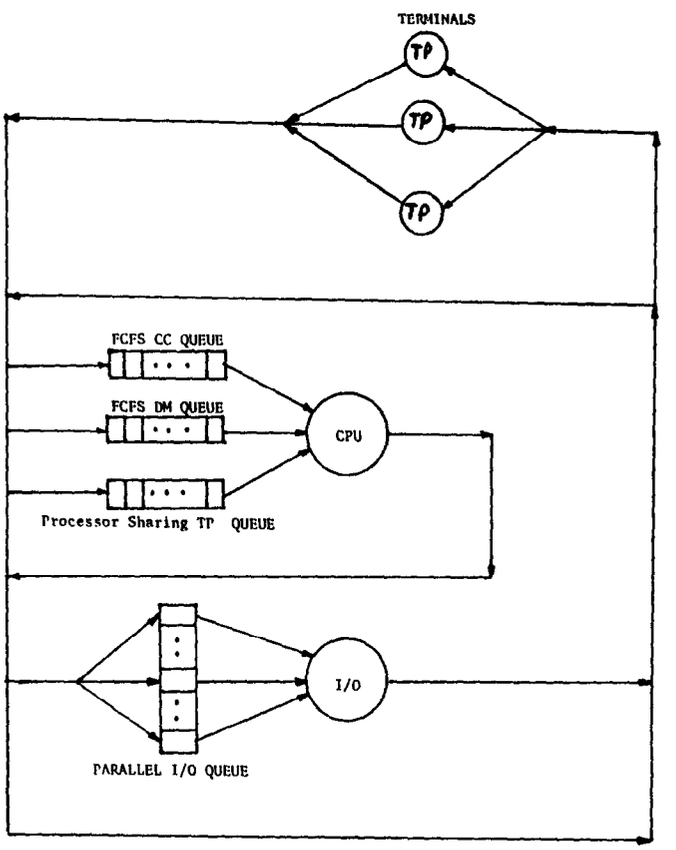

Figure 6 - DBMS Physical Model

model the real world situation in which the system is CPU bound and I/O utilization is low. Otherwise, transaction response time would mostly measure I/O queuing delays and hence differences between concurrency control algorithms would be difficult to detect (as preliminary experiments showed).

There may be pending service requests in all CPU queues. In such a case, CC requests are given first priority, DM requests are given second priority and TP requests are given the lowest priority. This policy approximately models a priority based system in which CC services are executed atomically at highest priority, lower priority is given to the DM and the lowest priority is identically given to all the TPs. Service in the CPU CC Queue and the CPU DM Queue is provided on a FCFS basis while service in the TP CPU Queue is provided using the processor sharing policy; the latter may be seen as a limiting case of the common Round Robin scheduling policy.

4.2. Experimental Setup

The experiments presented here were designed to investigate the performance of 2PL *relative* to a concurrency control method, named ICCA1[11], in which readers use Certification (Serialization Graph Checking) whereas writers use 2PL and do not wait for readers. Since the relative performance of the algorithms depends on the *conflict rate* among the transactions, we have decided to vary the amount of data contention by fixing the database size and then varying the number of concurrently executing transactions. In the experiments the **database size was fixed at 1024 data items** and the Multi Programming Level (MPL) ranged from 16 to 128 TPs.

The duration of an experiment run is defined by a ***run count*** parameter which is the number of transactions that must be committed before the experiment is halted. For each MPL value the simulation is initiated for a run count of 1000 during which no statistics are collected. The simulation is continued for a run count of 10000 during which statistics are gathered.

Five transaction classes were considered: *short writers*, *short readers*, *medium writers*, *medium readers* and *long sequential readers* (see Table 1).

Class	Readset Distribution	Writeset Distribution	Prw
Short Writers	unif(4,6)	unif(2,4)	0.50
Short Readers	unif(4,6)		
Medium Writers	unif(8,12)	unif(4,6)	0.50
Medium Readers	unif(8,12)		
Long Readers	seq(62,66)		

Table 1. Transactions Classes

The readset size of short transactions is uniformly distributed in the range [4,6] and the readset is assigned by randomly selecting data items without replacement from the entire database. The writeset size for short writers is uniformly distributed in the range [2,4] and data items for the writeset are first selected from the readset with probability 0.50 (for each data item in the readset), and then the rest of the items are uniformly selected from the entire database. The readset and the writeset distributions for medium transactions are similarly defined. To form a script, the readset and the writeset are interleaved randomly under the constraint that if a transaction reads and writes the same data item then the read request must precede the write request in the transaction script. The readset size of long sequential readers is uniformly selected in the range [62,66] and the readset is assigned a random collection of *adjacent* data items.

The transaction classes in Table 1 model transactions which result from precompiled programs. Therefore, the system parameters used in the experiments (see Table 4) display no startup I/O, no local I/O, low startup CPU and short local CPU processing time. It is assumed that the writing of post-images to disk takes place during the commit phase.

[11] Using the ICCA terminology in [BORAL84] this concurrency control method is defined as:

ICCA1 = ({ 2PL-ET-rw,CERT-ET-rw } ,
{ 2PL-ET-ww },
F:if T_i is a reader then
Cert-ET-rw X 2PL-ET-ww
else
2PL-ET-rw X 2PL-ET-ww)

4.3. Experiments and Results

In designing the experiments we were faced with the unpleasant fact that the number of possible experiments, involving various transaction workloads, is enormous. However, we have noticed that experiments may be characterized by their *effect* on the system work environment rather than by transaction workloads. This effect may be measured by the *blocking rate* of transactions and by the system *resource utilization*. The space of possibilities is defined by table 2.

Experimental Environment	**Resource Utilization**		
	low	high	Resource Contention
Blocking Rate			
low	exp 1.	exp 4.	
high	exp 2.	exp 1. exp 3.	exp 4.
Data Contention		exp 2.	exp 3.

Table 2 - Experimental Environment for 2PL.

Notice that the effect of each experiment span one or more table entries. So, to reliably cover the table, only a small number of experiments is needed; we have designed 4 experiments which cover the interesting entries of Table 2.

Experiment 1 represents a mix of short and long transactions (see Table 5). The moderate arrival rate implies that by increasing the MPL, 2PL moves from a low blocking rate and low CPU utilization environment into a high blocking rate and high CPU utilization environment. Experiment 2 represents the same mix of transactions as experiment 1 but with a higher arrival rate (see Table 6). Thus, by increasing the MPL, 2PL moves from a high blocking rate and low CPU utilization environment into an environment with data contention and high CPU utilization. Experiment 3 represents a mix of medium length transactions with high arrival rate (see Table 7). So, by increasing the MPL, 2PL moves from a high blocking rate and high CPU utilization environment into an environment with data contention and CPU contention. Experiment 4 represents a mix of short transactions with high arrival rate (see Table 8). So, by increasing the MPL, 2PL moves from a low blocking rate and high CPU utilization environment into a high blocking rate and CPU contention environment.

These results suggest that with no CPU contention the blocking phenomenon is dominant and restarts are cheap. ICCA1 makes better use of the CPU by letting readers use certification; the number of blocked transactions and the waiting time are reduced which leads to a significant improvement in transaction response time and throughput. These observations may be verified by the performance tables of experiments 1 and 2 (Table 5 and 6). Note that the 2PL performance degradation seems to be caused by the increased blocking rate rather than by the increased restart rate. This may be seen from the fact that long readers with a higher number of restarts outperform long readers with a lower number of restarts as long as the former do not enter the CPU contention area.

Under high CPU contention, restart is expensive and it becomes a dominant factor. As the MPL increases, ICCA1 can no longer take advantage of the decreased blocking rate since waiting due to blocking is replaced by increased waiting for CPU service. This explains the observations in experiment 4 where 2PL performs almost as well as ICCA1. The significant improvement in the performance of ICCA1 in experiment 3 starts as soon as 2PL approaches the trashing area and is due to the reduced number of restarted transactions.

5. Conclusions

The *feasibility* of private workspace based concurrency control mechanisms is exhibited by presenting an efficient parallel commit phase algorithm in which no I/O is associated with a critical section of the Transaction Manager. The use of post-images for recovery purposes and the placement of the commit record writing requests on a commit queue enhances performance. Immediately after queuing its commit request and issuing its updates a committing transaction may release all transactions blocked by it. The transaction is completed as soon as its commit record is known to reside in stable storage (read-only transactions must also queue a null commit request which induces no I/O activity).

In order to demonstrate the performance advantages of private workspace based concurrency control mechanisms, "ordinary" 2PL was compared to an ICCA method in which readers (queries) use certification (Serializa-

tion Graph Checking) and writers (updating transactions) use 2PL and are never blocked by readers. The results of the experiments conducted show that the ICCA method tested is usually superior to the ordinary 2PL. In an environment with high data contention and no system resource contention (i.e., low to high CPU utilization), the ICCA method performs significantly better. The results suggest that with no CPU contention blocking is dominant and restart cost is cheap. On the other hand, when there is high CPU contention restart is expensive and it becomes a dominant factor.

These results confirm some previously observed phenomena; in particular, the effects of high data contention predicted in [TAY84]. There, restarting a transaction upon conflict offers a method for overcoming the disadvantages of blocking in 2PL. It also confirms the effect of high I/O resource utilization where transaction restarts have a more negative effect on throughput than blocking (stated in [CARE83]). In the simulation model the choice of I/O with infinite parallel processing capability seems to capture (current) reality. Many of today's mainframe systems with multiple I/O channels tend to be CPU bound.

6. References

[AGRA83] Agrawal R., "Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation", *PhD Dissertation*, University of Wisconsin, (1983).

[BADAL79] Badal D., "Correctness of Concurrency Control and Implications in Distributed Database", *Proc. COMPSAC Conf.*, Chicago Ill., (1979).

[BERN81] Bernstein P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems", *Computing Surveys*, Vol. 13, No. 2, (1981).

[BERN83] Bernstein P.A., N. Goodman and V. Hadzilacos, "Recovery Algorithms for Database Systems", *Technical Report*, TR-10-83, Aiken Computation Laboratory, Harvard University, (1983).

[BORAL84] Boral H. and I. Gold., "Towards a Self-Adapting Centralized Concurrency Control Algorithm", *ACM SIGMOD*, Vol. 14, No. 2, (1984).

[CARE83] Carey M.J., "Modeling and Evaluation of Database Concurrency Control Algorithms", *PhD Dissertation*, University of California Berkeley, (1983).

[ESWA76] Eswaran K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Vol. 19, No. 11, (1976).

[GALL82] Galler B., "Concurrency Control Performance Issues", *PhD Dissertation*, University of Toronto, (1982).

[GOLD85] Gold I. and H. Boral, "The Power of The Private Workspace Model", Submitted for publication, (1985).

[GRAY78] Gray J., "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, (1978).

[KUNG81] Kung H.T. and J.T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM TODS*, Vol. 6, No. 2, (1981).

[LAMP76] Lampson B. and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", *Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center*, Palo Alto, Calif., (1976).

[LIN82] Lin W. and J. Nolte, "Distributed Database Control and Allocation: Semi-Annual Report", *Technical Report, CCA*, Cambridge, Massachusetts, (1982).

[LIND79] Lindsay B.G. et al., "Notes on Distributed Databases", *IBM Research Report*, No. RJ2571, (1979).

[PAPA79] Papadimitriou C.H., "The Serializability of Concurrent Database Updates", *Journal of the ACM*, Vol. 26, No. 4, (1979).

[PEIN83] Peinl P. and A. Reuter., "Empirical Comparison of Database Concurrency Control Schemes", *Proc. International Conference on Very Large Databases*, Florence, Italy, pp. 97-108 (1983).

[ROBI82] Robinson J.T., "Experiments with Transaction Processing on a Multi-Microprocessor", *IBM Research Report*, No. RC9725, T.J. Watson Research Center, (1982).

[TAY84] Tay Y.C., "A Mean Value Performance Model for Locking in Databases", *PhD Dissertation*, Harvard University, (1984).

[WILK81] Wilkinson W.K., "Database Concurrency Control and Recovery in Local Broadcast Networks", *PhD Dissertation*, University of Wisconsin, (1981).

System Parameters		
Process	**Parameter**	**Description**
TP	THINKtime	mean exponential TP think time
	STARTUPcpu	CPU time for transaction startup
	STARTUPio	I/O time for transaction startup
	LOCALcpu	CPU time for transaction local processing
	LOCALio	i/O time for transaction local processing
CC	CCinit	CC CPU time for transaction initialization
	CCconflict	CC CPU time for request conflict analysis
	CCcommit	CC CPU time for committing a transaction
	CCcleanup	CC CPU time for transaction cleanup
	REASTARTdelay	transaction delay time before restart
DM	DMcpu	DM CPU time for reading/writing a data item
	DMio	DM I/O time for reading/writing a data item
	PRE_WRITEcpu	DM CPU time for writing a post-image at Execution Time
	PRE_WRITEio	DM I/O time for writing a post-image at Execution Time
	DM_POSTcpu	DM CPU time for writing a post-image at Commit Time
	DM_POSTio	DM I/O time for writing a post-image at Commit Time disk access time + post-image transfer time
	DM_COMMITcpu	DM CPU time for writing a commit record
	DM_COMMITio	DM I/O time for writing a commit record

Table 3. System Parameters

Experiments Parameters Setup		
Process	**Parameter**	**Time (msec.)**
TP	STARTUPcpu	5
	STARTUPio	0
	LOCALcpu	1
	LOCALio	0
CC	CCinit	1
	CCconflict	1
	CCcommit	1
	CCcleanup	1
DM	DMcpu	1
	DMio	30
	PRE_WRITEcpu	0
	PRE_WRITEio	0
	DM_POSTcpu	1
	DM_POSTio	28+2
	DM_COMMITcpu	1
	DM_COMMITio	30

Table 4. Experiments Parameters Setup

Class	% of MPL
Short Writers	50.0
Short Readers	37.5
Long Readers	12.5

Experiment 1 Classes Mix.

Class	% of MPL
Short Writers	50.0
Short Readers	37.5
Long Readers	12.5

Experiment 2 Classes Mix.

Class	% of MPL
Medium Writers	50.0
Medium Readers	50.0

Experiment 3 Classes Mix.

Class	% of MPL
Short Writers	50.0
Short Readers	50.0

Experiment 4 Classes Mix.

Experiment 1. - 2PL Performance A Mix of Short and Long Transactions

THINKtime=5000; RESTARTdelay=250

MPL	Short Writers			Short Readers			Long Readers			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	1.53	313.9	13	1.17	176.6	0	0.28	2106.8	20	1.25	0.14
32	2.96	410.3	27	2.30	187.4	0	0.54	2231.0	73	1.90	0.27
48	4.36	477.3	49	3.43	195.0	0	0.81	2240.5	50	2.53	0.41
64	5.68	635.1	98	4.57	210.5	0	1.06	2446.9	94	4.14	0.54
80	6.86	790.3	141	5.66	240.2	0	1.35	2826.1	120	5.46	0.68
96	7.71	1293.0	292	6.74	333.0	0	1.43	3527.1	210	10.96	0.78
112	7.21	2805.1	545	7.06	512.0	3	1.33	5528.2	414	23.13	0.80
128	7.02	4315.6	1016	8.05	966.1	3	1.17	8493.2	598	40.53	0.85

Experiment 1. - ICCA1 Performance A Mix of Short and Long Transactions

THINKtime=5000; RESTARTdelay=250

MPL	Short Writers			Short Readers			Long Readers			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	1.53	251.9	5	1.17	175.6	0	0.27	2125.3	33	1.00	0.14
32	2.99	258.0	12	2.39	179.8	1	0.54	2240.6	79	1.05	0.28
48	4.51	264.4	12	3.43	184.5	0	0.80	2325.3	114	1.02	0.42
64	6.02	274.6	24	4.70	191.7	0	1.10	2409.7	136	1.03	0.56
80	7.46	295.2	23	5.84	206.4	2	1.36	2607.4	195	1.05	0.71
96	8.98	353.3	38	6.85	250.7	1	1.51	3031.7	244	1.08	0.84
112	10.08	482.0	31	7.93	351.7	0	1.57	3976.7	340	1.22	0.95
128	10.86	878.7	75	8.47	641.7	3	1.26	7653.7	527	1.49	0.99

Table 5 - Experiment 1 Performance Tables.

Experiment 2. - 2PL Performance A Mix of Short and Long Transactions

THINKtime=2500; RESTARTdelay=250

MPL	Short Writers			Short Readers			Long Readers			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	2.83	360.2	27	2.22	185.6	2	0.43	2191.5	41	1.74	0.25
32	5.37	502.4	59	4.46	199.3	1	0.81	2309.1	53	2.83	0.47
48	7.62	702.5	158	6.58	234.1	0	1.13	2647.1	99	5.20	0.68
64	8.45	1286.6	289	8.33	344.6	2	1.30	3660.1	207	11.14	0.83
80	8.46	2169.4	446	9.87	598.3	8	1.25	5503.7	258	18.70	0.88
96	7.35	4065.0	863	10.48	918.2	3	1.09	8744.3	443	33.76	0.88
112	5.09	9092.3	1668	10.30	1554.3	9	0.76	16400.4	845	52.40	0.81
128	3.53	14151.7	2378	9.84	2337.3	15	0.63	21350.9	1124	73.34	0.76

Experiment 2. - ICCA1 Performance A Mix of Short and Long Transactions

THINKtime=2500; RESTARTdelay=250

MPL	Short Writers			Short Readers			Long Readers			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	2.87	256.5	11	2.23	178.1	0	0.42	2201.1	53	1.02	0.25
32	5.75	269.4	15	4.47	189.2	1	0.80	2441.2	135	1.04	0.50
48	8.53	297.2	23	6.59	210.3	0	1.19	2624.4	154	1.08	0.73
64	10.99	411.2	31	8.78	295.8	2	1.31	3514.6	223	1.15	0.93
80	12.14	786.6	81	9.97	570.6	1	1.04	6960.9	390	1.44	1.00
96	12.39	1408.3	131	10.14	1018.9	6	0.42	19542.8	773	2.02	1.00
112	12.34	1996.9	218	10.87	1391.5	5	0.20	51360.8	895	3.22	1.00
128	12.32	2678.5	333	11.20	1749.5	6	0.15	84968.9	923	5.57	1.00

Table 6 - Experiment 2 Performance Tables.

Experiment 3. - 2PL Performance Medium Transactions

THINKtime=1000; RESTARTdelay=500

MPL	Medium Writers			Medium Readers			Overall			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	5.20	636.1	182	5.78	380.3	11	10.98	454.1	193	1.52	0.54
32	8.26	928.1	576	10.30	549.3	42	18.56	717.9	618	3.55	0.94
48	7.01	2414.4	1318	12.07	1005.4	67	19.08	1522.9	1385	11.45	0.99
64	4.96	5398.5	2416	13.34	1387.2	119	18.30	2473.5	2535	25.21	0.98
80	3.04	11845.8	3688	14.59	1732.5	216	17.63	3477.1	3904	39.89	0.98
96	1.44	27412.4	4955	16.05	1995.7	229	17.49	4082.4	5184	50.97	0.99
112	0.84	55943.8	4993	16.84	2377.4	252	17.48	4937.9	5245	66.34	0.99
128	0.48	86837.6	4126	18.35	2988.0	279	18.83	5143.0	4405	86.74	1.00

Experiment 3. - ICCA1 Performance Medium Transactions

THINKtime=1000; RESTARTdelay=500

MPL	Medium Writers			Medium Readers			Overall			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	5.39	496.2	122	5.79	363.9	13	11.18	427.7	135	1.19	0.55
32	8.81	817.9	378	10.45	518.0	47	19.26	655.2	425	2.09	0.96
48	7.81	2081.5	814	12.22	964.6	52	20.03	1400.2	866	5.68	1.00
64	5.64	4664.9	1494	14.34	1226.1	46	19.98	2198.6	1540	13.42	1.00
80	3.56	10091.9	2024	16.81	1376.9	34	20.37	2901.1	2058	23.16	1.00
96	2.27	19880.4	1966	18.92	1540.7	22	21.19	2503.1	1988	33.44	1.00
112	1.21	40058.8	2588	19.86	1822.2	22	21.07	4020.8	2590	41.24	1.00
128	0.94	57479.4	2227	20.74	2096.2	16	21.68	4507.3	2243	49.87	1.00

Table 7 - Experiment 3 Performance Tables.

Experiment 4. - 2PL Performance Short Transactions

THINKtime=1000; RESTARTdelay=250

MPL	Short Writers			Short Readers			Overall			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	6.33	264.8	16	6.78	185.9	0	13.11	224.0	16	1.06	0.39
32	12.22	309.5	29	13.23	220.3	0	25.45	263.1	29	1.21	0.76
48	15.62	533.2	61	17.23	387.6	1	32.85	456.8	62	1.67	0.98
64	15.32	1090.6	150	18.04	772.3	3	33.36	918.5	153	3.23	1.00
80	14.70	1695.3	249	18.51	1144.7	5	33.21	1388.4	254	6.50	1.00
96	14.03	2419.9	413	18.89	1528.9	2	32.92	1908.6	415	11.70	1.00
112	13.32	3219.9	530	19.60	1858.1	5	32.92	2409.1	535	19.59	1.00
128	11.79	4414.1	845	20.91	2066.9	19	32.70	2913.1	892	35.19	1.00

Experiment 4. - ICCA1 Performance Short Transactions

THINKtime=1000; RESTARTdelay=250

MPL	Short Writers			Short Readers			Overall			BlockedQ	CPU
	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	Commtd tran/sec.	Res. Time msec.	Rest-arted	mean size	utili-zation
16	6.42	264.0	21	6.82	185.2	0	13.24	223.4	21	1.04	0.39
32	12.28	307.6	38	13.25	217.4	2	25.53	260.8	40	1.13	0.76
48	15.74	534.8	68	17.29	383.7	1	33.03	455.7	69	1.36	0.98
64	15.49	1082.0	122	18.11	767.4	2	33.60	912.5	124	2.16	1.00
80	14.87	1692.5	234	18.60	1156.4	5	33.47	1394.6	239	3.31	1.00
96	14.41	2349.8	309	19.04	1508.7	5	33.45	1871.0	314	5.67	1.00
112	13.61	3109.8	413	19.97	1807.4	3	33.58	2335.4	416	9.86	1.00
128	12.22	4243.6	619	21.42	1990.1	7	33.64	2808.6	626	14.68	1.00

Table 8 - Experiment 4 Performance Tables.