THE PRIVATE WORKSPACE MODEL FEASIBILITY AND APPLICATIONS TO 2PL PERFORMANCE IMPROVEMENTS

by

I. Gold, O. Shmueli and M. Hofri

Technical Report #358

March 1985
The Private Workspace Model Feasibility and Applications to 2PL Performance Improvements

Israel Gold, Oded Shimueli 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 management component of a database management system 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 of new values into the private workspace of the transactions do not affect the database state. First, the Concurrency Control has complete freedom in choosing how to synchronize conflicting transactions. Second, a transaction restart may not cause cascading restarts.

One may suspect that it would be difficult to implement the private workspace idea in a real world database system. In this paper we show the feasibility of the model by introducing a parallel-commit-phase algorithm. By simulation experiments we show that system performance may be significantly improved by using a concurrency control method in which readers use certification, writers use 2PL, and writers do not wait for readers.

1. Introduction

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.
It is possible to construct correct concurrency control algorithms that employ concurrently both blocking and transaction restart for synchronizing conflicting transactions [BORAL84]. This leads to the notion of an Integrated Concurrency Control Algorithm (ICCA) which is composed of several rw and several ww synchronization techniques¹ used concurrently. ICCA algorithms based on 2PL and Certification synchronization techniques were presented in [BORAL84]. The major likely advantage of the ICCA concept is in matching transactions (or in general transaction classes) with appropriate synchronization techniques as dictated by performance objectives.

ICCA advantages depend on transactions using a private workspace and on using Serialization Graph based synchronization techniques which only detect actual conflicts among transactions (as opposed to possible conflicts as done by the optimistic method in [KUNG81]). One may suspect that committing a transaction using a private workspace and at the same time detecting only actual conflicts (i.e. those affecting serialization) may be difficult. To complicate matters, some ICCAs use Commit Time Synchronization (CTS) before allowing a transaction to commit; the integrity of this test must be preserved as well. In this paper, a parallel commit algorithm using post-images for recovery purposes is introduced.

The whole commit phase (including writing post-images, writing a commit record, issuing database updates and updating various tables) could be performed in a critical section of the concurrency control, an obviously correct solution. Performing I/O operations in a critical section may slow the system considerably. The proposed 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 and immediately unblocking all transactions blocked by committing transactions. A transaction is completed as soon as its commit record is known to reside in stable

¹ see [HERN81]
storage (observe that once the transaction's update operations are issued, another transaction trying to read an updated item will obtain the correct result, either from the disk or from the buffer).

Once the feasibility of private workspace based concurrency control mechanisms is shown, it remains to demonstrate performance advantages in using these mechanisms. To this end, a substantial comparative simulation study was conducted. 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 ICCA method tested consists of readers (queries) using certification (Serialization Graph Checking) and writers (updating transactions) using 2PL. In this ICCA a writer is never blocked by a reader. The results of the experiments conducted show that the ICCA method tested is almost always at least as good as ordinary 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.

Recently, researchers have begun to compare various concurrency control algorithms 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) [WILK81, GALL82, LIN82, ROBI82, ROBI82a, AGRA83, CARE83, TAY84]. Our results confirm some previously observed phenomena. In particular, the 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* a private workspace is allocated to an active transaction in which it caches its previously read data items and those data values written by the transaction during its execution\(^6\).

Following the separation proposed in [BERN81], a centralized database management system may be decomposed into two components: a transaction manager (TM) and a data manager (DM). The TM controls interactions between users and the database management system and is charged with functions such as concurrency control. All reference to data items in the private workspace are made through the TM\(^6\). The DM is responsible for managing the database (i.e., accessing and modifying data). Two data manipulation operations are recognized by the DM: \(\text{DM\_READ}(X)\) — in which data item \(X\) is read; and \(\text{DM\_WRITE}(X, \text{NEW\_VALUE})\) — in which \(\text{NEW\_VALUE}\) is assigned to data item \(X\) in the database.

A transaction execution is divided into two phases. In the *Execution Phase* the transaction reads values from the database, performs various computations and writes results into its private workspace. In the *Commit Phase*, which takes place after the transaction finishes all computations, the transaction first goes through (a possibly empty) *Commit Time Synchronization* to ensure that commitment does not cause inconsistencies, and then it issues (a possibly empty) sequence \(\text{DM\_WRITE}\) operations (which instructs the DM to update the database). The Commit Phase should be

---

\(^6\) This model was previously used (e.g. [BERN81], [KUNG81]); the fact that write operations by distinct transactions are performed into distinct workspaces was used in developing ICCAs [BORAL84]. The term "PRE\_WRITE" denotes a write operation into a private workspace; in [BORAL84] this operation was also used as a synchronization primitive.

\(^3\) The notion of the (logical) private workspace in this model differs from that used by Network based database management system, 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.
In the next section we present a physical implementation of the commit procedure.

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 equivalent serial execution order, transaction \(T_i\) precedes transaction \(T_j\). SG is used to represent all such precedence relationships whether they originated in the deadlock detection phase of 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 \(<\text{INDICATOR}, \text{TRANSACTION IDENTIFIER}>\); 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 the transaction \(TID\).
2. A \(p\)-indicator indicates that a PRE\_WRITE operation was executed on this item on behalf of an active transaction \(TID\). At commit time, 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 the committed transaction \(TID\).

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

---

4 The actual implementation of the commit procedure need not be atomic as long as it appears atomic to the outside world. Kung and Robinson [KUNG81] have previously discussed several ways of implementing non-atomic commit.

5 Two transactions conflict if they access the same data item and one issues a WRITE request.
request. In the waiting phase, a transaction using 2PL is forced to wait until 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 the synchronization phase, the request is synchronized with conflicting operations from other transactions. The result of this synchronization may be aborting the issuing transaction or continuing execution. Edges are appropriately added to SG in order to reflect the precedence relation imposed by executing the request.

Request execution includes appending the appropriate indicator to IT and issuing the appropriate DM_READ or PRE_WRITE operation.

3. Parallelism of the Commit Phase

The atomic commit phase of a transaction has thus far been treated from a logical point of view; any implementation aspects were ignored. Execution of the commit phase in a critical section (of the TM) is unreasonable; this blocks the entire database for the duration of the commit thereby causing system performance degradation. On the other hand, if the commit phase is executed in parallel, several problems may result. The DM_WRITE operations performed by transactions committing in parallel may be interleaved so that the Commit Time Synchronization would be incorrect (see discussion in section 3.2). In this section a parallel commit phase algorithm which solves these problem and guarantees database consistency is proposed.

3.1. Supporting Atomic Commit

Supporting atomic commit means reaching a well defined commit point during transaction execution so that if a system failure occurs before that point then the transaction's effects will eventually be undone and if a system failure occurs after this point then the transaction's 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 of a transaction's DM_WRITE operations are processed or none are.

The "standard" implementation of atomic commitment is a procedure called *two-phase commit* [LAMP76, GRAY78]. Let \( T_i \) be a committing transaction. After \( T_i \) is validated by the Commit Time Synchronization, the *first phase* of two-phase commit begins. In this phase the TM does not issue \( T_i \)'s DM_WRITE operations directly; rather, it instructs the DM to force out to \( T_i \)'s virtual workspace on stable storage (disk) the *post-images* of those data items values written into \( T_i \)'s private workspace, 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.

It is incorrect, in general, to write the *pre-images* of data items into \( T_i \)'s virtual workspace on stable storage and use *undo* operations in case of system failure. The reason is that a transaction \( T_j \), using certification, might read a data item updated by \( T_i \) which is subsequently restored to its pre-image due to a system failure. If \( T_j \) has completed successfully before the system failed, it has seen a value which never "existed". This may be the case when a short reader using certification follows a committed transaction with a large write set.
3.2. A Parallel Commit Phase Algorithm

The Commit Phase of a transaction takes place after it has finished its Execution Phase. During this phase the TM must complete three tasks. First, the (possibly empty) Commit Time Synchronization (CT-Synchronization) which ensures that committing the transaction will not cause any database inconsistency. Second, the (possibly empty) two-phase commit procedure which ends by issuing the transaction's DM_WRITE operations and conversion of its p-indicators into c-indicators. Third, a (possible 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 the transaction being restarted.

The basic commit phase procedure is given in Figure 1.

```plaintext
procedure BasicCommitPhase(T1);
begin
  certify <- CT-Synchronization(T1);
  if certify then
    begin
      (* first phase of commit *)
      \forall X \in \text{Writset}(T1) force out the post-image of X to T1's virtual workspace on disk;
      wait for DM "Ready to Commit" message;
      TCN <- TCN+1; (* get new commit number *)
      force out a commit record to T1's virtual workspace on disk;
      wait for DM "committed" message;
    end
    (* second phase of commit *)
    \forall X \in \text{Writset}(T1) execute DM_WRITE(X);
    convert T1's p-indicators into c-indicators;
  end
  (* cleanup phase *)
  if certify then remove T1 from SG and IT
  else RESTART(T1);
end

Figure 1. Basic Commit Phase Procedure.
```

---

8 Information about a committed transaction may be removed from SG and IT only when the node representing it in SG becomes a root node with no incoming edges.
If the BasicCommitPhase procedure is executed in a critical section of the TM then, clearly, the database is kept consistent. However, in general, executing it in parallel is incorrect. Let \( T_i \) be a committing transaction requiring Commit Time Synchronization. The subtle point is that \( T_i \)'s Commit Time Synchronization is only partially correct. At the time \( T_i \) is validated by Commit Time Synchronization 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 \( \text{wr} \) and/or \( \text{ww} \) conflicts with \( T_j \) may not be resolved since \( T_j \) is not yet committed and does not yet own its c-indicators. If \( T_j \) subsequently commits and issues its DM_WRITE operations before \( T_i \) does, a database inconsistency may result.

Even when the Commit Time Synchronization is guaranteed a priori to be successful (e.g., when using the W2PL algorithm [GOLDB85]), the writesets of transactions committing in parallel may have nonempty intersections and as a result their DM_WRITE operations may be interleaved so as to cause database inconsistency. One way to overcome these problems is by encasing the second phase of \( T_i \)'s commit phase preceded by an additional Commit Time Synchronization in a critical section of the TM. This is implemented by the ParallelCommitPhase procedure shown in Figure 2. The critical section is indicated by using " < " and " > ".

```plaintext
procedure ParallelCommitPhase(T_i)
begin

(*) first phase of commit *)

certify <- CT-Synchronization(T_i); (* preliminary Commit Time Synchronization *)
if certify then
begin
    \( \forall X \in \text{Writset}(T_i) \) force out the post-image of \( X \)
to \( T_i \)'s virtual workspace;
    wait for DM "Ready to Commit" message;

(*) second phase of commit *)

< certify <- CT-Synchronization(T_i); (* second phase Commit Time Synchronization *)
if certify then
begin
    TCN <- TCN+1; (* get new commit number *)

end
```

Technion - Computer Science Department - Technical Report CS0358 - 1985
force out a commit record to $T_1$'s virtual workspace on disk;  
wait for DM "committed" message;  
\[ \forall X \in \text{Writset}(T_1) \text{ execute DM.WRITE}(X); \]
convert $T_1$'s p-indicators into c-indicators;  
end \>
end

(* cleanup phase *)

if certify then remove $T_1$ from SG and IT
else RESTART($T_1$);
end

Figure 2. Parallel Commit Phase Procedure

3.2.1. Some Practical Considerations

Given the ParallelCommitPhase procedure, consider a transaction $T_1$ having a large writeset. After $T_1$ is validated by the preliminary Commit Time Synchronization, the TM begins forcing $T_1$'s post-images to disk. Throughout this phase, $T_1$ is in "doubt" since it may be restarted later by the Commit Time Synchronization of the second phase. In order to decrease the probability that a transaction which requires Commit Time Synchronization 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 Commit Time Synchronization test is empty or is a priori successful.

A transaction $T_1$ is completed at the end of its commit phase, even though its DM.WRITE operations may not yet have been processed by the DM. Responding to $T_1$ with a completion message at the end of its commit phase without waiting for its DM.WRITE operations to be processed may speed up transactions execution and increase system throughput. The TM, however, need keep track of $T_1$'s DM.WRITE operations carried by the DM, since $T_1$'s private workspace may be cleared only after all of DM.WRITE operations are complete.
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 is moved outside the critical section thereby allowing more parallelism.

The suggested optimization may result in some certified transactions updating the database before their commit record has reached the disk, i.e., before they were actually committed. The possibility of "dirtying" the database need not worry us; in case of a system failure, the database is restored into a consistent state by reinstalling the post-images of the committed transactions (in their commit order).

However, transactions reading "uncommitted" data items may violate database consistency. For example, let $T_i$ be a non-committed transaction whose DM_WRITE operations have already been issued by the TM. If a transaction $T_j$ has managed to read a data item updated by $T_i$ (the value may be served from the DM buffer) and completes successfully just before the system fails and before $T_i$'s commit record has been installed in the disk, then $T_i$'s effects on the database will eventually be undone and thus $T_j$ has seen a value which never existed. However, if we require that $T_j$ be committed after $T_i$, then this problem would disappear as both transactions would be undone.

The above leads to a general solution for treating transactions that read "uncommitted" values. The solution is to require that transactions force their commit records in TCN order (but outside the critical section). This may be implemented by queuing commit record writing requests on a dedicated COMMIT QUEUE. This queue is serially processed by the DM on a FCFS (First Come First Served) basis; this is done in parallel with the processing of transactions' DM operations. Queuing the commit requests may not cause transaction delay since under heavy transaction load these requests may be "batched" together to amortize the cost of the commit operation over several transac-
The improved Parallel Commit Phase procedure below incorporates this solution.

```plaintext
procedure Improved Parallel Commit Phase (T_i):
begin
  (* first phase of commit *)
  certify <- CT-Synchronization (T_i);
  if certify then
    begin
      V X \in Writeset (T_i) force out the post-image of X
      to T_i's virtual workspace on disk;
      wait for DM "Ready to Commit" message;
    end
  (* second phase of commit *)
  if certify <- CT-Synchronization (T_i);
  if certify then
    begin
      (* NO I/O is associated with this phase *)
      TCN <- TCN + 1; (* get new commit number *)
      Enqueue T_i's commit request on the COMMIT QUEUE;
      V X \in Writeset (T_i) execute DM_WRITE (X);
      convert T_i's p-indicators into c-indicators;
    end
  (* cleanup phase *)
  if certify then
    begin
      remove T_i from SG and IT;
      wait for DM "committed" message;
      send completion message to T_i;
    end
  else
    RESTART (T_i);
end
```

Figure 3. Improved Parallel Commit Phase Procedure

So far, the issue of read-only transactions (queries), has been ignored. Since queries do not update the database, their first commit phase is empty and they can proceed directly to their second commit phase. Although those "readers" do not update the database, they are still required to queue a null commit request (which need not involve any I/O) onto the COMMIT QUEUE, and complete only after their-null commit request is dequeued.

\* The idea of "batching" commit records is pointed out in [WILK81].
4. Performance Experiments

This section describes the simulation experiments used to compare 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. 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. Note that although a concurrency control algorithm may have a very low execution time overhead, it may still place great demands on primary memory. Thus, the relative storage and CPU overheads required by various concurrency control mechanisms is a subject that implementors should be aware of. (See, for example, comparison studies in [CARE83].) However, with dropping memory cost, it seems that this should not be a source of concern. Thus, 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.
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 at 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 the 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 that 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. 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.

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) Commit Time Synchronization which ensures that committing T_i will cause no database inconsistency. When 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 out to disk T_i's commit record), 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 prior to issuing its SNART request, T_i has not written its post-images to disk, then following the preliminary Commit Time Synchronization 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 that 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*\(^\text{10}\) algorithm. 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.

\(^\text{10}\) 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.
DMcpu and DMio are the CPU and I/O costs associated with the DM reading or writing a data item. For the sake of clarity we give different names to the parameters DMcpu, PRE_WRITEcpu and DM_POSTcpu, although they actually have identical values. This also holds for the parameters DMio, PRE_WRITEio and DM_POSTio. 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, the 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 model the real world situation in which the system is CPU bound and I/O utilization is low. Otherwise, transaction response time would measure mostly I/O queuing delay 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 TPs requests are given the lowest priority. This policy (approximately) models a priority based system in which the 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, in which readers use

\[ \text{ICCA1} = \left( \{ 2PL-ET-rw, CERT-ET-rw \}, \{ 2PL-ET-ww \} \right) \]

\[ \begin{align*}
\text{If } T_1 \text{ is a reader then} & \\
\text{Cert-ET-rw } \times & \text{2PL-ET-ww} \\
\text{else} & \\
\text{2PL-ET-ww } \times & \text{2PL-ET-ww}
\end{align*} \]

\[ \text{Using the } \text{ICCA terminology in [BORAL84] this concurrency control method is defined as:} \]

\[ \text{ICCA1} = \left( \{ 2PL-ET-rw, CERT-ET-rw \}, \{ 2PL-ET-ww \} \right) \]

\[ \text{Cert-ET-rw } \times \text{2PL-ET-ww} \]

\[ \text{If } T_1 \text{ is a reader then} \\
\text{2PL-ET-ww } \times \text{2PL-ET-ww} \]

\[ \text{else} \]
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).

<table>
<thead>
<tr>
<th>Class</th>
<th>Readset Distribution</th>
<th>Writeset Distribution</th>
<th>Prw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Writers</td>
<td>uniform(4,6)</td>
<td>uniform(2,4)</td>
<td>0.50</td>
</tr>
<tr>
<td>Short Readers</td>
<td>uniform(4,6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium Writers</td>
<td>uniform(6,12)</td>
<td>uniform(4,6)</td>
<td>0.50</td>
</tr>
<tr>
<td>Medium Readers</td>
<td>uniform(5,12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long Readers</td>
<td>seq(62,66)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. (This may be considered as a model for the cost of the deferred updates recovery mechanism [GRAY78]).

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.

<table>
<thead>
<tr>
<th>Experimental Environment</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>low</td>
</tr>
<tr>
<td>Blocking Rate</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>exp 1.</td>
</tr>
<tr>
<td>high</td>
<td>exp 2.</td>
</tr>
<tr>
<td>Data Contention</td>
<td></td>
</tr>
</tbody>
</table>

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 experiment 1 performance tables. (Table 5). 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 transactions with higher number of restarts perform better than transactions with lower number of restarts and vice versa. The above may also be verified by experiment 2 in the situation where ICCA1 does not enter the CPU contention area.
Under high CPU contention, restart is very expensive and it becomes a dominant factor. As the MPI 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 readers.

5. Conclusions

The feasibility of private workspace-based concurrency control mechanisms is exhibited by presenting an efficient general parallel commit phase algorithm. 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).

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 (Serialization 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 almost always at least as good as 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 very 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 of overcoming the disadvantages of blocking in 2PL. It also confirms the effect of high I/O resource utilization where transaction restarts have more negative effect on throughput than blocking (stated in [CARE83]).

The choice of I/O with infinite parallel processing capability in the simulation model, seems to capture (current) reality. Many of today's mainframe systems with multiple I/O channels tend to be CPU bound.

6. References


---

Figure 6 - DBMS Physical Model.
### System Parameters

<table>
<thead>
<tr>
<th>Process</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>THINKtime</td>
<td>mean exponential TP think time</td>
</tr>
<tr>
<td></td>
<td>STARTUPcpu</td>
<td>CPU time for transaction startup</td>
</tr>
<tr>
<td></td>
<td>STARTUPio</td>
<td>I/O time for transaction startup</td>
</tr>
<tr>
<td></td>
<td>LOCALcpu</td>
<td>CPU time for transaction local processing</td>
</tr>
<tr>
<td></td>
<td>LOCALio</td>
<td>I/O time for transaction local processing</td>
</tr>
<tr>
<td>CC</td>
<td>CCinit</td>
<td>CC CPU time for transaction initialization</td>
</tr>
<tr>
<td></td>
<td>CConlict</td>
<td>CC CPU time for request conflict analysis</td>
</tr>
<tr>
<td></td>
<td>CCommit</td>
<td>CC CPU time for committing a transaction.</td>
</tr>
<tr>
<td></td>
<td>CCcleanup</td>
<td>CC CPU time for transaction cleanup</td>
</tr>
<tr>
<td></td>
<td>RESTARTdelay</td>
<td>transaction delay time before restart</td>
</tr>
<tr>
<td>DM</td>
<td>DMcpu</td>
<td>DM CPU time for reading/writing a data item</td>
</tr>
<tr>
<td></td>
<td>DMio</td>
<td>DM I/O time for reading/writing a data item</td>
</tr>
<tr>
<td></td>
<td>PRE_WRITEcpu</td>
<td>DM CPU time for writing a post-image at Execution Time</td>
</tr>
<tr>
<td></td>
<td>PRE_WRITEio</td>
<td>DM I/O time for writing a post-image at Execution Time</td>
</tr>
<tr>
<td></td>
<td>DM_POSTcpu</td>
<td>DM CPU time for writing a post-image at Commit Time</td>
</tr>
<tr>
<td></td>
<td>DM_POSTio</td>
<td>DM I/O time for writing a post-image at Commit Time</td>
</tr>
<tr>
<td></td>
<td>DM_COMMITcpu</td>
<td>DM CPU time for writing a commit record</td>
</tr>
<tr>
<td></td>
<td>DM_COMMITio</td>
<td>DM I/O time for writing a commit record</td>
</tr>
</tbody>
</table>

Table 3. System Parameters

### Experiments Parameters Setup

<table>
<thead>
<tr>
<th>Process</th>
<th>Parameter</th>
<th>Time (msec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>STARTUPcpu</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>STARTUPio</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>LOCALcpu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LOCALio</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>CCinit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CConlict</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CCommit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CCcleanup</td>
<td>1</td>
</tr>
<tr>
<td>DM</td>
<td>DMcpu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DMio</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>PRE_WRITEcpu</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>PRE_WRITEio</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DM_POSTcpu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DM_POSTio</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>DM_COMMITcpu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DM_COMMITio</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4. Experiments Parameters Setup
### Experiment 1 - MPL Performance: A Mix of Short and Long Transactions

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Long Readers</th>
<th>BlockedQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Commtd tran/sec</td>
<td>Res. Time</td>
<td>Rest-</td>
<td>Commtd tran/sec</td>
<td>Res. Time</td>
</tr>
<tr>
<td>10</td>
<td>1.50</td>
<td>381.0</td>
<td>17</td>
<td>1.16</td>
<td>177.5</td>
</tr>
<tr>
<td>20</td>
<td>2.92</td>
<td>457.5</td>
<td>32</td>
<td>2.32</td>
<td>184.2</td>
</tr>
<tr>
<td>40</td>
<td>4.23</td>
<td>564.1</td>
<td>54</td>
<td>3.45</td>
<td>192.1</td>
</tr>
<tr>
<td>64</td>
<td>5.61</td>
<td>687.6</td>
<td>107</td>
<td>4.65</td>
<td>213.5</td>
</tr>
<tr>
<td>80</td>
<td>6.65</td>
<td>918.7</td>
<td>177</td>
<td>5.62</td>
<td>223.1</td>
</tr>
<tr>
<td>96</td>
<td>7.38</td>
<td>1351.2</td>
<td>344</td>
<td>6.69</td>
<td>287.9</td>
</tr>
<tr>
<td>112</td>
<td>7.85</td>
<td>2000.9</td>
<td>544</td>
<td>7.75</td>
<td>493.6</td>
</tr>
<tr>
<td>128</td>
<td>8.85</td>
<td>4188.9</td>
<td>881</td>
<td>7.92</td>
<td>981.0</td>
</tr>
</tbody>
</table>

Table 5 - Experiment 1 Performance Tables.
Table 6 - Experiment 2 Performance Tables.
### Experiment 3 - mixer: MPL performance Medium Transactions

**THINKtime=1000; RESTART/delay=500**

<table>
<thead>
<tr>
<th>MPL</th>
<th>Medium Writers</th>
<th>Medium Readers</th>
<th>Overall</th>
<th>Misses</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4.80</td>
<td>864.4</td>
<td>22</td>
<td>5.76</td>
<td>268.7</td>
</tr>
<tr>
<td>32</td>
<td>7.50</td>
<td>1152.2</td>
<td>33</td>
<td>10.11</td>
<td>576.0</td>
</tr>
<tr>
<td>48</td>
<td>6.47</td>
<td>2702.3</td>
<td>1251</td>
<td>11.94</td>
<td>1039.9</td>
</tr>
<tr>
<td>64</td>
<td>4.39</td>
<td>6322.6</td>
<td>2478</td>
<td>13.33</td>
<td>1850.9</td>
</tr>
<tr>
<td>80</td>
<td>5.80</td>
<td>12704.0</td>
<td>2655</td>
<td>14.67</td>
<td>1712.2</td>
</tr>
<tr>
<td>96</td>
<td>1.39</td>
<td>27102.7</td>
<td>4688</td>
<td>15.50</td>
<td>2019.4</td>
</tr>
<tr>
<td>112</td>
<td>0.66</td>
<td>80606.7</td>
<td>5186</td>
<td>16.59</td>
<td>2324.8</td>
</tr>
<tr>
<td>128</td>
<td>0.22</td>
<td>192595.4</td>
<td>4767</td>
<td>17.47</td>
<td>2662.7</td>
</tr>
</tbody>
</table>

### Table 7 - Experiment 3 Performance Tables
### Class and % of MPL

<table>
<thead>
<tr>
<th>Class</th>
<th>% of MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Writers</td>
<td>50.0</td>
</tr>
<tr>
<td>Short Readers</td>
<td>50.0</td>
</tr>
</tbody>
</table>

Experiment 4 Classes Mix.

#### Experiment 4 - ZPL Performance Short Transactions

| MPL | Short Writers | Short Readers | Overall | Blocksize | CPU
|-----|---------------|---------------|---------|-----------|-----
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.65</td>
<td>324.5</td>
<td>18</td>
<td>6.74</td>
<td>186.8</td>
<td>0</td>
<td>12.69</td>
<td>251.4</td>
<td>18</td>
<td>1.11</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>11.79</td>
<td>370.8</td>
<td>39</td>
<td>10.15</td>
<td>221.1</td>
<td>0</td>
<td>24.94</td>
<td>291.9</td>
<td>30</td>
<td>1.27</td>
<td>0.74</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>15.31</td>
<td>378.4</td>
<td>63</td>
<td>17.45</td>
<td>371.9</td>
<td>2</td>
<td>32.70</td>
<td>448.4</td>
<td>95</td>
<td>1.95</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>15.30</td>
<td>111.6</td>
<td>148</td>
<td>18.19</td>
<td>750.5</td>
<td>5</td>
<td>33.39</td>
<td>916.7</td>
<td>101</td>
<td>3.46</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>14.54</td>
<td>1703.0</td>
<td>283</td>
<td>18.59</td>
<td>1144.8</td>
<td>5</td>
<td>35.02</td>
<td>1412.4</td>
<td>268</td>
<td>5.52</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>13.89</td>
<td>2532.5</td>
<td>420</td>
<td>19.52</td>
<td>1485.5</td>
<td>5</td>
<td>35.70</td>
<td>1907.9</td>
<td>425</td>
<td>12.99</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>12.98</td>
<td>3338.1</td>
<td>626</td>
<td>19.61</td>
<td>1528.0</td>
<td>15</td>
<td>32.70</td>
<td>2419.7</td>
<td>629</td>
<td>21.65</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>11.12</td>
<td>4757.7</td>
<td>878</td>
<td>21.80</td>
<td>1844.9</td>
<td>18</td>
<td>32.92</td>
<td>2398.7</td>
<td>892</td>
<td>41.72</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

#### Experiment 4 - ECEM Performance Short Transactions

| MPL | Short Writers | Short Readers | Overall | Blocksize | CPU
|-----|---------------|---------------|---------|-----------|-----
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.09</td>
<td>323.0</td>
<td>21</td>
<td>6.63</td>
<td>183.8</td>
<td>2</td>
<td>12.92</td>
<td>249.3</td>
<td>23</td>
<td>1.05</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>11.86</td>
<td>367.7</td>
<td>24</td>
<td>13.16</td>
<td>215.4</td>
<td>1</td>
<td>25.04</td>
<td>297.6</td>
<td>25</td>
<td>1.19</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>15.24</td>
<td>531.6</td>
<td>63</td>
<td>17.54</td>
<td>323.2</td>
<td>3</td>
<td>32.70</td>
<td>450.0</td>
<td>98</td>
<td>1.48</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>16.17</td>
<td>1150.0</td>
<td>118</td>
<td>16.54</td>
<td>740.0</td>
<td>3</td>
<td>33.70</td>
<td>908.1</td>
<td>121</td>
<td>2.18</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>14.69</td>
<td>1724.8</td>
<td>262</td>
<td>16.97</td>
<td>1123.0</td>
<td>1</td>
<td>35.06</td>
<td>1365.8</td>
<td>203</td>
<td>3.02</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>13.47</td>
<td>3197.3</td>
<td>401</td>
<td>20.11</td>
<td>1804.9</td>
<td>10</td>
<td>33.58</td>
<td>2251.3</td>
<td>411</td>
<td>9.70</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>15.41</td>
<td>4157.3</td>
<td>561</td>
<td>21.84</td>
<td>1926.2</td>
<td>2</td>
<td>33.75</td>
<td>2770.1</td>
<td>633</td>
<td>18.57</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

Table 8 - Experiment 4 Performance Tables.