THE PRIVATE WORKSPACE MODEL OF CONCURRENCY
CONTROL: THEORY AND PRACTICE
(D.Sc. Thesis)

by

Israel Gold

Technical Report #426

July 1986
The Private Workspace Model of Concurrency Control: Theory and Practice

Research Thesis submitted in partial fulfillment of the requirements for the degree of DOCTOR OF SCIENCE

By

ISRAEL GOLD

submitted to the Senate of the TECHNION - Israel Institute of Technology
TEVET 5745 HAIFA DECEMBER 1985
In the *private workspace model* of concurrency control, the transaction management (TM) component of a database management system (DBMS) maintains a *private workspace* for each active 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 work studies the design, feasibility and performance advantages of private workspace based concurrency control algorithms.

The fact that write operations of distinct transactions are performed into distinct private workspaces motivates the introduction of the PRE_WRITE operation (a "temporary" write into a private workspace) as a *synchronization primitive*. This facilitates the construction of various 2PL and Certification based synchronization techniques and leads to the notion of an *Integrated Concurrency Control Algorithm (ICCA)*. An ICCA consists of several rw and several ww synchronization techniques and has the capability of using any number of them concurrently. Some ICCA algorithms which employ 2PL and Certification based synchronization techniques are presented. The dynamic nature of these algorithms facilitates a smooth transition of a transaction class from one synchronization technique to another. Thus, for a given transaction system, the concurrency control may concurrently employ various synchronization policies, ranging from strict locking to full certification.

Two inherent properties of the private workspace model are examined. First, it is observed that the freedom implied by using PRE_WRITEs in synchronizing conflicting operations can be utilized for optimizations which seem to decrease the number of transaction restarts. An ICCA which guarantees that no transaction will be restarted...
CHAPTER 1

INTRODUCTION

In a data base management system (DBMS) users are provided with an abstraction known as a transaction which denotes a program accessing the database. Concurrency Control is the activity of synchronizing concurrent access to the database by multiple transactions; it permits transactions to access the database in a multiprogrammed fashion while preserving the illusion that each transaction is executing by itself on a dedicated system. The main technical goal is the preservation of database consistency, i.e., prevent database updates performed by one transaction from interfering with database retrievals and updates performed by another.

In the private workspace model of concurrency control, the transaction management (TM) component of a DBMS maintains a private workspace for each active transaction which is inaccessible to other transactions. 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, there is no need to undo updates made by an incomplete transaction.

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. Other algorithms rely on methods which allow conflicting transactions to run concurrently and use transaction restart to preserve database consistency [BADA79, BERN82, KUNG81]. These methods are called Certification methods because
they either abort a transaction or certify that a transaction may perform additional processing or commit.

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

Various centralized concurrency control algorithms have been compared in an attempt to evaluate their operational merits. The studies range from the purely abstract [PAPA79] to the more realistic where such measures as system throughput and response times are taken into account [AGRA83, CARE83, GAJA82, KIES83, LIN82, PEIN83, ROBI82, TAYB84, WILK81]. Naturally, if a clear cut conclusion could be reached about one algorithm being “best” at almost all times, then this algorithm should be employed by all database management systems. Unfortunately, the conclusions so far are not definitive. They tend to contradict each other primarily because of the variety of underlying system models and assumptions. The role of transaction restarts has not been thoroughly studied; to the best of my knowledge no one has attempted to evaluate an SG checking algorithm.

It is my thesis that the concurrency control mechanism in a DBMS should be a versatile piece of software that has the ability to adapt itself to various system workloads (or environments). We envision a mechanism equipped with a variety of locking based and certification based algorithms. It may operate in one of the following ways:

1. It uses one algorithm for all the transactions; it periodically switches from one algorithm to another (based on the system workload).
(2) It uses several algorithms concurrently; each for a given predetermined class of transactions and ensures that conflicts between transactions from different classes are correctly resolved.

A disadvantage of the first approach is that at any given time exactly one algorithm is used to synchronize all transactions. In an environment in which several transaction classes are executing concurrently, a concurrency control mechanism of this type may not perform well because of the need to frequently change algorithms. This approach has been investigated by Robinson [ROB182].

In the second approach, the allocation of algorithms to classes may be done statically (as was done in the SDD-1 concurrency control mechanism [BERN80]) or quasi-dynamically — by using an initial static allocation that may change at run time. The quasi-dynamic approach seems to be the most promising approach.

The notion of an Integrated Concurrency Control Algorithm (ICCA) is introduced in order to support the quasi-dynamic approach. An ICCA consists of a set of rw synchronization techniques and a set of ww synchronization techniques running concurrently. Each transaction may be mapped statically (using some predetermined decision) and dynamically (according to system state parameters) to one rw technique and one ww technique.

This thesis studies the design, feasibility and performance advantages of private workspace based ICCAs. The following is an outline of the thesis and a summary of its main results. Parts of this thesis have recently appeared in [BORAL84, GOLD85, GOLD85a].

Chapter 2 contains an overview of concurrency control theory and performance evaluation studies. In Section 2.1 an informal discussion of the concurrency control problem is presented. In section 2.2 the mathematical foundations of concurrency con-

---

1 We envision a class as composed of transactions that share some common feature, i.e. "short writers" as opposed to the characterization of classes in [BERN80].
control are reviewed. In section 2.3 2PL and Certification based synchronization strategies are discussed. In section 2.4 previous research in the area of concurrency control performance evaluation is surveyed with an emphasis on studies related to this work.

The private workspace model is described in in chapter 3. This model differs from the one known in the literature [e.g. BERN81, KUNG81] by employing an atomic write phase (the effect is not a disk access but rather the placement of DM_WRITE operations on appropriate queues).

The fact that write operations of distinct transactions are performed into distinct private workspaces motivates the utilization of the PRE_WRITE operation (a "temporary" write into a private workspace) as a synchronization primitive. In section 3.2 basic serializability theory is extended in order to capture the serializability implications of the private workspace model. It is shown that conflicts involving a PRE_WRITE operation essentially foresee future conflicts between DM_READ and DM_WRITE operations.

Chapter 4 is concerned with the design of private workspace based ICCAs. Since PRE_WRITE operations do not affect the database state, the concurrency control has complete freedom in choosing how to use PRE_WRITEs in synchronizing conflicting operations. This facilitates the construction of various 2PL and Certification based synchronization techniques and leads to the design of ICCA algorithms which concurrently employ both blocking and transaction restart in synchronizing conflicting transactions.

Section 4.1 describes in detail the TM component of the private workspace model. Section 4.2 presents a taxonomy of concurrency control synchronization techniques. Section 4.3 introduces the notion of a transaction's Wait For Set (WFS) as a parametric means for controlling the concurrency level in the system. Some 2PL and Certification based synchronization techniques are described in section 4.4. Two ICCAs based on these techniques, along with a proof of their correctness, are given in section 4.5.
Finally, some practical implementation issues are discussed in section 4.6. The dynamic nature of the ICCAs presented facilitates a smooth transition of a transaction class from one synchronization technique to another. Thus, for a given transaction system, the concurrency control may \textit{concurrently} employ synchronization policies ranging from strict locking to full certification.

Two properties of the private workspace model are examined in chapter 5. First, we observe that the freedom of using \texttt{PRE\_WRITE}s in synchronizing conflicting operations can be utilized for optimizations which seem to decrease the number of restarts in the system. Second, we show that due to the \textit{atomic write phase}, there is no need for \texttt{ww} synchronization; this results in simpler algorithms. A strategy for avoiding cycles in the Serialization Graph is presented in section 5.1. It is formalized in section 5.2 by introducing the notion of a transaction's \texttt{Optimized Wait For Set (OWFS)}. Section 5.3 presents ICCAW 2PL and shows that by means of the OWFS we can guarantee that no transaction will be restarted due to a READ request or because of conflicting WRITE requests. Section 5.4 proves the correctness of two concurrency control algorithms which include no \texttt{ww} synchronization.

The \textit{feasibility} of the private workspace model is addressed in chapter 6. Following a detailed physical description of model in section 6.1, realizable \textit{efficient} implementations of the \textit{atomic commit phase} are studied in section 6.2. Complications arise in integrating the notion of an atomic write phase with Commit Time Synchronization (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 integrity of CTS. In order to exactly record READ/WRITE conflicts no DM\_READ operations are allowed to 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 \textit{after} the CTS and \textit{before} issuing the DM\_WRITEs. Performing I/O
on a Read request or because of conflicting WRITE requests is presented. Second, it is shown that due to the atomic write phase requirement, there is no need for write synchronization; this leads to simpler algorithms.

The main difficulty in practically using the private workspace model lies in attaining an efficient implementation of the atomic commit phase. By exhibiting a parallel commit phase algorithm, which performs I/O operations only outside a critical section of the TM, it is shown that concurrency control methods developed under the private workspace assumption can be matched with an efficient recovery management procedure.

Once the feasibility of the model is exhibited a substantial comparative simulation study was conducted in order to demonstrate the performance advantages of private workspace based ICCAs. The simulation study is novel in three aspects. First, by employing a simulation model which integrates concurrency control and recovery functions. Second, by modeling a CPU bound system with infinite parallel I/O capability and third, by separating the effects of data and resource contention. The simulation study compares various private workspace based concurrency control algorithms which are modeled by means of a single ICCA. These, represent various synchronization policies ranging from strict 2PL to pure SG-checking. The results of the experiments conducted indicate that major advantages of the ICCA concept is in the ability to control the concurrency level in the system and in dynamically matching transactions (or in general transaction classes) with appropriate synchronization techniques according to performance objectives. For the system that we modeled, controlling the blocking level proved to have a more far reaching impact on system performance than attempting to minimize the number of restarts.
ACKNOWLEDGEMENTS

I would like to thank several people who helped me along the way.

My advisors, Dr. Haran Boral and Dr. Oded Shmueli, gave me generous support, were always willing to listen to my ideas and provided constructive criticisms. Discussions with Haran inspired the Private Workspace Model and the ICCA idea. Oded's insight helped in exhibiting the feasibility of the model; his encouragement and support during the last stages of the research have been invaluable.

Prof. Micha Höffri provided helpful suggestions in the design of the simulation model.

Dr. Eliezer Kantarovitz gave me advice and help in the early steps of the way.

Finally, I am grateful to my wife, Ruth, and my family for their understanding and support throughout the years.
# TABLE OF CONTENTS

1. Introduction ........................................................................................................... 1

2. Overview of Concurrency Control Theory and Practice ........................................ 8
   2.1 The Concurrency Control Problem ................................................................... 9
   2.1.1 Transaction Processing and Database Consistency ..................................... 9
   2.1.2 Concurrency Control Anomalies ................................................................ 11
   2.2 Serializability Theory ..................................................................................... 13
      2.2.1 Basic Definitions .................................................................................... 14
      2.2.2 Serializability ........................................................................................ 15
   2.3 2PL and Certification based Synchronization Strategies .................................. 18
      2.3.1 Two-Phase Locking (2PL) ..................................................................... 18
      2.3.2 Certification ............................................................................................ 21
   2.4 Performance of Concurrency Control Algorithms .......................................... 25
      2.4.1 Concurrency Control Effects on System Performance ......................... 25
      2.4.2 Performance Research .......................................................................... 27

3. The Private Workspace Model ............................................................................... 32
   3.1 Model Description ........................................................................................... 32
   3.2 Serializability ................................................................................................ 34

4. Integrated Concurrency Control Algorithms (ICCA) ............................................ 43
   4.1 The Transaction Manager Model ...................................................................... 44
   4.2 Characterization of Synchronization Techniques ............................................ 46
   4.3 The Wait For Set of a Transaction (WFS) ....................................................... 47
   4.4 The Synchronization Techniques ..................................................................... 49
      4.4.1 RW Synchronization Techniques ............................................................. 49
      4.4.2 WW Synchronization Techniques ........................................................... 51
      4.4.3 Formalization of the Synchronization Techniques ................................. 55
   4.5 2PL and Certification based ICCAs ................................................................. 56
   4.6 Practical Considerations, ................................................................................. 63
      4.6.1 Management of SG and IT ................................................................. 63
      4.6.2 Management of a data item WAIT QUEUE ........................................... 65

5. The Power of the Private Workspace Model ......................................................... 68
   5.1 A Strategy for Avoiding Cycles in SG ............................................................. 68
   5.2 The Optimized Wait For Set (OWFS) of a Transaction .................................. 70
   5.3 The Workspace 2PL(W2PL) ICCA ................................................................. 73
LIST OF FIGURES

Fig. 2.1 Lost Updates ................................................................. 12
Fig. 2.2 Inconsistent Retrievals .................................................. 13
Fig. 2.3 Waits-For Graph .......................................................... 20
Fig. 2.4 2PL Strategy versus Certification Strategy ......................... 22
Fig. 2.5 Concurrency Control Effects on System Performance ............. 26
Fig. 3.1 The Execution Schedule modeled by its Synchronization and Execution Logs ......................................................... 38
Fig. 3.2 Serializable Logs .......................................................... 39
Fig. 4.1 The Synchronization Techniques ....................................... 47
Fig. 4.2 C-a Matrices for the RW Synchronization Techniques .......... 54
Fig. 4.3 C-a Matrices for the WW-Synchronization Techniques .......... 54
Fig. 4.4 Typical cut of SG ......................................................... 64
Fig. 6.1 The Private Workspace Model .......................................... 84
Fig. 6.2 The TM - DM Relation ................................................... 89
Fig. 6.3 Basic Commit Phase Procedure ....................................... 93
Fig. 6.4 A justification of the requirement that the CTS and the write phase must be executed in a critical section ...................... 94
Fig. 6.5 Parallel Commit Phase Procedure ..................................... 95
Fig. 6.6 System Block Diagram .................................................. 97
Fig. 6.7 Improved Commit Phase and Recovery Procedures ............... 98
Fig. 7.1 DBMS Logical Structure ................................................ 104
Fig. 7.2 TP Model .................................................................... 105
Fig. 7.3 CC Model .................................................................... 106
Fig. 7.4 Transaction State Diagram .............................................. 109
Fig. 7.5 DBMS Physical Model .................................................... 111
Fig. A.1 The TM-DM logical relationship ....................................... 142
Fig. A.2 The TM-Parallelizer-DM logical relationship ...................... 143
Fig. A.3 Parallelizer version supporting redundant DM_WRITEEs Cancellation .... 146
operations in a critical section may slow the system considerably. Following a detailed description of the Private Workspace Model physical abstraction in section 6.1, in section 6.2.3 a parallel commit phase algorithm which performs I/O operations only outside a critical section of the TM together with a recovery procedure is proposed. Even in those cases where no CJS is necessary, the commit algorithm enhances performance by queuing commit record writing requests on a dedicated commit queue. Immediately after queuing its commit request and issuing its updates, all transactions blocked by a committing transaction, which is not yet actually committed, may resume operation. A 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).

Once the feasibility of the model is exhibited a substantial comparative simulation study was conducted in order to demonstrate performance advantages of private workspace based ICCAs. To this end, a "complete" simulation model of a transaction processing system which integrates concurrency control and recovery functions was implemented. In this model, the ICCAs introduced in chapter 4 along with all discussed functions and optimizations were available. The simulation model and the experiments conducted are described in chapter 7. In order to aid simulation monitoring and allow for interactive studies, a high level experiment specification language (see Appendix B) and an interpreter for it were developed on top of the simulation model.

Three important aspects of the simulation are the following. 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 components 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 (conflict) rate and system resource utilization. This substantially reduced the number of needed experiments and separated the effects of data and resource contention.

The simulation study compares various private workspace based concurrency control algorithms which are modeled by means of a single ICCA. These, represent various synchronization policies, ranging from strict 2PL to pure SG checking. The results of the experiments conducted indicate that the major advantages of the ICCA concept is in the ability to control the concurrency level in the system and in dynamically matching transactions (or in general transaction classes) with appropriate synchronization techniques as dictated by performance objectives. For the system that we modeled, controlling the blocking level proved to have a more far reaching impact on system performance than attempting to minimize the number of restarts. A synchronization policy in which readers (queries) use certification (SG checking) whereas writers (updating transactions) use 2PL significantly improved the performance of "ordinary" 2PL policy.
CHAPTER 2

AN OVERVIEW OF CONCURRENCY CONTROL THEORY AND PRACTICE

In a data base management system (DBMS) users are provided with an abstraction known as a transaction which denotes a program accessing the database. Concurrency Control is the activity of synchronizing concurrent access to the database by multiple transactions; it permits transactions to access the database in a multiprogrammed fashion while preserving the illusion that each transaction is executing by itself on a dedicated system. The main technical goal is the preservation of database consistency, i.e., prevent database updates performed by one transaction from interfering with database retrievals and updates performed by another.

During the past decade, the area of concurrency control has seen intensive research activity. Some of this research has focused on serializability theory, a discipline used in proving the correctness of concurrency control algorithms [BERN79, BERN81, CASA79, ESWA76, PAPA77, PAPA79, STEA76]. This theory has led to the development of new concurrency control algorithms; the majority of which are based, to one degree or another, on three basic synchronization techniques: locking [ESWA76, GRAY79, LIND79, MENA79, ROSE78], timestamping [BERN80, BERN81, REED78, THOM79], and certification [BADA79, BAYE80, BERN82, CASA79, KUNG81]. Recently, researchers have begun to study the relative performance of various algorithms [AGRA83, CARE83, GALL82, LIN82, RIES77, ROB82, TAY84, WILK81]. These studies make use of analytical methods as well as simulation and experimental models, where such measures as system throughput and response times are taken into account.

In Section 2.1 an informal discussion of the concurrency control problem is presented. In section 2.2 we review the mathematical theory of concurrency control.
In section 2.3 we discuss 2PL and Certification based synchronization strategies. In section 2.4 previous research in the area of concurrency control performance evaluation is surveyed with an emphasis of studies related to this work.

2.1. The Concurrency Control Problem

In this section the notion of a transaction as an atomic unit preserving database consistency is introduced. To illustrate the concurrency control problem, some examples of transaction interference which cause database inconsistency are presented.

2.1.1. Transaction Processing and Database Consistency

A database is a model of some aspects of the real world. The state of the database is the set of all values in the database together with their logical relationships. Each database is associated with a set of integrity constraints which distinguish between "consistent" database states and "inconsistent" states. A database is consistent if it satisfies its integrity constraints.

For example, a constraint on a banking database might be that the total credit balance in all accounts must be equal to the total debit balance in all accounts.

Money transfer from account A to account B involves two activities:

(1) debit account A
(2) credit account B

Each of these activities involves in turn two operations:

(a) retrieve old account balance from the database
(b) update new account balance in the database

A customer money transfer request is translated into a number of operations on the database. A single change to the database, e.g. debiting account A, clearly violates the database integrity constraints. If these constraints were checked following each change, then no update would be allowed. Therefore, it is only necessary to check for
integrity violations at the end of a series of database updates corresponding to a change in the real world. One source of database inconsistency problems is that arbitrary interleaving of database operations may produce incorrect results. These observations led to the notion of a database transaction. A transaction is a series of atomic database operations (e.g., read or write a record) which correspond to a change in the real world. Since a transaction is assumed to satisfy the integrity constraints of the application, a transaction may be viewed as a function which maps the database from one consistent state into another.

It should be apparent that if transactions are executed serially (i.e., one must complete before the next one begins) and the system never fails, then database consistency will be preserved. Unfortunately, systems do fail and restricting transaction processing to a single transaction at a time reduces transaction throughput and leads to poor resource utilization. In order to solve these problems, a DBMS usually contains two mechanisms: a concurrency control and a recovery manager.

The task of the concurrency control component is to preserve database consistency under simultaneous accesses and updates by multiple transactions. The common approach is to use a scheduler which interleaves transactions in a way that is "equivalent" to executing the transactions serially; i.e., executing the transactions simultaneously will produce the same outcomes that are the results of executing the transactions in some serial order.

The task of the recovery manager is to prevent system or transaction failures from resulting in database inconsistency. In other words, the recovery mechanism must ensure that only transactions which execute to completion will have an effect on the database; effects of interrupted transactions must be transparent to completing transactions, either by preventing them (using deferred updates) or by removing them before the database is made available following a failure. Typically, a recovery
manager maintains sufficient redundant information on stable storage (secondary storage which survives system crash), which makes it possible to reconstruct a consistent database state following the failure. With the recovery mechanism transactions are said to be "atomic", because, they effectively run to completion or not at all. Thus, the combined effect of the concurrency control and the recovery mechanisms is similar to that of running transactions serially and atomically.

2.1.2. Concurrency Control Anomalies

In this section we clarify the concurrency control problem by presenting two examples of transaction interference. The examples assume the existence of an Automated Teller Machines (ATMs) linked on-line to a banking DBMS [BERN81]. Using these ATMs, customers can deposit money, retrieve account balances and transfer money from account to account. Each request triggers the execution of a suitable transaction which retrieves data from the database, performs computations and stores results back into the database.

Lost Updates

Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control these two requests could interfere (see Figure 2.1). The two transactions handling these requests may read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances in the database one after the other. The end result is incorrect due to the lost update (of T1).
Inconsistent Retrievals

Suppose a customer and the bank manager execute the following requests.
Customer A: transfer $1000 from my checking account to my savings account.
Bank Manager: print total balance in customer A's checking and savings accounts.

In the absence of concurrency control the two requests may interfere (see Figure 2.2). The first transaction handling the customer's request might read the checking account balance, subtract $1000 and store the result in the database. Then the second transaction, handling the manager's request, might read customer A's checking and savings account balances and print the total. Finally, the first transaction might complete the money transfer by reading the savings account balance, adding $1000, and storing the result in the database.

Figure 2.1: Lost Updates
Unlike the lost updates anomaly, the final values of accounts placed into the database is correct. Still, the execution is incorrect because the balance printed by the bank manager is $1000 short.

\[ T_1 \quad \begin{array}{c} \leftarrow C \ 2,000 \\ \downarrow \ \ S \ 1,000 \end{array} \]

\[ \text{READ checkings balance} \quad \left| \begin{array}{c} 2,000 \end{array} \right| \]

\[ \text{Subtract 1,000} \]

\[ \text{WRITE result} \quad \left| \begin{array}{c} 1,000 \end{array} \right| \]

\[ \quad \downarrow \rightarrow C \ 1,000 \]

\[ \quad \leftarrow S \ 4,000 \]

\[ \text{READ Savings balance} \quad \left| \begin{array}{c} 1,000 \end{array} \right| \]

\[ \text{Add 1,000} \]

\[ \text{WRITE result} \quad \left| \begin{array}{c} 2,000 \end{array} \right| \]

\[ \quad \downarrow \rightarrow C \ 1,000 \quad \leftarrow S \ 2,000 \]

\[ \text{READ Checkings balance} \quad \left| \begin{array}{c} C \ 1,000 \end{array} \right| \]

\[ \text{READ Checkings balance} \quad \left| \begin{array}{c} S \ 1,000 \end{array} \right| \]

\[ \text{READ Checkings balance} \quad \left| \begin{array}{c} \text{Sum} = 2,000 \end{array} \right| \quad \text{Print Sum} \]

Figure 2.2: Inconsistent Retrievals

2.2. Serializability Theory

Serializability theory is a mathematical theory which gives precise conditions under which a concurrency control algorithm is correct [BERN79, CASA79, ESWA76,
2.2.1. Basic Definitions

A database consists of a set of logical data items, denoted $A, B, \ldots, X, Y, Z$. The granularity of a data item is left unspecified; in practice, a data item can be a file, page, record, etc. Data items are accessed by means of read and write operations; $\text{read}(x)$ returns the current value of $x$ and $\text{write}(x, \text{new value})$ updates the current value of $x$.

Users interact with the database by executing transactions. A transaction is an ordered sequence of read and write operations bracketed by two distinguished transaction and snapshot operations signaling the start and the end of the transaction. The logical readset (writeset) of a transaction is the set of logical data items the transaction reads (or writes). A reader (or query) is a transaction with an empty writeset. A writer is a transaction which updates at least one data item.

The computation performed by a transaction totally and deterministically depends on the values that the transaction reads. Furthermore, the value produced by a write operation of a transaction $T_1$ is considered as some (uninterpreted) function of the values returned by the read operations that precede this write operation in the execution order of $T_1$. Each transaction is assumed to be a complete and correct computation; i.e., each transaction, if executed by itself on an initially consistent database, would terminate, produce correct results, and leave the database in a consistent state.

A concurrency control algorithm is a scheduler which receives the operations queued by users' transactions and controls the order by which these operations are processed by the DBMS. When a concurrency control algorithm receives an operation, it can either output the operation immediately for execution, delay the operation by
holding it for later action, or reject the operation. A rejection causes the DBMS to restart (i.e., abort) the transaction which issued the operation; every write executed on behalf of the transaction is undone (thereby, removing its effects), and every transaction that read a value written by a restarted transaction is also restarted. This phenomenon of one restart triggering other restarts is called cascading restarts. Note that the actual implementation of the undo operation depends on the recovery mechanism used.

2.2.2: Serializability

Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions and let $E$ denote an execution of the transactions in $T$. $E$ is modeled by its execution log, $L$, which consists of the sequence of read and write operations of transactions in $T$, in the order output by the concurrency control algorithm. The semantics is that each output operation is executed atomically and only then the next output operation is executed.

In the following we shall assume that $T, E$ and $L$ as defined above are given. Furthermore, references to $T_i$ and $T_j$ are to any two distinct transactions in $T$. An $r_i[X]$ (resp., $w_i[X]$) is used to denote a read (resp., write) operation on $X$ executed by $T_i$: $O_i[X] < O_j[X]$ means that $O_i[X]$ precedes $O_j[X]$ in $L$. Two operations $O_i[X]$ and $O_j[X]$ conflict if one of them is a write operation. An rw-conflict is a conflict between $r_i[X]$ and $w_j[X]$ operations. A ww-conflict is a conflict between $w_i[X]$ and $w_j[X]$ operations.

Two execution logs are computationally equivalent\footnote{Another commonly accepted equivalence definition is finite state equivalence, computational equivalence implies finite state equivalence but not vice versa.} [PAPA77, PAPA79] if:

1. each $r_i[X]$ operation reads the same value of a data item $X$ that was produced by the same $w_j[X]$ operation in both logs, and
2. the final $w_i[X]$ operation on each data item $X$ is the same in both logs.
Condition (1) ensures that each transaction reads the same values in both executions (and therefore performs the same computation). Combined with (2), it ensures that both executions leave the database in the same final state (under the assumption of an uninterpreted function for a write operation).

A serial execution log is an execution log in which for every pair of transactions $T_i$ and $T_j$, either all of $T_i$'s operations precede all of $T_j$'s or vice versa. A serial execution log models an execution in which each transaction is executed to completion before another is started.

An execution log is serializable if it is computationally equivalent to some serial execution log. An execution is serializable if it is modeled by a serializable execution log. A concurrency control algorithm is correct if it allows only serializable executions.

The serializability theorem (PAPA77, PAPA79, STEA76):

Let $E$ be an execution of $T$ modeled by $L$. If there is a total ordering of $T$ such that for each pair of conflicting operations $O_i[X]$ and $O_j[X]$, $O_i[X]$ precedes $O_j[X]$ in $L$ if and only if $T_i$ precedes $T_j$ in the total ordering, then $L$ is serializable.

Decomposition of the Concurrency Control Problem (BG81):

Let $E$ be an execution of $T$ modeled by $L$. For every pair of transactions $T_i, T_j$ define the following binary relations, associated with $L$ if there exists a data item $X$ for which the condition holds.

1. $T_i \rightarrow_{rw} T_j$ if $r_i[X] < w_j[X] \text{ in } L$.
2. $T_i \rightarrow_{wr} T_j$ if $w_i[X] < r_j[X] \text{ in } L$.
3. $T_i \rightarrow_{ww} T_j$ if $w_i[X] < w_j[X] \text{ in } L$.
4. $T_i \rightarrow_{rwr} T_j$ if $T_i \rightarrow_{rw} T_j$ or $T_i \rightarrow_{wr} T_j$.
5. $T_i \rightarrow_{wwr} T_j$ if $T_i \rightarrow_{rwr} T_j$ or $T_i \rightarrow_{ww} T_j$.

\[^{b}\text{This is not iff.}\]
A relation \( R \) is acyclic if there is no sequence \( T_1 \rightarrow R T_2 \rightarrow R T_3 \rightarrow \ldots \rightarrow R T_n \) such that \( n > 1 \) and \( T_1 = T_n \).

The decomposition theorem [BG81]:

Let \( R \) and \( W \) be associated with an execution \( E \) modeled by \( L \). \( L \) is serializable if

1. \( R \) and \( W \) are acyclic, and
2. There is a total ordering of the transactions consistent with all \( R \) and all \( W \) relationships.

The decomposition theorem suggests designing concurrency control algorithms consisting of:

1. An \( R \) \( W \) synchronization technique which is responsible for synchronizing \( R \) \( W \) conflicts and guarantees the acyclicity of the \( R \) relation.
2. A \( W \) \( W \) synchronization technique which is responsible for synchronizing \( W \) \( W \) conflicts and guarantees the acyclicity of the \( W \) relation.
3. A coupler between the above synchronization techniques which attains a total order on the transactions consistent with both the \( R \) \( W \) and \( W \) \( W \) relations.

The Serialization Graph

The serialization graph is a convenient device for analyzing and understanding various concurrency control algorithms and is also useful for their implementation.

Let \( R \) be associated with an execution \( L \). The serialization graph \( SG(L) \) is a directed graph whose nodes are \( T_1, \ldots, T_n \), and whose edges are all \( (T_i, T_j) \) such that \( T_i \rightarrow R T_j \) is in \( R \).

The restated serializability theorem (using \( SG \)) [BERN79, PAPA79]:

Let \( E \) be an execution of \( T \) modeled by \( L \). If \( SG(L) \) is acyclic then \( E \) is serializable.

This theorem follows immediately from the fact that all conflicts in \( L \) are represented in \( SG(L) \).
2.3. 2PL and Certification based Synchronization Strategies

In the previous section, the notion of serializability was introduced as a basis for discussing concurrency control correctness. An important result is that an execution is serializable (correct) if the serialization graph associated with its execution log is acyclic. Unfortunately, this result provides little insight into how to construct a concurrency control algorithm. The construction problem is far from being overlooked in the literature; there are numerous published algorithms, which are referenced in [BERN81, HSIO81]. Using the decomposition paradigm it was shown that there are hundreds of conceivable algorithms [BERN81].

The majority of the algorithms make use, to one degree or another, of three synchronization strategies: locking, timestamping and certification. Locking algorithms are based on the Two Phased Locking method (2PL) [ESWA76] in which blocking is used to synchronize conflicting transactions. Timestamp based algorithms [BERN80, BERN81, REED78, THOM79] select a priori an equivalent serial order for transaction execution, usually by assigning a timestamp to each transaction upon its initiation. These algorithms resolve conflicts by forcing operations to obey the a priori order (and if this is impossible, then restart is used). Certification algorithms [BADA79, BAYE80, BERN82, CASA79, KUNGB1, WILK81] allow conflicting transactions to run concurrently. Transaction restart is used in cases where inconsistencies could result (these are known as certification methods because they certify a transaction (or additional processing or restart it).

2.3.1. Two-Phase Locking (2PL)

Two-Phase Locking (2PL) is the earliest and most common concurrency control method. 2PL synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations as defined by the following three rules [ESWA76]:
(1) Before outputting $r_i[X]$ (resp. $w_i[X]$), set a read-lock (resp. a write-lock) for $T_i$ on $X$. The lock must be held (at least) until the operation is executed.

(2) Different transactions cannot concurrently hold "conflicting" locks. Two locks conflict if they are on the same data item and (at least) one of them is a write-lock. If rw and ww synchronizations are done separately, the definition of "conflict" is modified. For rw synchronization, two locks on the same data item conflict if exactly one is a write-lock; i.e., write-locks do not conflict with each other. For ww-synchronization, both locks must be write-locks.

(3) After releasing a lock, a transaction cannot obtain additional locks.

Rule (3) causes locks to be obtained in a two-phase manner. During its growing phase, a transaction obtains locks. By releasing a lock, the transaction enters the shrinking phase during which it can only release locks. When a transaction is restarted, all of its locks are automatically released.

2PL is a correct synchronization technique, i.e. it attains an acyclic $\rightarrow_{\text{rw}}$ ( $\rightarrow_{\text{ww}}$ ) when used for rw (ww) synchronization [BERN79, ESWA76, PAPA79]. The serialization order attained by 2PL is determined by the order in which transactions obtain conflicting locks. Another observation is that unless write-locks are held until transaction termination, 2PL may suffer from cascading restarts [GRAY75]. As a result, rule(3) is usually implemented by holding all of a transaction's locks until termination.

Deadlock

Due to rule(2) transactions are forced to wait for unavailable locks. This may lead to deadlocks (see Figure 2.3). Deadlocks can be characterized by a wait-for graph [HOLT72, KING74], a directed graph whose nodes represent transactions and whose edges represent the waiting relations; edge $(T_i, T_j)$ means that $T_i$ is waiting for a lock owned by $T_j$. A deadlock exists if and only if the wait-for graph is cyclic. A common
A method for handling deadlocks is to maintain the wait-for graph and periodically check for cycles. (see [AH075 chap. 5, HOLT72, SHMU83] for cycle detection and prevention algorithms). When a deadlock cycle is detected one of the transactions on the cycle, called a victim, is restarted, thereby breaking the cycle.

\[
\begin{align*}
T_1 &: \text{trans \ read}(X); \text{write}(Y); \text{snart} \\
T_2 &: \text{trans \ read}(Y); \text{write}(Z); \text{snart} \\
T_3 &: \text{trans \ read}(Z); \text{write}(X); \text{snart}
\end{align*}
\]

**Figure 2.3 Wait-For Graph**

The form of locking we have considered above is also called dynamic locking, where a transaction requests additional locks as it proceeds with its computation. There is an alternative, called static locking, in which a transaction acquires all the locks it needs before beginning its computation. In static locking, when a transaction starts, it *predeclares* the locks it needs. If they are all available, the transaction receives those locks and begins computation; otherwise, it waits for some time before retrying.
2.3.2. Certification

2PL may be viewed as a synchronization strategy for achieving serializable executions by explicitly preventing the creation of non-serializable ones. The synchronization is achieved using blocking. This approach suffers from an inherent delay caused by the blocking of transactions; transactions must wait for the release of conflicting locks owned by other transactions. This may cause a group of transactions to block each other, thereby prolonging the waiting. Even readers which do not affect database consistency are required to set locks in order to guarantee that the data items they read are not concurrently updated.

The certification approach may be viewed as a synchronization strategy for achieving serializable executions by detecting non-serializable ones. The synchronization is achieved by restarting transactions which, if not restarted, will cause a non-serializable execution. The rational for this approach is based on an optimistic assumption regarding run-time conflicts: if few run-time conflicts are expected, then most executions are likely to be serializable and thus very few transactions will be restarted.

Figure 2.4 illustrates the limitation imposed by the blocking employed in 2PL; by preventing conflicts between concurrent operations 2PL may also prevent faster serializable executions. Using 2PL T2 waits until T1 terminates, however, by using certification T2 could terminate much earlier (before T1 does). In both cases the executions are serializable.

The above observations lead to the development of concurrency control algorithms in which transaction operations are arbitrarily interleaved and are never delayed while being processed. Restarting transactions is the only method used to avoid non-serializable executions. The decision to restart a transaction can be made either on a per-operation basis (by checking for serializability following every operation received) or on a per-transaction basis by a (hopefully) fast serializability test at
transaction termination. The way in which transactions serializability is checked for
distinguishes between the various algorithms.

\[ L = r_1[A]r_1[B]...w_1[Z]w_2[A] \]

\[ L = r_1[A]w_2[A]r_1[B]...w_1[Z] \]

\[ \text{2PL execution log} \]

\[ \text{Certification execution log} \]

**Figure 2.4** 2PL Strategy versus Certification Strategy

**Serialization Graph (SG) Checking**

The SG checker \([\text{BERN82}]\) works by continuously building the serialization graph
and checking it for cycles for every received operation. The basic algorithm is defined
by the following rules:

1. When a \textit{trans} operation from transaction \(T_i\) is received, add node \(T_i\) to SG.
2. When a \textit{read} or \textit{write} operation from transaction \(T_i\) is received, add all edges \((T_j, T_i)\)
such that \(T_j\) is a node of SG, and the algorithm has already output a conflicting
operation from \(T_j\) (the definition of "conflicting operation" is modified if \(r_w\) and \(w_w\)
conflicts are treated separately). If SG remains acyclic, then output the
operation. Otherwise, reject it by deleting node $T_i$ and all edges $(T_i, T_j)$ or $(T_j, T_i)$ from SG (SG is acyclic again).

Since every execution log produced by the algorithm has an acyclic SG, by the serializability theorem every such execution is serializable.

There are two main problems with the SG checker. First, a (node for a) transaction must remain in SG even after the transaction has terminated. A transaction (node) can only be deleted from SG if it is a root node of the graph, i.e., when it has no incoming edges. See [CASA79] for a discussion of this problem and techniques for efficiently encoding information about terminated transactions that must remain in SG. Second, the SG checker may induce cascading restarts.

A certification algorithm using (a version of) SG checking for rw synchronization and 2PL for ww synchronization is proposed in [BAYEB0]. In this algorithm a read operation is never delayed or rejected. The cascading restarts phenomenon is avoided by delaying the termination of a writer until all "old readers" preceding it in SG have terminated.

Serial Validation

An algorithm using certification on a per-transaction basis is proposed in [KUNG81]. In this algorithm transaction execution is divided into two phases. In the computation phase the transaction reads values from the database, performs various computations and writes results into its private workspace. Only read and no write operations are output by the algorithm during this phase. In the validation phase (which takes place after the transaction finishes all computations) the transaction first goes through a (possible empty) serializability test to ensure that certifying it will not cause database inconsistencies. If the serializability test fails, then the transaction is restarted. Otherwise, it is certified and all of its write operations are output thereby installing the transaction's updates in the database. The validation phase takes place...
in a critical section of the validator which imposes serial validation phases, i.e., only a single transaction at a time can be validated and output its write operations at a time. Read operations, however, are not limited and may be carried out in parallel.

Two sets of data items are maintained for each transaction T_j:

T_j's readset, RS(T_j) = \{ X | the algorithm has output r_j[X] \}

T_j's writeset, WS(T_j) = \{ X | the algorithm has output w_j[X] \}

At the time transaction T_i goes through the serializability test, the algorithm determines the range of timestamps of transactions with which T_i could conflict. The lower limit of the range, T_i.start, is the latest timestamp that was already allocated when T_i started its computation phase, i.e., the timestamp of the last transaction which was certified before T_i started. The upper limit of the range, T_i.finish, is the latest timestamp allocated when T_i entered the validation phase, i.e., the timestamp of the last transaction certified during T_i's computation phase.

The test is as follows:

\[
\text{if } RS(T_i) \cap \bigcap_{j=n_{\text{start}}+1}^{n_{\text{finish}}} WS(T_j) = \emptyset \text{ then certify } T_i \text{ else restart } T_i
\]

Note that in the example given in Figure 2.4, T_1 will be certified by the SG checker but will fail in the strict serial validation test (since T_1's readset intersects with T_2's writeset and T_2 completes during T_1's computation phase).
2.4. Performance of Concurrency Control Algorithms

The correctness of concurrency control algorithms is now well understood, and attention has shifted to their relative performance. Researchers have begun to evaluate the performance of different algorithms and their operational merits. The popularity of concurrency control algorithms using locking (static or dynamic) is reflected by the large number of performance studies concerning such algorithms [AGRA83, BAL982, CARE83, CARE84, DOVE82, GALL82, IRAN79, KIES83, LIN82, LIN82a, LIN82b, MENA82, MUNZ77, PEIN83, POT80, RIES77, RIES79, RIES79a, ROBI82, ROBI82a, TAY84, TAY84a]. Certification algorithms, mostly Serial Validation, were studied and compared to locking in [AGRA83, CARE83, CARE84, FRAN83, KIES83, MENA82, PEIN83, ROBI82, ROBI82a, WILK81].

Some of these studies are of an experimental nature using simulations, while others are based on analytic models using queuing networks or markov processes. These have produced a wealth of data and some empirical results. However, the data are open to misinterpretation and the results can not be stated with any generality (in fact some are contradictory) because of the variety of underlying system models and assumptions used. In section 2.4.1 general concurrency control effects on system performance are demonstrated. Section 4.2.2 surveys previous research in the area of concurrency control performance evaluation with an emphasis of studies related to this work.

2.4.1. Concurrency Control Effects on System Performance

Figure 2.5 demonstrates the general effects of concurrency control on system performance. It presents system throughput (i.e., number of completed transactions per unit time) and transaction mean response time as a function of the concurrency level (i.e., the number of concurrently executing transactions). The throughput grows gradually to an optimum point. If the concurrency level continues to increase, the system
reaches a trashing point. Pushing the system beyond the trashing point, lowers throughput, increases response time and causes system instability. In the trashing region, the increased throughput from each added transaction is paid by a higher number of restarts per completion. In locking algorithms this is accompanied by prolonged waiting for a lock acquisition. One may therefore wish not to push the system up to the trashing point in order to maximize throughput.

![Concurrency Control Effects on System Performance](image)

**Figure 2.5 Concurrency Control Effects on System Performance**

The most dominant factor affecting concurrency control performance is the conflict rate among transactions. Suppose the database consists of $D$ data items, a transaction length is $k$ (the total number of reads and writes) and the number of concurrently executing transactions is $N$. The *workload parameter* given by $k^2 N/D$ [TAY84] models transactions' contention for data. Heavy workload characterizes environments with a high probability for transactions conflict, while light workload characterizes environments with a small conflict probability. Note that the workload parameter is
very sensitive to transaction length. Workloads of long transactions make the probability of a conflict, as well as the penalty associated with restarting a transaction, higher. Thus, workloads of long transactions adversely affect system performance; it is therefore desirable that transactions are as short as possible.

2.4.2. Performance Research

Ries and Stonebraker [RIES77, RIES79, RIES79a] conducted an extensive simulation study on the effects of database granularity on system performance. One of their findings was that granularities of 100 to 1000 were sufficient to achieve best performance for each algorithm studied under most system parameters. Therefore, simulation studies do not need to consider huge databases.

Balter, Berard and Decitre [BALTB2] observed that deadlocks are secondary to blocking in causing performance degradation for 2PL. They simulated several concurrency control algorithms and found that, when the system trashes for locks the waiting time spent to obtain locks is much greater than the time lost due to restarts. This is so even when the restart rate is significant. They conclude that blocking alone can cause trashing. Their main contribution is in warning that blocking should be controlled (say, by limiting the lengths of data item wait queues).

The observation that blocking can cause trashing before restarts do is supported by Tay [TAY84]. In studying the performance of locking, using an analytic mean value performance model, Tay separated the effects of data contention (i.e. conflicts over data) and resource contention (i.e. competition for memory, CPU, I/O etc). He modeled resource contention through an inter-request time parameter $T$; as resource contention increases, the rate of transactions execution slows down, and $T$ increases. Tay observed that 2PL performance degradation occurs even if the restart rate and the resource contention are negligible, and is the result of transactions holding on to locks while waiting for a new lock. By studying the no waiting protocol, in which a
transaction is restarted whenever it encounters a conflict. Tay concluded that restarting transactions without waiting offers a way to overcome the limitation that blocking imposes on 2PL, i.e., the no waiting protocol leads to higher throughput than 2PL. However, this is limited to situations in which restart rate is low and there is little resource contention. Under high resource contention 2PL outperforms the no waiting protocol. Because some transactions are blocked by 2PL whereas none is in the no waiting protocol, the resource contention adversely affects the latter’s performance. The significance of Tay’s study of the no waiting protocol is in the identification of the conditions under which a pure restart strategy is viable.

Using an extended version of the simulation model of Ries [RIES79], Carey conducted simulation experiments of 2PL and timestamp and Serial Validation algorithms [CARE83, CARE84]. Carey observed that for workloads in which conflicts are not rare, the concurrency control algorithms which performed best were those which minimized the number of transaction restarts. Thus, a major conclusion of his study is that blocking is preferable to restarts as a conflict-resolution technique, even when the workload is heavy. This seems to contradict Tay’s result of the no waiting protocol performance in low resource contention and high workload environments. The explanation given by Tay et al [TAY84a], which I tend to agree with, lies mainly in the fact that Carey modeled an I/O bound system and his simulations suffered from serious resource contention. As a result, restarting loses its advantages as a conflict-resolution technique for heavy workloads. Furthermore, for restarts to be preferable, it seems that one must impose a long enough conflict-avoidance delay on restarted transactions so that the transactions which caused conflicts can terminate. In Carey’s simulation, however, the response time is typically several times longer than the conflict-avoidance delay, which makes repeated conflicts likely. This also makes restarting less advantageous.
In addition to Carey, several researchers have compared 2PL to Serial Validation. Robinson performed research on the design of general transaction processing systems [ROB182, ROB182a]. He designed and implemented such a system on the Cm* multiprocessor system [FULL78]. As a test of the generality of his design, Robinson performed experiments in which several different concurrency control algorithms were executed by the system. Robinson observed that for his system the effect of blocking proved to be negligible, and so 2PL produced higher throughput than Serial Validation. The results of his experiments agree with those of Carey and Agrawal.

Agrawal [AGRAB3] performed a comparison study of several combinations of concurrency control and recovery algorithms. One result of his study, which was based on an analytical model of the "burden" experienced by transactions operating under the various concurrency control and recovery algorithm combinations, was that 2PL (combined with several different recovery mechanisms) is the best algorithm examined. He also concludes that Serial Validation is only reasonable under workloads consisting of short transactions for which conflicts are rare.

One can improve the concurrency of readers and writers (or read and write access in general) by creating new versions of a data item for each update, and keeping the old versions for read operations. One such algorithm is the so-called (r,a,c)-protocol [BAYE80]. Its performance was compared to that of the (single version) 2PL protocol by Keissling and Landherr [KIES83]. They conducted two sets of simulations, one with writers, and one with a mix of readers and writers (Tay et al argues that the results of these simulations are largely based on trashing data [TAY84a]). The motivation for the (r,a,c)-protocol is reducing the probability that a read access is blocked. This is a good idea, provided reducing the probability of blocking also reduces the time spent in waiting, and does not drastically increase restart costs. The results in [KIES83] show that the two-version protocol does in fact have a lower probability of conflict (compared to
2PL) and no big increase in restart costs. It therefore seems that having two versions would significantly improve performance. This observation is contradicted by Carey who studied multi-versions algorithms [CAREY83]. Carey concludes that multiple versions do little to improve 2PL, as 2PL does well for the workloads he examined without multiple versions. He observes that multiple versions were only effective for workloads consisting on many short updates and a few long read-only transactions.

Pienl and Reuter [PIEN83] used a benchmark-like technique to study 2PL, a two-version variant of 2PL [BAYE80], and Serial Validation algorithms. They took page reference strings from some applications on a CODASYL-like database, and fed them into a string converter, which generates reference strings for a given number of concurrent transactions. These in turn were run on a fictitious database with different concurrency controllers, in order to compare the performance of the algorithms. One conclusion from the experiments, contradicting the results in [KIES83], is that 2PL performs as well as the two-version protocol in [BAYE80]. Another conclusion, contradicting Carey’s results, is that Serial Validation performs well compared to 2PL although it leads to the largest number of restarts. A possible reason for the disagreements between the studies in [CAREY83, KIES83, PIEN83] is the difference in their choice of performance measures. Kiessling and Landherr based their performance comparison on the number of conflicts and restarts whereas Pienl and Reuter used the ratio of number of executing transactions to the number of restarts. Carey’s metric was transaction throughput.

In summary, results in the literature are not definitive and tend to contradict one another because of the variety of underlying system models and assumption used. The role of transaction restarts has been touched on only lightly; it seems that no one has attempted to evaluate an SG checking algorithm. This suggests a different approach to concurrency control design. One which is based on the thesis that the concurrency
control mechanism in a DBMS should be a versatile piece of software that has the ability to adapt itself to various system workloads (or environments). We envision a mechanism equipped with a variety of locking based, and certification based algorithms that may be used concurrently, each for a given predetermined class of transactions. Allocation of algorithms to classes may be done by using an initial predetermined decision that may be changed, at run time, according to performance objectives. The private workspace model serves as a framework for the design of such adaptive concurrency control mechanisms.
CHAPTER 3

THE PRIVATE WORKSPACE MODEL

In this chapter a logical abstraction of a private workspace based transaction-processing model is described. The notion of basic serializability theory is extended in order to capture the serializability implications of the model.

3.1. Model Description

In the Private Workspace Model a private workspace is allocated for each active transaction which is inaccessible to other transactions. 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. A similar model has been previously used in [BERN81, KUNG81]. The fact that write operations of distinct transactions are performed into distinct private workspaces motivates the use of the PRE_WRITE operation (a "temporary" write into a private workspace) as a synchronization primitive.

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 a database. 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. The granularity of a data item is left unspecified; in practice, a data item can be a file, page, record, etc.

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 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 which enables it to control the concurrency level in the system. 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.

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.

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 (i.e. as one indivisible and non-overlapping action); after all have been issued the transaction is complete.

---

1 The notion of the (logical) private workspace in this model differs from that used by Network based database management systems, where the user program "contains" its own private workspace (or user work area) which can be accessed at any time independently of the database management system.
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). Once a transaction has completed successfully its Commit Phase, it is known to be committed; the commit point is reached when the transaction executes its first DM_WRITE operation. The Commit Phase should effectively be atomic. A transaction \( T_i \) may be restarted by the TM any time before a DM_WRITE operation 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.

### 3.2. Serializability

In this section we study the schedules of operations output by the TM. We review basic serializability theory results and show that our use of PRE_WRITE as a synchronization primitive is compatible with known results. We also introduce several new precedence relations and show that conflicts involving a PRE_WRITE operation essentially foresee possible conflicts between DM_READ and future DM_WRITE operations.

Since transactions use private workspaces, restarted transactions leave no traces in the database and do not affect the operations of other transactions. Therefore, we can ignore all restarted transactions as far as serializability is concerned.

---

8 The actual implementation of the commit procedure need not be atomic as long as it appears atomic to the outside world. Realizable implementations of the commit procedure are presented in chapter 6.

9 For definitions of various terms, e.g. computational equivalence, see chapter 2.
Definition 3.1: Let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions. $E$, the execution schedule of $T$ in which each transaction executed to completion, is modeled by $L^S$, the synchronization log of $T$, which consists of DM_READ, PRE_WRITE, and DM_WRITE operations in the order in which they were scheduled by the TM. $L$, the execution log of $T$, is derived from $L^S$ by removing from it all PRE_WRITE operations.

In subsequent lemmas and theorems we shall assume that $T$, $E$, $L$, and $L^S$ as defined in Definition 3.1 are given. Furthermore, references to $T_i$ and $T_j$ are to any two distinct transactions in $T$. Let $\pi_i(L^S)$, denote the script of transaction $T_i$ derived from $L^S$, by removing from $L^S$ all operations which were not issued by $T_i$. A DM_READ($X$) operation by transaction $T_i$ is denoted by $r_i[X]$. Similarly, $p_i[X]$ and $w_i[X]$ will, respectively, denote PRE_WRITE and DM_WRITE operations on $X$ issued by transaction $T_i$. Finally, $O_i[X] < O_j[X]$ means that $O_i[X]$ precedes $O_j[X]$ in $L^S$. Figure 3.1 illustrates the notions of execution schedule, synchronization log and execution log.

Basic Assumptions of the Model

The private workspace model enforces certain structure on the synchronization logs produced.

1. \( \forall X \in \text{Writeset}(T_i), \ p_i[X] < w_i[X] \) in $L^S$. The $p_i[X]$ operation does not affect the database state; furthermore, the value it produces is considered as some uninterpreted function of the values returned by the DM_READ operations which precede this PRE_WRITE in $\pi_i(L^S)$. The value produced by the $w_i[X]$ operation is the same value that was produced by the final $p_i[X]$ operation in $\pi_i(L^S)$.

2. All $T_i$'s DM_WRITE operations are executed atomically with no interleaving operations from other transactions, i.e. $w_i[X] < O_j[Y] < w_i[Z], i \neq j$, is impossible in $L^S$.

3. Once $T_i$'s DM_WRITE operations have been executed, $T_i$ is complete. Thus, no $O_i[X]$ operation may follow $T_i$'s last $w_i[X]$ operation in $L^S$. 
Definition 3.2: Two synchronization logs, over a set of transactions $T$, are computationally equivalent if their associated execution logs are computationally equivalent.

Lemma 3.1, below, states that the above definition of computational equivalence between two synchronization logs indeed captures the fact that both synchronization logs represent the same computation.

Lemma 3.1: Let $L_1^\mathcal{F}$ and $L_2^\mathcal{F}$, two synchronization logs over $T$, be given with their respective associated execution logs, $L_1$ and $L_2$. If $L_1$ is computationally equivalent to $L_2$ then the following observations hold:

(a) each $r_1[X]$ operation, in both $L_1^\mathcal{F}$ and $L_2^\mathcal{F}$, returns the same value.

(b) each $p_1[X]$ operation, in both $L_1^\mathcal{F}$ and $L_2^\mathcal{F}$, computes the same value.

(c) each $w_1[X]$ operation, in both $L_1^\mathcal{F}$ and $L_2^\mathcal{F}$, produces the same value.
Proof: \( L_1^{\mathcal{F}} \) and \( L_2^{\mathcal{F}} \) represent two execution schedules over the same set of transactions.

Therefore, \( \forall i, \pi_i(L_1^{\mathcal{F}}) = \pi_i(L_2^{\mathcal{F}}) \) (the transaction scripts in both logs are identical) and \(|L_1^{\mathcal{F}}| = |L_2^{\mathcal{F}}| = m\), for some \( m > 0 \).

Let \( L_1^{\mathcal{F}} = 0_1, O_2, \ldots, O_m \). We will prove that (a)-(c) hold for each operation \( O_k \) in \( L_1^{\mathcal{F}} \), \( 1 \leq k \leq m \), by induction on \( k \) - the index of an operation.

basis \( k=1 \) : \( O_1 \) may stand for a \( r_1[X] \) or a \( p_i[X] \) operation (by rule(1) above), it cannot be a \( w_j[X] \) operation).

(a) \( O_1 = r_i[X] \). The \( r_i[X] \) operation in \( L_1^{\mathcal{F}} \) is preceded by no \( w_j[X] \) operation.

This must be also the case in \( L_2^{\mathcal{F}} \); otherwise, \( L_1 \) and \( L_2 \) would contain different conflicts and would not be computationally equivalent. Therefore, we may conclude that the \( r_i[X] \) operation in both \( L_1^{\mathcal{F}} \) and \( L_2^{\mathcal{F}} \) returns the same value.

(b) \( O_1 = p_i[X] \). The \( p_i[X] \) operation in \( L_1^{\mathcal{F}} \) is preceded by no \( r_i[X] \) operation.

This must be also the case in \( L_2^{\mathcal{F}} \), since \( \pi_i(L_1^{\mathcal{F}}) = \pi_i(L_2^{\mathcal{F}}) \). By rule(1) above, we conclude that the value computed by the \( p_i[X] \) operation, in both \( L_1^{\mathcal{F}} \) and \( L_2^{\mathcal{F}} \), is the same.

induction step : Assume the hypothesis of Lemma 3.1 holds for each operation \( O_k \) in \( L_1^{\mathcal{F}} \), \( k < m \), and prove it for \( O_{k+1} \).

There may be three cases.

(a) \( O_{k+1} = r_i[X] \). The \( r_i[X] \) operation, in both \( L_1^{\mathcal{F}} \) and \( L_2^{\mathcal{F}} \), returns the value which was produced by the same \( w_j[X] \) operation; otherwise, \( L_1 \) and \( L_2 \) would not be computationally equivalent (if no such \( w_j[X] \) operation exists then we are done). Since \( w_j[X] < r_i[X] \) in \( L_1^{\mathcal{F}} \), we have, by the induction assumption, that the value produced by the \( w_j[X] \) operation in both \( L_1^{\mathcal{F}} \) and \( L_2^{\mathcal{F}} \) is the same. We thus conclude the value returned by the \( r_i[X] \) operation in both logs is identical.

(b) \( O_{k+1} = p_i[X] \). The value computed by the \( p_i[X] \) operation, in \( L_1^{\mathcal{F}} \), is a
function of all values returned by $r_i[Y]$-operations, such that $r_i[Y] < p_i[X]$ in $L_f^1$. By the induction assumption these $r_i[Y]$ operations, in both $L_f^1$ and $L_f^2$, return the same values. Since the $p_i[X]$ operation, in both logs, depends on the same set of values, it therefore computes the same value.

(c) $0_k + 1 = w_i[X]$. By rule (1) above, the value produced by the $w_i[X]$ operation, in $L_f^1$, is the one computed by $T_i$'s last $p_i[X]$ operation where $p_i[X] < w_i[X]$ in $L_f^1$. By the induction assumption, this $p_i[X]$ operation, in both $L_f^1$ and $L_f^2$, computes the same value. By applying rule (1) above, also to $w_i[X]$ in $L_f^2$, we may conclude that the $w_i[X]$ operation in both logs produces the same value.

**Definition 3.3:** $L^S$ is serial if for every pair of transactions $T_i$ and $T_j$, either all of $T_i$'s operations precede all of $T_j$'s operations or vice versa.

A serial synchronization log models an execution schedule in which each transaction is executed to completion before the next is initiated.

**Definition 3.4:** $L^S$ is serializable if it is computationally equivalent to a serial synchronization log.

Figure 3.2 illustrates the notion of serializable logs.

**Theorem 3.1:** $L^S$ is serializable iff $L$ is serializable.

**Proof:**

(==>) This follows immediately from definitions 3.2 and 3.4.

(<=)

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

(2) Define the serial synchronization log $L_f^S$, derived from $L_1$, by augmenting each transaction $T_i$, in the appropriate places, with its set of PRE_WRITE operations, as defined by its script, $\pi_i(L_f^S)$.

(3) Since $L^S$ is computationally equivalent to $L_f^S$ (by means of $L$ and $L_1$) $L^S$ is serializable.

Theorem 3.1 establishes that in order to obtain serializable synchronization logs it is sufficient to maintain serializable execution logs. That is, the use of the PRE_WRITE
operation as a synchronization primitive, while maintaining serializable execution logs, does not affect the basic theory.

\[ L^S = p_1(Y)r_2(Z)r_1(X)p_2(Y)p_1(Z)r_2(Y)p_1(Z)w_2(Y)w_1(Y)w_1(Z). \]
\[ L^I = r_2(Z)p_2(Y)w_2(Y)p_1(Y)r_1(X)p_1(Z)w_1(Y)w_1(Z). \]
\[ L = r_2(Z)r_1(X)w_2(Y)w_1(Y)w_1(Z). \]
\[ L_1 = r_2(Z)w_2(Y)r_1(X)w_1(Y)w_1(Z). \]

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

**Figure 3.2: Serializable Logs.**

Definition 3.5 sets the stage for stating the Decomposition Theorem in our extended notation.

**Definition 3.5:** For every pair of transactions \( T_i, T_j \) and data item \( X \) define the binary relations \( \rightarrow_u \), where names for \( u \) are given below, as follows:

1. \( T_i \rightarrow_{rp} T_j \) if \( r_i[X] < p_j[X] \) in \( L^S \)
2. \( T_i \rightarrow_{pr} T_j \) if \( p_j[X] < r_i[X] < w_j[X] \) in \( L^S \)
3. \( T_i \rightarrow_{rpr} T_j \) if \( T_i \rightarrow_{rp} T_j \) or \( T_i \rightarrow_{pr} T_j \)
4. \( T_i \rightarrow_{wp} T_j \) if \( w_i[X] < p_j[X] \) in \( L^S \)
5. \( T_i \rightarrow_{pw} T_j \) if \( p_j[X] < w_i[X] < w_j[X] \) in \( L^S \)
6. \( T_i \rightarrow_{pp} T_j \) if \( w_i[X] < p_j[X] \) and \( w_i[X] < w_j[X] \) in \( L^S \)
7. \( T_i \rightarrow_{rw} T_j \) if \( r_i[X] < w_j[X] \) in \( L^S \)
8. \( T_i \rightarrow_{wr} T_j \) if \( w_i[X] < r_j[X] \) in \( L^S \)
9. \( T_i \rightarrow_{ww} T_j \) if \( w_i[X] < w_j[X] \) in \( L^S \)
The binary relations (8)-(13), are exactly those defined in [BERN81]. The remaining relations are new relations based on our use of the PRE_WRITE operation as a synchronization primitive. All of these relations can be easily derived from $L^S$.

Definition 3.6: Two operations from distinct transactions, conflict if both operate on the same data item and one of the operation is a DM_WRITE or a PRE_WRITE. A $u,v$ conflict denotes a conflict between two operations of types $u$ and $v$, where $T_i$'s operation of type $u$ precedes $T_i$'s operation of type $v$ in the execution order. Transaction designators will be omitted when the order of the conflicting transactions is clear from the text or when the information is irrelevant.

Note that the order in which two operations conflict does not necessarily impose the same serialization order on the conflicting transactions. For example, a $p_j r_i$ conflict, between two active transactions $T_i$ and $T_j$, indicates that $p_j[X] < r_i[X]$ in $L^S$ but that $T_i$ precedes $T_j$ in $\rightarrow$.

Theorem 3.2 (Decomposition): Let $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ be associated with an execution schedule $E$ modeled by $L^S$. $L^S$ is serializable if

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

Proof: (1) and (2) guarantee the serializability of the execution log [BERN81]. This fact and Theorem 3.1 guarantee the serializability of $L^S$. $\square$

Lemmas 3.2 and 3.3 will be used subsequently. They formally capture an important fact, that by using the PRE_WRITE operation the system is able to foresee future conflicts in $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ relations.

Lemma 3.2: $\rightarrow_{rpr} \supset \rightarrow_{rw}$

Proof: If $T_i \rightarrow_{rw} T_j$, then by Definition 3.5 there exists a data item $X$ such that $r_i[X] < w_j[X]$ in $L^S$. In $L^S$ each $w_j[X]$ is preceded by $p_j[X]$ i.e., $p_j[X] < w_j[X]$. Theorem 3.2 guarantees that the execution log is serializable. Hence $r_j[X] < w_j[X]$ in $L^S$. Therefore, $r_i[X] < w_j[X]$ in $L^S$.
(1) if \( r_i[X] < p_j[X] < w_j[X] \) then \( T_i \rightarrow_{rp} T_j \)
(2) if \( p_j[X] < r_i[X] < w_j[X] \) then \( T_i \rightarrow_{pr} T_j \)

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

**Corollary 3.1:** \( \rightarrow_{rpr} \cup \rightarrow_{wr} \supseteq \rightarrow_{rwr} \)

**Lemma 3.3:** \( \rightarrow_{pwp} \supseteq \rightarrow_{ww} \)

*Proof:* If \( T_i \rightarrow_{ww} T_j \) then by Definition 3.5 there exists a data item \( X \) such that \( w_i[X] < w_j[X] \) in \( L^S \).

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

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

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

**Lemma 3.4:** If \( T_i \rightarrow_{wwr} T_j \) then \( T_i \) committed before \( T_j \).

*Proof:* \( T_i \rightarrow_{wwr} T_j \) implies \( w_i[X] < w_j[X] \) or \( w_i[X] < r_j[X] \) in \( L^S \), i.e., \( T_j \) was active after \( T_i \) executed a \( DM\_WRITE \) operation. Since all \( DM\_WRITE \) operations of a transaction are executed *atomically* in its commit phase (i.e., with no interleaving operations of other transactions) we conclude that \( T_i \) committed before \( T_j \).

**Theorem 3.3:** \( \rightarrow_{wwr} \) is acyclic.

*Proof:* We show that there is no cycle \( T_i \rightarrow_{wwr} T_j \rightarrow_{wwr} T_k \rightarrow_{wwr} T_i \) (the proof for the case \( T_j = T_k \) is identical). Suppose such a cycle exists. Using Lemma 3.4 directly and transitively we have that \( T_i \) committed before \( T_k \) and \( T_k \) committed before \( T_i \), which is impossible since the execution of \( DM\_WRITE \)s is the last atomic action executed by a transaction. Hence, our assumption is false and \( \rightarrow_{wwr} \) is acyclic.

**Corollary 3.2:** Let \( \rightarrow \) be associated with an execution schedule \( E \) modeled by \( L^S \).

\( L^S \) is serializable if \( T_i \rightarrow_{rw} T_j \) implies \( T_i \) committed before \( T_j \).

*Proof:* If the above condition holds, using Lemma 3.4 we have:

- if \( T_i \rightarrow_{rw} T_j \) or \( T_i \rightarrow_{wr} T_j \) or \( T_i \rightarrow_{ww} T_j \) then \( T_i \) committed before \( T_j \).

Equivalently, \( T_i \rightarrow_{rw} T_j \) implies \( T_i \) committed before \( T_j \). Hence, using a similar
argument to that in theorem 3.3 we conclude that $\rightarrow$ is acyclic. It follows that $L$ is serializable and so $L^3$ is also serializable.

The atomic write phase a priori guarantees the acyclicity of $\rightarrow_{WW}$ and suggests that we can design concurrency control algorithms which can do without $WW$ synchronization. This is formalized by Corollary 3.2. One such algorithm is Serial Validation in [KUNG81] and its timestamp version in [CARE83]. In the sequel we present such a 2PL derivative.
CHAPTER 4

INTEGRATED CONCURRENCY CONTROL ALGORITHMS (ICCAs)

The Decomposition Theorem (Theorem 3.2) enables the designer of a concurrency control mechanism to address two sub-problems, synchronization of READ and WRITE requests (rw-synchronization) and synchronization of WRITE and WRITE requests (ww-synchronization) rather than the single more complex problem of concurrency control. An \textit{rw (ww) synchronization technique} is defined to be the procedure that guarantees correct \textit{rw (ww)} synchronization. The concurrency control mechanism must ensure that the use of a given \textit{rw} synchronization technique together with a given \textit{ww} synchronization technique will yield serializable execution orders.

The decomposition theorem serves as the foundation for the notion of an \textit{Integrated Concurrency Control Algorithm (ICCA)}. Rather than the use of one \textit{rw} technique together with one \textit{ww} technique as is the norm in working systems and the plethora of proposed algorithms, it is suggested to design an ICCA that would be composed of \textit{several \textit{rw} and several \textit{ww}} synchronization techniques all used \textit{concurrently}. The manner and the number of techniques to be used at a given instance is something that would be left up to the system designer. 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.

Section 4.1 describes in detail the \textit{TM} model. Section 4.2 presents a taxonomy of concurrency control synchronization techniques. Section 4.3 introduces the notion of a transaction's \textit{Wait For Set} as a means for controlling the concurrency level in the system. Some 2PL and Certification based synchronization techniques are described in section 4.4. Two ICCAs based on these techniques along with a proof of their correct-
ness are given in section 4.5. Finally, section 4.6 discusses some practical implementation issues concerning the management of SG, IT, and data items' wait queues.

4.1. The Transaction Manager Model

In this section we examine in more detail the actions taken by the TM upon receiving a transaction's request. The data structures involved as well as the operations performed on them are described.

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. SG is maintained acyclic at all times. A transaction causing a cycle in SG is restarted.

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 \(<I, \text{TID}>\):

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 transaction executes four types of requests: TRANS, READ, WRITE, and SNART. 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 stage; then a (possibly empty) synchronization stage followed by execution of the request. A transaction using 2PL must first obtain an appropriate indicator on a data item it accesses; in the waiting stage, it is forced to wait as long as there is some other active transaction which "holds" a conflicting indicator on the same data item. Edges are appropriately added to SG in order to reflect the precedence relation imposed by the blocking. A transaction using Certification is allowed to continue immediately to the synchronization stage.

In the synchronization stage, 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 atomic commit phase of the issuing transaction as described in the previous chapter. In this phase all the transaction's p-indicators are converted into c-indicators. Its r-indicators remain unchanged. 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.
4.2. Characterization of Synchronization Techniques

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

The idea behind ET synchronization is that the arrival order of READ and WRITE operations is important and that the synchronization algorithm must take that order into account. In a 2PL algorithm, this order is determined by the order in which transactions obtain their indicators.

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

Using these two characterization parameters, we can obtain four synchronization techniques for each of the two types of synchronization. For example, we may use certification synchronization at transaction execution time or transaction commit time for either the rw or ww synchronizations. The following notation is used to represent the possible techniques. Each technique's name will be composed of three components: synchronization strategy (2PL or CERT), synchronization time (ET or CT), and synchronization type (rw or ww). For example, CERT-CT-ww is a technique that uses the certification approach to achieve ww-synchronization at transaction commit time. Figure 4.1 summarizes all eight possible techniques. In Section 4.3 we give a description
of the actions taken by five of the techniques. As explained subsequently, the others are of no interest. Before describing the techniques we introduce, in the next section, the notion of a transaction’s Wait For Set.

---

<table>
<thead>
<tr>
<th>Srw Synchronization Techniques</th>
<th>Sww Synchronization Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PL-ET-rw</td>
<td>2PL-ET-ww</td>
</tr>
<tr>
<td>2PL-CT-rw</td>
<td>2PL-CT-ww</td>
</tr>
<tr>
<td>CERT-ET-rw</td>
<td>CERT-ET-ww</td>
</tr>
<tr>
<td>CERT-CT-rw</td>
<td>CERT-CT-ww</td>
</tr>
</tbody>
</table>

**Figure 4.1: The Synchronization Techniques.**

---

### 4.3. The Wait For Set of a Transaction (WFS)

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

Definition 4.1, below, formalizes this notion.

---

1 This part of the design was influenced to a great degree by Robinson’s notion of separation of policy from correctness in concurrency control design [ROB182].
Definition 4.1: The *Wait For Set (WFS)*\(^1\) of a transaction \(T_i\) for \(rw(ww)\) synchronization, \(WFS_{\text{rw}(ww)}(T_i)\), is the set of transactions for which \(T_i\) is required to wait in its request waiting stage.

For a transaction \(T_i\) using CERT synchronization strategy for \(rw(ww)\) synchronization, \(WFS_{\text{rw}(ww)}(T_i)\) is empty. For a transaction \(T_i\) using 2PL strategy we define the following WFS definition parameters for \(rw(ww)\) synchronization: *strict-2PL-rw(ww)* denotes that \(WFS_{\text{rw}(ww)}(T_i) = T\); *standard-2PL-rw(ww)* denotes that \(WFS_{\text{rw}(ww)}(T_i) = \{ T_j | T_j \text{ is using 2PL for } rw(ww) \text{ synchronization} \} \). *user-def* denotes any user defined set of transactions and *null* means that \(WFS_{\text{rw}(ww)}(T_i) = \emptyset\).

The *Wait For Set* of a transaction \(T_i\) is implementation dependent and may be defined by means of synchronization techniques, transaction classes, and even as a function of the concurrency level desired.

1.1. The Synchronization Techniques

Five of the synchronization techniques, defined in Figure 4.1, are presented in this section. The three other techniques, 2PL-CT-rw, 2PL-CT-ww and CERT-ET-ww were ignored. 2PL-CT-rw is meaningless as it forces a transaction to wait for an r-indicator after the corresponding DM_READ operation has already been executed. Likewise, CERT-ET-ww is meaningless since there is no way to a priori determine the serialization order, upon pp conflicts between active transactions, 2PL-CT-ww requires the inclusion of a waiting stage in the commit phase, which is unacceptable.

The differences among the synchronization techniques presented have to do with the manner in which they operate in the Execution and Commit phases. What is common to all the techniques is that they all update SG and IT; if adding an edge to SG

---

\(^1\) WFS determines, during system operation, the set of transactions for which each transaction in \(T\) is required to wait in its request waiting stage; in the general case, this set may change from one request to another (see for example the Optimized Wait For Set (OWFS) defined in chapter 5). WFS may be formally looked upon as function \(WFS: T \rightarrow 2^T\) (the function is "determined" once all transactions have completed execution).
causes a cycle then the transaction that caused the cycle creation is restarted, all its indicators are released and all information concerning it is removed from SG.

To facilitate the description of the various techniques below, we utilize a compatibility-action matrix (c-a matrix). An entry in the matrix indicates how a synchronization technique interprets the values of existing indicators on a data item when processing a transaction request to access that item. In addition, the entry specifies what edges are added to SG. The c-a matrices for five of techniques described below are shown in Figure 4.2 and Figure 4.3. Note that the CT-synchronization techniques include no waiting stage.

Recall that regardless of what synchronization technique $T_i$ is using, execution of a READ($X$) (resp. WRITE($X$)) request includes appending the appropriate r-indicator (resp. p-indicator) to $X$ and issuing an $r_i[X]$ (resp. $p_i[X]$) operation. At the commit phase of $T_i$ following the CTS, all its p-indicators are converted into c-indicators and the appropriate DM_WRITE operations are issued on its behalf.

4.4.1. RW Synchronization Techniques.

RW synchronization techniques synchronize conflicting DM_READ and PRE_WRITE operations as well as conflicting DM_READ and DM_WRITE operations.

2PL-ET-rw

2PL-ET-rw generalizes the rw synchronization of conventional 2PL; each READ or a WRITE request undergoes a (possibly empty) waiting stage followed by a (possibly empty) synchronization stage. The c-a matrix in Figure 4.2 for 2PL-ET-rw describes the actions taken in each of these stages.

Assume $T_i$ is using 2PL-ET-rw and let $T_j$ be any active transaction in WFS$_{rw}(T_i)$. $T_i$'s $r_jp_i$ and $p_jr_i$ conflicts with $T_j$ are synchronized using blocking in the waiting stage of a request. Upon issuing a READ($X$) (resp. WRITE($X$)) request, $T_i$ must first obtain an
r-indicator (resp. a p-indicator) on X and it can not proceed to the synchronization stage as long as Tj owns a conflicting indicator on X. The edge (Tj, Ti) is added to SG in order to reflect the precedence relation imposed by the blocking. Note that this action of blocking and edge addition to SG is continuous; i.e., if during T1's waiting stage some new transaction Tk in WFSrw(T1) obtains a conflicting indicator on X then T1 will be forced to wait also for Tk and an edge (Tk, T1) will be added to SG.

In the synchronization stage, T1's rjpj, pjr1 and wjr1 conflicts with other transactions are synchronized, using a pure restart strategy, by adding the appropriate edges to SG in order to reflect the precedence relation imposed by execution of the request. Upon a READ(X) request an edge (T1, Tj) is added to SG for each active transaction owning a p-indicator on X, while an edge (Tj, T1) is added to SG for each committed transaction owning a c-indicator on X. Upon a WRITE(X) request, an edge (Tj, T1) is added to SG for each transaction Tj owning an r-indicator on X.

CERT-ET-rw

Assume T1 is using CERT-ET-rw and is requesting to access some data item X. T1's waiting stage is null; it appends the appropriate indicator to X and immediately proceeds to the synchronization stage. During the synchronization stage, T1's rjpj, pjr1 and wjr1 conflicts on X with other transactions, are resolved using the c-a matrix for CERT-ET-rw shown in Figure 4.2. This is accomplished by adding the appropriate edges to SG in order to reflect the precedence relation imposed by execution of the request; the details of the actions taken are exactly those (described above) for a transaction using 2PL-ET-rw in its synchronization stage.

CERT-CT-rw

Note that the direction of the edges added to SG due to a transaction waiting is opposite to the one usually used (see Figure 2.3). This makes the waiting relations between transactions consistent with \( \rightarrow \).
In this technique rw synchronization is performed at commit time, by resolving rw and wr conflicts using a procedure similar to the one used in [KUNG81]. Assume Ti is using CERT-CT-rw. At the execution phase, Ti's requests are allowed to run unhindered; the appropriate indicators are appended to IT but no edges are added to SG. At commit time, edges are added to SG as shown in the c-a matrix for this technique (Figure 4.2). An edge(Tj,Ti) is added to SG for each transaction Tj such that, (1) Tj owns a c-indicator on X and X ∈ Readset(Ti) and (2) Tj owns an r-indicator on X and X ∈ Writeset(Ti). In other words, the committing transaction Ti will follow all conflicting transactions committed earlier and all conflicting active readers. Note that if during Ti’s commit phase, an edge (Tj,Ti) for an active reader Tj using CERT-CT-rw is added to SG, then it is guaranteed that Tj will be restarted during its own commit phase (and as an optimization it may be restarted earlier, i.e. during Ti’s commit phase).

4.4.2. WW Synchronization Techniques

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

2PL-ET-ww

2PL-ET-ww generalizes the ww synchronization of conventional 2PL; each WRITE request undergoes a (possibly empty) waiting stage followed by a (possibly empty) synchronization stage. The meaning of the c-a matrix entries in 2PL-ET-ww (see Figure 4.3) is as in the matrix for 2PL-ET-rw.

Assume Ti is using 2PL-ET-ww and let Tj be an active transaction in WFS_{ww}(Ti). Ti’s p_j pi conflicts with Tj are synchronized using blocking in the waiting stage of a WRITE request. Upon issuing a WRITE(X) request, Ti must first obtain a p-indicator on X and can not proceed to the synchronization stage as long as Tj owns a p-indicator on X.
The edge \((T_j, T_i)\) is added to SG in order to reflect the precedence relation imposed by the blocking (as in 2PL-ET-rw this blocking and the edge addition is continuous).

\(T_i\)'s \(wp\) conflicts with committed transactions are synchronized during the request's synchronization stage, by adding to SG an edge \((T_j, T_i)\) for each transaction \(T_j\) owning a \(c\)-indicator on \(X\).

**CERT-CT-ww**

In this technique \(ww\) synchronization is performed at commit time by resolving \(ww\) and \(pw\) conflicts. Assume \(T_i\) is using CERT-CT-ww. At the execution phase \(T_i\)'s \(WRITE\) requests are allowed to run unhindered; its \(p\)-indicators are appended to IT but no edges are added to SG. The \(c\)-a matrix for this technique (see Figure 4.3) applies to the commit phase of the transaction. Let \(X \in \text{Writeset}(T_i)\). An edge \((T_j, T_i)\) is added to SG for each committed transaction \(T_j\) owning a \(c\)-indicator on \(X\). An edge \((T_i, T_j)\) is added to SG for each active transaction \(T_j\) owning a \(p\)-indicator on \(X\) and using 2PL-ET-ww. In other words, the committing transaction \(T_i\) follows all transactions committed earlier and indicates that active writers will follow it (synchronization with active writers is required only when they use the 2PL-ET-ww synchronization technique).
### Legend

- $T_i$: issuing the request; $T_j$: owning an indicator
- $+$: request granted
- $-$: request not granted on a conflict with an active transaction
- $\cdot$: irrelevant
- $T_i \rightarrow T_j$ add to SG edge $(T_i,T_j)$
- $T_j \rightarrow T_i$ add to SG edge $(T_j,T_i)$

### Table 4.2: C-a Matrices for the RW Synchronization Techniques

**Table 4.2: C-a Matrices for the RW Synchronization Techniques**

<table>
<thead>
<tr>
<th>2PL-ET-rw</th>
<th>Waiting Stage</th>
<th>Synchronization Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_j \in WFSrw(T_i)$</td>
<td>$T_j \rightarrow T_j$</td>
<td>$\forall T_j$</td>
</tr>
<tr>
<td>$p$</td>
<td>$c$</td>
<td>$p$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>$+$</td>
<td>$T_i \rightarrow T_j$</td>
</tr>
<tr>
<td>$T_j \rightarrow T_i$</td>
<td>$+$</td>
<td>$T_j \rightarrow T_i$</td>
</tr>
</tbody>
</table>

**Table 4.2: C-a Matrices for the RW Synchronization Techniques**

<table>
<thead>
<tr>
<th>CERT-ET-rw</th>
<th>Waiting Stage</th>
<th>Synchronization Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>$NULL$</td>
<td>$\forall T_j$</td>
</tr>
<tr>
<td>$p$</td>
<td>$+$</td>
<td>$T_i \rightarrow T_j$</td>
</tr>
<tr>
<td>$T_j \rightarrow T_i$</td>
<td>$+$</td>
<td>$T_j \rightarrow T_i$</td>
</tr>
</tbody>
</table>

**Table 4.2: C-a Matrices for the RW Synchronization Techniques**

<table>
<thead>
<tr>
<th>CERT-CT-rw</th>
<th>Waiting Stage</th>
<th>Synchronization Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>$NULL$</td>
<td>$\forall T_j$</td>
</tr>
<tr>
<td>$c$</td>
<td>$+$</td>
<td>$T_j \rightarrow T_i$</td>
</tr>
<tr>
<td>$T_j \rightarrow T_i$</td>
<td>$+$</td>
<td>$T_j \rightarrow T_i$</td>
</tr>
</tbody>
</table>

**Figure 4.2: C-a Matrices for the RW Synchronization Techniques**
### LEGEND

- $T_i$: Issuing the request; $T_j$: Owning an indicator
- $+$: Request granted
- $-$: Request not granted on a conflict with an active transaction
- $*$: Irrelevant
- $T_i \rightarrow T_j$: Add to SG edge $(T_i, T_j)$
- $T_j \rightarrow T_i$: Add to SG edge $(T_j, T_i)$

#### Figure 4.3: C-ad Matrices for the WW Synchronization Techniques
4.4.3. Formalization of the Synchronization Techniques

The following Lemma characterizes the correspondence between the relations defined in chapter 3 and edges in SG.

Lemma 4.1: For each transaction $T_i$ (regardless of which synchronization technique $T_j$ is using) the relations $\rightarrow_{rp}$, $\rightarrow_{pr}$, $\rightarrow_{rw}$, $\rightarrow_{wr}$, $\rightarrow_{pw}$, $\rightarrow_{wp}$, and $\rightarrow_{ww}$ derived from $L_S$ and $SG$ which are maintained by the synchronization techniques satisfy the following conditions:

(1) If $T_i$ is using 2PL-ET-rw then:
   (a) if $T_i \rightarrow_{wr} T_j$ then $(T_i, T_j) \in SG$ (synchronization stage $rc$ entry).
   (b) if $T_i \rightarrow_{rp} T_j$ then $(T_i, T_j) \in SG$ (synchronization stage $pr$ entry).
   (c) if $T_j \in WFS_{rw}(T_i)$ then $T_i \rightarrow_{pr} T_j \not\rightarrow_{pr}$ (in the waiting stage $r_i$ operation would have been forced to wait).
   (d) if $T_j \not\rightarrow_{w} WFS_{rw}(T_i)$ then $T_i \rightarrow_{pr} T_j$ then $(T_i, T_j) \in SG$ (edge added in the synchronization stage $rp$ entry).

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

(3) If $T_i$ is using 2PL-ET-ww then:
   (a) if $T_i \rightarrow_{wp} T_j$ then $(T_i, T_j) \in SG$ (synchronization stage $pc$ entry).
   (b) if $T_j \in WFS_{ww}(T_i)$ and $T_i \in WFS_{ww}(T_j)$ then $T_i \rightarrow_{pw} T_j \not\rightarrow_{pw}$. (by waiting stage $pp$ entry, if $p_j[X] < p_i[X]$ then $p_i[X]$ would have been forced to wait; if $p_i[X] < p_j[X]$ then $p_j[X]$ would have been forced to wait).

(4) If $T_i$ is using CERT-CT-rw then:
   (a) if $T_j \rightarrow_{rw} T_i$ then $(T_j, T_i) \in SG$ (synchronization stage $cr$ entry).
   (b) if $T_j \rightarrow_{wr} T_i$ then $(T_j, T_i) \in SG$ (synchronization stage $rc$ entry).
(5) If $T_i$ is using CERT-CT-ww then:

(a) if $T_j \rightarrow\text{ww} T_i$ then $(T_j, T_i) \in SG$ (synchronization stage cc entry).

(b) if $T_j$ is using $2\text{PL-ET-ww}$ then if $T_i \rightarrow\text{pw} T_j$ then $(T_i, T_j) \in SG$ (synchronization stage cp entry).

4.5. 2PL and Certification based ICCAs

An Integrated Concurrency Control Algorithm (ICCA) consists of a set of rw synchronization techniques, a set of ww synchronization techniques and their associated WFS definition parameters. By proper selection of a single rw technique, a single ww technique and appropriate WFS definition parameters, a synchronization policy is assigned to a transaction. The ICCA mechanism may employ several synchronization policies concurrently, each for a given class of transactions and still ensure that conflicts between transactions from different classes are correctly resolved. The major likely advantage of the ICCA idea is in matching transactions (or in general transaction classes) with appropriate synchronization policies as dictated by performance objectives.

An ICCA instance models the selection made by the ICCA mechanism for allocating synchronization policies to transactions in the course of their execution. In reality, allocation of synchronization policies to transactions may be done either statically, by using some pre determined criteria (as in Example 4.1 below) or dynamically, by using an initial static allocation which may change according to system state parameters (see for example, the DYNAMIC algorithm in sect. 7.4). In case of dynamic allocation, the selection of synchronization policies for transactions is made during execution; it depends on the execution order and may be looked upon as an ICCA instance only after all transactions have completed execution.

The notions of an ICCA, an ICCA instance and their correctness are formally defined in this section. Some ICCAs based on the synchronization techniques presented
in the previous section are introduced together with a proof of their correctness.

Before proceeding with the discussion let us consider for a moment the deadlock problem. Consider the operation of a concurrency control algorithm employing 2PL-ET-rw and 2PL-ET-ww synchronization techniques. Whenever a transaction \( T_i \) waits for a transaction \( T_j \), upon \( r_j p_i \), \( p_j r_i \) or \( p_j p_i \) conflicts, an edge \((T_j, T_i)\) is added to SG. Since SG is maintained acyclic at all times no deadlock can result. Thus we conclude that any ICCA employing 2PL-ET-rw and 2PL-ET-ww synchronization techniques is deadlock-free.

**Definition 4.2:** An ICCA is a quadruple \((S_{rw}, WFS_{drw}, S_{ww}, WFS_{dww})\), where
- \( S_{rw} \) is a non-empty set of rw-synchronization techniques,
- \( WFS_{drw} \) is a non-empty set of parameters which denotes WFSs for rw-synchronization,
- \( S_{ww} \) is a non-empty set of techniques for \( S_{ww} \) techniques, and
- \( WFS_{dww} \) is a non-empty set of parameters which denotes WFSs for \( S_{ww} \) synchronization.

Let \( T = \{T_1, ..., T_n\} \) be an arbitrary set of transactions. An ICCA instance is a pair \((ICCA, F)\), where - ICCA is a quadruple \((S_{rw}, WFS_{drw}, S_{ww}, WFS_{dww})\) as defined above, and - \( F \) is a function \( F: T \rightarrow S_{rw} \times WFS_{drw} \times S_{ww} \times WFS_{dww} \).

\( F \) determines the synchronization method by which each transaction in \( T \) is synchronized. \( F \) maps a transaction \( T_i \) to one \( S_{rw} \) technique with \( WFS_{rw}(T_i) \) defined by a \( WFS_{drw} \) parameter and to one \( S_{ww} \) technique with \( WFS_{ww}(T_i) \) defined by a \( WFS_{dww} \) parameter. An asterisk replacing the \( rw(ww) \) WFS definition parameter for \( T_i \) means that we do not care about the value of \( WFS_{rw(ww)}(T_i) \) (\( T_i \) may be using certification for \( rw(ww) \) synchronization where \( WFS_{rw(ww)}(T_i) \) is irrelevant as there is no waiting stage).
Example 4.1 below illustrates the notions of an ICCA and an ICCA instance.

Example 4.1:

\[
\text{ICCA}_1 = ( \{ 2\text{PL-ET-rw}, \text{CERT-ET-rw} \}, \{ \text{standard-2PL-rw} \} \\
\{ \text{CERT-CT-ww} \}, \{ \text{null} \})
\]

An ICCA\(_1\)'s instance = (ICC\(_1\), F\(_1\) : \text{if } T_i \text{ is a reader, then}
\[
(\text{CERT-ET-rw}, *, \text{CERT-CT-ww}, *)
\]
\[\text{else} \]
\[
(2\text{PL-ET-rw}, \text{standard-2PL-rw}, \text{CERT-CT-ww}, *)
\]

Under F\(_1\), readers use certification (SG checking); writers use 2PL for rw synchronization and certification for ww synchronization and writers do not wait for readers (by definition of standard-2PL).

**Definition 4.3:** An ICCA instance = (ICCA, F) is correct, if any execution schedule of T, in which T\(_i\) ∈ T is synchronized according to F(T\(_i\)), produces a serializable synchronization log. An ICCA is correct; if all of its instances are correct.

**Corollary 4.1:** An ICCA employing a subset of the techniques or a subset of the WFS definition parameters used by a correct ICCA is also correct.

**Proof:** Let A\(_1\) be a correct ICCA and let A\(_2\) be an ICCA employing a subset of the techniques or a subset of the WFS definition parameters used by A\(_1\). The set of all A\(_2\)'s instances is a subset of the set of all A\(_1\)'s instances. Since all A\(_1\)'s instances are correct we conclude that all A\(_2\)'s instances are correct. 

**Definition 4.4:** Define the following ICCAs and some possible mapping functions.

Let WFS\(_d\)-rw = \{ strict-2PL-rw, standard-2PL-rw, user-def, null \} and

WFS\(_d\)-ww = \{ strict-2PL-ww, standard-2PL-ww \}.

\[
\text{ICCA}_2 = ( \{ 2\text{PL-ET-rw}, \text{CERT-ET-rw} \}, \text{WFS\(_d\)-rw}, \\
\{ 2\text{PL-ET-ww}, \text{CERT-CT-ww} \}, \text{WFS\(_d\)-ww} )
\]
ICCA\textsubscript{3} = ( \{ CERT-CT-rw \}, \{ null \}, \\
\{ 2PL-ET-ww, CERT-CT-ww \}, WFSd-ww )

\[ F_{2PL} : \forall T_i \in T. \ F_{2PL}(T_i) = (2PL-ET-rw, \text{standard-2PL-rw}, 2PL-ET-ww, \text{standard-2PL-ww}) \]

\[ F_{CERT} : \forall T_i \in T. \ F_{CERT}(T_i) = (CERT-ET-rw, *, CERT-CT-ww, *) \]

\[ F_{2} : \forall T_i \in T. \ F_{2}(T_i) = (CERT-ET-rw, *, 2PL-ET-ww, \text{standard-2PL-ww}) \]

\[ F_{SV} : \forall T_i \in T. \ F_{SV}(T_i) = (CERT-CT-rw, *, CERT-CT-ww, *) \]

It will be shown subsequently that ICCA\textsubscript{2} and ICCA\textsubscript{3} are correct. Consider ICCA\textsubscript{1}, introduced in example 4.1. The set of ICCA\textsubscript{1}'s instances is a proper subset of all of ICCA\textsubscript{2}'s instances. Therefore, if ICCA\textsubscript{2} is shown to be correct, then ICCA\textsubscript{1} is also correct. ICCA\textsubscript{2} is important because it can model a transaction system employing a synchronization policy varying from strict locking to full certification. ICCA\textsubscript{2}'s instance defined by \( F_{2PL} \) represents the "standard" 2PL algorithm. ICCA\textsubscript{2}'s instance defined by \( F_{2} \) is reminiscent of the algorithm defined in [BAYE80]. ICCA\textsubscript{2}'s instance defined by \( F_{CERT} \) is an SG checking algorithm. ICCA\textsubscript{3}'s instance defined by \( F_{SV} \) is similar to the Serial Validation algorithm in [KUNG91].

**Correctness**

How do we prove the correctness of a given ICCA? We need to show that for any instance of the given ICCA and for any execution schedule of transactions using this instance, a serializable synchronization log is produced. In other words, we need to select an arbitrary instance of the given ICCA and show that each constraint placed on the total execution order by an operation is represented in SG regardless of the transactions involved, and more importantly, regardless of the synchronization method selected to synchronize the conflicting transactions. In the following we show that ICCA\textsubscript{2} and ICCA\textsubscript{3} defined above indeed satisfy this condition.

**Convention:** In the following Lemmas and Theorems, and throughout this work, \( rw \) and
WW synchronizations are treated separately, therefore when specifying the mapping of a transaction we will omit, for reasons of clarity, the ww part of the mapping when rw synchronization is treated and the rw part of the mapping when ww synchronization is treated.

Lemma 4.2: Let A=(ICCA₂, Fa) be an instance of ICCA₂. Let E be an execution schedule of T using A and modeled by L⁹ with its associated $\rightarrow$ relation. If $T_j \rightarrow T_i$, then $(T_j, T_i) \in SG$.

Proof: Let $wfs-rw_1, 2 \in WFSd-rw$ and let $wfs-ww_1, 2 \in WFSd-ww$.

(a) First we show that if $T_j \rightarrow_{rw} T_i$ then $(T_j, T_i) \in SG$ (*).

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

(1) Let $Fa(T_i) = (2PL\rightarrow ET-rw, wfs-rw_1), Fa(T_j) = (2PL\rightarrow ET-rw, wfs-rw_2)$

By Lemma 4.1.1.a if $T_j \rightarrow_{wr} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.1.b if $T_j \rightarrow_{rp} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.1, substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we have:

If $T_i \in WFS_{rw}(T_j)$ then by Lemma 4.1.1.c $T_j \rightarrow_{pr} T_i$ is not possible, hence vacuously: if $T_j \rightarrow_{pr} T_i$ then $(T_j, T_i) \in SG$.

If $T_i \not\in WFS_{rw}(T_j)$ then by Lemma 4.1.1.d if $T_j \rightarrow_{pr} T_i$ then $(T_j, T_i) \in SG$.

(2) Let $Fa(T_i) = (2PL\rightarrow ET-rw, wfs-rw_1), Fa(T_j) = (CERT\rightarrow ET-rw, *)$

By Lemma 4.1.1.a if $T_j \rightarrow_{wr} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.1.b if $T_j \rightarrow_{rp} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.2.a substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we get if $T_j \rightarrow_{pr} T_i$ then $(T_j, T_i) \in SG$.

(3) Let $Fa(T_i) = (CERT\rightarrow ET-rw, *), Fa(T_j) = (2PL\rightarrow ET-rw, wfs-rw_2)$

By Lemma 4.1.2.c if $T_j \rightarrow_{wr} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.2.b if $T_j \rightarrow_{rp} T_i$ then $(T_j, T_i) \in SG$.

By Lemma 4.1.1, substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we have:
if $T_i \in \text{WFS}_{rw}(T_j)$ then by Lemma 4.1.1.c $T_j \not\rightarrow_{pr} T_i$ is not possible, hence we can write, if $T_j \not\rightarrow_{pr} T_i$ then $(T_j, T_i) \in \text{SG}$.

if $T_i \not\in \text{WFS}_{rw}(T_j)$ then by Lemma 4.1.1.d if $T_j \not\rightarrow_{pr} T_i$ then $(T_j, T_i) \in \text{SG}$.

(4) Let $F_a(T_i) = (\text{CERT-ET-rw}, \ast)$, $F_a(T_j) = (\text{CERT-ET-rw}, \ast)$

By Lemma 4.1.2.c if $T_j \not\rightarrow_{wr} T_i$ then $(T_j, T_i) \in \text{SG}$.
By Lemma 4.1.2.b if $T_j \not\rightarrow_{rp} T_i$ then $(T_j, T_i) \in \text{SG}$.
By Lemma 4.1.2.a substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we get that if $T_j \not\rightarrow_{pr} T_i$ then $(T_j, T_i) \in \text{SG}$.

In all possible mappings of $T_i$ and $T_j$ by $F_a$ we have that if $T_j \not\rightarrow_{wp} T_i$ or $T_j \not\rightarrow_{wr} T_i$ then $(T_j, T_i) \in \text{SG}$. By Corollary 3.1 we conclude $(\ast)$.

(b) Next we show that if $T_j \not\rightarrow_{ww} T_i$ then $(T_j, T_i) \in \text{SG}$ $(\ast\ast)$

$F_a$ may map $T_i$ and $T_j$ to 2PL-ET-ww and CERT-CT-ww in 4 ways:

(5) Let $F_a(T_i) = (\text{2PL-ET-ww}, \text{wfs-ww}_1)$, $F_a(T_j) = (\text{2PL-ET-ww}, \text{wfs-ww}_2)$

By Lemma 4.1.3.a if $T_j \not\rightarrow_{wp} T_i$ then $(T_j, T_i) \in \text{SG}$.

$\text{wfs-ww}_1, 2$ (i.e. strict-2PL or standard-2PL) imply that $T_i \in \text{WFS}_{ww}(T_j)$ and that $T_j \in \text{WFS}_{ww}(T_i)$.

By Lemma 4.1.3.b substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we have that $T_j \not\rightarrow_{pw} T_i$ is not possible, hence vacuously: if $T_j \not\rightarrow_{pw} T_i$ then $(T_j, T_i) \in \text{SG}$.

(6) Let $F_a(T_i) = (\text{2PL-ET-ww}, \text{wfs-ww}_1)$, $F_a(T_j) = (\text{CERT-CT-ww}, \ast)$

By Lemma 4.1.3.a if $T_j \not\rightarrow_{wp} T_i$ then $(T_j, T_i) \in \text{SG}$.
By Lemma 4.1.5.b substituting $T_j$ for $T_i$ and $T_i$ for $T_j$ we get if $T_j \not\rightarrow_{pw} T_i$ then $(T_j, T_i) \in \text{SG}$.

(7) Let $F_a(T_i) = (\text{CERT-CT-ww}, \ast)$, $F_a(T_j) = (\text{2PL-ET-ww}, \text{wfs-ww}_2)$

By Lemma 4.1.5.a if $T_j \not\rightarrow_{ww} T_i$ then $(T_j, T_i) \in \text{SG}$. 

---

Technion - Computer Science Department - Technical Report CS0426 - 1986
(8) Let $F_a(T_i) = (\text{CERT-CT-rw. *})$. $F_a(T_j) = (\text{CERT-CT-rw. *})$

By Lemma 4.1.5.a if $T_j \rightarrow_{\text{ww}} T_i$ then $(T_j, T_i) \in \text{SG}$.

In all possible mappings of $T_i$ and $T_j$ by $F_a$ we have that if $T_j \rightarrow_{\text{pwp}} T_i$ or $T_j \rightarrow_{\text{ww}} T_i$ then $(T_j, T_i) \in \text{SG}$. By Lemma 3.3 we conclude (**).

From (a) and (b) we have if $T_j \rightarrow_{\text{rwr}} T_i$ or $T_j \rightarrow_{\text{ww}} T_i$ then $(T_j, T_i) \in \text{SG}$. ■

**Theorem 4.1:** ICCA$^2$ is correct.

**Proof:** Let $A=(\text{ICCA}_2, F_a)$ be an instance of ICCA$^2$. Let $E$ be an execution schedule of $T$ using $A$ and modeled by $L^5$ with its associated $\rightarrow$ relation. By Lemma 4.2 if $T_j \rightarrow T_i$ then $(T_j, T_i) \in \text{SG}$. Since SG is maintained acyclic at all times it follows that $\rightarrow$ is acyclic. By the fact that $\rightarrow$ is acyclic and Theorem 3.2 we conclude that $L^5$ is serializable. ■

**Lemma 4.3.** Let $A=(\text{ICCA}_3, F_a)$ be an instance of ICCA$_3$. Let $E$ be an execution schedule of $T$ using $A$ and modeled by $L^5$ with its associated $\rightarrow$ relation. If $T_j \rightarrow T_i$ then $(T_j, T_i) \in \text{SG}$.

**Proof:** (a) First we show that if $T_j \rightarrow_{\text{rwr}} T_i$ then $(T_j, T_i) \in \text{SG}.$

$F_a(T_i) = F_a(T_j) = (\text{CERT-CT-rw. *})$

(1) by Lemma 4.1.4.b if $T_j \rightarrow_{\text{wr}} T_i$ then $(T_j, T_i) \in \text{SG}$.

(2) by Lemma 4.1.4.a if $T_j \rightarrow_{\text{rw}} T_i$ then $(T_j, T_i) \in \text{SG}$.

(b) Next by argument (b) of Lemma 4.2 we have: if $T_j \rightarrow_{\text{ww}} T_i$ then $(T_j, T_i) \in \text{SG}$.

From (a) and (b) we have if $T_j \rightarrow_{\text{rwr}} T_i$ or $T_j \rightarrow_{\text{ww}} T_i$ then $(T_j, T_i) \in \text{SG}.$ ■

**Theorem 4.2:** ICCA$_3$ is correct.

**Proof:** By the argument of Theorem 4.1 and using Lemma 4.3, we conclude that any instance of ICCA$_3$ produces serializable synchronization logs. ■
4.6. Practical Considerations

There are several implementation issues that need to be addressed: efficient implementation of atomic commits, implementation of the private workspaces by the TM, management of SG and IT, management of the wait queue associated with a data item, etc. In this section we consider two of these issues which are related to efficient implementation of an ICCA: management of SG and IT and the wait queues (atomic commit is discussed in chapter 6).

4.6.1. Management of SG and IT

Let \( T \) be a set of transactions using some correct ICCA. SG and IT should not grow without bound. An SG node, corresponding to a committed transaction \( T_i \), may be deleted when it becomes a root node, i.e., it has no incoming edges. In this case there is no transaction \( T_j \) such that \( T_j \rightarrow T_i \) (since all conflicts in \( \rightarrow \) are resolved through SG) and \( T_i \) cannot participate in a cycle in \( \rightarrow \). At that point in time \( T_i \)'s indicators (r-indicators and c-indicators) may be removed from IT.

**Lemma 4.4:** If \( T_i \) is an SG root node of a committed transaction \( T_i \), then \( T_i \) may be deleted from SG with its corresponding IT indicators without affecting database consistency.

The condition that only root nodes may be removed from SG may cause SG to contain, nodes or chain of nodes, of committed transactions which can not yet be removed. The process of removing those nodes is recursive, as the following example shows. Consider Figure 4.4. Suppose that \( T_{11} \) is a root node of an active transaction. When \( T_{11} \) commits we are able to remove node \( T_{11} \). As a result \( T_{22} \) becomes a root node, and since \( T_{22} \) has already committed it can also be removed. Later, when \( T_{10} \) commits and becomes a root node we will be able to remove, recursively, nodes \( T_{21} \) and \( T_{31} \).
Algorithm 4.1 implements the procedure discussed above:

Algorithm 4.1: Removal of a committed transaction node from SG.

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

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

Method:

(1) Define the following sets of nodes in SG:

- $\text{FOLLOWING}(T_i) = \{ T_j \mid (T_i, T_j) \in SG \}$
- $\text{PRECEDING}(T_i) = \{ T_j \mid (T_j, T_i) \in SG \}$

(2) Execute the following recursive procedure:

\begin{verbatim}
procedure DeleteTransaction($T_i$)
begin
if $T_i$ committed and $\text{PRECEDING}(T_i) = \emptyset$ then
end
\end{verbatim}
begin
for each \( T_j \) in \text{FOLLOWING}(T_i) \) do
begin
Remove edge \((T_i, T_j)\);
DeleteTransaction\((T_j)\);
end,
Remove \( T_i \) from \( SG \);
Remove \( T_i \)'s IT indicators;
end;
end;

4.6.2. Management of a data item WAIT-QUEUE

The management of a data item WAIT-QUEUE is an implementation policy decision which does not affect basic correctness of an ICCA. An ICCA correctness is based on synchronization with \textit{actual executed operations}, thus \textit{FIFO} (First In First Out) or any other queue management policy is acceptable.

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

For example:

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

\begin{itemize}
\item \( T_1 \): TRANS READ(X); WRITE(X); WRITE(Y); SNART
\item \( T_2 \): TRANS READ(Y); WRITE(Y); WRITE(W); SNART
\item \( T_3 \): TRANS READ(Z); READ(X); WRITE(Y); SNART
\end{itemize}
Define an ICCA2's instance by the following mapping:

\[ F(T_1) = (2\text{-PL-ET-rw, standard-2PL-rw, 2PL-ET-ww, standard-2PL-ww}) \]

\[ F(T_2) = (2\text{-PL-ET-rw, standard-2PL-rw, 2PL-ET-ww, standard-2PL-ww}) \]

\[ F(T_3) = (\text{CERT-ET-ww}, 2\text{PL-ET-ww, standard-2PL-ww}) \]

In a round robin execution of T, after T3 executes WRITE(Y) the following partial LS:


\[ \text{SG} = \{(T_3,T_1), (T_2,T_1), (T_2,T_3)\} \text{ where T}_1 \text{ and T}_3 \text{ are waiting for T}_2. \]

If the FIFO policy is used for wait queue management, then T3 will follow T1 in the wait queue for Y, and will be aborted when the edge (T1,T3) is added to SG. However, if T3 precedes T1 in the wait queue for Y as \( \rightarrow \) dictates, execution of T will complete successfully with no aborts.

**Algorithm 4.2: FIFO type QUEUE management consistent with \( \rightarrow \).

**Input:** Transaction T\(_i\) and a queue of transactions \( Q \).

**Output:** Q augmented by \( T_j \) such that: \( \forall T_j \in Q, \text{if } T_i \rightleftharpoons T_j, \text{ then } T_i \text{ precedes } T_j \text{ in } Q. \)

**Method:** Execute procedure AddTransactionToQ.

```
procedure AddTransactionToQ(T\(_i\))
begin
    T\(_j\) <- Head of Q;
    while T\(_j\) is defined do
        begin
            if there is a path in SG joining T\(_i\), and T\(_j\) then
                begin
                    insert T\(_i\) in front of T\(_j\) in Q;
                    stop;
                end
        else T\(_j\) <- Next transaction on Q;
        end;
    add T\(_i\) to Tail of Q;
end;
```

**Lemma 4.7:** Algorithm 4.2 guarantees that for every pair of transactions T\(_i\), T\(_j\) on Q, if T\(_i\) \rightleftharpoons T\(_j\) then T\(_i\) precedes T\(_j\) in Q.
Proof: Trivial if we note that \( r \rightarrow \) is maintained by SG, which is acyclic at all times.

Conclusion In order to prevent unnecessary transaction restarts, as demonstrated above, and in addition prevent hidden deadlocks, the management of data items wait queue is implemented as follows:

Let \( Q[X] \) denote the wait queue for data item \( X \).

1. Insertion of a new transaction, \( T_i \), into \( Q[X] \) is accomplished by Algorithm 4.2 introduced above. When \( T_i \) is inserted into \( Q[X] \), no attempt is made to resolve its wait for relation with other transactions preceding it in the queue, i.e. no edge \( (T_j, T_i) \) where \( T_j \) precede \( T_i \) in the queue, is added to SG.

2. Upon commitment of a transaction \( T_i \), for which some transaction \( T_j \) is waiting due to a \( T_i \) r-indicator or a p-indicator (currently converted into a c-indicator) held on \( X \), \( Q[X] \) is reorganized by applying the pending transaction requests in their queue order; blocked transactions are re-inserted into \( Q[X] \).

Note that the above wait queue management policy is not by itself a source for starvation problems. Starvation, however, may result from the concurrent use of 2PL and Certification synchronization policies. For example, consider an ICA2's instance in which writers are mapped by (2PL-ET-rw, strict-2PL-rw, 2PL-ET-ww, strict-2PL-ww) whereas readers are mapped by (CERT-ET-rw, *, CERT-CT-ww, *). A writer starvation may occur when it waits for an r-indicator on some data item \( X \), while readers continuously access \( X \). Nevertheless, it is expected (and verified later in the simulations) that the performance advantages which follow from matching transactions with appropriate 2PL and Certification synchronization policies will probably outweigh the disadvantage of a possible starvation problems.
CHAPTER 5

THE POWER OF THE PRIVATE WORKSPACE MODEL

The fact that distinct write operations by distinct transactions are performed into distinct private workspaces allows the concurrency control mechanism complete freedom in choosing how to use PRE_WRITEs in synchronizing conflicting operations. This has been utilized, in the previous chapter, for constructing ICCAs which integrate 2PL and Certification based synchronization techniques.

In this chapter two properties of the privateworkspace model are examined. First, we observe that the freedom of using PRE_WRITEs in synchronizing conflicting operations can be utilized for optimizations which seem to decrease the number of restarts in the system. Second, it is shown that due to the atomic write phase, the need for ww synchronization may be removed, leading to simpler algorithms.

A strategy for avoiding cycles in SG is presented in section 5.1. It is formalized, in section 5.2, by introducing the notion of the transaction's Optimized Wait For Set (OWFS). Section 5.3 presents ICCAW_2PL, which is based on the OWFS, and shows that no instance of it will restart a transaction due to a READ request or because of conflicting WRITE requests (a transaction may however be restarted on a WRITE request due to a conflict with a READ request). In section 5.4 we show that ww synchronization is redundant. It is proved that both 2PL-ET-rw with the OWFS and CERT-CT-rw are correct concurrency control algorithms.

5.1. A strategy for avoiding cycles in SG

Conflicts involving PRE_WRITE operations essentially foresee future conflicts in the rwr and ww relations. This information may be used to avoid unnecessary cycles
in SG. The idea is to avoid a transaction wait when the waiting causes a cycle in SG (possibly a deadlock) or to force a transaction wait when execution of the request causes a cycle in SG (a future data inconsistency). Note that this strategy may be applied only to transactions using some ET-synchronization and upon conflicts with active transactions (a transaction conflicting with a committed transaction is always serialized after it, and nothing can be done about it).

5.1.1. Examples for Avoiding cycles in SG

Let the following set of transactions be given: T={T₁, T₂, T₃, T₄}

T₁: TRANS WRITE(X); WRITE(Y); SNART
T₂: TRANS READ(Y); READ(X); SNART
T₃: TRANS READ(Y); WRITE(X); SNART
T₄: TRANS WRITE(Y); READ(X); SNART

and consider the two ICAA₂'s instances given by F₂PL and FCERT (see definition 4.4).

Avoiding a Deadlock on a READ request

Let T₁ and T₂ execute concurrently using ICAA₂'s instance given by F₂PL. Their round robin execution will deadlock in T₂'s READ(X) waiting stage with the following partial LS and SG:

L₅ = p₁[X]r₂[Y]
SG = { (T₂,T₁), (T₁,T₂) }

The edge (T₁,T₂), which was added because of T₂'s READ(X) request, causes a cycle in SG (deadlock). However, if T₂ gives up its waiting and skips immediately to its synchronization stage it will complete successfully (the new added edge (T₂,T₁), due to the synchronization stage, already exists in SG).

The resulting Synchronization Log is L₅ = p₁[X]r₂[Y]r₂[X]p₁[Y]w₁[X]w₁[Y].
Avoiding Inconsistency on a READ request

Let $T_3$ and $T_4$ execute concurrently using ICCA2's instance given by $F_{CERT}$. Their round robin execution will cause $T_4$ to restart on its READ($X$) request with the following partial $L^S$ and $SG$:

$$SG = \{(T_3,T_4), (T_4,T_3)\}$$

The edge $(T_4,T_3)$, which was added because of $T_4$'s READ($X$) request, causes a cycle in $SG$ since $T_4$ may cause a data inconsistency. However, if $T_4$ is forced to wait for $T_3$, both transactions will complete successfully (the added edge $(T_3,T_4)$, due to the waiting, already exists in $SG$).


Avoiding a deadlock on a WRITE request

Let $T_1$ and $T_3$ execute concurrently using ICCA2's instance given by $F_{2PL}$. Their round robin execution will deadlock in $T_3$'s WRITE($X$) waiting stage with the following partial $L^S$ and $SG$:

$$L^S = p_1[X]r_3[Y]$$
$$SG = \{(T_3,T_1), (T_1,T_3)\}$$

The edge $(T_1,T_3)$, which was added because of $T_3$'s WRITE($X$) request, causes a cycle in $SG$ (deadlock). However, if $T_3$ gives up its waiting (since $T_1$ already waits for it) and executes the WRITE request, both transactions will complete successfully.


5.2. The Optimized Wait For Set (OWFS) of a Transaction

The optimizations based on trying to avoid cycles in $SG$ may be applied only to $pr$ and $pp$ conflicts. As pointed out before, there is nothing we can do to change the serialization order upon $wr$, $wp$ or $ww$ conflicts. This observation is also valid for $rp$
conflicts because old readers always precede the conflicting writer in the serialization order. Proposition 5.1, below, identifies the conditions under which pr and pp conflicts cause a cycle in SG.

**Definition 5.1:** For two active transactions $T_i$ and $T_j$, we say that transaction $T_j$ is waiting-for transaction $T_i$ to complete, if edge $(T_i, T_j)$ was added to SG and the corresponding c-a matrix entry included a '-' sign (denoting that $T_j$'s request was delayed).

**Definition 5.2:** Let $T_i$ be an active transaction executing $O_i[X]$ operation. Define the following sets of transactions:

- $\text{WRITERS}(X) = \{ T_j \mid T_j \text{ owns a p-indicator on X} \}$
- $\text{WAITING}(T_i) = \{ T_j \mid T_j \text{ is waiting-for } T_i \text{ to complete or} \}
  \begin{align*}
  &T_j \text{ is waiting-for } T_k \text{ to complete and } T_k \in \text{WAITING}(T_i) \\
  &T_j \text{ is waiting-for } T_i \text{ to complete} \}
\end{align*}$
- $\text{FOLLOWING}(T_i) = \{ T_j \mid \text{there is a path from } T_i \text{ to } T_j \text{ in SG} \}$
- $\text{PRECEDING}(T_i) = \{ T_j \mid \text{there is a path from } T_j \text{ to } T_i \text{ in SG} \}$
- $\text{FOLLOWING-ON-X}(T_i) = \text{FOLLOWING}(T_i) \cap \text{WRITERS}(X)$
- $\text{PRECEDING-ON-X}(T_i) = \text{PRECEDING}(T_i) \cap \text{WRITERS}(X)$

**Proposition 5.1:** Let $T_i$ be an active transaction executing $O_i[X]$ operation and using some ET-synchronization technique for either rw or ww synchronization.

1. $T_i$ will create a cycle in SG in its request waiting stage in the following cases:
   a. on a p$_j$r$_i$ conflict if $T_j \in \text{FOLLOWING-ON-X}(T_i) \cap \text{WFS}_{\text{rw}}(T_i)$
   b. on a p$_j$p$_i$ conflict if $T_j \in \text{FOLLOWING-ON-X}(T_i) \cap \text{WFS}_{\text{ww}}(T_i)$

2. $T_i$ will create a cycle in SG in its request synchronization stage on a p$_j$r$_i$ conflict if $T_j \in \text{PRECEDING-ON-X}(T_i)$
Proof: In cases (1.a) and (1.b), there exists a transaction $T_j$ owning a $p$-indicator on $X$ and $T_j \in \text{FOLLOWING}(T_i)$. The edge $(T_j, T_i)$ which is added to SG, according to waiting stage entries in the c-a matrices for 2PL-ET-rw and 2PL-ET-ww, respectively, creates a cycle in SG.

In case (2), $T_i$ executes an $r_i[X]$ operation while there exists a transaction $T_j$ owning a $p$-indicator on $X$ such that $T_j \in \text{PRECEDING}(T_i)$. The edge $(T_i, T_j)$ added to SG, according to the synchronization stage entries in the c-a matrices for 2PL-ET-rw and CERT-ET-rw, creates a cycle in SG.

Proposition 5.1 suggests that in order to avoid cycles in SG, $T_i$ should, on the one hand, wait for all active transactions in $\text{PRECEDING-ON-X}(T_i)$, and on the other hand $T_i$ should not wait for any active transaction in $\text{FOLLOWING-ON-X}(T_i)$. This strategy is based on a dynamic definition of $T_i$'s wait for set. i.e., it needs to be recomputed on each request. This computation induces some extra cost. Definition 5.3, below, formalizes a special case of the above strategy by introducing the concept of the Optimized Wait For Set.

Definition 5.3: The Optimized Wait For Set of a transaction $T_i$ using 2PL, $\text{OWFS}_{\text{rw}}(\text{ww})(T_i)$, for rw(ww) synchronization, is the set of transactions that $T_i$ should wait for, in its waiting stage, in order to avoid unnecessary cycles in SG, on $p_j r_i$ or $p_j p_i$ conflicts.

(1) for $T_i$ using 2PL-ET-rw

$$\text{OWFS}_{\text{rw}}(T_i) = \begin{cases} \text{if } \text{READ request then } T \cdot \text{FOLLOWING}(T_i); \\ \text{if WRITE request then } T_i; \end{cases}$$

(2) for $T_i$ using 2PL-ET-ww

$$\text{OWFS}_{\text{ww}}(T_i) = \{ T_j | T_j \text{ is using 2PL-ET-ww } \} \cdot \text{FOLLOWING}(T_i)$$
Let \textit{optimized-2PL-rw} denote the \textit{Optimized Wait For Set}, for \textit{rw(ww)} synchronization, as defined above. Note that the synchronization policy defined by \textit{optimized-2PL-rw} may be employed by ICCA\textsubscript{2} when \textit{optimized-2PL-rw} is added to the WFS\textsubscript{drw} parameters of ICCA\textsubscript{2} as \textit{user-def}.

5.3. The Workspace 2PL(W2PL) ICCA

ICCA\textsubscript{W2PL} is introduced in this section with a proof of its correctness. It is shown that no instance of ICCA\textsubscript{W2PL} will restart a transaction due to a READ request or because of conflicting WRITE requests. However, a WRITE request may cause restart upon a conflict with a READ request.

\textbf{Definition 5.4:}

ICCA\textsubscript{W2PL} \(\equiv\) \{ 2PL-ET-rw \}, \{ optimized-2PL-rw \}
\{ 2PL-ET-ww, CERT-CT-ww \}, \{ optimized-2PL-ww \}

In the following consider the system's operation on some set of transactions using an instance of ICCA\textsubscript{W2PL} given by \(F_{W2PL}\) and resulting in an execution schedule \(E\). Let \(T_i, T_j\) and \(T_k\) be any distinct transactions in this set. We prove (1) that \(E\) is serializable, and (2) that no transaction \(T_i\) will be restarted due to a READ request or because of conflicting WRITE requests.

\textbf{Lemma 5.1, below, states formally the fact that the SG node corresponding to an active (i.e. non-blocked and prior to entering the commit phase) transaction \(T_i\) is preceded by no node of some active transaction \(T_j\) (otherwise, \(T_i\) would be waiting for \(T_j\)). In other words, there is no path in SG joining some active transaction \(T_j\) with active \(T_i\). As a consequence, if \(T_i\) subsequently commits or it is restarted it can no longer participate in a cycle in SG; its SG node may be deleted with its corresponding IT indicators without affecting database consistency.}
Lemma 5.1: If $T_i$ is an active (blocked or non-blocked) transaction prior to entering its commit phase then $\text{FOLLOWING}(T_i) \subseteq \text{WAITING}(T_i)$.

Proof: Consider $SG$ at any point in time while it is acyclic. A path $(T_1, T_2, ..., T_k)$ is maximal if there is no longer path joining $T_1$ and $T_k$. We first prove that for all maximal paths, $(T_1, ..., T_j)$, $T_j \in \text{WAITING}(T_i)$ by induction on $n$ - the length of a maximal path. Since any transaction $T_j \in \text{FOLLOWING}(T_i)$ is on some maximal path starting at $T_i$ this will prove the statement of the Lemma.

Basis $n=1$: At any point prior to $T_i$ entering its commit phase an edge $(T_i, T_j)$ may be added to $SG$ in two cases: (a) while $T_j$'s request is being processed and (b) while $T_i$'s request is being processed. For each of these cases, the two stages of a request processing must be considered, the waiting stage and the synchronization stage. Recall, that a $u_j v_i$ conflict denotes a conflict between two operations of types $u$ and $v$, where $T_j$'s operation of type $u$ precedes $T_i$'s operation of type $v$.

(a) An edge $(T_i, T_j)$ may be added to $SG$, in the waiting stage of a $T_j$'s request, upon $p_j r_j$, $r_j p_j$ or $p_i p_j$ conflicts with $T_i$ (see the waiting stage entries in the c-a matrices for 2PL-ET-rw and 2PL-ET-ww) where the request is not granted and $T_j$ is waiting-for for $T_i$ to complete.

The cases in which the edge $(T_i, T_j)$ may be added to $SG$, in the synchronization stage of a $T_j$'s request, upon $r_i p_j$, $w_i r_j$, $w_i p_j$, or $w_i w_j$ conflicts with $T_i$ (see the synchronization stage entries in the c-matrices for 2PL-ET-rw, 2PL-ET-ww and CERT-CT-ww) do not apply since they contradict our assumption that $T_i$ is an active transaction; in these cases $T_i$ must have committed prior to processing of $T_j$'s request.

(b) No edge $(T_i, T_j)$ may be added to $SG$ in the waiting stage of $T_i$'s request. However, it may be added in the synchronization stage of $T_i$'s request upon a $p_j r_i$ conflict with $T_j$ (see the synchronization stage entry for
2PL-ET-rw c-a matrix). $T_i$'s READ request could have reached its synchronization stage only if $T_j \in \text{FOLLOWING}(T_i)$ (see the OWFS$_{rw}$ definition in Definition 5.3). As the path from $T_i$ to $T_j$ is of length 1 we deduce that the edge $(T_i, T_j)$ must have been already in SG. Thus, the first time this edge was added it must have been due to a previous case (a) conflict, i.e., $T_j$ is waiting-for for $T_i$ to complete.

induction step: Assume that for all maximal paths $(T_i, \ldots, T_j)$ of length $n$, $T_j \in \text{WAITING}(T_i)$, and prove it for all maximal paths of length $n+1$.

A maximal path in SG, from $T_i$ to $T_j$, of length $n+1$ consists of two paths: a path $(T_i, \ldots, T_k)$ of length $n$, where $T_k$ is the predecessor of $T_j$, and a path $(T_k, T_j)$ of length 1. Since SG is a Directed Acyclic Graph it may be easily verified that both of the above two paths are maximal (otherwise, either the path $(T_i, \ldots, T_j)$ would not be maximal or SG is cyclic). By the induction hypothesis we have that $T_k \in \text{WAITING}(T_i)$ and that $T_k$ is a non-committed transaction (since it is waiting-for $T_i$). Using the basis we have in addition (replacing $T_i$ by $T_k$) that $T_j$ is waiting-for $T_k$, i.e., $T_j \in \text{WAITING}(T_k)$.

**Lemma 5.2**: Let $E$ be modeled by $L^5$ with its associated $\rightarrow$ relation. If $T_i \rightarrow_{rw} T_j$ then $T_i$ committed before $T_j$.

**Proof**: Suppose Lemma 5.2 is false, i.e. there exist transactions $T_i$ and $T_j$ such that $T_i \rightarrow_{rw} T_j$ and $T_j$ committed before $T_i$. By Lemma 3.2, $T_i \rightarrow_{rw} T_j$ implies that $T_i \rightarrow_{rpr} T_j$.

(1) If $T_i \rightarrow_{rpr} T_j$ then, by definition, there exists a data item $X$ such that $r_i[X] < p_j[X]$. Since (by our assumption) $T_j$ committed before $T_i$, $T_i$ must have been an active transaction owning an r-indicator on $X$ while $T_j$ executed $p_j[X]$, contradicting the fact that $T_j$'s PRE_WRITE($X$) request could not be granted (see the c-a matrix for 2PL-ET-rw waiting stage). (Recall that for a WRITE request OWFS$_{rw} = T$.)
(2) If $T_i \rightarrow_{pr} T_j$ then, by definition, there exists a data item $X$ such that $p_j[X] < r_i[X] < w_j[X]$. Execution of an $r_i[X]$ operation while $T_j$ is active and owns a p-indicator on $X$, could have been allowed only if $T_j \in \text{FOLLOWING}(T_i)$, i.e. $T_i$ precedes $T_j$ in SG. But if this is the case, then $T_j \in \text{WAITING}(T_i)$, i.e. $T_j$ must have been waiting for $T_i$ to terminate (see Lemma 5.1) and release its indicators, in particular the indicator that $T_j$ was waiting on. Thus, $T_j$ cannot commit before $T_i$.

From (1) and (2) it follows that, if $T_i \rightarrow_{rw} T_j$ then $T_j$ could not have committed before $T_i$. This completes the proof.

**Theorem 5.1:** ICCAW2PL is correct.

**Proof:** Let $E$ be modeled by $L^S$ with its associated $\rightarrow$ relation. By Lemma 5.2 and Corollary 3.2 we conclude that $L^S$ is serializable.

**Corollary 5.1:** An active transaction $T_i$ will not create a cycle in SG, in its synchronization stage, on a $w_j r_i$ or a $w_j p_i$ conflict with some committed transaction $T_j$.

**Proof:** Suppose $T_i$ creates a cycle in SG on a $w_j r_i$ or a $w_j p_i$ conflict with some committed transaction $T_j$ by adding the edge $(T_j, T_i)$. Since the edge causes a cycle in SG it must be that $T_j \in \text{FOLLOWING}(T_i)$ and by Lemma 5.1 that $T_j \in \text{WAITING}(T_i)$. However, $T_j$ has already committed and thus could not be waiting for $T_i$. Hence, our assumption is false and the edge $(T_j, T_i)$ could not have created a cycle in SG.

**Lemma 5.3:** An active transaction $T_i$ will not create a cycle in SG, in its waiting stage, on a $p_j r_i$ or a $p_j p_i$ conflict with some other active transaction $T_j$.

**Proof:** Suppose $T_i$ creates a cycle in SG in its waiting stage on a $p_j r_i$ or a $p_j p_i$ conflict with some active writer $T_j$ by adding the edge $(T_j, T_i)$. It follows that $T_j \in \text{FOLLOWING}(T_i)$. However, by definition of the OWFS, $T_i$'s request upon conflict with $T_j$ would have been granted (since $T_j$ is not in $\text{OWFS}_{rw}(T_i)$ or $\text{OWFS}_{ww}(T_i)$) and no new edge would be added to SG.
Lemma 5.4: $T_1$ will not be restarted on $ww$ synchronization.

Proof: $T_1$ may be mapped in two ways:

1. Let $F_{W2PL}(T_1) = (2PL-ET-ww, \text{optimized-2PL-ww})$

   $T_1$'s $ww$ synchronization is done at its Execution Phase by resolving $p_jp_i$ and $w_jw_i$ conflicts. $T_1$ may not create a cycle in $SG$ in its waiting stage on $p_jp_i$ conflicts, due to Lemma 5.3 and may not create a cycle in $SG$ in its synchronization stage on $w_jw_i$ conflicts, due to Corollary 5.1.

2. Let $F_{W2PL}(T_1) = (\text{CERT-CT-ww}, \ast)$

   $T_1$'s $ww$ synchronization is done at its Commit Phase by resolving $w_jw_i$ and $p_jw_i$ conflicts. We will show that no such conflict may cause a cycle in $SG$.

   a) Suppose $T_1$ creates a cycle in $SG$ on a $w_jw_i$ conflict with a committed transaction $T_j$ by adding the edge $(T_j, T_i)$. Since the edge creates a cycle in $SG$ it must be that $T_j \in \text{FOLLOWING}(T_i)$. We argue that this is impossible as a path $(T_1, \ldots, T_j)$ cannot exist in $SG$. Consider a path $(T_k, \ldots, T_j)$ in $SG$, at any time following $T_j$'s commit and prior to $T_i$ entering its commit phase. $T_k$ can not be active; otherwise, we have by Lemma 5.1 (replacing $T_i$ by $T_k$ in the Lemma) that $T_j \in \text{WAITING}(T_k)$ contradicting the fact that $T_j$ has already committed. Thus all transactions along the path $(T_k, \ldots, T_j)$ must have already been committed and a path $(T_i, \ldots, T_j)$ could not have existed in $SG$. During $T_i$'s atomic commit phase, no edge of the form $(T_i, T_k)$, where $T_k$ is a committed transaction, is added to $SG$ (see the $c-a$ matrix for CERT-CT-ww). Therefore, a path $(T_i, \ldots, T_j)$ can not be created in $T_i$'s commit phase. This contradicts our assumption that $w_jw_i$ conflict caused a cycle in $SG$.

   b) Suppose $T_i$ discovers a $p_jw_i$ conflict with some active writer $T_j$ and the new added edge $(T_i, T_j)$ creates a cycle in $SG$. It follows that $T_i \in \text{FOLLOWING}(T_j)$.

   We argue that this is impossible as a path $(T_j, \ldots, T_i)$ can not exist in $SG$. Con-
sider a path \((T_k, \ldots, T_i)\) in SG, prior to \(T_i\) entering its commit phase and while it was an active non-blocked transaction. \(T_k\) can not be active; otherwise, we have by Lemma 5.1 (replacing \(T_i\) by \(T_k\) in the Lemma) that \(T_i \in \text{WAITING}(T_k)\) contradicting the fact that \(T_i\) is waiting for no transaction. Thus all transactions along the path \((T_k, \ldots, T_i)\) must have already been committed and a path \((T_j, \ldots, T_i)\) could not have existed in SG. During \(T_i\)'s atomic commit phase, no edge of the form \((T_j, T_k)\), where \(i \neq j\) and \(T_j\) is an active transaction, is added to SG (see the c-a matrix for CERT-CT-ww). Therefore, a path \((T_j, \ldots, T_i)\) can not be created in \(T_i\)'s commit phase. This contradicts our assumption and completes the proof.

Lemma 5.5: An active transaction \(T_i\) will not create a cycle in SG on a READ request.

Proof: Let \(T_i\) execute a READ\((X)\) request. By Lemma 5.3 \(T_i\) will not create a cycle in SG in its waiting stage on \(p_jr_i\) conflicts. By Corollary 5.1 no cycle will be created in \(T_i\)'s synchronization stage on \(w_jr_i\) conflicts. We will also show that no cycle may develop in \(T_i\)'s synchronization stage on \(p_jr_i\) conflicts.

Suppose \(T_i\) conflicts in its synchronization stage with an active transaction \(T_j \in \text{WRITERS}(X)\) and the new added edge \((T_i, T_j)\) creates a cycle in SG. It follows that \(T_i \in \text{FOLLOWING}(T_j)\). By Lemma 5.1 (replacing \(T_i\) by \(T_j\)) \(T_i \in \text{WAITING}(T_j)\) and must be a blocked transaction waiting for \(T_j\) to complete. This contradicts our assumption that \(T_i\) is an unblocked transaction (following a waiting stage). This completes our proof.

Theorem 5.2: No instance of ICCAW2PL will restart a transaction on a READ request or because of conflicting WRITE requests.

Proof: Follows from Lemma 5.4 and Lemma 5.5.
5.4. Discarding WW Synchronization

The atomic write phase a priori guarantees the acyclicity of \( \rightarrow_{\text{wwr}} \) and suggests that we can design concurrency control algorithms which can do without \( \text{ww} \) synchronization. This observation is formalized by Corollary 3.2 which states that the condition that \( T_i \rightarrow_{\text{rw}} T_j \) implies \( T_i \) committed before \( T_j \) is sufficient for guaranteeing serializable executions. In this section we show that \( 2\text{PL-ET-rw} \) with the OWFS, denoted as \textit{optimized-2PL-ET-rw}, as well as \( \text{CERT-CT-rw} \), each, satisfy this condition.

Both \textit{optimized-2PL-ET-rw} and \( \text{CERT-CT-rw} \) may be modeled by an ICCA employing a NOSYNCH synchronization technique which denotes null synchronization. \textit{optimized-2PL-ET-rw} is the single instance of the following ICCA:

\[
( \{ 2\text{PL-ET-rw} \}, \{ \text{optimized-2PL-rw} \} \{ \text{NOSYNCH} \} \{ \text{null} \} ).
\]

Similarly, \( \text{CERT-CT-rw} \) is the single instance of the following ICCA:

\[
( \{ \text{CERT-CT-rw} \}, \{ \text{null} \} \{ \text{NOSYNCH} \} \{ \text{null} \} ).
\]

\textit{optimized-2PL-ET-rw}

We show now that \textit{optimized-2PL-ET-rw}, by itself, is a correct concurrency control algorithm. Furthermore, it will not restart a transaction due to a READ request.

**Theorem 5.3:** If \( E \) is an execution schedule of \( T \) using \textit{optimized-2PL-ET-rw} then \( E \) is serializable. Furthermore, during the operation of \textit{optimized-2PL-ET-rw} no transaction will be restarted on a READ request.

**Proof:** Consider the ICCA below,

\[
A = ( \{ 2\text{PL-ET-rw} \}, \{ \text{optimized-2PL-rw} \} \{ \text{CERT-CT-rw} \} \{ \text{null} \} ).
\]

ICCA \( A \) has a single instance which is identical to ICCA \( W_{2\text{PL}} \)'s instance given by \( W_{2\text{PL}} \).

\[
W_{2\text{PL}} : \forall T_i \in T, F_{W_{2\text{PL}}}(T_i) = (2\text{PL-ET-rw}, \text{optimized-2PL-rw}, \text{CERT-CT-rw}, *).
\]

Consider ICCA \( A \)'s operation on some set of transactions resulting in an execution schedule \( E \). Let \( T_1, T_2, T_k \) and \( T_m \) be in this set. We will show (1) that \( E \) is serializable
and that no transaction $T_i$ is ever restarted on a READ request, and (2) that no edge added to SG during the CTS of a transaction $T_i$, i.e. CERT-CT-ww, may affect the operation of $A$. Therefore, CERT-CT-ww is effectively a no-op. By replacing CERT-CT-ww with a NOSYNCH synchronization technique in $A$, we obtain optimized-2PL-ET-rw and conclude the theorem.

(1) By theorem 5.1, $E$ is serializable and by theorem 5.2, no transaction $T_i$ is ever restarted on a READ request.

(2) Consider an arbitrary edge in SG. The edge may affect the operation of $A$, i.e. the decisions made by $A$, only if the edge resides on some cycle in SG or if it affects the Wait For Set of some transaction. We prove that no edge which may affect $A$'s computation is added to SG during the CTS of a committing transaction.

Let $T_i$ be a committing transaction. By case (a) of Lemma 5.4.2 we have that no edge $(T_j,T_i)$ added to SG during $T_i$'s atomic CTS causes a cycle (since no transaction uses 2PL-ET-ww the only possible edges are of the form $(T_j,T_i)$ where $T_j$ is committed). As result $T_i$ successfully commits. Furthermore, once $T_i$ has committed (i.e. completed its commit phase) there is no path in SG joining some active transaction $T_k$ with $T_i$. Otherwise, we have by Lemma 5.1 (replacing $T_i$ by $T_k$ in the Lemma) that $T_i \in WAITING(T_k)$, contradicting the fact that $T_i$ has already committed.

Suppose in the future an edge $(T_j,T_i)$ which was added to SG during $T_i$'s commit phase participates in a cycle caused by an edge added on behalf of some transaction $T_m$ ($m \neq j$ since $T_j$ has already completed execution). Consider two paths $(T_i,T_m)$ and $(T_k,T_j,T_i)$ which are closed into a cycle by adding an edge of the form $(T_k,T_m)$ during $T_m$'s execution phase. Since a path $(T_k,T_j,T_i)$ contains only committed transactions, a cycle including this path may be created in SG during $T_m$'s execution phase, i.e. while $T_m$ is active, only by adding an edge of the form $(T_m,T_k)$, where $T_k$ is a committed transaction. However, by 2PL-ET-rw no such edge is added to SG on $T_m$'s behalf.
(see the c-a matrix for 2Pl-ET-rw, replacing \( T_1 \) by \( T_m \) and \( T_j \) by \( T_k \)). Thus \( T_m \) can not create at its execution phase a cycle in SG, including some edge which was added during \( T_1 \)'s CTS. Furthermore, since by Lemma 5.4.2 \( T_m \)'s CTS is a priory successful, \( T_m \) may not create any cycle in SG (in particular not one including \( (T_j, T_i) \)) during its commit phase. Therefore, no edge which was added to SG during \( T_1 \)'s CTS may subsequently participate in a cycle.

Also, since \( T_i \) is inaccessible via a path in SG from an active transaction, no edge which was added to SG during \( T_1 \)'s CTS may affect the Wait For Set computation of some active transaction (see OWFS \( r_w \) in Definition 5.3). This completes the proof that no edge added to SG during the CTS of a committed transaction may affect A's computation.

**CERT-CT-rw**

We show that CERT-CT-rw, by itself, is also a correct concurrency control algorithm.

**Lemma 5.6:** Let \( E \) be an execution schedule of \( T \) using CERT-CT-rw and modeled by \( L_S \) with its associated \( \rightarrow \) relation. If \( T_i \rightarrow_{r_w} T_j \) then \( T_i \) committed before \( T_j \).

**Proof:** If \( T_i \rightarrow_{r_w} T_j \) then, by definition, there exists a data item \( X \) such that \( r_i[X] < w_j[X] \). Suppose \( T_j \) committed before \( T_i \) did. Then, \( T_i \) must have been an active transaction owning an r-indicator on \( X \) while \( T_j \) owned a c-indicator on \( X \) and executed \( w_j[X] \).

By CERT-CT-rw an edge \( (T_i, T_j) \) would have been added to SG at \( T_j \)'s commit phase, while a second edge, \( (T_j, T_i) \) causing a cycle in SG, would have been added later when \( T_i \) attempts to commit. This would have caused \( T_i \) to restart contradicting our assumption that it was committed, i.e., \( T_i \) must have committed before \( T_j \) or else it would have been restarted.

**Theorem 5.4:** If \( E \) is an execution of \( T \) using CERT-CT-rw then \( E \) is serializable.
Proof: Follows immediately from Lemma 5.6 and Corollary 3.2.

CERT-CT-rw is similar to the Serial Validation algorithm presented in [KUNG81] and formulated by (3.1) in section 2.3.2. Both algorithms allow transaction to run unhindered and use a commit time test (synchronization) as the only means for achieving serializability. The difference between the algorithms lies in that CERT-CT-rw relies on an atomic commit phase with no intereaving DM operations, while Serial Validation relies on a critical section which blocks the commit of new transactions but allows DM_READ operations to be carried out in parallel.
CHAPTER 6

THE PRIVATE WORKSPACE MODEL FEASIBILITY

In this chapter an integrated physical TM-DM model is described in detail. For the reader's benefit we again present the details of the TM operation and expand the logical model described in chapter 3 to include a detailed description of the DM and the TM-DM interaction. Likewise, we examine in detail the feasibility of implementing this model in a realistic system environment. Following a detailed description of the integrated TM-DM model in section 6.1, a parallel commit phase algorithm which performs I/O operations only outside a critical section of the TM together with a recovery procedure is proposed in section 6.2.

6.1. Integrated TM-DM Model

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 (see Figure 6.1). The storage consists of a stable storage and a buffer storage. 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 to access. Buffer storage models a limited capacity fast main memory which does not survive system failures. The TM workspace and the DM buffer reside in the buffer storage. The database consists of a set of logical pages stored in a portion 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 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.

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

---

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.
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 (in several works the DM equally treats all
pending operations and the TM must enforce any desired order).

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 which enables it to control the con­
currency level in the system. 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.

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 updat­
ing the previous value of X in the private workspace to NEW_VALUE. Other­
wise, 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.
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 (i.e. as one indivisible and non-overlapping action); 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). Once a transaction has completed successfully its Commit Phase, it is known to be committed. The Commit Phase should effectively be atomic². A transaction Tᵢ may be restarted by the TM any time before a DM_WRITE operation has been executed on its behalf. The effect of restarting Tᵢ is to obliterate its private workspace and to treat it as a new incoming transaction.

6.1.1. The Transaction Manager Model

In this Section we examine in more detail the actions taken by the TM upon receiving a transaction's request. The data structures involved as well as the operations performed on them are described.

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

² Realizable implementations of the commit procedure are presented in the following section.
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. **SG is maintained acyclic at all times.** A transaction causing a cycle in SG is restarted.

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 \(<i, \text{TID}>\):

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 stage; then a (possibly empty) synchronization stage followed by execution of the request. A transaction using 2PL must first obtain an appropriate indicator on a data item it accesses; in the waiting stage, it is forced to wait as long as there is some other active transaction which "holds" a conflicting indicator on the same data item. Edges are appropriately added to SG in order to reflect the precedence relation imposed by the blocking. A transaction using Certification is allowed to continue immediately to the synchronization stage.
In the synchronization stage, 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 atomic commit phase of the issuing transaction as described in the previous section. In this phase all the transaction’s p-indicators are converted into c-indicators. Its r-indicators remain unchanged. 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.

6.1.2. 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 serializability constraints it must execute conflicting operations serially (i.e., one must complete before the other begins) in the order output by the TM. This may be implemented by associating a queue of requests with each “active” data item (see Figure 6.2). Note that the schedule of operations output by the DM may be different from the serializable schedule output by the TM. Nevertheless, since the DM is conflict preserving its schedule is also serializable (because conflicts are
preserved, the DM log is computationally equivalent to the TM log).

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 copy of desired stable database page.

A DM_WRITE(X,NEW_VALUE) overwrites page X if it is in the buffer; otherwise, an

---

Figure 6.2 The TM - DM Relation
empty page frame is selected and \texttt{NEW\_VALUE} is simply moved into the buffer. At some later time the DM may write the page to the stable database. A page involved in a \texttt{DM\_WRITE} operation must be pinned (see [LIND79]) to 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 policy.

6.2. Parallellism of the Commit Phase

The \textit{atomic commit phase} of a transaction has thus far been treated from a logical point of view; any implementation aspects were ignored. However, the main difficulty in practically using the private workspace model lies in attaining an efficient implementation of the \textit{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 integrity of CTS (see discussion in section 6.2.2). In order to exactly record rw conflicts no \texttt{DM\_READ} operations are allowed to 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 \textit{after} the CTS and \textit{before} issuing the \texttt{DM\_WRITE}s. Performing I/O operations in a critical section may slow the system considerably. In this chapter a parallel commit phase algorithm which performs I/O operations only outside a critical section together with a recovery procedure are proposed.

6.2.1. Supporting Atomic Commit

Supporting \textit{atomic commit} means reaching a well defined \textit{commit point} during transaction execution. If a system failure occurs \textit{before} that point then the

\footnote{If data item granularity is smaller than a page then a \texttt{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}
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 or none of a transaction's DM_WRITE operations are processed.

Atomic commitment is done in two phases [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 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.
6.2.2. 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 cause no database inconsistencies. Second, the (possibly empty) commit procedure which ends by issuing the transaction's DMWRITE operations and the conversion of its p-indicators into c-indicators. Third, a (possibly empty) cleanup phase in which the transaction is (possibly) 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 6.3.

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 the execution schedule of T_i and T_j in Figure 6.4. Both T_i and T_j use SG checking and require CTS. At the time T_i is validated by CTS, T_j has already been certified but has not yet completed its first commit phase. T_i's write conflict 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-indicator on X. Since T_j subsequently commits and issues its DMWRITE operation before T_i does, a database inconsistency results (T_j's update is lost). The subtle point is that the CTS and the write phase must be executed in a critical section; otherwise, the CTS would be only partially correct.

---

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.
procedure BasicCommitPhase(Tj)
begin
  certify <= CTS(Tj);
  if certify then
    begin
      (* first phase of commit *)
      V X E Writeset(Tj) send DM a request to force out
      the post-image of X to Tj's virtual workspace;
      Wait* for DM "Ready to Commit" message;
      TCN <= TCN+1; (* new commit number *)
      send DM a request to force out
      a commit record to Tj's virtual workspace;
      wait for DM "committed" message;
      send a completion message to Tj;
    (* second phase of commit *)
    V X E Writeset(Tj) do
      begin
        X.tid <= Tj, X.tcn <= TCN;
        execute DM.WRITE(X);
      end
      convert Tj's p-indicators into c-indicators;
    end
    (* cleanup phase *)
    if certify then CLEANUP(Tj) else RESTART(Tj);
  end

Figure 6.3: Basic Commit Phase Procedure

In certain cases (see ICCA1 instance given by example 4.1) 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.

* A wait frees the TM to serve other transactions; Tj is eligible for TM service when the wait condition holds.
Figuer 6.4: A justification of the requirement that the CTS and the write phase must be executed in a critical section.

One way to overcome these problems is by enclosing T<sub>i</sub>’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 6.5. The critical section is enclosed by "<<" and ">>". Note that this is a critical section of the entire 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.

Given the ParallelCommitPhase procedure, consider a transaction T<sub>i</sub> having a large writeset. After T<sub>i</sub> is validated by the preliminary CTS, the TM begins forcing T<sub>i</sub>’s post-images to disk. Throughout this phase, T<sub>i</sub> is in “doubt” since it may be restarted later by the CTS of the second phase. In order to decrease the probability that a transac-tion 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.
procedure ParallelCommitPhase(T_i)
begin

(* first phase of commit *)
certify <- CTS(T_i);
if certify then
begin
\forall X \in \text{Writset}(T_i) \text{ send DM a request to 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 <- CTS(T_i);
if certify then
begin
TCN <- TCN+1; (* new commit number *)
send DM a request to force out
a commit record to T_i's virtual workspace;
wait for DM "committed" message;
send a completion message to T_i;

\forall X \in \text{Writset}(T_i) \text{ do}
begin
X.tid <- T_i, X.tcn <- TCN;
execute DM.WRITE(X);
end
convert T_i's p-indicators into c-indicators;
end >>
(* cleanup phase *)
if certify then CLEANUP(T_i) else RESTART(T_i);
end

Figure 6.5: Parallel Commit Phase Procedure

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.
6.3. An Improved Parallel Commit Phase Algorithm

The ParallelCommitPhase procedure (Figure 6.5) 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 state of the system is illustrated by the system block diagram in Figure 6.6. Commit requests are enqueued on the COMMIT Q while DM operations are independently enqueued on a dedicated page frame queues. All the queues reside in buffer storage and are processed on a FCFS basis. 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 (TCN) order.

Procedure RECOVERY implements system restart (see Figure 6.7). The main difficulty lies in finding the TCN of the certified non-committed writer which might have affected the stable database. 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 anywhere, then its virtual workspace may be discarded (since \( T_1 \) 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 (see Figure 6.6), let \( T_J \) be the current non-committed writer whose \texttt{DM\_WRITE} operations have already been issued by the TM. Suppose transaction \( T_K \) has managed to read a data item updated by

\[
\text{Figure 6.6 System Block Diagram}
\]
procedure ImprovedParallelCommitPhase(T_i)
begin
 (* first phase of commit *)
certify <- CTS(T_i);
if certify then
begin
∀ X ∈ Writeset(T_i) send DM a request to force out
the post-image of X to T_i's virtual workspace;
wait for DM "Ready to Commit" message;
if T_i is a writer then wait until
cur_committing_writer is known to be committed;
(* second phase of commit *)
<< certify <- CTS(T_i);
if certify then
begin
(* no I/O is associated with this phase *)
if T_i is a writer then
cur_committing_writer <- T_i;
TCN <- TCN + 1; (* new committ number *)
Enqueue T_i's commit request on the COMMIT QUEUE;
∀ X ∈ Writeset(T_i)
do begin
X.tid <- T_i, X.tcn <- TCN;
end
convert T_i's p-indicators into c-indicators;
end
end
(* cleanup phase *)
if certify then
begin
CLEANUP(T_i);
wait for DM "committed" message;
T_i is known to be committed;
send a completion message to T_i;
end
else RESTART(T_i);
end

procedure RECOVERY
begin
AL <- { Ti | T_i has a virtual workspace, CL <- φ;
(* add to CL all pairs (T_i, T_i's TCN)
such that T_i has affected the database *)
∀ Ti ∈ AL do
begin
(* add T_i to CL if it is known to be committed *)
if ∃commit-record(T_i) then begin
ten <- T_i's TCN found in the commit-record;
CL <- CL ∪ (T_i, ten);
end else begin
(* Add T_i to CL if it is a certified non-committed
transaction which has affected the database;
otherwise discard it. *)
ten <- null;
∀ X ∈ Writeset(T_i) do begin
DM.READ(X);
if X.tid = T_i then ten <- X.tcn;
end
else CL <- CL ∪ (T_i, ten);
end
end
(* of else *)
end
(* of RECOVERY *)
end

Figure 6.7: Improved Parallel Commit Phase and RECOVERY procedures

T_j (the value may be served from the DM buffer) and has completed successfully before
a system failure. If the failure occurs before T_j's commit record has reached stable
storage and before $T_j$ has affected the stable database, then $T_j$'s effects will eventually be undone. Thus $T_k$ has seen a value which never existed. However, had we required that $T_k$ be committed after $T_j$ 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.

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 currently committing writer's commit record is known to be in stable storage. This restriction may be relaxed resulting in more parallelism. The idea is based on the observation that a new writer need wait only until "old" transactions, from which it has read, are known to be committed. In other words, updates of a newly committing writer should be delayed only if there exists a certified non-committed transaction from which it has read from.

Let $T_i$ be a committing writer which has completed its Execution Phase. Define the following:

$$\text{View}(T_i) = \{ (X, \text{tid}, \text{tcn}) | X \in \text{Readset}(T_i) \}$$

$$\text{OPEN} = \{ (T_j, T_j's \text{ tcn}) | T_j \text{ is an updating transaction on the COMMIT QUEUE} \}$$

$$\text{NonCommitedView}(T_i) = \text{View}(T_i) \cap \text{OPEN}$$

$$\text{MaxTCNseenByTi} = \max \{ T_j's \text{ tcn} | T_j \in \text{NonCommitedView}(T_i) \}$$

$$\text{SystemTCN} = \text{the TCN of the transaction whose commit record is the latest to have reached stable storage.}$$
Prior to entering its second phase of commit, $T_i$ should wait until its NonCommittedView becomes empty. Since commit record writing requests are processed in ascending TCN order, this is equivalent to requiring that $T_i$ should wait until $\text{SystemTCN} \geq \text{MaxTCNseenByTi}$. A variation of this solution puts the burden of delaying $T_i$'s updates on the DM [SHEM85]. No limitation is imposed by the TM on $T_i$'s entering its second phase of commit; however, the DM must guarantee that none of the pages updated by $T_i$ will migrate to the stable database until $\text{SystemTCN} \geq \text{MaxTCNseenByTi}$. A simpler but more restrictive solution [ELHA84] requires that none of the pages updated by $T_i$ will migrate to the stable database until $T_i$'s commit record has reached stable storage. In other words, page $X$ will not migrate to the stable database as long as $X.tcn < \text{SystemTCN}$.

The above restriction imposed on $T_i$, by the TM or by the DM, may be removed resulting in even more parallelism (and increased overhead). The idea is that prior to entering its second phase of commit, $T_i$ should force out to stable storage the set of pairs in $\text{NonCommittedView}(T_i)$ (thereby committing the transactions in $\text{NonCommittedView}(T_i)$). Since the set is usually small this may be accomplished, with no extra cost, by simply piggy-bagging each pair in this set onto $T_i$'s post-images, before the latter are forced out to stable storage. Thus, the cost of forcing out the pairs in $\text{NonCommittedView}(T_i)$ may be charged to the I/O cost of forcing out $T_i$'s post-images to stable storage. In case the size of $\text{NonCommittedView}(T_i)$ is too large to be piggy-bagged onto $T_i$'s post-images, $T_i$ must wait until some transactions are removed from OPEN (and thus the size of its $\text{NonCommittedView}$ is reduced).

Procedure RECOVERY is 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 spreads the commit.
operation I/O cost over several transactions?.
CHAPTER 7

PERFORMANCE EXPERIMENTS

This chapter describes the simulation experiments conducted in order to demonstrate the performance advantages of private workspace based Integrated Concurrency Control Algorithms. A major goal of the experiments is to characterize the tradeoff between blocking, overhead induced by 2PL and restart costs induced by Certification.

In section 7.1 we discuss our simulation approach. Section 7.2 describes the simulation model in detail. The experimental setup is given in section 7.3. Finally, section 7.4 presents the set of concurrency control experiments conducted and concludes with a summary of observations.

7.1. Basic Approach

There are three traditional approaches to performance evaluation: queuing models, simulation models and system benchmarks. Analytic queuing models of concurrency control algorithms are difficult to develop and are usually done at a rather coarse level of detail because of simplifications required to keep the solution tractable. For example, many analytic models assume one or more of the following: there is only one transaction class, transactions have fixed lengths, transactions predeclare their locks and the locks are exclusive (i.e. no sharing). System benchmark is the most accurate and detailed method. However, it requires the monitoring of an existing database system. The accuracy of simulations varies, depending on how detailed the simulation is. One advantage of simulation models is that they allow the modeling of complicated scheduling policies and are capable of producing distributions of various performance measures.
Based on previous performance evaluation studies [AGRA83, CAREB3, GALL82, ROBI82, TAYB4, WILK81], we do not expect differences between concurrency control algorithms to result in "order of magnitude" differences in throughput. Since we expect rather fine variations, we chose to construct a fairly-detailed simulation model and to conduct experimental studies under controlled conditions by measuring of throughput and restart rates, response times, resource utilizations and queue lengths.

7.2. The Simulation Model

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. The model is a closed queuing system: a transaction processor initiates a new transaction only when a previous one completed, thus the multiprogramming level is kept constant. 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.

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 prime source of concern. Thus, our model assumes unlimited memory and some fixed cost is associated with concurrency control and various system services.

In order to aid simulation monitoring and to allow for interactive studies, a high level experiment specification language (see Appendix B) and an interpreter for it were
developed on top of the simulation model.

7.2.1. The Logical Model

The logical structure of the model is illustrated by Figure 7.1. 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 7.1 DBMS Logical Structure

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 which are bracketed by TRANS and SNART requests (signaling
start: THINK;
STARTUP cpu and i/o;
ON restart goto begin;

begin: TRANS;
LOCAL cpu and i/o;
READ/ WRITE request;
LOCAL cpu and i/o;
READ/ WRITE request;
LOCAL cpu and i/o;
SNAR;

goto start;

Figure 7.2 TP Model

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. Once a transaction is restarted by the CC it repeats its sequence of local processing and database requests.

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 Model is given by Figure 7.3.

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 issues a DM_READ or a PRE_WRITE operation on behalf of $T_i$. In an operation mode in which $T_i$’s post-images are 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 blocked.
put on the front of the request queue\(^1\). 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\(^2\) period.

A SNART request triggers the commit phase of transaction \(T_i\); it is implemented by the improved parallel commit phase algorithm proposed in section 6.3, in which multiple commit requests of updating transactions on the COMMIT QUEUE are allowed. The commit phase begins with the (possibly empty) CTS 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 CTS, \(T_i\) is not yet ready to commit. In this case the CC must first issue 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 by performing functions which are similar to those performed by a back-end database processor. The DM accepts the

---

\(^1\) \(T_i\) is given higher priority over other transactions in the request queue in order to minimize the possibility of starvation.

\(^2\) The purpose of this delay is to allow the transactions with which \(T_i\) has conflicted to finish before \(T_i\) is restarted.
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\(^3\) algorithm (see Appendix A). 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 removal 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.

7.2.2. Transaction State Diagram

Using the above description of the TP, CC and DM components, the transaction state diagram given in Figure 7.4 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.

A summary of the parameters used to determine the delay time or request service time at each logical state is given in Table 7.1. All parameter values are specified in milliseconds.

---

\(^3\) 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.
The THINKtime parameter models TP thinking time. The parameters STARTUPcpu and STARTUPio are the amounts of CPU and I/O associated with transaction startup. Similarly, LOCALcpu and LOCALio are the amounts of CPU and I/O associated with transaction local processing.

![Figure 7.4 Transaction State Diagram](image)

The parameters CCinit, CCconflict, CCcommit and CCcleanup determine the CPU cost associated with the CC computation of transaction initialization, READ/WRITE request conflict analysis, transaction commit and transaction cleanup respectively. The conflict analysis cost is taken in consideration only when the transaction is using some ET-Synchronization. The cost associated with CCcommit is only considered when the transaction is a writer. The CTS cost associated with a
A transaction using CERT-CT-rw synchronization technique is \( N_r \cdot CC_{\text{conflict}} \), where \( N_r \) is the number of data items read by the transaction, while the CTS cost associated with a transaction using CERT-CT-ww synchronization technique is \( N_w \cdot CC_{\text{conflict}} \), where \( N_w \) is the number of data items written by the transaction.

<table>
<thead>
<tr>
<th>Process</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>THINKtime</td>
<td>TP think time</td>
</tr>
<tr>
<td></td>
<td>STARTUPcpu</td>
<td>CPU 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>CCconflict</td>
<td>CC CPU time for request conflict analysis</td>
</tr>
<tr>
<td></td>
<td>CCcommit</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 7.1 System Parameters**

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 DMcpu and PRE_WRITEcpu parameters, although they actually have identical values. This also holds for the DMio and PRE_WRITEio parameters. DM_POSTcpu and DM_POSTio model the CPU and I/O 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 post-image). Finally, the RESTARTdelay parameter 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.

7.2.3. 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 7.5. The CPU server has three queues servicing requests for the CC, the DM and the TPs.

![Figure 7.5 DBMS Physical Model.](image-url)
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 is not a limiting factor (i.e. the "bottleneck") of the system. CPU requirements by the DM have traditionally been underestimated. It is not that we truly have (need) an I/O server with infinite parallel processing capability, but, due to the very large CPU demand of each I/O operation we can not derive more than a few in parallel (because of the lack of CPU power).

There may be pending service requests in all CPU queues. In this 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.

7.2.4. The Workload Model

When a transaction initiated by a TP is assigned a script, consisting of the data items that it has to read and write during its execution (i.e. its readset and writeset). Each TP is associated with a class type which determines how this script is created, i.e., how the transaction’s readset and writeset are selected and interleaved. The workload parameters defined in Table 7.2, enable us to set up environments by mixing TP classes and transaction workloads.

The DB parameter determines the number of data items in the database. Transaction requests are made on a data item basis. Thus, in modeling read and write requests data items are given values ranging from 1 to DB.
The $R_{Si}$ parameter defines the readset distribution for a $CLASS_i$ transaction; it can be specified as either $\text{unif}(n,m)$, $\text{exp}(n)$ or $\text{seq}(n,m)$. If $R_{Si}$ is $\text{unif}(n,m)$ then the readset size is uniformly distributed in the range $[n,m]$ and the readset is assigned by randomly selecting data items \textit{without replacement} from the entire database. If $R_{Si}$ is $\text{exp}(n)$ then the readset size is selected from an exponential distribution with mean $n$ and truncated to an integer value. The readset is assigned as in the uniform case. If $R_{Si}$ is $\text{seq}(n,m)$ then the readset size is uniformly selected in the range $[n,m]$ and the readset is assigned a random collection of \textit{consecutive} data items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DB$</td>
<td>Database Size</td>
</tr>
<tr>
<td>$R_{Si}$</td>
<td>Readset Distribution of a $CLASS_i$ transaction</td>
</tr>
<tr>
<td></td>
<td>- $\text{unif}(n,m)$</td>
</tr>
<tr>
<td></td>
<td>- $\text{exp}(n)$</td>
</tr>
<tr>
<td></td>
<td>- $\text{seq}(n,m)$</td>
</tr>
<tr>
<td>$WS_i$</td>
<td>Writeset Distribution of a $CLASS_i$ transaction</td>
</tr>
<tr>
<td></td>
<td>- $\text{unif}(n,m)$</td>
</tr>
<tr>
<td></td>
<td>- $\text{exp}(n)$</td>
</tr>
<tr>
<td></td>
<td>- $\text{seq}(n,m)$</td>
</tr>
<tr>
<td>$Pr_{Wi}$</td>
<td>$\text{Prob}(\text{WRITE } X \mid \text{READ } X)$ for a $CLASS_i$ transaction</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Number of $CLASS_i$ Transaction Processors</td>
</tr>
</tbody>
</table>

\textbf{Table 7.2: Workload Parameters}

The parameter $Pr_{Wi}$ defines the probability that a data item read by a $CLASS_i$ transaction will be updated by it. If it is not specified we assume as default the "80/20 rule", i.e., 80% of the data items read by the transaction are updated by it and $Pr_{Wi}$ is set to 0.80. The $WS_i$ parameter, similarly, defines the writeset distribution of a $CLASS_i$ transaction. The writeset size is defined as for $RS_i$, however, the data items for the writeset are first selected from the readset with probability $Pr_{Wi}$ (for each data item in
the readset), and then the rest are selected from the entire database (according to the distribution function). If $\text{WS}_i$ is not specified, then the writeset is assumed to be null.

To form a script, the readset and the writeset are randomly interleaved 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 script.

The parameter $N_i$ determines the number of \text{CLASS}_i Transaction Processors. The system \textit{multiprogramming level} is $\Sigma N_i$.

Using a high level specification language developed for the simulation (Appendix B), example 7.1 sets up a sample environment consisting of a database with 1024 data items. The multiprogramming level is 16 Transaction Processors: 8 originate short transactions which randomly read and update the database, 6 originate short queries which randomly read the database and 2 originate long queries which sequentially read the database.

\begin{verbatim}
set DB=1024
def Writers readset=unif(4,6) writeset=unif(2,4) Prw=0.50
def ReadersS readset=unif(4,6)
def ReadersL readset=seq(62,66)
add Writers: 8
add ReadersS:8
add ReadersL:2
\end{verbatim}

\textbf{Example 7.1: Sample Workload Setup}

\section*{7.3. Experimental Setup}

The experiments presented were designed to investigate the \textit{relative performance} of various concurrency control algorithms. Since the relative performance of the algorithms depends on the \textit{conflict rate} among transactions, it was decided to vary the amount of data contention by fixing the database size and varying the number of concurrently executing transactions. In the experiments, the \textbf{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 determines the number of transactions that must be committed during the experiment. 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 7.3).

<table>
<thead>
<tr>
<th>Class</th>
<th>Readset Distribution</th>
<th>Writeset Distribution</th>
<th>Prw</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Writers</td>
<td>unif(4,6)</td>
<td>unif(2,4)</td>
<td>0.50</td>
<td>8</td>
</tr>
<tr>
<td>Short Readers</td>
<td>unif(4,6)</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Medium Writers</td>
<td>unif(8,12)</td>
<td>unif(4,6)</td>
<td>0.50</td>
<td>15</td>
</tr>
<tr>
<td>Medium Readers</td>
<td>unif(8,12)</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Long Readers</td>
<td>seq(62,66)</td>
<td></td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

Table 7.3: Transactions Classes

The transaction classes in Table 7.3 model transactions which result from precompiled programs. Therefore, the system parameters used in the experiments (see Table 7.4) 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 (and not during the Execution Phase).
Table 7.4: 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>LOCALpio</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>CCinit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CCconflict</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CCcommit</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>28+2</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>

7.4. Experiments and Results

A major difficulty in designing the experiments is 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 for these factors is defined in Table 7.5. Notice that the effect of each experiment spans one or more table entries. Therefore, to reliably cover the table, only a small number of experiments is needed. The 2PL method serves as the basis for the performance comparison study. Based on 2PL behavior, 4 experiments were designed which cover the interesting entries of Table 7.5.

Experiment 1 represents a mix of short and long transactions and a moderate arrival rate (see Table 7.11). 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 7.12). Here, 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 7.13). 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 7.14). Here, 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.

<table>
<thead>
<tr>
<th>Experimental Environment</th>
<th>Blocking/Conflict Rate</th>
<th>CPU Utilization</th>
<th>Resource Contention</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>low</td>
<td>high</td>
<td></td>
</tr>
<tr>
<td>Data Contention</td>
<td>exp 1.</td>
<td>exp 4.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exp 2.</td>
<td>exp 1.</td>
<td>exp 4.</td>
</tr>
<tr>
<td></td>
<td>exp 3.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.5: Experimental Environment for 2PL

7.4.1. The Algorithms Studied

The concurrency control algorithms used in the experiments represent different synchronization policies, ranging from strict locking (e.g. 2PL) to full certification (e.g. SG checking). The algorithms are described in Table 7.6. 2PL, ICCA1 and CERT algorithms represent three distinct instances of ICCA2. The DYNAMIC algorithm represents a set of ICCA2's instances and 02PL represents an instance of ICCA\(_W\) 2PL.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PL</td>
<td>$F(T_i) = (2PL_{ET-rw}, \text{standard-}2PL_{rw}, 2PL_{ET-ww}, \text{standard-}2PL_{ww})$</td>
<td>&quot;ordinary&quot; 2PL.</td>
</tr>
<tr>
<td>O2PL</td>
<td>$F(T_i) = (2PL_{ET-rw}, \text{optimized-}2PL_{ET-rw}, 2PL_{ET-ww}, \text{optimized-}2PL_{ET-rw})$</td>
<td>2PL with the Optimized Wait For Set, no transaction is restarted on a READ request or due to ww synchronization.</td>
</tr>
<tr>
<td>ICCA1</td>
<td>$F$: if $T_i$ is a reader then $(\text{CERT-ET-rw, } *, \text{CERT-CT-ww, } *)$ else $(2PL_{ET-rw}, \text{standard-}2PL_{rw}, 2PL_{ET-ww}, \text{standard-}2PL_{ww})$</td>
<td>An ICCA in which readers use SG checking whereas writers use 2PL and do not wait for the readers.</td>
</tr>
<tr>
<td>CERT</td>
<td>$F(T_i) = (\text{CERT-ET-rw, } *, \text{CERT-CT-ww, } *)$</td>
<td>SG checking.</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>$F$: if $T_i$ is a writer then $(2PL_{ET-rw}, \text{standard-}2PL_{rw}, 2PL_{ET-ww}, \text{standard-}2PL_{ww})$ else if cpu utilization $&gt; 0.94$ and $T_i$ is a long reader then $(2PL_{ET-rw}, \text{null, CERT-CT-ww, } *)$ else $(\text{CERT-ET-rw, } *, \text{CERT-CT-ww, } *)$</td>
<td>An ICCA which dynamically changes the transactions' synchronization policy according to the CPU utilization.</td>
</tr>
</tbody>
</table>

Table 7.6: The Algorithms Studied

The 2PL algorithm examined is similar to the "ordinary" 2PL with a deferred update recovery mechanism [GRAY76]. The 2PL and CERT algorithms represent two extremes of synchronization policies, i.e., strict locking and full certification (SG checking), respectively. The O2PL algorithm models 2PL with the Optimized Wait For Set in which no transaction is restarted on a READ request or because of ww-synchronization. It is supposed to reduce the number of transaction restarts induced by 2PL. By letting readers use certification, whereas writers use 2PL and do not wait for the readers, ICCA1 is supposed to reduce the overhead of blocking employed in 2PL. Finally, the DYNAMIC algorithm synchronization method changes according to CPU utilization. In an environment with no CPU contention, DYNAMIC uses the same method as ICCA1, i.e., readers use certification whereas writers use 2PL and do not wait for readers. In an environment in which the CPU approaches the contention area, the writers' synchronization is changed and writers are forced to wait for long readers.
7.4.2. The Performance Comparison Metric and Total Results

The designed experiments model transaction environments consisting of various transaction classes. The performance of a given transaction class is measured by its throughput, i.e., by the number of committed transactions in that class per unit time. Sometimes it is difficult to compare the performance of two algorithms because they induce different performance on different transaction classes. To resolve this difficulty, a performance metric which compares the overall number of committed transaction requests per unit time is used. This number may be computed by multiplying the throughput of each class by its average transaction size (as defined by Table 7.3). For example, a throughput rate of 1 Short Writer/sec., 1 Short reader/sec. and 1 Long reader/sec. is equivalent to a throughput rate of 77 (8+5+64) requests/sec.

The four designed experiments were performed for each of the 2PL, 02PL, IÇCA1 and CERT algorithms (see Table 7.6). For the DYNAMIC algorithm, only experiment 1 and 2 were performed since DYNAMIC is identical to IÇCA1 in experiments 3 and 4. The detailed results are shown in Tables 7.11-7.24. These represent a total of more than 2,000,000 executed transactions. The total throughputs, derived according to the above performance comparison metric, are shown in Tables 7.7 - 7.10. Each of these tables corresponds to a single experiment.

<table>
<thead>
<tr>
<th>Experiment 1.</th>
<th>Total Throughputs in committed requests/second.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPL</td>
<td>2PL</td>
</tr>
<tr>
<td>16</td>
<td>35.72</td>
</tr>
<tr>
<td>32</td>
<td>71.07</td>
</tr>
<tr>
<td>48</td>
<td>103.36</td>
</tr>
<tr>
<td>64</td>
<td>136.68</td>
</tr>
<tr>
<td>80</td>
<td>164.80</td>
</tr>
<tr>
<td>96</td>
<td>189.13</td>
</tr>
<tr>
<td>112</td>
<td>188.94</td>
</tr>
<tr>
<td>128</td>
<td>176.49</td>
</tr>
</tbody>
</table>

Table 7.7 - Experiment 1 Total Throughputs.
## Table 7.8 - Experiment 2 Total Throughputs.

<table>
<thead>
<tr>
<th>MPL</th>
<th>2PL</th>
<th>02PL</th>
<th>ICCA1</th>
<th>CERT</th>
<th>DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>60.91</td>
<td>61.52</td>
<td>61.28</td>
<td>61.20</td>
<td>61.28</td>
</tr>
<tr>
<td>32</td>
<td>116.76</td>
<td>118.55</td>
<td>120.38</td>
<td>119.53</td>
<td>120.38</td>
</tr>
<tr>
<td>48</td>
<td>164.17</td>
<td>168.81</td>
<td>175.68</td>
<td>175.13</td>
<td>175.68</td>
</tr>
<tr>
<td>64</td>
<td>197.28</td>
<td>205.50</td>
<td>228.32</td>
<td>204.92</td>
<td>217.95</td>
</tr>
<tr>
<td>80</td>
<td>185.18</td>
<td>215.89</td>
<td>214.45</td>
<td>195.26</td>
<td>235.72</td>
</tr>
<tr>
<td>96</td>
<td>177.00</td>
<td>212.00</td>
<td>179.56</td>
<td>182.39</td>
<td>232.28</td>
</tr>
<tr>
<td>112</td>
<td>150.91</td>
<td>195.44</td>
<td>164.65</td>
<td>156.64</td>
<td>241.14</td>
</tr>
<tr>
<td>128</td>
<td>133.63</td>
<td>176.81</td>
<td>160.63</td>
<td>148.02</td>
<td>258.56</td>
</tr>
</tbody>
</table>

## Table 7.9 - Experiment 3 Total Throughputs.

<table>
<thead>
<tr>
<th>MPL</th>
<th>2PL</th>
<th>02PL</th>
<th>ICCA1</th>
<th>CERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>136.65</td>
<td>137.00</td>
<td>138.40</td>
<td>136.55</td>
</tr>
<tr>
<td>32</td>
<td>232.30</td>
<td>233.05</td>
<td>241.95</td>
<td>222.00</td>
</tr>
<tr>
<td>48</td>
<td>227.90</td>
<td>230.60</td>
<td>239.40</td>
<td>234.02</td>
</tr>
<tr>
<td>64</td>
<td>206.05</td>
<td>212.65</td>
<td>232.80</td>
<td>203.50</td>
</tr>
<tr>
<td>80</td>
<td>194.65</td>
<td>198.15</td>
<td>226.35</td>
<td>195.90</td>
</tr>
<tr>
<td>96</td>
<td>186.15</td>
<td>189.10</td>
<td>210.35</td>
<td>186.75</td>
</tr>
<tr>
<td>112</td>
<td>183.40</td>
<td>187.90</td>
<td>215.15</td>
<td>176.65</td>
</tr>
<tr>
<td>128</td>
<td>179.95</td>
<td>189.80</td>
<td>216.05</td>
<td>171.15</td>
</tr>
</tbody>
</table>

## Table 7.10 - Experiment 4 Total Throughputs.

<table>
<thead>
<tr>
<th>MPL</th>
<th>2PL</th>
<th>02PL</th>
<th>ICCA1</th>
<th>CERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>84.08</td>
<td>84.08</td>
<td>84.29</td>
<td>83.58</td>
</tr>
<tr>
<td>32</td>
<td>163.82</td>
<td>163.78</td>
<td>164.26</td>
<td>161.87</td>
</tr>
<tr>
<td>48</td>
<td>214.37</td>
<td>214.30</td>
<td>214.50</td>
<td>194.44</td>
</tr>
<tr>
<td>64</td>
<td>212.41</td>
<td>212.39</td>
<td>213.69</td>
<td>192.73</td>
</tr>
<tr>
<td>80</td>
<td>209.17</td>
<td>209.16</td>
<td>212.28</td>
<td>191.25</td>
</tr>
<tr>
<td>96</td>
<td>206.14</td>
<td>205.63</td>
<td>209.45</td>
<td>186.19</td>
</tr>
<tr>
<td>112</td>
<td>202.69</td>
<td>203.74</td>
<td>208.79</td>
<td>186.98</td>
</tr>
<tr>
<td>128</td>
<td>200.50</td>
<td>200.50</td>
<td>205.67</td>
<td>184.23</td>
</tr>
</tbody>
</table>
7.4.3. Study 1: O2PL Vs. 2PL

We investigate the advantages of O2PL (which aims at minimizing the number of transactions restarts) relative to 2PL. The results of the experiments (see tables 7.11 - 7.14) suggest that O2PL usually outperforms 2PL and has a smaller restart rate. O2PL does not restart readers and typically the writers restart rate is decreased by 10% - 20%. Surprisingly, the performance of short and medium transactions, using 2PL, is as good as in O2PL. This does not hold for long readers; under high workloads, O2PL has better throughput and response time compared with long readers using 2PL (by 100%). An important outcome of this study is that, the blocking level, as measured by the average number of blocked transactions, is almost the same in both algorithms.

It is interesting to compare the behavior of both algorithms in experiment 1 relative to experiment 2 with MPL greater than 96. Although the arrival rate of the transactions in experiment 2 is twice that of experiment 1, both algorithms have a higher throughput and a better CPU utilization in experiment 1 due to the severe blocking overhead suffered in experiment 2. This suggests that in work environments in which the CPU is underutilized and the system experiences a high blocking rate the throughput may be increased by letting some transactions use certification.

7.4.4. Study 2: of ICCA1 Vs. O2PL

Performance study 1 indicates that blocking, by itself, can cause trashing. An example for that is the behavior of O2PL and 2PL in experiment 2. In this work environment in which the CPU is underutilized, reducing the blocking rate seems appealing. This study investigates the performance of ICCA1 relative to that of O2PL. ICCA1 is supposed to reduce the overhead of the blocking employed by O2PL by letting readers use certification (SG checking) whereas writers use 2PL and do not wait for readers.

The performance of ICCA1 is given in Tables 7.15-7.18. To facilitate the comparison, the performance tables of O2PL (Tables 7.11 -7.14) are presented again. The
results of the experiments suggest that in most environments ICCA1 outperforms 02PL. In environments consisting of short writers and long readers ICCA1 induces a better performance by writers at the expense of long readers. Another observation is that with no CPU contention the blocking phenomenon is dominant and restarts are cheap. ICCA1 makes a better use of the CPU; both the number of blocked transactions and the waiting times are reduced; this lead to a significant improvement in the overall system performance. These observations may be verified by inspecting the tables of experiments 1 and 2 (Tables 7.15-7.16). Note that the long readers using ICCA1, with a significant restart rate, perform comparably well to the long readers using 02PL, which are never restarted, 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 for long readers. As MPL increases, ICCA1 no longer takes advantage of the decreased blocking rate since waiting due to blocking is replaced by increased waiting for CPU service. This explains the sharp decrease in throughput for long readers using ICCA1 the moment they enter the CPU contention area (Tables 7.15-7.16). It also explains the observations in experiment 4, where 02PL performs almost as well as ICCA1. The significant improvement in ICCA1's performance in experiment 3 commences as soon as 02PL approaches the trashing area and is due to the 20% - 40% cut down in the number of restarted transactions. On the other hand, 02PL suffers from a high blocking overhead coupled with an increased CPU contention which adversely affects its performance.

7.4.5. Study 3: CERTVs. 2PL

In this study we examine how transaction performance is affected by using pure SG checking (which entirely avoids blocking). In particular, we examine how the SG checking algorithm performs in comparison to 2PL. Recall that the SG checking algorithm modeled induces an additional CPU overhead due to CTS.
The results of the experiments (see Table 7.19-7.22) suggest that the *SG checking algorithm improves the performance of writers at the expense of readers*. This is similar to ICCA1's behavior carried to an extreme. For example, in experiment 2 (Table 7.20), as the MPL increases, the throughput of long readers which use CERT drops to zero whereas the performance of writers is improved three fold in comparison to 2PL. Another observation is that the restart rate of the SG checking algorithm seems reasonable.

For 2PL, increased transaction conflicts result in increased waiting time due to blocking. However, as MPL increases, there is also increased contention for the CPU. If the waiting time due to blocking dominates the waiting time on the CPU then CERT usually outperforms 2PL. This explains the observations in experiments 1 and 2 where CERT outperforms 2PL (Tables 7.7-7.9). It also explains the observations in experiment 3 where the medium writers using CERT usually outperform the medium writers using 2PL (Table 7.21). The improvement starts at MPL=48 as soon as the blocking rate induced by 2PL increases dramatically. At this point, it is interesting to compare this behavior with the performance of medium writers using ICCA1 (Table 7.17). As medium writers using ICCA1 do not have to wait for readers they suffer less blocking overhead. As result, the domination of medium writers using CERT over medium writers using ICCA1 starts only at MPL=64; the writer's using CERT perform significantly better although they suffer from 100% increase in the restart rate.

In environments where the waiting time due to the blocking is negligible as compared to the waiting on the CPU the effect of restarts becomes a dominant factor. Since CERT induces more CPU overhead and leads to a higher number of restarts its performance is adversely affected in such environments. This explains the results of experiment 4 (Table 7.22) where CERT has a lower throughput than 2PL (Table 7.10).
In all the experiments the total throughput differences between 2PL and CERT are marginal. In addition the restart rate of CERT is relatively reasonable; thus, we conclude that the SG checking algorithm is a good possible alternative to 2PL.

7.4.6. Study 4: DYNAMIC vs. ICCA1

The performance degradation of ICCA1 in experiments 1 and 2 is due to the sharp decrease in the throughput of long readers (Tables 7.16-7.17). This effect has been noticed and discussed in study 2 and is the result of the long readers’ high restart cost in the CPU contention environment. In order to improve ICCA1’s performance it is mandatory to minimize the restart rate of the long readers as the system approaches the CPU contention area. This idea is implemented by the DYNAMIC algorithm; when CPU utilization increases beyond 94% writers are forced to wait for long readers (whereas long readers continue to use certification). This substantially reduces the throughput of the writers and as a result the restart rate of long readers. The overall effect is a significant increase in the total throughput (see Table 7.7-7.8).

This synchronization policy improves the performance of all transaction classes, in experiments 1 and 2, relative to 2PL. The improvement of 2PL’s total throughput reaches 36% in experiment 1 and 93% in experiment 2.

7.4.7. Summary of Observations

For the CPU bound system that we modeled, controlling the blocking level proved to have a much higher impact on system performance than attempting to minimize the number of restarts. The results of the experiments suggest that there is a tradeoff between the cost of waiting due to blocking and the cost of waiting for CPU service. With no CPU contention blocking is dominant and restart cost is cheap. On the other hand, under high CPU contention restart is more expensive and it becomes a dominant factor for long readers; for short and medium transactions restart is a dominant factor.
only under low blocking overhead, however, as MPL increases the cost of waiting due to
blocking also increases up to a point where restart becomes less expensive than block-
ing. These results confirm some previously observed phenomenon; in particular, the
effects of high data contention and low resource utilization predicted in [BALTB2,
TAY84]. There, restarting a transaction upon conflict lead to a better performance
than 2PL.

In regard to the specific algorithms (synchronization policies) examined, the
results of the experiments conducted indicate the following:

1. 02PL usually outperforms 2PL.

2. ICCA1 usually outperform 2PL and 02PL. In environments consisting of short writ-
ers and long readers ICCA1's performance may significantly improved by forcing
writers to wait for long readers as the latter enter the CPU contention area.

3. SG checking algorithm has a reasonable restart rate and performs almost as good
as 2PL. In environments consisting of readers and writers, SC checking improves
the performance of writers at the expanse of readers, relative to 2PL.
## Experiment 1 Classes Mix

<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>37.5</td>
</tr>
<tr>
<td>Long Readers</td>
<td>12.5</td>
</tr>
</tbody>
</table>

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

**THINKtime=5000; RESTARTdelay=200**

<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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.50</td>
<td>323.3</td>
<td>12</td>
<td>1.16</td>
<td>176.6</td>
<td>0</td>
<td>0.28</td>
<td>2032.8</td>
<td>14</td>
<td>1.30</td>
<td>0.14</td>
<td>1.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2.92</td>
<td>410.7</td>
<td>39</td>
<td>2.34</td>
<td>183.4</td>
<td>0</td>
<td>0.56</td>
<td>2175.9</td>
<td>36</td>
<td>1.91</td>
<td>0.28</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>4.29</td>
<td>506.1</td>
<td>54</td>
<td>3.44</td>
<td>238.9</td>
<td>0</td>
<td>0.81</td>
<td>2284.8</td>
<td>77</td>
<td>3.51</td>
<td>0.61</td>
<td>0.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>5.55</td>
<td>650.1</td>
<td>109</td>
<td>4.60</td>
<td>218.5</td>
<td>1</td>
<td>1.07</td>
<td>2511.0</td>
<td>116</td>
<td>4.29</td>
<td>0.55</td>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>6.72</td>
<td>805.4</td>
<td>212</td>
<td>5.68</td>
<td>278.3</td>
<td>1</td>
<td>1.29</td>
<td>2832.8</td>
<td>129</td>
<td>7.10</td>
<td>0.68</td>
<td>0.58</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>7.55</td>
<td>2319.3</td>
<td>521</td>
<td>7.50</td>
<td>599.8</td>
<td>2</td>
<td>1.41</td>
<td>4910.0</td>
<td>316</td>
<td>21.88</td>
<td>0.83</td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>8.50</td>
<td>3410.9</td>
<td>772</td>
<td>8.37</td>
<td>739.2</td>
<td>1</td>
<td>1.31</td>
<td>7238.7</td>
<td>448</td>
<td>30.48</td>
<td>0.87</td>
<td>0.72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Experiment 1 - O2PL Performance A Mix of Short and Long Transactions

**THINKtime=5000; RESTARTdelay=200**

<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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.50</td>
<td>313.2</td>
<td>7</td>
<td>1.16</td>
<td>177.3</td>
<td>0</td>
<td>0.26</td>
<td>2071.3</td>
<td>0</td>
<td>1.29</td>
<td>0.14</td>
<td>1.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2.92</td>
<td>402.7</td>
<td>33</td>
<td>2.34</td>
<td>187.6</td>
<td>0</td>
<td>0.56</td>
<td>2087.3</td>
<td>0</td>
<td>1.02</td>
<td>0.27</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>4.35</td>
<td>518.0</td>
<td>64</td>
<td>3.46</td>
<td>196.9</td>
<td>0</td>
<td>0.86</td>
<td>2156.7</td>
<td>0</td>
<td>2.32</td>
<td>0.41</td>
<td>0.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>5.62</td>
<td>698.3</td>
<td>125</td>
<td>4.59</td>
<td>226.6</td>
<td>0</td>
<td>1.10</td>
<td>2208.5</td>
<td>0</td>
<td>4.21</td>
<td>0.53</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>6.83</td>
<td>883.7</td>
<td>143</td>
<td>5.68</td>
<td>270.7</td>
<td>0</td>
<td>1.34</td>
<td>2446.0</td>
<td>0</td>
<td>6.38</td>
<td>0.65</td>
<td>0.56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>7.68</td>
<td>1246.2</td>
<td>297</td>
<td>7.76</td>
<td>340.4</td>
<td>0</td>
<td>1.55</td>
<td>2746.5</td>
<td>0</td>
<td>10.41</td>
<td>0.76</td>
<td>0.69</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>8.25</td>
<td>1771.4</td>
<td>346</td>
<td>8.66</td>
<td>466.6</td>
<td>0</td>
<td>1.67</td>
<td>3348.0</td>
<td>0</td>
<td>18.32</td>
<td>0.83</td>
<td>0.78</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.11 - O2PL Vs. 2PL in Experiment 1.**
### Table 7.12 - OZPL vs. 2PL in Experiment 2.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.77</td>
<td>386.9</td>
<td>25</td>
<td>2.24</td>
<td>183.8</td>
<td>0</td>
<td>0.44</td>
<td>2053.2</td>
<td>0</td>
<td>1.48</td>
<td>0.24</td>
</tr>
<tr>
<td>32</td>
<td>5.26</td>
<td>545.2</td>
<td>48</td>
<td>4.43</td>
<td>206.3</td>
<td>0</td>
<td>0.85</td>
<td>2190.9</td>
<td>0</td>
<td>2.87</td>
<td>0.47</td>
</tr>
<tr>
<td>48</td>
<td>7.26</td>
<td>804.5</td>
<td>78</td>
<td>6.53</td>
<td>257.8</td>
<td>0</td>
<td>1.22</td>
<td>2492.5</td>
<td>0</td>
<td>5.26</td>
<td>0.96</td>
</tr>
<tr>
<td>64</td>
<td>8.61</td>
<td>1210.5</td>
<td>215</td>
<td>8.58</td>
<td>394.2</td>
<td>0</td>
<td>1.48</td>
<td>2870.3</td>
<td>0</td>
<td>10.08</td>
<td>0.82</td>
</tr>
<tr>
<td>80</td>
<td>8.85</td>
<td>2300.6</td>
<td>414</td>
<td>9.07</td>
<td>533.7</td>
<td>0</td>
<td>1.56</td>
<td>3035.2</td>
<td>0</td>
<td>19.91</td>
<td>0.87</td>
</tr>
<tr>
<td>96</td>
<td>7.22</td>
<td>4125.1</td>
<td>736</td>
<td>10.77</td>
<td>630.0</td>
<td>0</td>
<td>1.57</td>
<td>5152.5</td>
<td>0</td>
<td>33.36</td>
<td>0.87</td>
</tr>
<tr>
<td>112</td>
<td>5.78</td>
<td>7240.6</td>
<td>1162</td>
<td>11.28</td>
<td>1220.0</td>
<td>0</td>
<td>1.45</td>
<td>7181.8</td>
<td>0</td>
<td>53.21</td>
<td>0.82</td>
</tr>
<tr>
<td>128</td>
<td>4.18</td>
<td>17292.4</td>
<td>1428</td>
<td>11.91</td>
<td>1670.1</td>
<td>0</td>
<td>1.38</td>
<td>9179.8</td>
<td>0</td>
<td>73.52</td>
<td>0.74</td>
</tr>
</tbody>
</table>

**Experiment 2 Classes Mix.**

**Table 7.12 - OZPL vs. 2PL in Experiment 2.**
<table>
<thead>
<tr>
<th>Class</th>
<th>% of MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium Writers</td>
<td>50.0</td>
</tr>
<tr>
<td>Medium Readers</td>
<td>50.0</td>
</tr>
</tbody>
</table>

Experiment 3 Classes Mix.

## Experiment 3 - 2PL Performance Medium Transactions

<table>
<thead>
<tr>
<th>MPL</th>
<th>Medium Writers</th>
<th>Medium Readers</th>
<th>Overall</th>
<th>höchst</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5.25</td>
<td>525.1</td>
<td>150</td>
<td>5.79</td>
</tr>
<tr>
<td>32</td>
<td>8.54</td>
<td>874.5</td>
<td>485</td>
<td>10.42</td>
</tr>
<tr>
<td>48</td>
<td>7.22</td>
<td>223.3</td>
<td>1207</td>
<td>11.98</td>
</tr>
<tr>
<td>64</td>
<td>6.05</td>
<td>281.9</td>
<td>2481</td>
<td>13.33</td>
</tr>
<tr>
<td>80</td>
<td>3.25</td>
<td>11045.7</td>
<td>3399</td>
<td>14.61</td>
</tr>
<tr>
<td>96</td>
<td>2.10</td>
<td>2137.7</td>
<td>4176</td>
<td>15.39</td>
</tr>
<tr>
<td>112</td>
<td>1.14</td>
<td>39945.0</td>
<td>4267</td>
<td>16.03</td>
</tr>
<tr>
<td>128</td>
<td>0.07</td>
<td>9327.3</td>
<td>4972</td>
<td>17.14</td>
</tr>
</tbody>
</table>

Table 7.13 - 2PL Vs. 2PL in Experiment 3
# Table 7.14 - O2PL vs. 2PL in Experiment 4

<table>
<thead>
<tr>
<th>Class</th>
<th>% of MPI</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 - 2PL Performance Short Transactions

THINKtime=1000, RESTART delay=250

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Overall</th>
<th>BlockedQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.31</td>
<td>268.0</td>
<td>8</td>
<td>6.72</td>
<td>169.3</td>
</tr>
<tr>
<td>20</td>
<td>12.20</td>
<td>302.2</td>
<td>31</td>
<td>13.10</td>
<td>261.0</td>
</tr>
<tr>
<td>40</td>
<td>16.89</td>
<td>509.4</td>
<td>88</td>
<td>17.45</td>
<td>376.0</td>
</tr>
<tr>
<td>60</td>
<td>12.27</td>
<td>1095.7</td>
<td>172</td>
<td>16.05</td>
<td>772.7</td>
</tr>
<tr>
<td>80</td>
<td>14.54</td>
<td>1758.8</td>
<td>305</td>
<td>18.57</td>
<td>1153.9</td>
</tr>
<tr>
<td>90</td>
<td>13.79</td>
<td>2489.3</td>
<td>459</td>
<td>19.19</td>
<td>1502.2</td>
</tr>
<tr>
<td>120</td>
<td>12.89</td>
<td>3329.9</td>
<td>670</td>
<td>18.80</td>
<td>1822.1</td>
</tr>
</tbody>
</table>

Table 7.14 - O2PL vs. 2PL in Experiment 4.
**Class** | **% of MPL**
--- | ---
Short Writers | 50.0
Short Readers | 37.5
Long Readers | 12.5

**Experiment 1 Classes Mix.**

**Table 7.15 - ICCAI Vs. O2PL in Experiment 1.**

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th></th>
<th></th>
<th>Short Readers</th>
<th></th>
<th></th>
<th>Long Readers</th>
<th></th>
<th></th>
<th>BlockQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.50</td>
<td>333.2</td>
<td>7</td>
<td>1.18</td>
<td>177.3</td>
<td>0</td>
<td>0.28</td>
<td>2071.3</td>
<td>0</td>
<td>1.29</td>
<td>0.14</td>
</tr>
<tr>
<td>32</td>
<td>2.95</td>
<td>422.3</td>
<td>33</td>
<td>2.31</td>
<td>187.6</td>
<td>0</td>
<td>0.56</td>
<td>2087.3</td>
<td>0</td>
<td>1.76</td>
<td>0.27</td>
</tr>
<tr>
<td>48</td>
<td>4.35</td>
<td>516.0</td>
<td>64</td>
<td>3.48</td>
<td>198.9</td>
<td>0</td>
<td>0.84</td>
<td>2156.7</td>
<td>0</td>
<td>2.52</td>
<td>0.41</td>
</tr>
<tr>
<td>64</td>
<td>5.82</td>
<td>605.9</td>
<td>125</td>
<td>4.59</td>
<td>228.6</td>
<td>0</td>
<td>1.10</td>
<td>2285.5</td>
<td>0</td>
<td>4.21</td>
<td>0.53</td>
</tr>
<tr>
<td>80</td>
<td>6.83</td>
<td>685.7</td>
<td>145</td>
<td>5.69</td>
<td>270.7</td>
<td>0</td>
<td>1.34</td>
<td>3446.0</td>
<td>0</td>
<td>6.30</td>
<td>0.65</td>
</tr>
<tr>
<td>96</td>
<td>7.86</td>
<td>748.2</td>
<td>207</td>
<td>6.76</td>
<td>304.0</td>
<td>0</td>
<td>1.55</td>
<td>3745.5</td>
<td>0</td>
<td>10.41</td>
<td>0.76</td>
</tr>
<tr>
<td>112</td>
<td>8.93</td>
<td>877.4</td>
<td>348</td>
<td>7.88</td>
<td>409.8</td>
<td>0</td>
<td>1.81</td>
<td>4296.0</td>
<td>0</td>
<td>16.32</td>
<td>0.83</td>
</tr>
<tr>
<td>128</td>
<td>9.95</td>
<td>995.5</td>
<td>722</td>
<td>8.43</td>
<td>621.6</td>
<td>0</td>
<td>1.88</td>
<td>4623.4</td>
<td>0</td>
<td>26.28</td>
<td>0.95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th></th>
<th></th>
<th>Short Readers</th>
<th></th>
<th></th>
<th>Long Readers</th>
<th></th>
<th></th>
<th>BlockQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.52</td>
<td>257.5</td>
<td>10</td>
<td>1.16</td>
<td>177.8</td>
<td>0</td>
<td>0.28</td>
<td>2143.5</td>
<td>41</td>
<td>1.00</td>
<td>0.14</td>
</tr>
<tr>
<td>32</td>
<td>3.04</td>
<td>281.0</td>
<td>9</td>
<td>2.32</td>
<td>181.7</td>
<td>0</td>
<td>0.56</td>
<td>2231.4</td>
<td>77</td>
<td>1.00</td>
<td>0.28</td>
</tr>
<tr>
<td>48</td>
<td>4.55</td>
<td>286.7</td>
<td>12</td>
<td>3.47</td>
<td>188.8</td>
<td>0</td>
<td>0.82</td>
<td>2346.5</td>
<td>134</td>
<td>1.00</td>
<td>0.42</td>
</tr>
<tr>
<td>64</td>
<td>6.06</td>
<td>270.7</td>
<td>12</td>
<td>4.67</td>
<td>194.6</td>
<td>0</td>
<td>1.01</td>
<td>2410.0</td>
<td>137</td>
<td>1.02</td>
<td>0.56</td>
</tr>
<tr>
<td>80</td>
<td>7.57</td>
<td>280.5</td>
<td>10</td>
<td>5.76</td>
<td>200.0</td>
<td>0</td>
<td>1.33</td>
<td>2566.4</td>
<td>154</td>
<td>1.02</td>
<td>0.70</td>
</tr>
<tr>
<td>96</td>
<td>9.02</td>
<td>322.1</td>
<td>15</td>
<td>6.87</td>
<td>238.1</td>
<td>0</td>
<td>1.61</td>
<td>2697.3</td>
<td>246</td>
<td>1.02</td>
<td>0.84</td>
</tr>
<tr>
<td>112</td>
<td>10.58</td>
<td>454.0</td>
<td>30</td>
<td>7.97</td>
<td>355.4</td>
<td>0</td>
<td>1.83</td>
<td>4107.6</td>
<td>390</td>
<td>1.13</td>
<td>0.97</td>
</tr>
<tr>
<td>128</td>
<td>12.10</td>
<td>481.2</td>
<td>70</td>
<td>8.53</td>
<td>428.0</td>
<td>4</td>
<td>1.25</td>
<td>7112.2</td>
<td>320</td>
<td>1.36</td>
<td>1.00</td>
</tr>
</tbody>
</table>
### Experiment 2 Classes Mix.

#### Experiment 2 - O2PL Performance: A Mix of Short and Long Transactions

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th></th>
<th>Short Readers</th>
<th></th>
<th>Long Readers</th>
<th></th>
<th>BlockedQ</th>
<th></th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.77</td>
<td>358.9</td>
<td>25</td>
<td>2.24</td>
<td>183.8</td>
<td>0</td>
<td>0.44</td>
<td>2083.2</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>5.25</td>
<td>545.2</td>
<td>68</td>
<td>4.43</td>
<td>206.3</td>
<td>0</td>
<td>0.65</td>
<td>2190.9</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>7.26</td>
<td>804.5</td>
<td>176</td>
<td>6.53</td>
<td>227.8</td>
<td>0</td>
<td>1.22</td>
<td>2430.6</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>8.61</td>
<td>1215.5</td>
<td>215</td>
<td>8.58</td>
<td>264.2</td>
<td>0</td>
<td>1.48</td>
<td>2876.3</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>8.35</td>
<td>2206.8</td>
<td>414</td>
<td>9.67</td>
<td>538.7</td>
<td>0</td>
<td>1.56</td>
<td>3603.2</td>
<td>0</td>
</tr>
<tr>
<td>112</td>
<td>7.72</td>
<td>4125.1</td>
<td>739</td>
<td>10.77</td>
<td>836.0</td>
<td>0</td>
<td>1.57</td>
<td>5129.2</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>5.78</td>
<td>7240.9</td>
<td>1162</td>
<td>11.28</td>
<td>1222.9</td>
<td>0</td>
<td>1.45</td>
<td>7161.8</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th></th>
<th>Short Readers</th>
<th></th>
<th>Long Readers</th>
<th></th>
<th>BlockedQ</th>
<th></th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.90</td>
<td>232.3</td>
<td>6</td>
<td>2.24</td>
<td>180.3</td>
<td>1</td>
<td>0.42</td>
<td>2223.0</td>
<td>42</td>
</tr>
<tr>
<td>32</td>
<td>5.78</td>
<td>206.5</td>
<td>0</td>
<td>4.40</td>
<td>191.2</td>
<td>1</td>
<td>0.61</td>
<td>2412.8</td>
<td>119</td>
</tr>
<tr>
<td>48</td>
<td>8.61</td>
<td>228.9</td>
<td>28</td>
<td>6.54</td>
<td>206.3</td>
<td>0</td>
<td>1.15</td>
<td>2628.2</td>
<td>191</td>
</tr>
<tr>
<td>64</td>
<td>8.13</td>
<td>374.7</td>
<td>49</td>
<td>8.54</td>
<td>277.8</td>
<td>0</td>
<td>1.36</td>
<td>3388.8</td>
<td>232</td>
</tr>
<tr>
<td>96</td>
<td>12.22</td>
<td>773.4</td>
<td>74</td>
<td>9.77</td>
<td>571.2</td>
<td>2</td>
<td>1.00</td>
<td>6656.8</td>
<td>371</td>
</tr>
<tr>
<td>112</td>
<td>12.42</td>
<td>1305.1</td>
<td>123</td>
<td>10.22</td>
<td>999.1</td>
<td>3</td>
<td>0.85</td>
<td>22231.4</td>
<td>726</td>
</tr>
<tr>
<td>128</td>
<td>12.41</td>
<td>2014.5</td>
<td>226</td>
<td>10.77</td>
<td>1398.6</td>
<td>2</td>
<td>0.18</td>
<td>61689.5</td>
<td>908</td>
</tr>
</tbody>
</table>

Table 7.16 - ICCA1 vs. O2PL in Experiment 2.
### Table 7.17 - ICCAl Vs. O2PL in Experiment 3.

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

#### Experiment 3 - O2PL Performance Medium Transactions

<table>
<thead>
<tr>
<th>MPL</th>
<th>Medium Writers</th>
<th>Medium Readers</th>
<th>Overall</th>
<th>BlockedQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6.32</td>
<td>502.7</td>
<td>143</td>
<td>5.86</td>
<td>368.4</td>
</tr>
<tr>
<td>32</td>
<td>9.01</td>
<td>776.3</td>
<td>365</td>
<td>10.86</td>
<td>498.5</td>
</tr>
<tr>
<td>48</td>
<td>7.82</td>
<td>2002.8</td>
<td>631</td>
<td>12.21</td>
<td>967.6</td>
</tr>
<tr>
<td>64</td>
<td>6.04</td>
<td>4294.7</td>
<td>1249</td>
<td>14.22</td>
<td>1252.7</td>
</tr>
<tr>
<td>80</td>
<td>5.89</td>
<td>7324.0</td>
<td>1742</td>
<td>16.80</td>
<td>1380.7</td>
</tr>
<tr>
<td>96</td>
<td>1.87</td>
<td>18500.0</td>
<td>2241</td>
<td>18.23</td>
<td>1651.3</td>
</tr>
<tr>
<td>112</td>
<td>1.09</td>
<td>45177.3</td>
<td>2565</td>
<td>18.88</td>
<td>1816.9</td>
</tr>
<tr>
<td>128</td>
<td>0.87</td>
<td>57944.5</td>
<td>2813</td>
<td>20.00</td>
<td>2105.3</td>
</tr>
</tbody>
</table>
### Experiment 4 - O2PL Performance Short Transactions

**THINKtime=1000; RESTARTdelay=200**

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Overall</th>
<th>HocksQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6.31</td>
<td>387.5</td>
<td>10</td>
<td>8.72</td>
<td>150.5</td>
</tr>
<tr>
<td>32</td>
<td>12.30</td>
<td>381.3</td>
<td>12</td>
<td>13.67</td>
<td>233.3</td>
</tr>
<tr>
<td>64</td>
<td>15.90</td>
<td>308.2</td>
<td>77</td>
<td>14.42</td>
<td>377.5</td>
</tr>
<tr>
<td>128</td>
<td>15.28</td>
<td>1094.2</td>
<td>109</td>
<td>13.03</td>
<td>774.1</td>
</tr>
<tr>
<td>256</td>
<td>14.57</td>
<td>1741.1</td>
<td>202</td>
<td>18.52</td>
<td>1159.1</td>
</tr>
<tr>
<td>512</td>
<td>13.86</td>
<td>2453.4</td>
<td>471</td>
<td>18.95</td>
<td>1528.6</td>
</tr>
<tr>
<td>1024</td>
<td>12.98</td>
<td>3239.9</td>
<td>570</td>
<td>19.66</td>
<td>1802.1</td>
</tr>
</tbody>
</table>

### Experiment 4 - ICAI Performance Short Transactions

**THINKtime=1000; RESTARTdelay=250**

<table>
<thead>
<tr>
<th>MPL</th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Overall</th>
<th>HocksQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6.33</td>
<td>264.8</td>
<td>7</td>
<td>6.73</td>
<td>189.0</td>
</tr>
<tr>
<td>32</td>
<td>12.32</td>
<td>268.2</td>
<td>23</td>
<td>13.14</td>
<td>217.7</td>
</tr>
<tr>
<td>64</td>
<td>15.90</td>
<td>510.1</td>
<td>95</td>
<td>17.40</td>
<td>374.0</td>
</tr>
<tr>
<td>128</td>
<td>15.28</td>
<td>1020.0</td>
<td>139</td>
<td>16.10</td>
<td>785.0</td>
</tr>
<tr>
<td>256</td>
<td>14.57</td>
<td>1890.3</td>
<td>226</td>
<td>18.68</td>
<td>1140.5</td>
</tr>
<tr>
<td>512</td>
<td>13.86</td>
<td>3104.4</td>
<td>381</td>
<td>19.97</td>
<td>1818.0</td>
</tr>
<tr>
<td>1024</td>
<td>12.99</td>
<td>4083.3</td>
<td>554</td>
<td>21.03</td>
<td>2348.3</td>
</tr>
</tbody>
</table>

### Table 7.18 - ICAI Vs. O2PL in Experiment 4
<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>37.5</td>
</tr>
<tr>
<td>Long Readers</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Experiment 1 Classes Mix.

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

- **THINKtime=5000, RESTARTdelay=250**

<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>18</td>
<td>1.50</td>
<td>323.3</td>
<td>12</td>
<td>1.18</td>
<td>178.0</td>
</tr>
<tr>
<td>22</td>
<td>2.98</td>
<td>410.7</td>
<td>39</td>
<td>2.31</td>
<td>189.4</td>
</tr>
<tr>
<td>48</td>
<td>4.29</td>
<td>698.1</td>
<td>54</td>
<td>2.44</td>
<td>238.9</td>
</tr>
<tr>
<td>94</td>
<td>5.65</td>
<td>860.1</td>
<td>109</td>
<td>4.00</td>
<td>210.0</td>
</tr>
<tr>
<td>96</td>
<td>6.73</td>
<td>985.4</td>
<td>212</td>
<td>5.68</td>
<td>278.3</td>
</tr>
<tr>
<td>98</td>
<td>7.73</td>
<td>1224.7</td>
<td>291</td>
<td>6.77</td>
<td>328.3</td>
</tr>
<tr>
<td>112</td>
<td>7.95</td>
<td>2319.3</td>
<td>521</td>
<td>7.50</td>
<td>599.8</td>
</tr>
<tr>
<td>128</td>
<td>7.60</td>
<td>3452.9</td>
<td>772</td>
<td>8.87</td>
<td>726.2</td>
</tr>
</tbody>
</table>

### Experiment 1 - CERT Performance: A Mix of Short and Long Transactions

- **THINKtime=5000, RESTARTdelay=250**

<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>18</td>
<td>1.52</td>
<td>294.3</td>
<td>12</td>
<td>1.10</td>
<td>177.9</td>
</tr>
<tr>
<td>22</td>
<td>3.04</td>
<td>285.3</td>
<td>35</td>
<td>2.31</td>
<td>194.3</td>
</tr>
<tr>
<td>48</td>
<td>4.55</td>
<td>277.3</td>
<td>44</td>
<td>3.47</td>
<td>193.3</td>
</tr>
<tr>
<td>94</td>
<td>6.06</td>
<td>286.0</td>
<td>44</td>
<td>4.61</td>
<td>203.7</td>
</tr>
<tr>
<td>96</td>
<td>7.54</td>
<td>303.7</td>
<td>70</td>
<td>5.74</td>
<td>220.8</td>
</tr>
<tr>
<td>98</td>
<td>8.97</td>
<td>346.1</td>
<td>59</td>
<td>6.84</td>
<td>258.9</td>
</tr>
<tr>
<td>112</td>
<td>10.04</td>
<td>579.1</td>
<td>144</td>
<td>7.73</td>
<td>434.0</td>
</tr>
<tr>
<td>128</td>
<td>10.48</td>
<td>1144.4</td>
<td>285</td>
<td>8.21</td>
<td>603.9</td>
</tr>
</tbody>
</table>

Table 7.19 - CERT Vs. 2PL in Experiment 1.
<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>37.5</td>
</tr>
<tr>
<td>Long Readers</td>
<td>12.5</td>
</tr>
</tbody>
</table>

**Experiment 2 Classes Mix.**

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

**THINKtime=2500; RESTARTdelay=250**

<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></td>
<td>msec.</td>
<td>arted</td>
<td></td>
<td>msec.</td>
</tr>
<tr>
<td>10</td>
<td>2.76</td>
<td>376.6</td>
<td>16</td>
<td>2.23</td>
<td>185.0</td>
</tr>
<tr>
<td>32</td>
<td>5.29</td>
<td>549.5</td>
<td>71</td>
<td>4.44</td>
<td>203.8</td>
</tr>
<tr>
<td>48</td>
<td>7.52</td>
<td>774.5</td>
<td>140</td>
<td>6.53</td>
<td>258.7</td>
</tr>
<tr>
<td>64</td>
<td>8.54</td>
<td>1022.9</td>
<td>287</td>
<td>8.38</td>
<td>385.9</td>
</tr>
<tr>
<td>96</td>
<td>8.88</td>
<td>2330.0</td>
<td>589</td>
<td>9.79</td>
<td>582.0</td>
</tr>
<tr>
<td>128</td>
<td>7.03</td>
<td>4511.9</td>
<td>773</td>
<td>10.20</td>
<td>1025.5</td>
</tr>
</tbody>
</table>

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

**THINKtime=2500; RESTARTdelay=250**

<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></td>
<td>msec.</td>
<td>arted</td>
<td></td>
<td>msec.</td>
</tr>
<tr>
<td>10</td>
<td>2.89</td>
<td>285.8</td>
<td>23</td>
<td>2.24</td>
<td>182.1</td>
</tr>
<tr>
<td>32</td>
<td>5.76</td>
<td>278.3</td>
<td>35</td>
<td>4.45</td>
<td>197.5</td>
</tr>
<tr>
<td>48</td>
<td>6.58</td>
<td>302.6</td>
<td>67</td>
<td>8.61</td>
<td>239.0</td>
</tr>
<tr>
<td>64</td>
<td>10.82</td>
<td>457.9</td>
<td>119</td>
<td>8.44</td>
<td>343.9</td>
</tr>
<tr>
<td>96</td>
<td>11.45</td>
<td>983.3</td>
<td>285</td>
<td>9.28</td>
<td>741.5</td>
</tr>
<tr>
<td>128</td>
<td>11.81</td>
<td>1035.7</td>
<td>426</td>
<td>9.61</td>
<td>1203.5</td>
</tr>
<tr>
<td>112</td>
<td>11.71</td>
<td>2282.7</td>
<td>641</td>
<td>10.10</td>
<td>1681.4</td>
</tr>
</tbody>
</table>

**Table 7.20 - CERT Vs. 2PL in Experiment 2.**
### Experiment 3 Class's Mix.

#### Table 7.21 - CERT Vs. 2PL in Experiment 3.
### Table 7.22 - CEKT Vs. 2PL in Experiment 4.

<table>
<thead>
<tr>
<th>M/L</th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Overall</th>
<th>Blocked</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Commit</td>
<td>Res Time</td>
<td>Rest</td>
<td>Commit</td>
<td>Res Time</td>
</tr>
<tr>
<td>16</td>
<td>6.80</td>
<td>269.0</td>
<td>6</td>
<td>6.72</td>
<td>188.3</td>
</tr>
<tr>
<td>32</td>
<td>12.29</td>
<td>302.2</td>
<td>31</td>
<td>13.10</td>
<td>221.9</td>
</tr>
<tr>
<td>48</td>
<td>15.89</td>
<td>259.4</td>
<td>86</td>
<td>17.45</td>
<td>376.0</td>
</tr>
<tr>
<td>64</td>
<td>15.27</td>
<td>1095.7</td>
<td>172</td>
<td>16.65</td>
<td>772.7</td>
</tr>
<tr>
<td>80</td>
<td>14.54</td>
<td>1763.6</td>
<td>505</td>
<td>15.57</td>
<td>1133.9</td>
</tr>
<tr>
<td>96</td>
<td>13.78</td>
<td>2489.3</td>
<td>459</td>
<td>15.18</td>
<td>1502.6</td>
</tr>
<tr>
<td>120</td>
<td>12.95</td>
<td>3329.9</td>
<td>670</td>
<td>16.80</td>
<td>1828.1</td>
</tr>
</tbody>
</table>

**Note:** The table above shows the performance metrics for CEKT vs. 2PL in Experiment 4 for different M/L values. The metrics include commit rate, res. time, rest. rate, committed transactions, res. time, rest. rate, overall committed transactions, res. time, rest. rate, blocked transactions, and CPU utilization.
### Table 7.23 - DYNAMIC Vs. ICCA1 in Experiment 1.

<table>
<thead>
<tr>
<th></th>
<th>Short Writers</th>
<th>Short Readers</th>
<th>Long Readers</th>
<th>BlockesQ</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.52</td>
<td>257.5</td>
<td>10</td>
<td>1.16</td>
<td>177.7</td>
</tr>
<tr>
<td>32</td>
<td>3.04</td>
<td>261.4</td>
<td>9</td>
<td>2.32</td>
<td>191.7</td>
</tr>
<tr>
<td>48</td>
<td>4.55</td>
<td>286.7</td>
<td>12</td>
<td>3.47</td>
<td>188.4</td>
</tr>
<tr>
<td>64</td>
<td>6.06</td>
<td>276.7</td>
<td>24</td>
<td>4.62</td>
<td>194.8</td>
</tr>
<tr>
<td>90</td>
<td>7.57</td>
<td>290.5</td>
<td>10</td>
<td>5.76</td>
<td>230.6</td>
</tr>
<tr>
<td>98</td>
<td>9.02</td>
<td>352.1</td>
<td>15</td>
<td>6.67</td>
<td>259.1</td>
</tr>
<tr>
<td>112</td>
<td>10.28</td>
<td>454.0</td>
<td>30</td>
<td>7.87</td>
<td>334.6</td>
</tr>
<tr>
<td>128</td>
<td>10.85</td>
<td>441.2</td>
<td>70</td>
<td>8.53</td>
<td>626.0</td>
</tr>
</tbody>
</table>
## Class % of MPL

| Short Writers | 50.0 |
| Short Readers | 37.5 |
| Long Readers  | 12.5 |

**Experiment 2 Classes Mix.**

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

**THINKtime:=250; RESTART-delay:=250**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0.92</td>
<td>2.90</td>
<td>259.3</td>
<td>4</td>
<td>2.24</td>
<td>180.3</td>
<td>1</td>
<td>0.42</td>
<td>222.0</td>
<td>0.25</td>
</tr>
<tr>
<td>32</td>
<td>0.92</td>
<td>5.78</td>
<td>268.3</td>
<td>6</td>
<td>4.48</td>
<td>191.2</td>
<td>1</td>
<td>0.81</td>
<td>212.0</td>
<td>1.07</td>
</tr>
<tr>
<td>48</td>
<td>0.61</td>
<td>8.61</td>
<td>268.1</td>
<td>28</td>
<td>8.64</td>
<td>200.3</td>
<td>0</td>
<td>1.15</td>
<td>206.4</td>
<td>0.74</td>
</tr>
<tr>
<td>64</td>
<td>11.19</td>
<td>11.19</td>
<td>374.7</td>
<td>49</td>
<td>8.84</td>
<td>277.8</td>
<td>0</td>
<td>1.30</td>
<td>336.8</td>
<td>0.95</td>
</tr>
<tr>
<td>80</td>
<td>12.22</td>
<td>12.22</td>
<td>773.4</td>
<td>74</td>
<td>9.77</td>
<td>571.2</td>
<td>2</td>
<td>1.08</td>
<td>695.8</td>
<td>0.90</td>
</tr>
<tr>
<td>96</td>
<td>12.42</td>
<td>12.42</td>
<td>1355.1</td>
<td>123</td>
<td>10.28</td>
<td>696.1</td>
<td>3</td>
<td>0.45</td>
<td>225.1</td>
<td>0.96</td>
</tr>
<tr>
<td>112</td>
<td>12.41</td>
<td>12.41</td>
<td>2014.5</td>
<td>225</td>
<td>10.77</td>
<td>1986.0</td>
<td>2</td>
<td>0.18</td>
<td>618.6</td>
<td>3.33</td>
</tr>
<tr>
<td>128</td>
<td>12.47</td>
<td>12.47</td>
<td>2255.3</td>
<td>364</td>
<td>11.23</td>
<td>1728.0</td>
<td>2</td>
<td>0.08</td>
<td>927.8</td>
<td>4.75</td>
</tr>
</tbody>
</table>

### Experiment 2 - DYNAMIC Performance A Mix of Short and Long Transactions

**THINKtime:=250; RESTART-delay:=250**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0.92</td>
<td>2.90</td>
<td>260.1</td>
<td>9</td>
<td>2.24</td>
<td>191.4</td>
<td>1</td>
<td>0.42</td>
<td>2216.4</td>
<td>0.00</td>
</tr>
<tr>
<td>32</td>
<td>0.92</td>
<td>5.78</td>
<td>269.3</td>
<td>13</td>
<td>4.48</td>
<td>191.9</td>
<td>1</td>
<td>0.82</td>
<td>2397.8</td>
<td>0.00</td>
</tr>
<tr>
<td>48</td>
<td>0.61</td>
<td>8.61</td>
<td>260.5</td>
<td>24</td>
<td>8.64</td>
<td>211.6</td>
<td>3</td>
<td>1.15</td>
<td>2873.5</td>
<td>0.00</td>
</tr>
<tr>
<td>64</td>
<td>11.19</td>
<td>11.19</td>
<td>374.4</td>
<td>25</td>
<td>8.83</td>
<td>279.8</td>
<td>1</td>
<td>1.34</td>
<td>3516.4</td>
<td>0.00</td>
</tr>
<tr>
<td>80</td>
<td>12.22</td>
<td>12.22</td>
<td>1700.0</td>
<td>336</td>
<td>10.38</td>
<td>384.1</td>
<td>1</td>
<td>1.70</td>
<td>3401.0</td>
<td>0.00</td>
</tr>
<tr>
<td>96</td>
<td>12.42</td>
<td>12.42</td>
<td>2800.1</td>
<td>454</td>
<td>11.84</td>
<td>503.5</td>
<td>1</td>
<td>1.66</td>
<td>487.0</td>
<td>0.00</td>
</tr>
<tr>
<td>112</td>
<td>12.41</td>
<td>12.41</td>
<td>4308.1</td>
<td>618</td>
<td>12.90</td>
<td>755.8</td>
<td>4</td>
<td>1.72</td>
<td>564.9</td>
<td>0.00</td>
</tr>
<tr>
<td>128</td>
<td>12.47</td>
<td>12.47</td>
<td>7230</td>
<td>844</td>
<td>14.32</td>
<td>850.7</td>
<td>1</td>
<td>1.82</td>
<td>616.3</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 7.24 - DYNAMIC Vs. ICCA1 in Experiment 2.
CHAPTER 8

CONCLUSIONS

8.1. Private Workspace Model Advantages

Three main advantages follow from the fact that the writing of new values into the private workspace of transactions do not affect the database state:

(1) Transactions are not subject to cascading restarts.

(2) There is no need to undo updates made by an incomplete transaction, therefore recovery may be achieved by using only redo operations; this simplifies the recovery problem.

(3) The concurrency control has complete freedom in choosing how to use PRE_Writes in synchronizing conflicting operations. This enables certain optimizations; for example:
   (a) The 2PL derivative (optimized-2PL-ET-rw) which includes no ww-synchronization.
   (b) The Optimized 2PL (02PL) algorithm in which there is never a restart due to a READ request or because of conflicting WRITE requests.
   (c) ICCA2 which can model a transaction system employing various synchronization policies, ranging from strict locking (e.g. 2PL) to pure restart (e.g. SC checking); allocation of a synchronization policy to a transaction may be done dynamically according to the system state.

8.2. Private Workspace Model Feasibility

The main difficulty in practically using the model is in efficiently implementing the atomic commit phase. The parallel commit phase algorithm introduced in this work
performs I/O operations outside a critical section of the TM while still detecting only "real" serialization conflicts. Aside from being a solution to the problem of atomic commit, the algorithm enhances system performance. Immediately after queuing its commit request and issuing its updates, a committing transaction which is not yet committed may unblock all transactions blocked by it. The transaction is completed as soon as its commit record is known to reside in stable storage.

8.3. Simulation Study Results

The results of the experiments conducted support the argument that the selection of an efficient concurrency control algorithm for a DBMS depends on both system and resource utilization. For example, studies made by Carey [CARE83] of an I/O bound system suggest that the best performance is achieved by minimizing the number of restarts. My study, of a CPU bound system, suggests that controlling the blocking level has a much stronger impact on system performance than attempting to minimize the number of restarts. Moreover, different synchronization policies induce different performance characteristics on different transaction classes.

These results emphasize the need for a concurrency control mechanism that has the ability to adapt itself to various environments in order to maximize throughput. The experiments in using the ICCA concept for tuning the concurrency control of a given transaction system indicate that ICCAs can form a basis for the design of such adaptive mechanisms. ICCA advantages stem from the ability to control the concurrency level in the system and in dynamically matching transactions (or in general transaction classes) with appropriate synchronization techniques as dictated by performance objectives.
APPENDIX A - PARALLELISM OF DM OPERATIONS

The logical relationship between the TM and the DM is described in Figure A.1. The TM accepts an input schedule of READ and WRITE requests from transactions and produces a serializable schedule of DM_READ and DM_WRITE operations which are processed by the DM on a FCFS basis.

<table>
<thead>
<tr>
<th>Input Schedule of READ,WRITE requests</th>
<th>Serializable Schedule of DM_READ,DM_WRITE operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TM</td>
</tr>
</tbody>
</table>

Figure A.1 The TM-DM logical relationship

Serially processing DM operations is perfectly suitable for a system having a single I/O server. However, when multiple I/O servers are available (which is often the case), the potential parallel processing capability of the DM should be exploited. Since conflicting DM operations must still be executed serially, it is incorrect to process the DM input schedule in parallel since this may violate serialization order of transactions.

For example, let schedule $S = r_k[X] w_i[X] w_j[Y] w_j[X] r_1[Y]$ be input to the DM. We may split $S$ into two disjoint schedules: $S_X$ consisting of operations related to data item $X$ and $S_Y$ consisting of operations related to data item $Y$

- $S_X = r_k[X] w_i[X] w_j[X]$  
- $S_Y = w_j[Y] r_1[Y]$  

By itself each of the schedules $S_X$ and $S_Y$ must be processed serially; however, $S_X$ and $S_Y$ may be processed in parallel.
To accommodate the DM parallel processing capability a new system component, the Parallelizer, is introduced. The Parallelizer uses the above splitting technique to convert the serial schedule which is output by the TM into a parallelized schedule in which no conflicting operations are concurrently scheduled, i.e., it will enable the execution of non-conflicting operations in parallel while maintaining the serial execution of conflicting operations (see Figure A.2). The parallelizer function may be considered as a "second phase concurrency control" embedded in the DM.

**Figure A.2 The TM-Parallelizer-DM logical relationship**

Note that when the concurrency control is based on standard 2PL, i.e., 2PL-ET-rw and 2PL-ET-ww and transactions indicators are released only after transactions updates have been processed by the DM, the TM does not schedule conflicting operations concurrently i.e., its schedule is already parallelized and thus the Parallelizer function is redundant. A similar effect is achieved by the strict checking in Kung & Robinson's serial validation algorithm [KUNG81] where a transaction conflicting with a committing transaction is subsequently restarted and thus the parallelizer function is redundant.

**A.1. The Parallelizer Algorithm**

The parallelizer maintains a table of queues (Q) indexed by data items in which the DM operations output by the TM are queued. The queue for data item X, Q[X], is a FIFO list of pairs <OPERATION, TRANSACTION IDENTIFIER> which identifies operations on
data item X.

The exact state of data item X is determined using the following information:

- \( Q[X].\text{busy} \) - true if the DM is still processing some \( O_i[X] \) operation and false otherwise.
- \( Q[X].\text{readers} \) - the number of \( R_i[X] \) operations that are currently being processed by the DM.

A \( DM._\text{WRITE}(X) \) operation is in progress (at the DM) exactly when \( Q[X].\text{readers}=0 \) and \( Q[X].\text{busy} \) is true.

The Parallelizer senses two events: the \textit{arrival} of a new \( O_i[X] \) operation from the TM and the \textit{completion} of an \( O_i[X] \) operation at the DM. Each event triggers the queue processing for the data item X.

The following procedure implements the Parallelizer.

```plaintext
procedure Parallelizer(event, \( O_i[X] \));
begin
  case event of
  begin
    arrival : begin Add \(<0,i>\) to Tail of \( Q[X] \);
      ProcessQueue(X);
    end;
    completion : begin if \( O=\"r\" \) then \( Q[X].\text{readers} \leftarrow Q[X].\text{readers}-1; \)
      if \( Q[X].\text{readers}=0 \) then \( Q[X].\text{busy} \leftarrow false; \)
      ProcessQueue(X);
    end;
    end
  end
end
```

Procedure \( \text{ProcessQueue} \) is a recursive procedure which schedules a new DM operation from the head of the queue for the data item given as a parameter. A new operation is scheduled only after previous conflicting operations in that queue have already been processed by the DM.
procedure ProcessQueue( Q[X] )
begin
  if Q[X] is not empty then
    case OPERATION on Head of Q[X] of
      begin
        "r": if ( not Q[X].busy ) or ( Q[X].readers >0 ) then
          begin
            Q[X].readers <- Q[X].readers+1;
            < O,i > <- remove Top of Q[X];
            Q[X].busy <- true;
            Output (O_i [X]);
            ProcessQueue( Q[X] );
          end;
        "w": if ( not Q[X].busy ) then
          begin
            < O,i > <- remove Top of Q[X];
            Q[X].busy <- true;
            Output (O_i [X]);
          end;
      end
    end
end

A.2. Canceling redundant DM_WRITEs

An optimization which avoids overwriting a data item by redundant DM_WRITE operations, may be implemented by the Parallelizer.

The formal definition of a redundant DM_WRITE operation is given below.

Definition A.1:

Let S be a schedule of DM operations output by the TM and let S_X be the projection of S on X, obtained by deleting from S all operations not on data item X. A w_i [X] operation in S is redundant if the the operation following it in S_X is some w_j[X] operations.

For example, let S_X = r_k[X] w_i[X] w_j [X] be the schedule of DM operations associated with data item X. Since w_j[X] will subsequently overwrite w_i[X], and there is no DM_READ operation in between, w_i[X] may be ignored (recall that all DM_WRITEs are committed operations and thus w_j[X] is bound to be processed).
Cancellation of redundant DM_WRITEs may be implemented as follows. Whenever a $w_j[X]$ operation is to be added to the queue for X and the last operation in the queue is some $w_i[X]$ operation, we may simply replace $w_i[X]$ by $w_j[X]$. The new Parallelizer version below implements this idea.

```pascal
procedure Parallelizer(event', O_i[X]);
begin
  case event of
  begin
    arrival: begin <O_b,I_b> <- Tail of Q[X];
      if (O="w") and (O_b="w")
        then replace Tail of Q[X] with <"w",i>
      else Add <O,I> to Tail of Q[X];
      ProcessQueue(X);
    end;
    completion: begin if O="r" then Q[X].readers <- Q[X].readers-1;
      if Q[X].readers=0 then Q[X].busy <- false;
      ProcessQueue(X);
    end;
  end
end
```

**Figure A.3** The Parallelizer version supporting redundant DM_WRITEs cancellation.
THE SIMULATOR SPECIFICATION LANGUAGE AND A SAMPLE RUN

B.1 The Simulator Specification Language Grammar

<Statement> ::= <Comment> | <Set Cmd> | <Def Cmd> | <Map Cmd> | <Add Cmd> | <Run Cmd> | <Show Cmd> | clear | help | stop

<Comment> ::= # <character string>

<Set Cmd> ::= set <parm> = <integer> { + <integer> } { set DELAY=<DelayType> }

<Def Cmd> ::= def <Class Id> readset=<Set Dist> { writeset=<Set Dist> Prw = <real> }

<Map Cmd> ::= map <Class Id>: <Map Function> WFS=<Wait For Set>

<Add Cmd> ::= add <integer>: <Class Id>

<Run Cmd> ::= run <integer>

<Show Cmd> ::= show <subject>

<parm> ::= DB | LOG | DEBUG | PRINT | <Physical Parm>

<Physical parm> ::= <entry from Table 7.1>

<DelayType> ::= fixed | dynamic

<Set Dist> ::= <Dist Function>(<integer>,<integer>)

<Dist Function> ::= unif | exp | seq | null

<Map Function> ::= <rw synch>+<ww synch> | null

<rw synch> ::= 2PL-ET-rw | CERT-ET-rw | CERT-CT-ww

<ww synch> ::= 2PL-ET-ww | CERT-CT-ww

<Wait For Set> ::= strict2PL | std2PL | optimized | null | dynamic.<integer>

<subject> ::= ENV | SG | TOTALS |QUEUES | UTILIZATION | PARMS

<Class Id> ::= <character string>
B.2 A Sample Experiment Specification

`ICTURE algorithm - A mix of Short and Long Transactions`
`Readers use SG checking whereas Writers use standard 2PL and do not wait for Readers`

set LOG=0
set PRINT=0
set DEBUG=0
set DELAY=fixed

`Set Physical Parameters`

set ThinkTime=2500
set DelayTime=250
set STARTUPcpu=5
set STARTUPio=0
set LOCALcpu=1
set LOCALio=0
set CCinit=1
set CCconflict=1
set CCcommit=1
set CCcleanup=1
set DMcpu=1
set DMio=30
set PREWRITEcpu=0
set PREWRITEio=0
set DMPOSTcpu=1
set DMPOSTio=28+2
set DMCOMMITio=30
show PARMS

`Define Transactions Workload`

map Writers:2PL-ET-rw+2PL-ET-ww WFS=std2PL
map ReaderL:CERT-ET-rw+2PL-ET-ww WFS=null
map ReaderS:CERT-ET-rw+2PL-ET-ww WFS=null

`RUN - MPL=16`

add 8:Writers
add 6:ReaderS
add 2:ReaderL
"
# Initiate Simulation and Cleanup
#
set PRINT=0
show ENV
run 1000
clear
#
# Start Running
#
set PRINT=5
run 10000
show SG
show UTILIZATION
show QUEUES
clear
stop
B.3 A Sample Experiment Run

Integrated Concurrency Control Simulator - Version 4.2 May 85

Israel Gold
C.Sc Dep. - Technion I.I.T.

```plaintext
> ICCA algorithm - A mix of Short and Long Transactions
> whereas Readers use SG checking
>
> set LOG=0
> set PRINT=0
> set DEBUG=0
> set DELAY=fixed

> Set Physical Parameters

> set ThinkTime=2500
> set DelayTime=250
> set STARTUpcpu=5
> set STARTUpio=0
> set LOCALcpu=1
> set LOCALio=0
> set CCinit=1
> set CCconflict=1
> set CCcommit=1
> set CCrecovery=1
> set DMcpu=1
> set DHie=30
> set PREWRITEcpu=0
> set PREWRITEio=0
> set DMPOSTcpu=1
> set DMPOSTio=28+2
> set DMCOMMITio=30
> show PARMS

VALUES OF PHYSICAL PARAMETERS

STARTUpcpu =  5
STARTUpio =  0
LOCALcpu =  1
LOCALio =  0
CCinit =  1
CCconflict =  1
CCcommit =  1
DCpu =  1
DHie =  30
PREWRITEcpu=  0
PREWRITEio =  0
DMPOSTcpu =  1
DMPOSTio =  28+2
DMCOMMITcpu =  1
DMCOMMITio =  30
ThinkTime =  2500
DelayTime =  250
```
> DEFINE TRANSACTIONS WORKLOAD
>*****************************************************************************
>\def Writers readset=unif(4,6) writeset=unif(2.4) Prw=0.50
>\def ReaderS readset=unif(4,6) writeset=null
>\def ReaderL readset=seq(62,66) writeset=null
>
>\map Writers:2PL-ET-rw+2PL-ET-ww WFS=std2PL
>\map ReaderL:CERT-ET-rw+2PL-ET-ww WFS=null
>\map ReaderS:CERT-ET-rw+2PL-ET-ww WFS=null
>
>*****************************************************************************
>\# RUN MPL=16
>
>\add 8:Writers
>\add 6: ReaderS
>\add 2: ReaderL
>\# Initiate Simulation and Cleanup
>\#
>\set PRINT=0
>show ENV

TIME = 0.00

DB = 1024
MPL = 16
LOG = 0
PRINT = 0
DELAY = fixed

CLASS Term. %MPL ReadSet WriteSet Prw map-function WFS

Writers 8 50.0 unif(4,6) unif(2,4) 0.50 2PL-ET-rw+2PL-ET-ww std2PL
ReaderS 6 37.5 unif(4,6) null(0,0) 0.00 CERT-ET-rw+2PL-ET-ww null
ReaderL 2 12.5 seq(62,66) null(0,0) 0.00 CERT-ET-rw+2PL-ET-ww null

>run 1000

TIME = 182091.00

CLASS exec- acti- bloc- aber- comm- Throu- Mean Mean Harm.
uted ked ted ited gput

Writers 531 8 0 2 521 2.86 272.7 1.5 1.1
ReaderS 409 6 0 0 403 2.21 184.5 0.0 1.1
ReaderL 84 2 0 5 77 0.42 2201.5 120.5 1.1
TOTAL 1024 16 0 7 1001 5.49 385.6 10.0 1.1

>clear
>\#
>\# Start Running
>\#
>\set PRINT=5
>\run 10000
### TIME = 542054.00

<table>
<thead>
<tr>
<th>CLASS</th>
<th>executed</th>
<th>active</th>
<th>blocked</th>
<th>aborted</th>
<th>committed</th>
<th>Throughput</th>
<th>Mean RestTime</th>
<th>Mean LastTime</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writers</td>
<td>1052</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1043</td>
<td>2.90</td>
<td>260.8</td>
<td>0.4</td>
<td>1.1</td>
</tr>
<tr>
<td>Readers</td>
<td>813</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>807</td>
<td>2.24</td>
<td>160.3</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadersL</td>
<td>168</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>150</td>
<td>0.42</td>
<td>2274.5</td>
<td>196.9</td>
<td>1.1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2033</td>
<td>16</td>
<td>0</td>
<td>17</td>
<td>2000</td>
<td>5.56</td>
<td>379.4</td>
<td>15.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### TIME = 901518.00

<table>
<thead>
<tr>
<th>CLASS</th>
<th>executed</th>
<th>active</th>
<th>blocked</th>
<th>aborted</th>
<th>committed</th>
<th>Throughput</th>
<th>Mean RestTime</th>
<th>Mean LastTime</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writers</td>
<td>2095</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2085</td>
<td>2.90</td>
<td>260.1</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>Readers</td>
<td>1619</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1613</td>
<td>2.24</td>
<td>179.6</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadersL</td>
<td>333</td>
<td>2</td>
<td>0</td>
<td>29</td>
<td>302</td>
<td>0.42</td>
<td>2252.4</td>
<td>174.6</td>
<td>1.1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>4047</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>4000</td>
<td>5.56</td>
<td>378.1</td>
<td>13.4</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### TIME = 1261001.00

<table>
<thead>
<tr>
<th>CLASS</th>
<th>executed</th>
<th>active</th>
<th>blocked</th>
<th>aborted</th>
<th>committed</th>
<th>Throughput</th>
<th>Mean RestTime</th>
<th>Mean LastTime</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writers</td>
<td>3139</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>3129</td>
<td>2.90</td>
<td>259.3</td>
<td>0.3</td>
<td>1.0</td>
</tr>
<tr>
<td>Readers</td>
<td>2423</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2417</td>
<td>2.24</td>
<td>180.2</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadersL</td>
<td>497</td>
<td>2</td>
<td>0</td>
<td>41</td>
<td>454</td>
<td>0.42</td>
<td>2243.4</td>
<td>165.6</td>
<td>1.1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>6059</td>
<td>16</td>
<td>0</td>
<td>43</td>
<td>6000</td>
<td>5.56</td>
<td>377.6</td>
<td>12.7</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### TIME = 1620687.00

<table>
<thead>
<tr>
<th>CLASS</th>
<th>executed</th>
<th>active</th>
<th>blocked</th>
<th>aborted</th>
<th>committed</th>
<th>Throughput</th>
<th>Mean RestTime</th>
<th>Mean LastTime</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writers</td>
<td>4182</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>4179</td>
<td>2.90</td>
<td>259.3</td>
<td>0.3</td>
<td>1.0</td>
</tr>
<tr>
<td>Readers</td>
<td>3228</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>3222</td>
<td>2.24</td>
<td>180.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadersL</td>
<td>661</td>
<td>2</td>
<td>0</td>
<td>52</td>
<td>607</td>
<td>0.42</td>
<td>2252.1</td>
<td>154.7</td>
<td>1.1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>8071</td>
<td>16</td>
<td>0</td>
<td>55</td>
<td>8000</td>
<td>5.56</td>
<td>377.1</td>
<td>11.9</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### TIME = 1980290.00

<table>
<thead>
<tr>
<th>CLASS</th>
<th>executed</th>
<th>active</th>
<th>blocked</th>
<th>aborted</th>
<th>committed</th>
<th>Throughput</th>
<th>Mean RestTime</th>
<th>Mean LastTime</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writers</td>
<td>5224</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>5218</td>
<td>2.90</td>
<td>259.3</td>
<td>0.3</td>
<td>1.0</td>
</tr>
<tr>
<td>Readers</td>
<td>4034</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>4027</td>
<td>2.24</td>
<td>160.3</td>
<td>0.1</td>
<td>1.0</td>
</tr>
<tr>
<td>ReadersL</td>
<td>825</td>
<td>2</td>
<td>0</td>
<td>62</td>
<td>761</td>
<td>0.42</td>
<td>2223.0</td>
<td>145.8</td>
<td>1.1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>10083</td>
<td>16</td>
<td>0</td>
<td>67</td>
<td>10000</td>
<td>5.56</td>
<td>376.9</td>
<td>11.3</td>
<td>1.1</td>
</tr>
</tbody>
</table>
The following SG statistics were gathered:

Cycle Length 2: 56 cycles
Cycle Length 3: 7 cycles
Cycle Length 4: 4 cycles

LENGTH OF CYCLES: MEAN = 2.22 VARIANCE = 0.29

Resource Utilization:

<table>
<thead>
<tr>
<th>RESOURCE</th>
<th>Service Time</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>442234</td>
<td>0.25</td>
</tr>
<tr>
<td>DMcpu</td>
<td>132805</td>
<td>0.30</td>
</tr>
<tr>
<td>CCcpu</td>
<td>137414</td>
<td>0.31</td>
</tr>
<tr>
<td>TPcpu</td>
<td>172015</td>
<td>0.39</td>
</tr>
<tr>
<td>IO</td>
<td>1605003</td>
<td>0.89</td>
</tr>
<tr>
<td>DMio</td>
<td>1605003</td>
<td>0.89</td>
</tr>
<tr>
<td>TPio</td>
<td>0</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Statistics on Queues Size:

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Mean</th>
<th>Variance</th>
<th>Observations No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockedQ</td>
<td>1.02</td>
<td>0.02</td>
<td>64</td>
</tr>
<tr>
<td>CommitQ</td>
<td>1.01</td>
<td>0.01</td>
<td>10000</td>
</tr>
<tr>
<td>CPU</td>
<td>1.21</td>
<td>0.43</td>
<td>386429</td>
</tr>
<tr>
<td>DMcpuQ</td>
<td>1.29</td>
<td>0.70</td>
<td>122353</td>
</tr>
<tr>
<td>CCcpuQ</td>
<td>1.00</td>
<td>0.00</td>
<td>132071</td>
</tr>
<tr>
<td>TPcpuQ</td>
<td>1.05</td>
<td>0.06</td>
<td>132005</td>
</tr>
</tbody>
</table>

Number of Concurrent IO Requests:

<table>
<thead>
<tr>
<th>IO Resource</th>
<th>Mean</th>
<th>Variance</th>
<th>Observations No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>2.96</td>
<td>2.49</td>
<td>122353</td>
</tr>
<tr>
<td>DMio</td>
<td>2.96</td>
<td>2.49</td>
<td>122353</td>
</tr>
<tr>
<td>TPio</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
</tr>
</tbody>
</table>

> clear
> step
REFERENCES


