Semantics-Based Concurrency Control: Beyond Commutativity [1]

B. R. Badrinath
Krithi Ramamritham

COINS Technical Report 86-18
Revised: April 1987

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[1]This work was supported in part by the National Science Foundation under grants DCR-8403097 and DCR-85000332.

Abstract

The concurrency of transactions executing on atomic data types can be enhanced through the use of semantic information about operations defined on these types. Hitherto, commutativity of operations has been exploited to provide enhanced concurrency while avoiding cascading aborts. We have identified a property known as ***recoverability*** which can be used to decrease the delay involved in processing non-commuting operations while still avoiding cascading aborts. When an invoked operation is ***recoverable*** with respect to an uncommitted operation, the invoked operation can be executed by forcing a commit dependency between the invoked operation and the uncommitted operation; the transaction invoking the operation will not have to wait for the uncommitted operation to abort or commit. Further, this commit dependency only affects the order in which the operations should commit, if both commit; if either operation aborts, the other can still commit thus avoiding cascading aborts. To ensure the serializability of transactions, we force the recoverability relationship between transactions to be acyclic. Simulation studies indicate that using recoverability, response time of transactions can indeed be reduced, especially at large transaction loads.

Contents

1 Introduction 1

2 Related Work 2

3 A Formal Definition of Recoverability 3
- 3.1 Operations and Recoverable Operations . . . 3
- 3.2 Examples . . . 5
  - 3.2.1 Stack . . . 5
  - 3.2.2 Set . . . 7
  - 3.2.3 Table . . . 7

4 A Concurrency Control and Commit Protocol 10
- 4.1 Correctness Requirements and Commit Dependency Graph . . . 11
- 4.2 A Two Phase Algorithm for Pseudo-committing Transactions . . . 13
  - 4.2.1 Centralized Algorithm . . . 13
  - 4.2.2 Distributed Algorithm . . . 17
- 4.3 Committing Pseudo-committed Transactions . . . 20

5 Results of Simulation Studies 21
- 5.1 The Simulation Model . . . 22
- 5.2 Simulation results . . . 24

6 Conclusions 26

1 Introduction

In object-oriented transaction environments it is desirable to attain as high a degree of concurrency as possible. Object specifications contain semantic information that can be exploited to increase concurrency. Several schemes based on the commutativity of operations have been proposed to provide more concurrency than obtained by the conventional classification of operations as *reads* or *writes* [Garcia-Molina 83, Weihl 84]. For example, two insert operations on a set object commute and hence, can be executed in parallel; further, regardless of whether one operation commits, the other can still commit. Applying the same rule, two push operations on a stack object do not commute and hence cannot be executed concurrently. We have identified a property we term ***recoverability*** to decrease the delay involved in processing non-commuting operations. It turns out that two push operations are recoverable and hence can be executed in parallel.

In protocols in which conflict of operations is based on commutativity, an operation o_i which does not commute with other uncommitted operations will be made to wait until these conflicting operations abort or commit. We would clearly prefer the operations to execute and return the results as soon as possible without waiting for the the transactions invoking the conflicting operations to commit. Such a feature will be especially useful when long-lived transactions are in progress. In our scheme, non-commuting but ***recoverable*** operations are allowed to execute in parallel; but the order in which the transactions invoking the operations should commit is fixed to be the order in which they are invoked. If o_j is executed after o_i, and o_j is ***recoverable relative to*** o_i, then, if transactions T_i and T_j that invoked o_i and o_j respectively commit, T_i should commit before T_j. Thus, based on the recoverability relationship of an operation with other operations, a transaction invoking the operation sets up a dynamic commit dependency relation between itself and other transactions. If an invoked operation is not recoverable with respect to an uncommitted operation, then the invoking transaction is made to wait. For example, two pushes on a stack do not commute, but if the push operations are forced to commit in the order they were invoked, then the execution of the two push operations is serializable in commit order. Further, if either of the transactions aborts the other can still commit.

Schemes for improving concurrency should be concerned with the problem of transaction rollback, in particular, the possibility of ***cascading aborts***. This phenomenon of cascading aborts occurs when abort of one transaction necessitates aborting other transactions that could have read its results. Thus, obliterating the effects of the aborted transaction involves not only undoing the effects of the aborted transactions but also causing abort of other transactions. This may propagate even further, with aborting transactions causing

some more transactions to abort and so on. What makes recoverability an attractive concept is that it permits more concurrency than commutativity while retaining the positive feature of commutativity, namely, avoiding cascade aborts. Cascade aborts are avoided because even if one of the transactions involved in a commit dependency aborts, the other can still commit.

When recoverable operations execute, they may form cyclic commit dependency relationships. To force this relationship to be acyclic and thus preserve serializability, one of the transactions involved in a cycle is aborted. We have developed a protocol to detect cyclic dependencies in a *dynamic* manner and abort transactions to ensure serializability. We have combined the process of checking for cyclic dependencies with the first phase of the commit protocol. This greatly reduces the overheads involved in providing additional concurrency through use of the notion of recoverability.

While Section 2 presents a brief survey of related work, Section 3 describes the model, assumptions, and definitions. Section 4 describes a concurrency control and commit protocol designed to utilize recoverability semantics. Results of extensive simulation studies are reported in Section 5. Section 6 concludes with a discussion.

2 Related Work

In optimistic concurrency schemes [Kung 81], conflicts are allowed to occur, but at the time of validation, transactions with conflicts are aborted. These algorithms can give rise to cascading aborts, thus introducing serious overhead. Further, conflicts are determined by a test of the intersection of read/write sets and is not efficient because semantics of the operations are not taken into account.

In [Buckley 85], locking protocols using structural information about the data items are developed to permit only non-cascading rollback. Their model has only read and write operations, and the database is structured as a directed hypergraph. In addition, associated with each transaction is a static set of entities which it must access first.

Most locking protocols used in semantics-driven concurrency control base conflicts between operations on the notion of commutativity of operations [Beeri 83, Weihl 84]. It is well known that if a protocol allows only commuting operations to execute concurrently then it prevents cascading aborts. When a transaction invokes an operation, the operation is executed if it commutes with every other uncommitted operation. Otherwise the transaction is made to wait. Some use operation return value commutativity [Weihl 85], wherein information about the results of executing an operation is used in determining commutativity, and some use the arguments of the operations in determining whether

or not two operations commute [Birman 85, Spector 84]. These protocols provide more concurrency than protocols using general commutativity [Beeri 83].

The term recoverability also appears in [Hadzilacos 84]. There the recoverability criterion defines a class of schedules in which no transaction commits before any transaction on which it depends. However, the definitions are based on a *free interpretation* of the operations invoked by the transactions [Papadimitriou 79]. That is, each value written by a transaction is some arbitrary function of the previous values read. Hence, their theory does not take into account semantics of the individual operations. For example, in their model, a transaction writing the *sum* of two values and another writing the *maximum* of two values are indistinguishable.

In our work we have used the notion of recoverability to define conflicts between operations. We use the semantic information that is available from the specifications of data types to determine recoverability of two operations. Use of the recoverability criterion provides more concurrency than commutativity while avoiding cascading aborts. To ensure serializability, we have developed an algorithm for detecting cycles in the transaction commit dependency relation. The algorithm is based on maintaining dependency graphs [Casanova 81]. However, we check for acyclicity only when transactions commit by using a new protocol. A node corresponding to a transaction remains in the graph only until the transactions on which it depends commit or abort. We have combined the process of checking for acyclicity of the dependency graph with the first phase of the commit protocol.

3 A Formal Definition of Recoverability

3.1 Operations and Recoverable Operations

Transactions in our system perform operations on instances of atomic data types. A transaction T is modeled by a tuple $(OP_T, <_T)$ where OP_T is a set of abstract operations and $<_T$ is a partial order on them.

Concurrent execution of a set of transactions $T_1, T_2, \ldots, T_n$ gives rise to a log $E = (OP_E, <_E)$. OP_E is $(\cup_i OP_{T_i})$ and $(\cup_i <_{T_i}) \subseteq <_E$. $<_E$ is a partial order on the operations in OP_E and the log represents the order in which they are executed by the system. If $o_i <_E o_j$ we say that o_j executed after o_i. The execution log is serializable if there exists a total order $<_s$ called a serialization order on the set $\{T_1, T_2, \ldots, T_n\}$ such that if an operation o_i in transaction T_i *conflicts* with an operation o_j in T_j, and if $T_i <_s T_j$, then $o_i <_E o_j$ [Eswaran 78]. Two operations conflict if they both operate on the same data item and one of them is a write. In this paper we will generalize the notion of conflict by considering the semantics of the operations. Execution of operations on different objects

can be thought of as generating logs E_j for each object O_j such that log E is the union of all these logs.

Each object has a type, which defines a possible set of states of the object, and a set of primitive operations that provide the only means to create and manipulate objects of that type. The specification of an operation indicates the set of possible states and the responses that will be produced by that operation when the operation is begun in a certain state. Formally, the specification is a total function: $S \mapsto S \times V$ where $S = \{s_1, s_2, \ldots\}$ is a set of *states* and $V = \{v_1, v_2, \ldots\}$ is a set of *return values*. For a given state $s \in S$ we define two components for the specification of an operation: *return*(o, s) which is the return value [2] produced by operation o, and *state*(o, s) which is the state produced after the execution of o.

Definition 1: For a given state $s \in S$ consider operations o_1 and o_2 such that o_1's execution is immediately followed by the execution of o_2. Let state$(o_1, s) = s'$, and state$(o_2, s') = s''$. We say that operation o_2 is ***recoverable relative to operation*** o_1 ***in state*** s denoted by $(o_2 \; RR_I \; o_1, s)$ iff

$$\text{return}(o_2, s') = \text{return}(o_2, s)$$

Henceforth we say that o_2 is ***recoverable relative to*** o_1, denoted by $(o_2 \; RR_I \; o_1)$, if for all states $s \in S$, o_2 is recoverable relative to o_1.

Intuitively, the above definition states that if o_2 executes immediately following o_1, the value returned by o_2, and hence the observable semantics of o_2, is independent of whether o_1 executed *immediately* before o_2.

Operations do not conflict if they commute. Operations commute if their effect on an object is independent of the order in which they are executed. This can be formally stated as follows.

Definition 2: Two operations o_1 and o_2 *commute* in state s if the final state of the object and the return values of the operations when begun in state s are independent of whether $o_1 <_E o_2$ and $o_2 <_E o_1$.

Lemma 1: If o_1 and o_2 commute in state s then $(o_2 \; RR_I \; o_1, s)$ and $(o_1 \; RR_I \; o_2, s)$.□

Since commuting operations are recoverable we restrict the term recoverability to those operations which are recoverable but do not commute. In the remaining sections, if we imply recoverability from commutativity, we will explicitly state so.

[2] It is assumed that every operation returns a value, at least a status or condition code.

In addition to the operations defined on objects, two special termination operations are abort and commit of a transaction. Commit (Abort) indicates the successful (unsuccessful) completion of a transaction. These will appear in the execution log with commit (abort) of a transaction T_i denoted by $C_i(A_i)$.

Terminology: An operation is *executable* if it can be scheduled for execution; it has *completed* once its results are available. When a transaction *aborts*, the effects (on the objects) of the operations executed by the transaction will be undone. If a transaction *commits*, all the effects will be made permanent and the changes will become visible to other transactions. A transaction *terminates* when it executes either a commit or an abort operation. A transaction *visits* an object if it executes at least one operation on the object.

We consider conflicts at the abstract level and it is assumed that the operations are executed indivisibly on the underlying implementation of the object. The conflicts are specified via an operation compatibility table. The table can be derived from the semantics of the operations on an object. Using the table, conflicts can be detected at run time by the manager of the object.

3.2 Examples

In this section we examine some objects. By use of a compatibility table we will elucidate the type of dependencies that exist between various operations. These examples focus on the type of conflicts that are permissible under commutativity and recoverability. Our derivation of the dependencies is based on the definitions of commutativity and recoverability.

3.2.1 Stack

The stack object provides three operations: *Push, pop,* and *top*. *Push* adds a specified element to the top of the stack. *Pop* removes and returns the top element if the stack is not empty, otherwise it returns *null*. *Top* returns the value of the top element if the stack is not empty, otherwise it returns *null*. Two push operations do not commute but are recoverable relative to each other. Similarly, though a push operation does not commute with a top operation, it is recoverable relative to top. These differences are indicated in the compatibility tables shown in Figures 1a and 1b.

Operation Requested	Operation Executed		
	Push	Pop	Top
Push	Yes-SP	No	No
Pop	No	No	No
Top	No	No	Yes

Figure 1a: Commutativity.

Operation Requested	Operation Executed		
	Push	Pop	Top
Push	Yes	Yes	Yes
Pop	No	No	Yes
Top	No	No	Yes

Figure 1b: Recoverability.

In the commutativity table, if an entry is *Yes*, it indicates that the operations associated with that entry are commutative; if the entry is *No*, it indicates that they are not. In the recoverability table, if an entry is *Yes*, then the requested operation associated with the entry is recoverable relative to the executed operation associated with the entry. A *No* entry indicates that the requested operation is not recoverable relative to the executed operation. A *qualified Yes*, in particular, a Yes-SP (Yes-DP), indicates that the operations involved are commutative or recoverable depending on whether the two operations have the Same input Parameter (Different input Parameter).

By examining the two tables, we can make the following observations: Commutativity is a symmetric property whereas recoverability is not. Yes entries in the commutativity table remain Yes in the recoverability table since commutativity implies recoverability. Because recoverability allows more concurrency, the recoverability table has more Yes entries than the commutativity table. The entry associated with two pushes in the commutativity table is Yes-SP because, two pushes having the same parameter, i.e., attempting to push the same element, are commutative.

A closer look at the tables will reveal that the entries in the above tables do not reflect state dependent commutativity and recoverability properties. Clearly, state dependent commutativity or recoverability can be used to extract further concurrency. However, as the following example shows, it will typically result in complex implementations: Two pop operations commute if the top two elements of the stack they are operating on are the same. Suppose the top two elements of a stack are the same and hence two pop operations are allowed to execute concurrently; before the two operations terminate, another pop request

arrives. In this case, it is not difficult to see that even though the pop request commutes with each of the pop operations in execution, it cannot be allowed to execute concurrently with them unless the top three elements of the stack are the same. Clearly, not only the specification, but also the implementation of such state-dependent notions of commutativity can become quite complex. However, use of commutativity and recoverability based on operation parameters does not result in appreciable increase in complexity. Hence in the remainder of this paper, we restrict ourselves to state-independent, but parameter-dependent notions of commutativity and recoverability.

3.2.2 Set

A set object provides three operations: *insert*, *delete*, and *member*. *Insert* adds a specified item to the set object. The parameter to *Delete* specifies the item to be deleted from the object. If the item is present in the set, it returns *Success*, otherwise, it returns *Failure*. *Member* determines whether a specified item is an element of the set object. Inserting two elements is commutative; so is deleting different elements. Similarly, insert and member involving different elements commute but do not commute when the specified elements are the same. However, insert is recoverable relative to member, as indicated by the Yes entry.

Operation	Operation Executed		
Requested	Insert	Delete	Member
Insert	Yes	Yes-DP	Yes-DP
Delete	Yes-DP	Yes-DP	Yes-DP
Member	Yes-DP	Yes-DP	Yes

Figure 2a: Commutativity.

Operation	Operation Executed		
Requested	Insert	Delete	Member
Insert	Yes	Yes	Yes
Delete	Yes-DP	Yes-DP	Yes
Member	Yes-DP	Yes-DP	Yes

Figure 2b: Recoverability.

3.2.3 Table

The *Table* type stores pairs of (key, item) values, where the keys are unique. The operation *insert* inserts a new (key, item) pair in the table. If the key is already present in the table, it returns a *Failure*, otherwise it returns *Success*. The operation *delete* deletes the pair with the given key from the table. If the key is not present in the table, it returns a *Failure*, otherwise it returns *Success*. The *size* operation returns the number of entries in

the table. *Lookup* returns the value of the item associated with a given key if it exists in the table. If no such item exists, the result returned is *not found*. *Modify* modifies the value of the item associated with the given key. If the key is not present in the table, it returns a *Failure*, otherwise it returns *Success*. A *size* operation does not commute with *insert* and *delete* operations. However, both *insert* and *delete* are recoverable relative to *size*; but the converse is not true: Because *size* returns the number of entries in the table, the value returned depends on prior *insert* and *delete* requests, whereas *insert* and *delete* are not affected by prior invocations of the *size* operation.

Operation Requested	Operation Executed				
	Insert	Delete	Lookup	Size	Modify
Insert	Yes-DP	Yes-DP	Yes-DP	No	Yes-DP
Delete	Yes-DP	Yes-DP	Yes-DP	No	Yes-DP
Lookup	Yes-DP	Yes-DP	Yes	Yes	Yes-DP
Size	No	No	Yes	Yes	Yes
Modify	Yes-DP	Yes-DP	Yes-DP	Yes	Yes-DP

Figure 3a: Commutativity.

Operation Requested	Operation Executed				
	Insert	Delete	Lookup	Size	Modify
Insert	Yes-DP	Yes-DP	Yes	Yes	Yes
Delete	Yes-DP	Yes-DP	Yes	Yes	Yes
Lookup	Yes-DP	Yes-DP	Yes	Yes	Yes-DP
Size	No	No	Yes	Yes	Yes
Modify	Yes-DP	Yes-DP	Yes	Yes	Yes

Figure 3b: Recoverability.

We find the notation used in [Weihl 84] convenient to describe a sequence of operations invoked on an object. We will consider operations to be events, where an event is a paired operation invocation and response. As an example, consider an object of type *set*. Invoking *insert*(i) inserts the element i into the set and returns "ok" when the operation is completed. Thus, if the integer set object *primeset* X is invoked to perform *insert*(3), 3 will be added to X and the result would be "ok". If this is followed by an invocation of the *member*(3) operation on *primeset* X to check for membership of 3 in *primeset* X, the result would be "yes". We will identify the object and the transaction invoking the operation when we describe a sequence of operations.

The following is a interleaved operation sequence invoked by transactions T_1 and T_2 on the set object *primeset* X.

$X : \langle insert(3), ok, T_1 \rangle$

$X : \langle member(3), yes, T_2 \rangle$

$X : \langle insert(7), ok, T_1 \rangle$

$X : \langle delete(3), ok, T_1 \rangle$ (1)

The abort of a transaction may cause other transactions to abort. This phenomenon is known as cascading aborts. In sequence (1), should T_1 abort for any reason, T_2 cannot commit (because it has seen effects of T_1), and hence has to abort. However, the following sequence of operations on two instances X and Y of a *set* object is free from cascading aborts:

$X : \langle member(3), no, T_2 \rangle$

$X : \langle insert(3), ok, T_1 \rangle$

$Y : \langle insert(4), ok, T_2 \rangle$

$Y : \langle delete(5), ok, T_2 \rangle$

$\langle commit, T_1 \rangle$

$\langle abort, T_2 \rangle$ (2)

Here, even though T_2 has aborted, the semantics of the operations invoked by T_1 is still the same.

Consider the sequence of operations invoked by transactions T_1 and T_2 on instances S of type stack and X of type set:

$S : \langle push, T_1, ok \rangle$

$X : \langle member(3), T_1, no \rangle$

$S : \langle push, T_2, ok \rangle$

$X : \langle insert(3), T_2, ok \rangle$

$\langle commit, T_1 \rangle$

$\langle commit, T_2 \rangle$ (3)

In concurrency protocols which consider operations to conflict if they are not commutative, the operations invoked by T_2 will have to wait until T_1 commits. However, in our scheme, since the relevant operations invoked by T_2 are recoverable they can be executed without waiting for T_1 to commit, while avoiding cascading aborts should T_1 abort for any reason. But the commit order is fixed: T_2 can commit only after T_1 terminates. In the next section, we now discuss a concurrency control and commit protocol where a transaction can *complete* execution even though the transactions on which it depends have not terminated.

So far, $(o_2 \ RR_I \ o_1)$ was used to denote the fact that o_2 was recoverable relative to o_1 when o_2 was executed immediately after o_1. Before we conclude this section, we extend the concept to include the case where o_2 is recoverable relative to o_1 in spite of intervening operations that have executed but have not yet committed.

Definition 3: Consider a set of operations $S = \{o_1, \ldots, o_n\}$ such that $\forall_{1 \leq i < n}\ o_i <_E o_{i+1}$. ($o_n$ RR o_1) if the return value of o_n is independent of whether o_1 executed before o_n (i.e., not necessarily immediately before). Hence $o_n\ RR\ o_1 \Longrightarrow o_n\ RR_I\ o_1$.

Lemma 2: Given the set of operations S defined above, if $\forall\ l,\ 1 \leq l < n$, ($o_n\ RR_I\ o_l$) then ($o_n$ RR o_1).

Proof: Let F denote the operations that execute between between o_n and o_1. The proof is by induction on k where $k = |F|$.

Induction base ($k = 1$ i.e., F contains only one operation): Let $S = \{o_n, o_2, o_1\}$. Given that ($o_n RR_I o_2$) and since o_2 is executed immediately before o_n, the results returned by o_n are independent of o_2. If o_2 aborts, o_1 will be the operation executed immediately before o_n; Since ($o_n\ RR_I\ o_1$) again results of o_n are independent of o_1 and hence (o_n RR o_1).

Induction hypothesis (F contains $k - 1$ operations): if $\forall\ l,\ 1 \leq l \leq k$, ($o_n\ RR_I\ o_l$), then ($o_n$ RR o_1).

Induction Step: Let $|F| = k$ and $S = \{o_n, o_{k+1}, \ldots, o_2, o_1\}$. Now ($o_n\ RR_I\ o_{k+1}$) and ($o_n\ RR_I\ o_k$) $\Longrightarrow$ (o_n RR o_k) by using a reasoning similar to the base case. From definition 3 we have o_n RR $o_k \Longrightarrow o_n\ RR_I\ o_k$, and by induction hypothesis $\forall\ l\ 1 \leq l \leq k$ $o_n\ RR_I\ o_l \Longrightarrow o_n\ RR\ o_1$.

Corollary : $\forall\ l, 1 \leq l < n\ o_n RR_I o_l \Longrightarrow \forall l\ 1 \leq l < n\ o_n RR o_l$.

4 A Concurrency Control and Commit Protocol

In this section we discuss the practical issues related to achieving enhanced concurrency using recoverability semantics.

We assume the existence of an object manager for each object. This manager schedules the executions of the operations invoked by transactions on that object. We also assume the existence of a transaction manager for each transaction, which is the system interface to the user transaction. The transaction manager forwards the user requests to the object managers. The manager of an object maintains an execution log of uncommitted operations on that object. Once an operation is requested on an object, the object manager determines the conflict between that operation and the operations in the log. Conflicts between operations are determined with recoverability in mind.

Since recoverable operations force commit dependencies, a transaction may commit only after other transactions on which it depends commit. However, the semantics of the execution of the transaction are not affected by the commit/abort of other transactions

with which it has a commit dependency. Hence a transaction can *complete* execution; with the exception that the operations and the transaction continue to remain in the execution log and commit dependency graph respectively. We call this sort of commit a *pseudo-commit*. Note that this is different from the conditional commit of nested transactions [Moss 81], wherein a transaction that has conditionally committed may be forced to abort by its parent. A transaction which has *pseudo-committed* will definitely commit, but only after all transactions on which it depends terminate, thus respecting the commit dependency relationship.

Transactions invoke operations on several objects. This leads to a problem: We must ensure that the executions on different objects agree on at least one serialization order for the committed transactions. To determine whether the execution is serializable we have to determine whether the commit dependency relationship is acyclic. This phase is similar to the validation phase in optimistic protocols [Kung 81]. We have combined the process of checking the dependency-graph for acyclicity with the first phase of the standard two phase commit protocol.

In section 4.1 we formally define the correctness requirements of the concurrency control and commit protocols and introduce the commit dependency graph. In section 4.2 we develop the two-phase protocol for pseudo-committing transactions. An Algorithm for committing pseudo-committed transactions is given in section 4.3.

4.1 Correctness Requirements and Commit Dependency Graph

Definition 4: An operation o_i invoked by transaction T_i is *sound* in a log E if for any *extension* $E' = E \parallel A_j$ for any $j \neq i$ ($\parallel$ indicates that the operation is appended to the log, A_j is abort of transaction T_j), return(o_i, s) =return(o_i, s') where s and s' are the states in which o_i is executed in E and E' respectively.

To ensure that the intended semantics of the operations are guaranteed in spite of transaction aborts, we shall require that all operations in a log be sound. As it turns out, this property can be achieved by allowing only operations that are either commutative or recoverable to execute.

Theorem 1: Let $o_1, \ldots, o_n$ be operations in the log E such that for any $o_i <_E o_j$, if o_i is uncommitted then either o_i and o_j commute or (o_j RR o_i) . Then all operations are sound in E.

Proof: By induction on the number of operations in the log L. From lemma 1 if o_i and o_j commute then (o_j RR o_i).

Induction Base. Consider a log E with two operations o_1 and o_2 such that $o_1 <_E o_2$. o_1 is sound. If o_1 is committed, then o_2 is trivially sound. If o_1 is not committed then since (o_2 RR o_1), o_2 is sound from definition of the relationship RR.

Induction Hypothesis. For a log E with operations $o_1, \ldots, o_{n-1}$ satisfying the conditions mentioned in the statement of the theorem, all operations in the log E are sound.

Induction Step. By the induction hypothesis, operations $o_1, \ldots, o_{n-1}$ are sound in E. Since $\bigwedge_{i=1}^{n-1} o_i <_E o_n$, and the relation ($o_n$ RR o_i) holds for any uncommitted o_i, again by the definition of recoverability, o_n is sound in E.

Lemma 3: A log E is free from cascading aborts if it has only sound operations.

The proof follows from the definition of soundness and recoverability.

The object manager uses compatibility tables for the objects to determine whether an operation is sound with respect to other uncommitted operations in the log. Once an operation is requested the object manager determines the type of conflict with other uncommitted operations. If the operation is neither recoverable nor commutative with other uncommitted operations, the transaction is made to wait. Deadlocks due to cyclic waits of non-recoverable operations can be handled using known techniques of deadlock avoidance, or deadlock detection and resolution [Bracha 84, Sinha 85].

The object manager for object O_k maintains a ***commit dependency graph*** G_k for object O_k. In the G_k, nodes indicate transactions and edges indicate the commit order which arises from conflicts between operations invoked by different transactions on object O_k. Thus absence of an edge between any two transactions implies that operations invoked by the two transactions on this object commute.

Definition 5: A commit dependency graph $G_k = (N, M)$, where N is the set of nodes corresponding to transactions that have executed some operation on object k and M is the set of edges e, where e is a directed edge from T_j to T_i if T_i has executed o_i and T_j has executed o_j such that 1) $o_i <_{E_k} o_j$, and 2) o_i and o_j are not commutative but (o_j RR o_i).

Lemma 4: An execution log E is serializable if the commit dependency graph $G = \cup_k G_k$ is acyclic. □

The proof follows from the definition of serializability.

Definition 6: An execution log E is correct if it is serializable and is free from cascading aborts.

Using Lemma 4, we will ensure serializability by forcing the commit dependency relationship resulting from the recoverability of operations in the log E to be acyclic. From Lemma 3, cascading aborts can be avoided by ensuring that all operations in the log are *sound*.

Figure 4 is an example of a dependency graph for an object. Here the operation invoked by T_1 is recoverable relative to operations invoked by T_2 and T_3, and operation invoked by T_2 is recoverable relative to operation invoked by T_3. The operation invoked by T_4 commutes with the rest of the operations. The dependency graph is constructed using the algorithm given in figure 5.

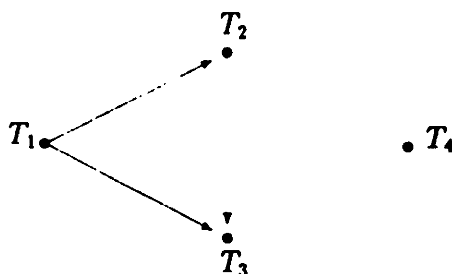

Figure 4: A dependency graph

Let G_k be the commit dependency graph and E_k the execution log at object k.
Let o_i be an operation invoked by transaction T_i.
For each operation $o_j \in E_k$ identify conflicting operations and update the commit dependency graph as follows:

1. If there is at least one ongoing operation with which o_i is not recoverable then T_i is made to wait.

2. If for all operations o_j, o_i and o_j are commutative or o_i is recoverable relative to o_j then

   - Insert a node corresponding to the transaction T_i. Insert directed edges from node T_i to other transactions which have invoked operations with which o_i is recoverable.

Figure 5: Algorithm to insert commit dependency edges.

4.2 A Two Phase Algorithm for Pseudo-committing Transactions

4.2.1 Centralized Algorithm

In this case, the transaction manager, in order to determine whether a transaction can pseudo-commit, via a two phase commit protocol interacts with the managers of the objects

visited by the transaction. As transactions attempt to pseudo-commit, as discussed below, care is taken to ensure that there does not exist a set of pseudo-committed transactions in a commit dependency cycle. Further, if there is a cycle of commit dependencies, it is sufficient for one of the transactions forming the cycle to abort. In our protocol the last transaction to pseudo-commit will be aborted.

Each transaction is initially assigned a unique timestamp, which serves as the transaction-id. We will maintain two sets $PRED_{obj}(T_i)$ and $SUCC_{obj}(T_i)$ for each transaction T_i (i.e., with each node in the commit dependency graph) at each object *obj*. Below we discuss how these sets are constructed. Roughly speaking, for a transaction T_i that has pseudo-committed, $PRED_{obj}(T_i)$ ($SUCC_{obj}(T_i)$) contains ids of pseudo-committed transactions that are predecessors(successors) in the commit dependency graph along paths consisting of only pseudo-committed nodes. Note that we are using the term predecessor(successor) to denote any ancestor(descendant), not necessarily immediate ones.

When a transaction T_i wants to pseudo-commit, as the reply to "prepare to pseudo-commit" message from the T_i's coordinator, the manager of each object *obj* visited by T_i, sends $PRED_{obj}(T_i)$ and $SUCC_{obj}(T_i)$. Since, in this section, we are considering a centralized system, we will assume that the process of pseudo-commit is done in an atomic manner i.e., only one transaction attempts to pseudo-commit at a time. The two sets of timestamps sent by the object managers are:

$$i)\ PRED_{obj}(T_i) = \bigcup_{T_p}(PRED_{obj}(T_p) \bigcup \{T_p\})$$

where T_p is an *immediate* pseudo-committed predecessor transaction of T_i; if no such T_p exists, $PRED_{obj}(T_i) = \emptyset$.

$$ii)\ SUCC_{obj}(T_i) = \bigcup_{T_s}(SUCC_{obj}(T_s) \bigcup \{T_s\})$$

where T_s is an *immediate* pseudo-committed successor transaction of T_i; if no such T_j exists, $SUCC_{obj}(T_i) = \emptyset$.

Having collected these sets the transaction manager then determines whether a transaction can pseudo-commit. The pseudo code for the entire algorithm is as shown below.

```
Begin

    Transaction Ti intends to Pseudo-commit;
    For each Obj visited by Ti do
    {
    Send "prepare to pseudo-commit" message to the manager
```

```
of Obj
Collect PRED_obj(T_i) and SUCC_obj(T_i);
}
PRED(T_i) = ∪_obj PRED_obj(T_i);
SUCC(T_i) = ∪_obj SUCC_obj(T_i);

If PRED(T_i) ∩ SUCC(T_i) ≠ ∅ then

    Send "Abort T_i" message to all object managers
    visited by T_i

else

    Send "pseudo-commit T_i" message along with
    PRED(T_i) and SUCC(T_i) to all object managers
    visited by T_i;
```

End

The object managers on receipt of a "pseudo-commit T_i" message update the $PRED_{obj}(T_i)$ and $SUCC_{obj}(T_i)$ sets as follows:

$$PRED_{obj}(T_i) = PRED(T_i)$$
$$SUCC_{obj}(T_j) = SUCC(T_i).$$

Also, the PRED and SUCC sets, for all pseudo-committed successors T_s and pseudo-committed predecessors T_p of T_i reachable via paths consisting of only pseudo-committed nodes are modified as follows:

$$SUCC_{obj}(T_p) = SUCC_{obj}(T_p) \cup SUCC(T_i)$$
$$PRED_{obj}(T_s) = PRED_{obj}(T_s) \cup PRED(T_i).$$

On the other hand, when the object managers receive an "abort T_i" message, they
remove the node corresponding to T_i from the commit dependency graph.

Step	T_i	SUCC(T_i)	PRED(T_i)	SUCC(T_i)	PRED(T_i)	Result
		At the end of first phase		At the end of second phase		
T_4 attempts pseudo-commit	T_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	P-C (pseudo-commit)
T_1 attempts pseudo-commit	T_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	P-C
T_2 attempts pseudo-commit	T_2	$\{T_1\}$	$\{T_4\}$	$\{T_1\}$	$\{T_4\}$	P-C
	T_4			$\{T_2, T_1\}$	$\emptyset$	-
	T_1			$\emptyset$	$\{T_4, T_2\}$	-
T_5 attempts pseudo-commit	T_5	$\emptyset$	$\{T_1, T_2, T_4\}$	$\emptyset$	$\{T_1, T_2, T_4\}$	P-C
T_3 attempts pseudo-commit	T_3	$\{T_4, T_2, T_1\}$	$\{T_1\}$	-	-	Aborts

Table 1: Stepwise execution of pseudo-commit algorithm

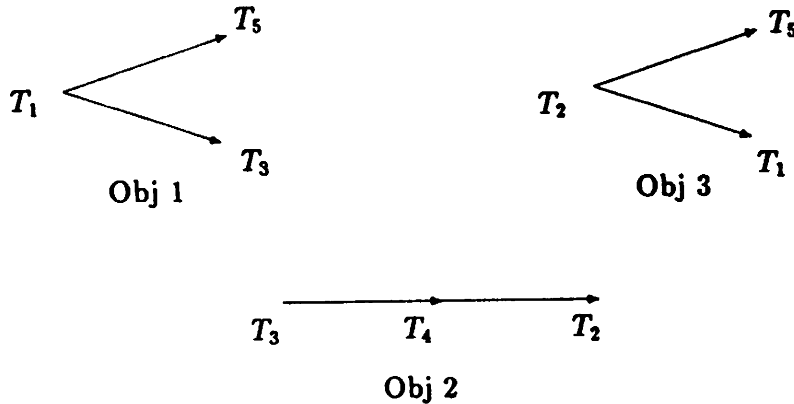

Figure 6: Dependency graphs at three objects

Example

Using figure 6, which shows dependency graphs at three objects, we illustrate, in table 1, the execution of the pseudo-commit algorithm in steps. At the beginning none of the transactions have pseudo-committed.

Correctness arguments for the pseudo-commit algorithm:

Each pseudo-committed transaction T_i at each object has a pair of sets: $SUCC_{obj}(T_i)$

$\{ T_n / T_i \stackrel{+}{\Longrightarrow} T_n, T_n$ is pseudo-committed and every T_m along the path from T_i to T_n is also pseudo-committed $\}$. $PRED_{obj}(T_l) = \{ T_j / T_j \stackrel{+}{\Longrightarrow} T_l, T_j$ is pseudo-committed and every T_k along the path from T_j to T_l is also pseudo-committed $\}$. To determine whether T_i can pseudo-commit or not, the transaction manager collects these sets from *immediate* successors and *immediate* predecessors from each object that T_i has visited to obtain $PRED(T_i)$ and $SUCC(T_i)$. The proof that a transaction is actually in a commit dependency cycle when the intersection of $PRED(T_i)$ and $SUCC(T_i)$ is non empty follows from the theorem.

Theorem 2: Let $T_1, \ldots, T_n$ be n nodes (transactions) in the CD (commit dependency) graph. Let T_k be the last transaction attempting to pseudo-commit. Then T_k is in a cycle of the CD graph iff $PRED(T_k) \bigcap SUCC(T_k) \neq \emptyset$.

Proof: *If:* If T_k attempts to pseudo-commit and it is in a CD cycle then $T_k \stackrel{+}{\Longrightarrow} T_l$ and $T_l \stackrel{+}{\Longrightarrow} T_k$ for all $T_l \neq T_k$ in the cycle. Let T_p and T_s be the immediate predecessor and immediate successor of T_k respectively. Since every transaction $T_l \neq T_k$ in the cycle has pseudo-committed, $SUCC(T_k)$ and $PRED(T_k)$ will contain T_l, and hence the intersection of $SUCC(T_k)$ and $PRED(T_k)$ will be non empty.

Only if: Assume $PRED(T_k) \bigcap SUCC(T_k) \neq \emptyset$. Let T_l belong to $PRED(T_k) \bigcap SUCC(T_k)$. Then $T_k \stackrel{+}{\Longrightarrow} T_l$ and $T_l \stackrel{+}{\Longrightarrow} T_k$ which implies T_k is in a cycle.

4.2.2 Distributed Algorithm

Since in a distributed system the commit process can be started by more than one transaction, we must provide for potential race conditions. We will present an algorithm to determine, in a distributed manner, whether a transaction can pseudo-commit.

In the distributed case, the local managers of the sites that contain objects visited by a transaction T_i intending to pseudo-commit are the cohorts of the manager of T_i. Each site has a local manager. Each transaction is assigned a unique timestamp using a system of Lamport clocks [Lamport 78]. The local managers maintain two sets known as PC and AC. The set AC contains transactions that have started the pseudo-commit process at this site and the set PC contains transactions that have pseudo-committed. The commit dependency graph, and hence the PRED and SUCC sets are still maintained by the object managers. When a cohort receives a pseudo-commit request, the local manager determines whether there is a cycle locally as in the centralized algorithm. If there is a cycle an abort message is sent to the transaction manager. On the other hand, if there exists no cycle at a site, then the cohort will send AC and PC along with PRED and SUCC sets to the

transaction manager. The set AC is then updated to include the transaction initiating the pseudo-commit. Note that the process of checking for a local cycle, sending the sets, and updating AC is assumed to be done in an atomic manner.

The sets AC and PC are used to determine a total order among transactions attempting to pseudo-commit at the same time, based on the order of starting the pseudo-commit process. Before we look at the pseudo code for the commit protocol we define some terms that will make it easier to reason about the correctness of the distributed commit algorithm.

Definition 7: T_1 started the commit process before T_2, denoted by T_1 BEFORE T_2, iff $T_2 \notin$ (AC or PC) collected by T_1, or $T_1 \in$ *every* AC or to *some* PC collected by T_2.

Definition 8: T_1 and T_2 have overlapping commit phases, denoted by T_1 OVERLAPS T_2, iff T_1 belongs to *some* AC collected by T_2 and vice versa.

Definition 9: Let us define $T_1 <_c T_2$ iff T_1 BEFORE T_2 or T_1 OVERLAPS T_2 and timestamp(T_1) $<$ timestamp(T_2). Note that given two transactions T_1 and T_2 either $T_1 <_c T_2$ or $T_2 <_c T_1$, i.e., $<_c$ defines a total order on transactions that have started the pseudo-commit process.

By using the sets PC and AC, as explained in the commit protocol below, transactions are allowed to pseudo-commit in the order defined by $<_c$. Thus potential race conditions in pseudo-committing are avoided, thereby reducing the distributed version of the algorithm essentially to the centralized algorithm whose correctness has been proved earlier.

- **Phase I**

  - During the first phase of the commit protocol for transaction T_i, the transaction manager initiates the pseudo-commit process sending prepare to pseudo-commit messages to its cohorts.

  - The cohorts then check for a local commit dependency cycle; if there is a local commit dependency cycle then an *abort* message is sent. However, if there is no cycle locally then the sets $PRED(T_i)$ and $SUCC(T_i)$ are sent along with the sets AC and PC. The set AC is then updated to $AC \cup T_i$. Note that the action of determining PRED and SUCC sets, and inserting T_i into AC is assumed to be in an atomic manner at each site.

  - If there was no abort message from any of the cohorts then the transaction manager determines whether there exists a $T_j <_c T_i$ and T_j has not completed the pseudo-commit process. If such a T_j exists then the transaction manager will send a request

to cohorts for obtaining the sets PRED and SUCC again. However, this time the cohorts will wait for all T_j such that $T_j <_c T_i$ to complete the pseudo commit process (as we shall see, this happens when T_j is removed from AC) and then send the sets $PRED$ and $SUCC$.

- **Phase II**

If the transaction manager has received an ***abort*** message at the end of phase I, or if the intersection of the two sets $\bigcup_{cohorts} PRED_{cohorts}$ and $\bigcup_{cohorts} SUCC_{cohorts}$ is non empty (i.e., there is a global commit dependency cycle) then

  - Transaction manager sends *abort* T_i message.

- If $\exists$ no cycle involving T_i in the commit dependency graph **then**

  Begin

  If $\bigcup_{cohorts} SUCC_{cohorts}(T_i) = \emptyset$ **then**

  - Transaction manager sends *Commit* T_i message.

  Else

  - Transaction manager sends *Pseudo Commit* T_i message.

  End

- A cohort does the following:

1) If it receives a commit or abort message: the node corresponding to T_i is removed from AC as well as from the commit dependency graph.

2) If it receives a Pseudo commit message: The node corresponding to T_i is removed from the set AC and inserted in the set PC.

When T_i completes the first phase of the commit protocol, using AC and PC, the transaction manager determines the set of transactions $B_f = \{T_j / T_j <_c T_i\}$. If this set is non empty, the transaction manager, in order to check for a global cycle, has to wait until all such T_j complete, at which point the cohorts send PRED and SUCC sets. While T_i is waiting, since T_i is in AC at each site visited by it, any other T_l which starts the pseudo-commit process will find $T_i <_c T_l$ and hence cannot belong to the set B_f, i.e., cannot pseudo-commit before T_i. Thus $<_c$ imposes an ordering on transactions attempting tp pseudo-commit thus avoiding race conditions.

Before we conclude this section we look at the problem of effecting aborts. When a transaction aborts, it is necessary to undo (backout) a transaction. Undo of a transaction

involves undo of all operations executed by a transaction. We will assume that each operation has an inverse (i.e., that it can be undone). To achieve transaction abort we need to maintain a log of events, which is a sequence of entries containing the operation, the object, and the transaction-id. One approach to undo an operation is to first undo all operations that come "after" it, and then apply the operation's inverse. Hence, we can roll back by applying inverses in reverse order. The current state can then be determined by redoing of all operations except the one being aborted. Nevertheless, to avoid digression, we do not investigate these strategies in this paper; details can be found in [Oki 85, Moss 86].

4.3 Committing Pseudo-committed Transactions

After a transaction pseudo-commits, the operations and the transaction continue to remain in the log and the commit dependency graph respectively. Because of this, operations executed by the pseudo-committed transactions will be used to determine conflicts with operations invoked by other transactions. The operations of pseudo-committed transactions can be removed from the log only when other transactions on which it depends terminate. If a transaction pseudo-commits, the object managers have to decide when actually to commit the transaction. To make this decision, the following information is required by each object manager.

- The set of object managers from which messages have to be received for an object manager to commit a transaction. This set is denoted by CD-SET. Basically CD-SET contains those objects at which a transaction has commit dependencies (the out-degree of the node corresponding to the transaction in the commit dependency graph at that object is non-zero).

- The set of object managers to which a message has to be sent when a transaction has no commit dependencies (the out-degree of the node corresponding to the transaction in the commit dependency graph becomes zero). This set is denoted by VISIT-SET. VISIT-SET contains those objects which a transaction has visited.

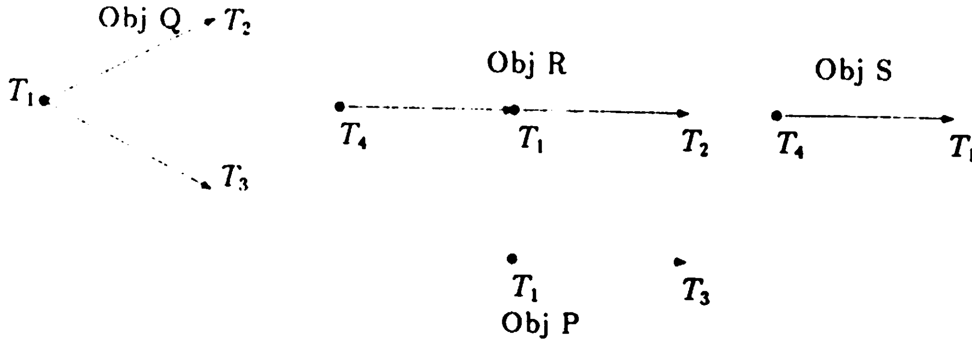

Figure 7: Dependency graphs at objects P, Q, R, and S

The members of CD-SET and VISIT-SET for each transaction can easily be determined by the transaction manager when it collects the dependency graph at each object as part of the commit protocol. Hence the transaction manager sends these sets to the object managers involved during the second phase of the commit protocol. Consider the scenario shown in Figure 7. At object P, CD-SET(T_1) = $\{Q, R\}$ and VISIT-SET(T_1) = $\{Q, R, S\}$. At object S, the sets CD-SET(T_1) = $\{P, Q, R\}$ and VISIT-SET(T_1) = $\emptyset$.

Each object manager uses information contained in CD-SET and VISIT-SET of transactions to commit a pseudo-committed transaction in a decentralized manner as follows. When the out-degree of the node corresponding to a particular transaction becomes zero (there exist no commit dependencies) the object manager sends **commit** messages to the object managers in VISIT-SET. If a transaction has no commit dependencies, and **commit** messages from all object managers in CD-SET have arrived, then the transaction is committed. The node and the incoming edges in the commit dependency graph, and the operations corresponding to the transaction in the execution log, are removed.

5 Results of Simulation Studies

We now report on simulation studies designed to evaluate the increased concurrency resulting from the use of recoverability. The purpose of this simulation study is to compare the amount of concurrency offered when both commutativity and recoverability are used to determine conflicts as opposed to using just commutativity. Hence, we are only interested in the effect of data contention rather than resource (for example, I/O) contention. Thus we are not modeling resource contention.

Simulation parameters	
Parameter	Value
Database size	400 objects
Transaction length	5 – 9 steps
Interrequest time	0.1 secs
Arrival rate	1 · 20 txns/sec
Time out (deadlock)	3 secs
Communication delay	0.6 secs
Time to retry	0.3 secs

Table 2: Parameters and their nominal values

5.1 The Simulation Model

To simplify the simulations, we focus on the effect of parameter-independent semantic properties. Thus an entry (i, j) in the recoverability (commutativity) table for an object will be *yes* only if operation i is recoverable relative to (commutative with) operation j independent of the input parameters to the two operations. In this case, we can merge the two tables into a single *compatibility table*; each entry in this table will be one of *commutative*, *recoverable*, or *null*.

To model different degrees of commutativity and recoverability, the properties of operations on an object are specified by two integers: P_c determines the number of *commutative* entries in an object's compatibility table; P_r determines the number of *recoverable* entries in this table. Thus, $(N^2 - P_c - P_r)$ is the number of *null* entries where N is the number of operations defined on the object. We experimented with even values of P_c and P_r. (In the graphs depicted in figures 8 through 13, each graph is for a fixed value of P_c and transaction length. The horizontal axis depicts different values of P_r.) At the beginning of a simulation run, given the values of P_c and P_r for an object, $P_c/2$ non-diagonal entries in its compatibility table are *randomly* chosen and set be *commutative*; their symmetric entries are then made *commutative*. P_r of the remaining entries are then randomly chosen using a uniform distribution and set to be *recoverable*. The rest of the entries are set to *null*. Some of the other parameters besides P_c and P_r and their nominal values are listed in Table 2. The values of the model parameters have been chosen similar to those in previous performance studies of locking protocols [Tay 85, Tay 85a] and commutativity-based protocols [Cordon 85].

To simplify the study, we assume a system consisting 400 objects. Each object has four operations defined on it. Each transaction makes a sequence of k requests where k is the

transaction length, and the time between the ith and the $(i+1)^{th}$ operation request is a random variable uniformly distributed on $(0, 2T)$ so that the average interrequest time is T.

The transaction interrequest time, the arrival rate, and the transaction length determine the overall transaction load. Since transactions compete for the shared objects, for a given transaction length, as transaction load increases, i.e., as either interrequest time or arrival rate increases, contentions will increase and hence transaction response time will increase. The transaction load is adjusted by changing the transaction arrival rate while keeping the interrequest time for operations at 0.1 secs for different transaction lengths. In essence we are simulating an open queuing model with a poisson transaction arrival process; a similar model has been used in [Yu 85].

We have conducted extensive simulation studies for various transaction lengths k = 5, 7, and 9. All of the defined operations can be invoked with equal probability. Further we assume, as in [Tay 85], there is uniform access, that is the probability that a transaction requests an operation on a particular object it has not accessed before is the same as that for any other object.

Arrival rate was determined in the following way: Under $P_r = 0$, i.e., maximum conflict, we determined maximum arrival rate before which the throughput begins to fall due to excessive aborts. For $k = 5$, this rate was found to be 20 transactions/sec and the response time for this arrival rate was $\approx$ 2.4 secs. For other transactions lengths k= 7, 9, we adjusted the arrival rates so as to obtain the same response time under maximum conflict. These were found to be 8 and 4 transactions/second respectively. We also experiment with arrival rates that are smaller than this maximum value.

Recall that an operation that is neither commutative with nor recoverable relative to all ongoing operations is made to wait. Such waits may lead to deadlocks. We have made use of time-outs to tackle this problem. If an operation request is not satisfied within 3 seconds, the invoking transaction is aborted. We term such aborts t-aborts.

Recall that one of the transactions in a commit dependency cycle is aborted. We call such aborts as r-aborts. An r-aborted transaction is resubmitted 0.3 secs after its abort. The response times of such transactions is computed with respect to their original arrival times.

We do not model the details of the communication between a transaction manager and the object managers; the communication delay involved in the commit protocol is fixed at 0.6 secs. This is the delay between when a transaction completes all its operations and when the managers of objects visited by the transaction are informed by the transaction manager about the commitment or abortion of the transaction. Thus the minimum communication

delay experienced by a transaction is 0.6 seconds. A transaction that is r-aborted once and succeeds in its second attempt experiences a total communication delay of 1.2 seconds. The only detail we do not model is the additional delay transactions experience when their pseudo-commit phases overlap.

Even though it might appear that the effect of modeling communication delays by a constant amount will be to increase the response times of all transactions by 0.6 seconds, there is an important secondary effect. This arises due to the fact that transactions executing nonrecoverable operations are now made to wait longer and hence our model for communication delays does model the impact on response times for conflicting transactions.

5.2 Simulation results

Typically, transaction response time is defined to be the length of the interval between transaction arrival time and the time the results of the transaction are available. In our case, the latter time is the same as the time when a transaction pseudo-commits.

The average transaction response time induced by a concurrency control algorithm will normally reflect the degree of concurrency allowed by that algorithm: The better the concurrency properties of the algorithm, the smaller the average transaction response time. Hence in this study, we use average transaction response time as the metric to evaluate the concurrency of operations having different commutativity and recoverability properties.

Given that recoverability is a weaker conflict predicate than commutativity, we expect significant reductions in response time for transactions. If recoverability properties are not considered, there will be an increase in the waiting time of transactions which invoke operations that do not commute with uncommitted operations. As recoverability increases we expect a decrease in average response time for transactions.

In this study each simulation is run till 400 transactions are generated. We measured various factors, including transaction response time, the average delay between the instant a transaction pseudo-commits and the instant it commits, and the number of transactions that t-abort. The graphs in figure 8 through 13 show the average results of 50 runs. These depict the increased concurrency, i.e., reduced average response time, the pseudo-commit time (mean time for a pseudo-committed transaction to commit) and the number of t-aborts. At large P_r values, a transaction experiences the minimum response time possible. This is equal to the total execution time of all operations plus the communication delay involved in commitment. In our case, this minimum transaction response time is $(k * 0.1) + 0.6$ which varies from 1.1, 1.3 and 1.5 secs for k= 5, 7, and 9 steps respectively. Since transactions are resubmitted when they are r-aborted, the minimum transaction response time is slightly higher than the minimum possible, as calculated above, for various

Parameter	k = 5	k = 7	k = 9
$P_c = 2$, $P_r = 2$	9.55%	9.199%	6.97%
$P_c = 2$, $P_r = 4$	20.4%	18.19%	13.3%
$P_c = 2$, $P_r = 6$	30.5%	25.74%	19.91%
$P_c = 4$, $P_r = 2$	11.62%	6.807%	6.92%
$P_c = 4$, $P_r = 4$	22.1%	14.699%	12.8%
$P_c = 4$, $P_r = 6$	30.96%	22.627%	18.08%

Table 3: Drop in response times under maximum arrival rate

transaction lengths.

Based on these graphs we can make the following observations which have been summarized in table 3 :

- The use of recoverability does result in smaller transaction response times; the larger the value of P_r, the smaller the response time. This decrease occurs in spite of transaction aborts due to cyclic commit dependencies. The percentage drop in response time for different values of P_r and P_c are shown in table 3.

- As the transaction length is increased, the response time decreases as recoverability increases; however, the percentage drop in response time for a given value of P_r reduces slightly as the length of the transaction increases.

- The notion of recoverability is especially useful for large loads. In this case, there is a almost linear drop in the response times with increased recoverability.

Our simulation experiments also show that the interval between when a transaction pseudo-commits and when it commits decreases as P_r increases. We can explain this behavior as follows: Transactions are made to wait by other transactions with which they have commit dependencies. At low values of P_r, there is a large probability of a transaction performing a non-recoverable operation and hence these waits are long. At higher P_r values, even though transactions form more commit dependencies, transactions experience less wait times for performing operations due to the increased concurrency made possible

by recoverability. This produces the fall in the aforementioned interval almost linearly with P_r value. The number of aborts caused by cyclic commit dependencies has been found to be very low. Even at maximum P_r value less than 5 % of the total transactions generated were aborted due to cyclic dependencies.

The average number of aborts due to timeout begins to decrease as P_r increases, but begins to rise after a certain value of P_r; it begins to decrease again for still larger P_r values. This is because transactions that execute recoverable operations increase as P_r increases, hence the level of multiprogramming also increases, thereby increasing the conflict among transactions executing nonrecoverable operations. In our simulation model, a transaction executing a nonrecoverable operation is made to wait, while a subsequently arriving transaction that invokes a recoverable operation will be allowed to execute. Waiting transactions are not given priority over transactions executing recoverable operations. As a result, when the number of transactions executing recoverable operations increases, the probability of a waiting transaction executing decreases. Hence the number of t-aborts increases as P_r increases beyond a certain value.

Even though the number of aborts increases after a certain value of P_r, it is still less than the number of aborts when $P_r = 0$ for k= 5. However, for k= 7 and 9 the number of aborts is higher than the corresponding value at $P_r = 0$. As P_r increases further, the number of operations that are non commutative, i.e., the number of conflicts, is further reduced thereby reducing the number of aborts.

In summary, for objects whose compatibility tables have a reasonable number of recoverable operations, as in examples of Section 3.2, the drop in response time is significant. Further, the number of t-aborts in such cases will be lower than the number of t-aborts when recoverability is not considered. Thus, we get a two-fold advantage by exploiting recoverability semantics.

6 Conclusions

We have described a concurrency control protocol which avoids cascading aborts by exploiting type-specific properties of objects. The protocol uses a conflict predicate known as recoverability in addition to commutativity. It is simple and effective because the algorithm is based on checking pre-defined conflicts between pairs of operations. Conflicts among operations executed by different transactions can be checked by using a compatibility table, and the table can be derived directly from the data type specification. The use of recoverability not only reduces the latency involved in processing non-commuting operations but also avoids cascading aborts. As we saw in the examples of Section 3.1,

non-commuting but recoverable operations are not uncommon and hence we expect to increase concurrency considerably.

Since the dynamic commit dependency relationship between transactions may be cyclic, serializability may be violated as transactions execute; during the commit phase transactions are aborted to maintain serializability. We have seen a scheme, to achieve this. Using recoverability as a conflict predicate it was possible to combine the process of commitment and validation of a transaction. This reduces some of the overhead involved in providing additional concurrency. This reduction can be achieved by piggybacking the PRED and SUCC sets on the messages used for the commit protocol.

Since the first phase of the commit protocol is also used for exchanging the dependency information at various objects, failures of nodes will affect our scheme in the same way as they affect the standard two phase commit protocol.

As our simulation studies indicate, the notion of recoverability is a powerful concept that produces appreciable drops in transaction response times. The magnitude of this drop is dependent on transaction loads as well as the commutativity and recoverability properties of operations on shared objects.

References

[Beeri83] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E., "Concurrency control theory for nested actions," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, (August 1983), pp. 45–62.

[Bernstein81] Bernstein, P. A., and Goodman, N., "Concurrency control in distributed database systems," *Computing Surveys*, Vol. 13, No.2 (June 1981), pp. 185-221.

[Birman85] Birman, K. P., et al., "Implementing fault-tolerant distributed objects," *IEEE Transactions on Software Engineering*, Vol. 11, Vol.6 (June 1985), pp. 520–530.

[Bracha84] Bracha, G., and Toueg, S., " Distributed algorithm for generalized deadlock detection," *Third ACM Symposium on Principles of Distributed Computing*, (August 1984), pp. 285–301.

[Buckley85] Buckley, G. N., and Silberschatz, A., "Beyond two phase locking", *Journal of the ACM*, Vol. 31, No. 2 (April 1985), pp. 314–326.

[Casanova81] Casanova, M. A., "The concurrency control problem for database systems," *Lecture Notes in Computer science*, Vol. 116, Springer-Verlag, (1981).

[Cordon 85] Cordon, R., and Garcia-Molina, H., "The performance of a concurrency mechanism that exploits semantic knowledge," *Proceedings of the fifth international conference on distributed systems*, (1985) pp. 350–358.

[Eswaran76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The notion of consistency and predicate locks in a database system," *Communications of the ACM*, Vol. 19, No. 11 (November 1976), pp. 624-633.

[Garcia-Molina83] Garcia-Molina, H., "Using semantic knowledge for transaction processing in a distributed database," *ACM Transactions on Database Systems*, Vol. 8, No. 2 (June 1983) pp. 186-213.

[Goodman85] Goodman, N., and Shasha, D., "Semantically-based concurrency control for search structures," *Fourth Annual Symposium on Principles of Database System*, (March 1985) pp. 8-19.

[Hadzilacos84] Hadzilacos, V., "Issues of fault tolerance in concurrent computations," Technical Report 11-84, Harvard University, Cambridge, MA, (June 1984).

[Kung81] Kung, H. T., and Robinson, J. T., " On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, Vol. 6, No. 2 (June 1981) pp. 213-226.

[Lamport78] Lamport, L., "Time, Clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol. 21, No. 7, (July 1978) pp. 558-565.

[Moss81] Moss, J. E. B., "Nested Transactions: An approach to reliable distributed computing," PhD Thesis, Tech. Rept. 260, Massachusetts Institute of Technology, Cambridge, MA (April 1981).

[Moss86] Moss, J. E. B., Griffeth, N. D., and Graham, M. H., "Abstraction in recovery management," *Proceedings of the ACM-SIGMOD international conference on management of data*, (May 1986) pp. 72-83.

[Oki85] Oki, M. B., Liskov, B. H., and Scheifler, R. W., " Reliable object storage to support atomic operations," *Proceedings of the tenth ACM symposium on operating systems principles*, Vol. 19, No. 5, (December 1985), pp. 147-159.

[Sinha85] Sinha, M., and Natarajan, N., " A priority based distributed deadlock detection algorithm," *IEEE Transactions on Software Engineering* , Vol. 11, No.1 (January 1985), pp. 67-80.

[Papadimitriou79] Papadimitriou, C. H., "The serializability of concurrent database updates," *Journal of the ACM*, Vol. 26, No. 4 (October 1979), pp. 631-653.

[Schwarz 84] Schwarz, M. P., and Spector, A. Z., "Synchronizing shared abstract data types," *ACM Transactions on Computer systems*, Vol. 2, No. 3 (August 1984), pp. 223-250.

[Tay85] Tay, Y. C., Rajan Suri and Goodman, N., "A mean value performance model for locking in databases: The no-waiting case," *Journal of the association for computing machinery*, Vol. 32, No. 3, (July 1985) pp. 618-651.

[Tay85a] Tay, Y. C., Rajan Suri and Goodman, N., "Locking performance in centralized databases," *ACM Transactions on Database Systems*, Vol. 8, No. 4, (December 1985) pp. 415-462.

[Weihl84] Weihl, W., "Specification and implementation of atomic data types," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA. (March 1984).

[Weihl85] Weihl, W., and Liskov, B., "Implementation of resilient, atomic data types," *ACM Transactions Programming Languages and Systems*, Vol. 7, No. 1 (April 1985), pp. 244-269.

[Yu85] Yu, S. P., Dias, M. D. and et al, " Modeling of centralized concurrency control in a multi-system environment," *Performance Review*, Vol. 13, No. 2, (August 1985) pp. 183-191.

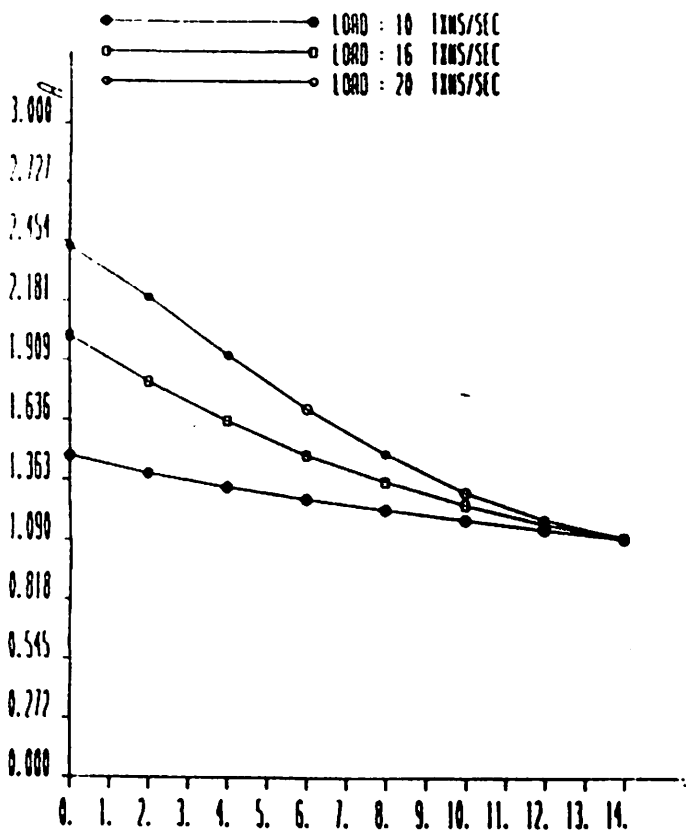

Figure 8a: Effect on response time

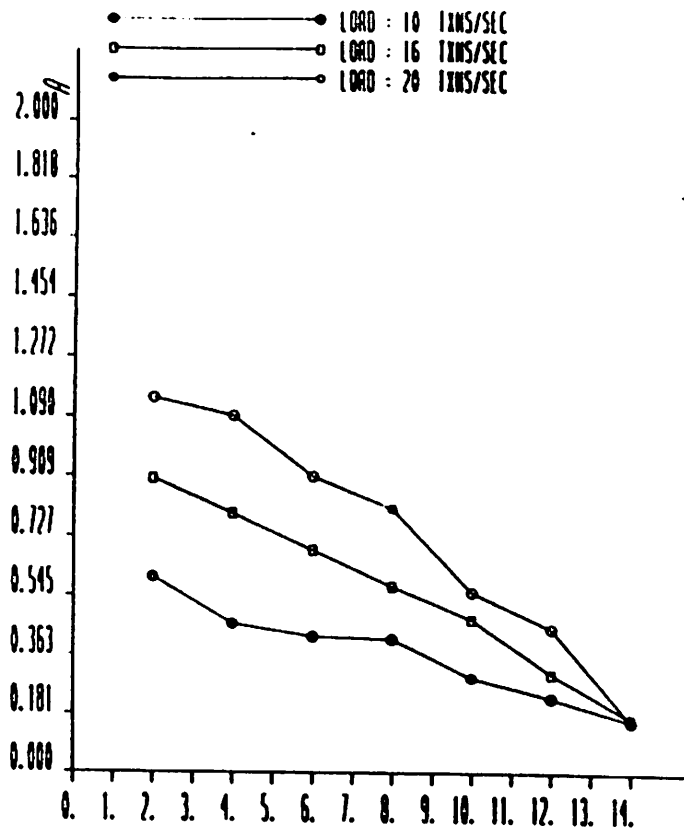

Figure 8b: Effect on pseudo-commit time

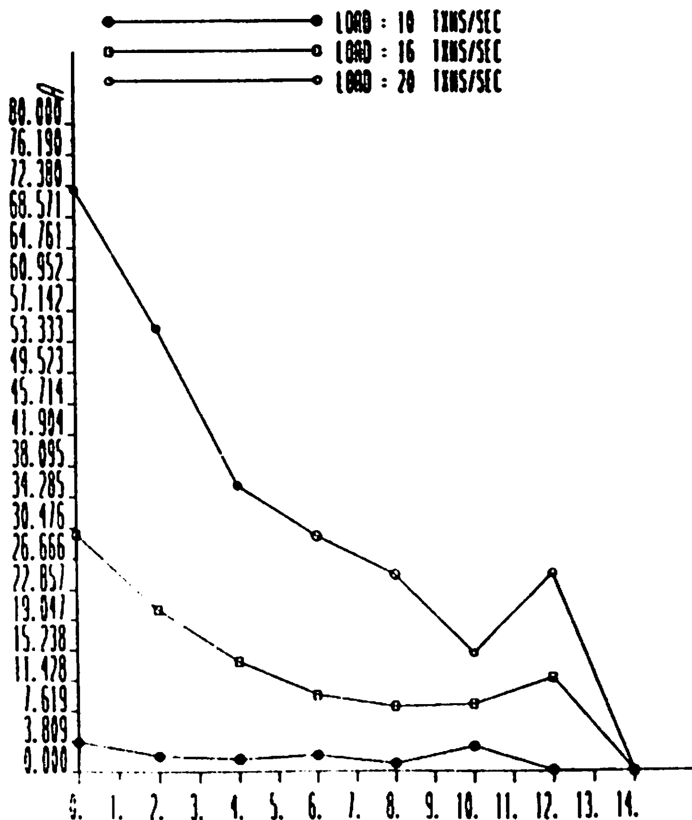

Figure 8c: Effect on aborts

Figure 8: Simulation Results for P 2, Transaction length   5 steps

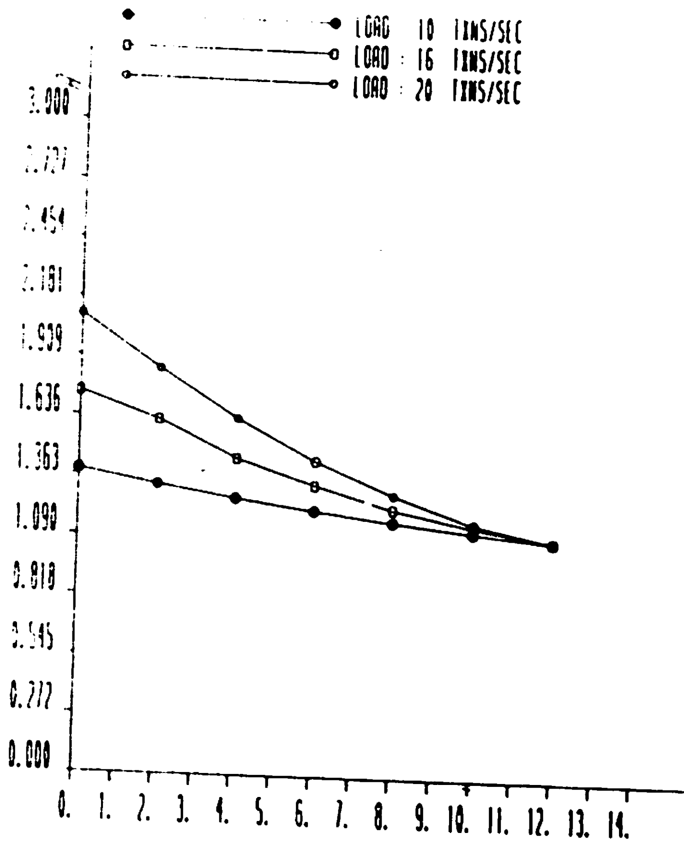

Figure 9a: Effect on response time

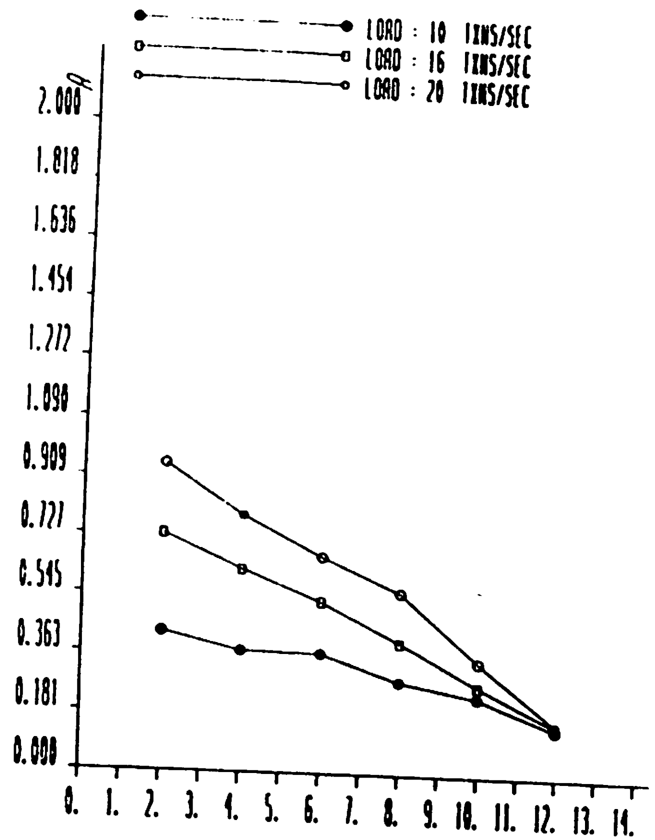

Figure 9b: Effect on pseudo-commit time

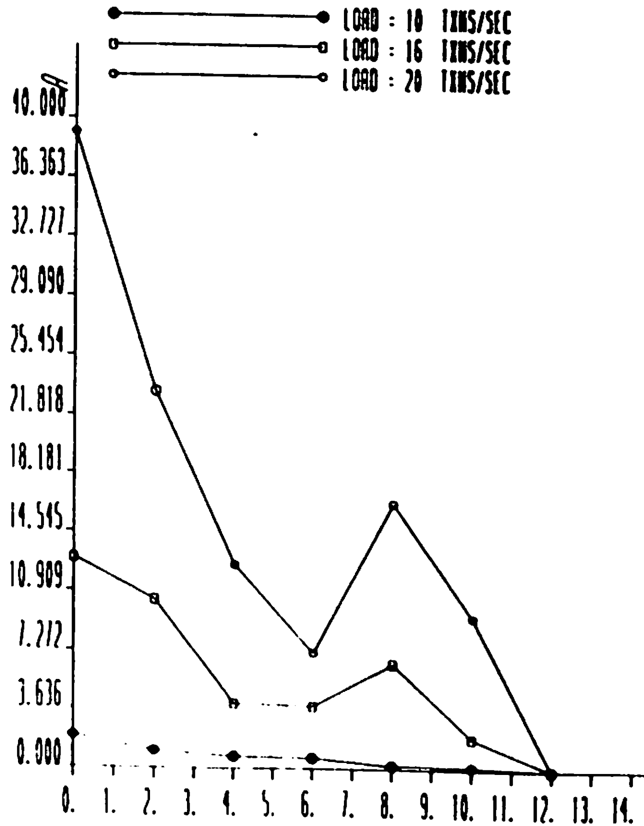

Figure 9c: Effect on aborts

Figure 9: Simulation Results for $P = 4$, Transaction length = 5 steps

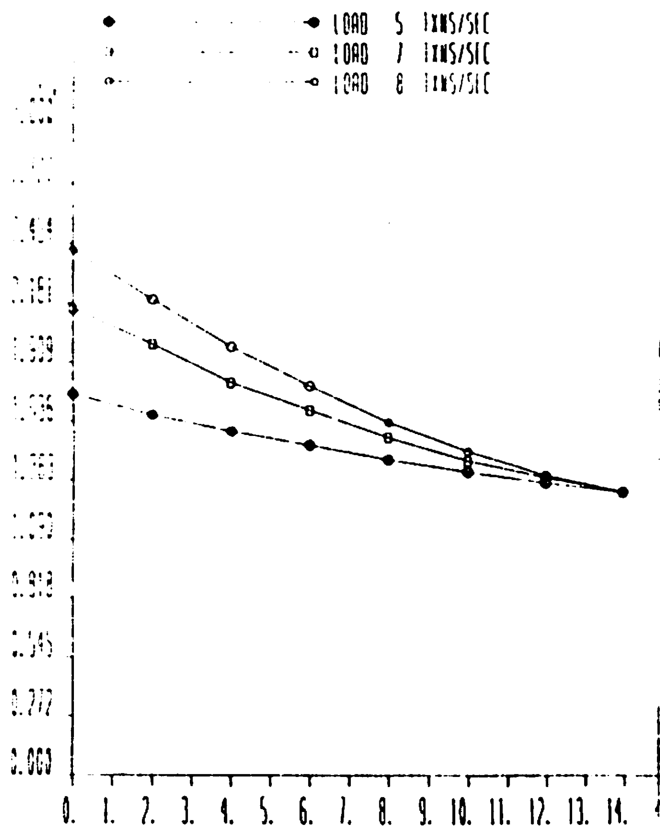

Figure 10a: Effect on response time

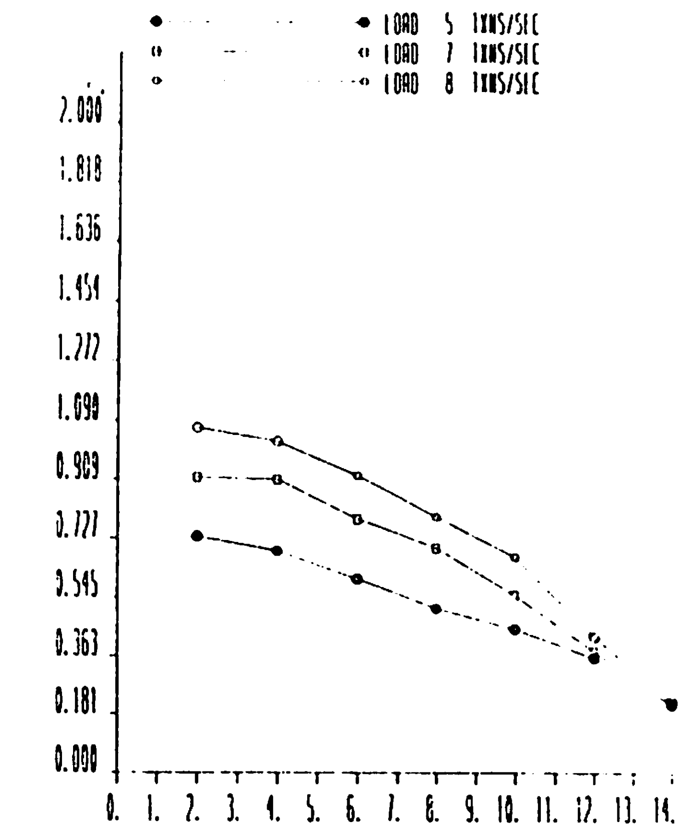

Figure 10b: Effect on pseudo-commit time

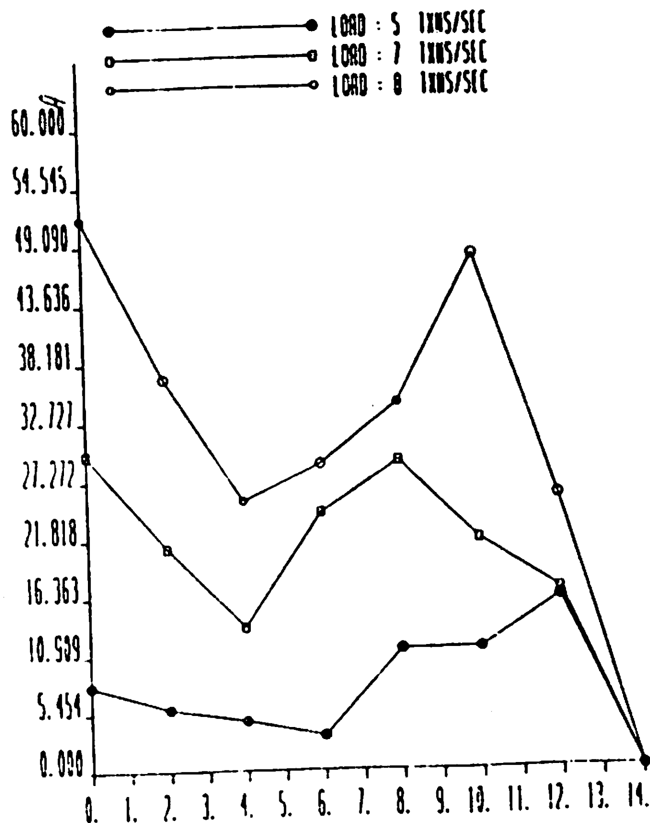

Figure 10c: Effect on aborts

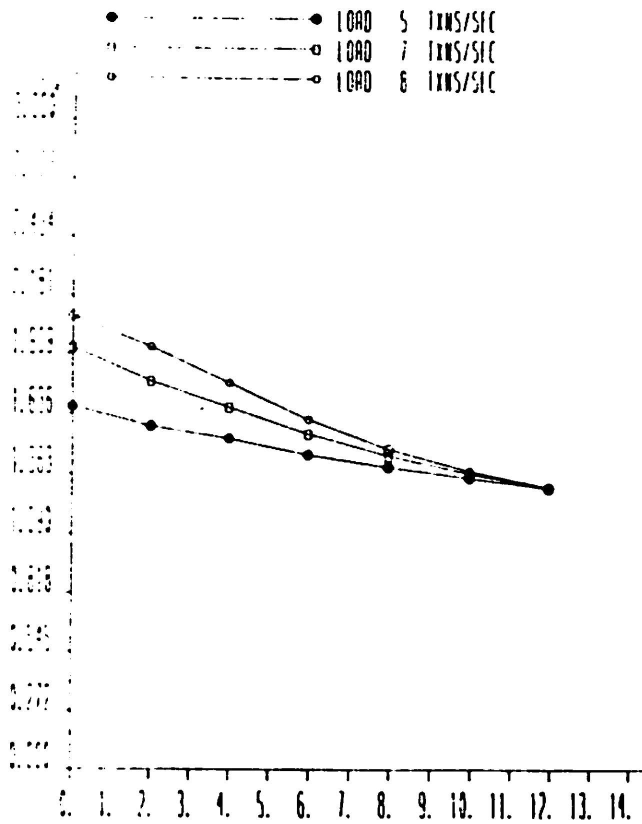

Figure 11a: Effect on response time

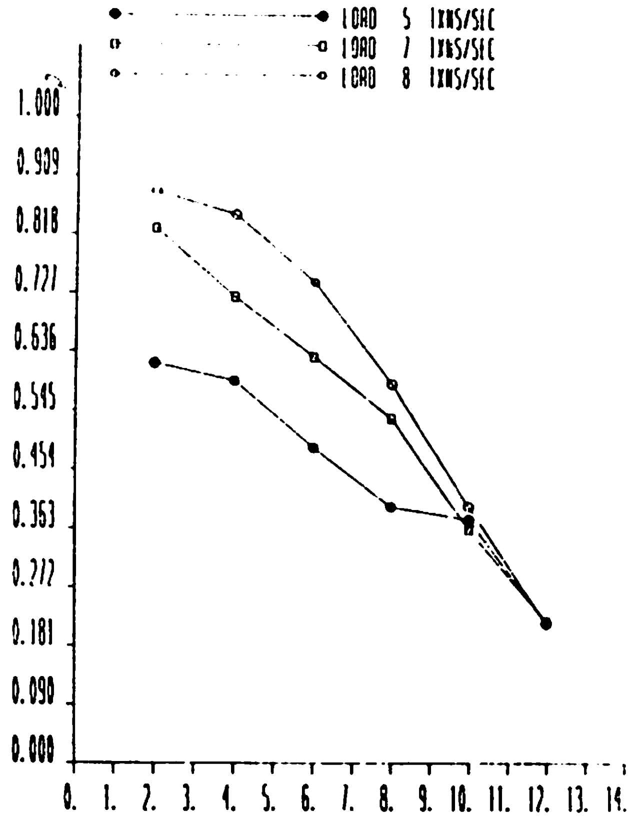

Figure 11b: Effect on pseudo-commit time

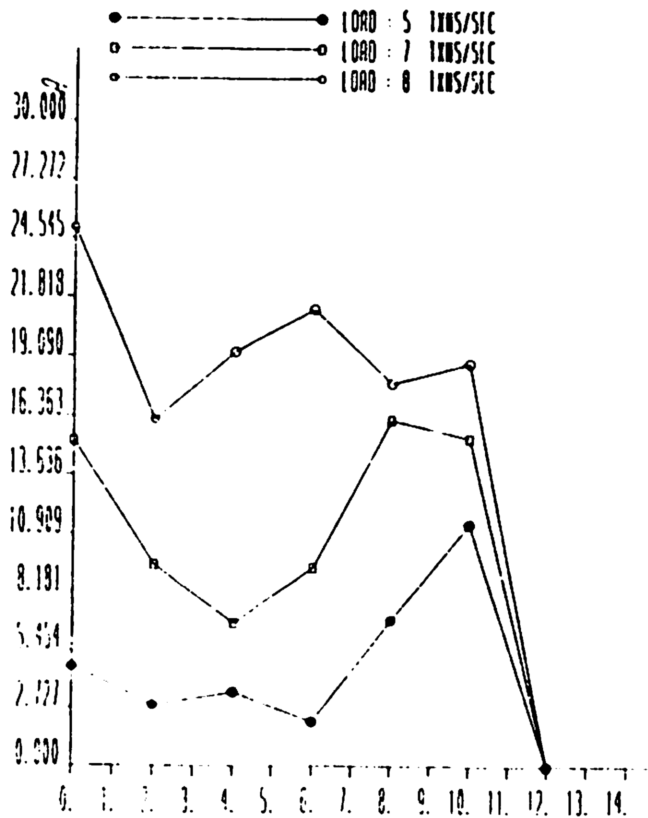

Figure 11c: Effect on aborts

Figure 11: Simulation Results for $P = 1$, Transaction length = 7 steps

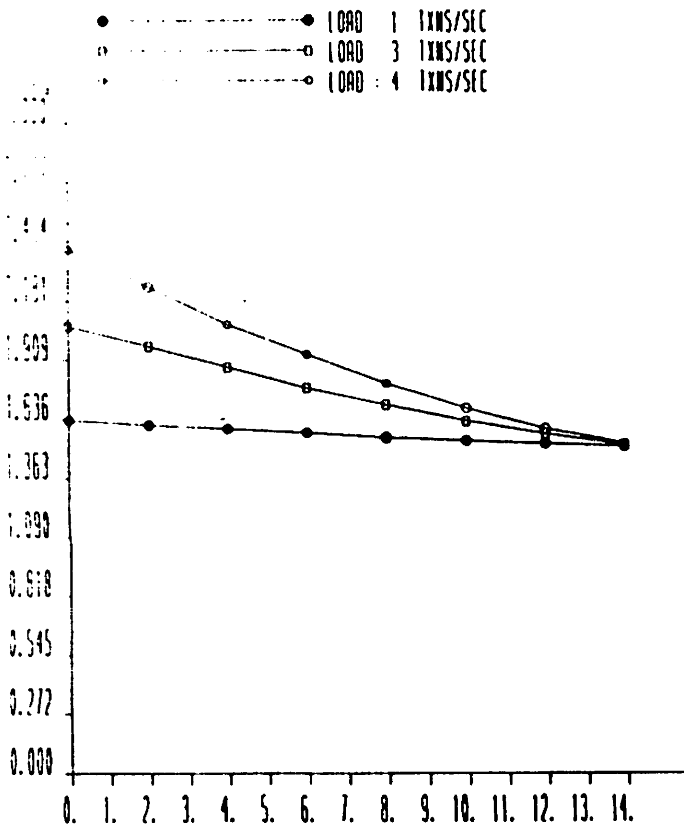

Figure 12a: Effect on response time

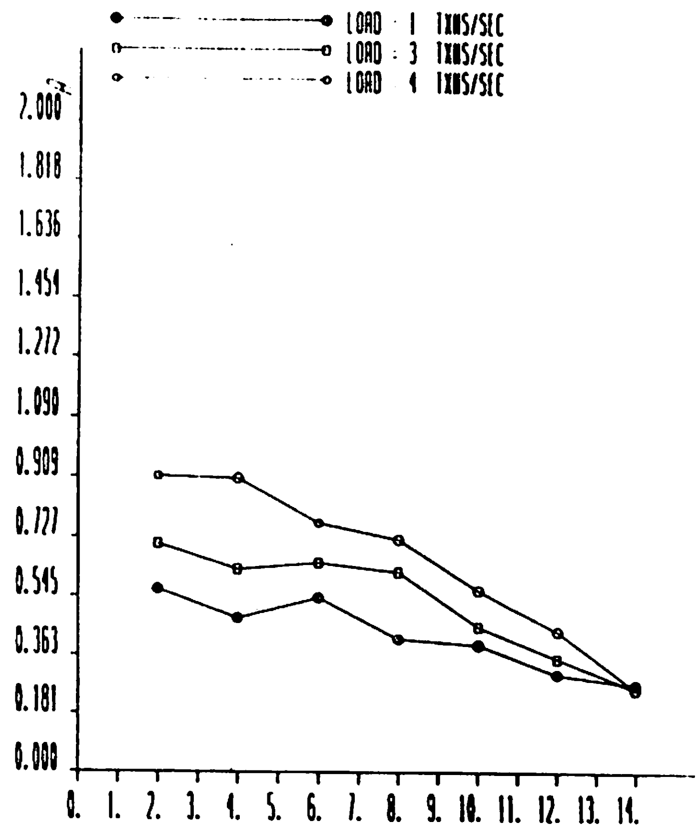

Figure 12b: Effect on pseudo-commit time

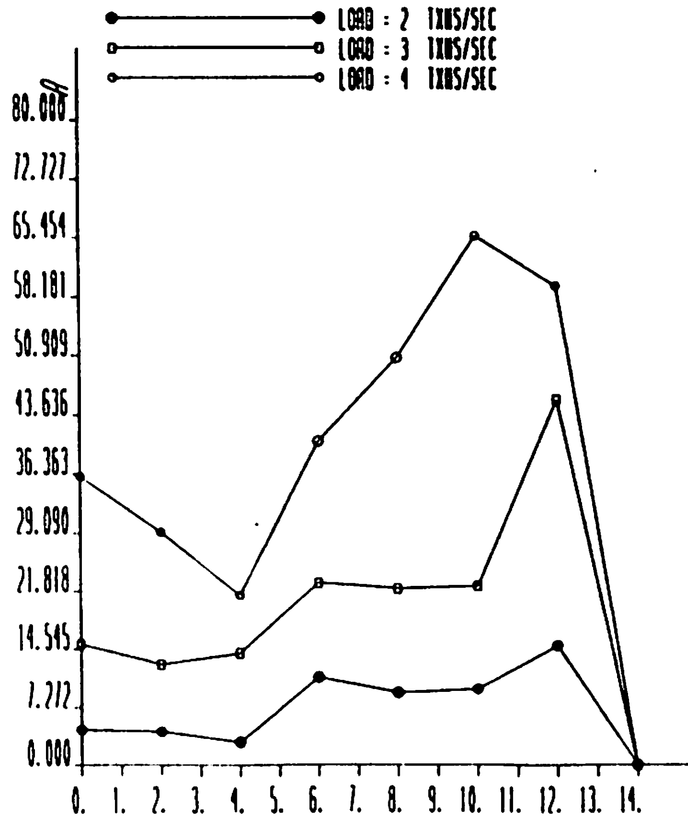

Figure 12c: Effect on aborts

Figure 12: Simulation Results for P_c = 2, Transaction length = 9 steps

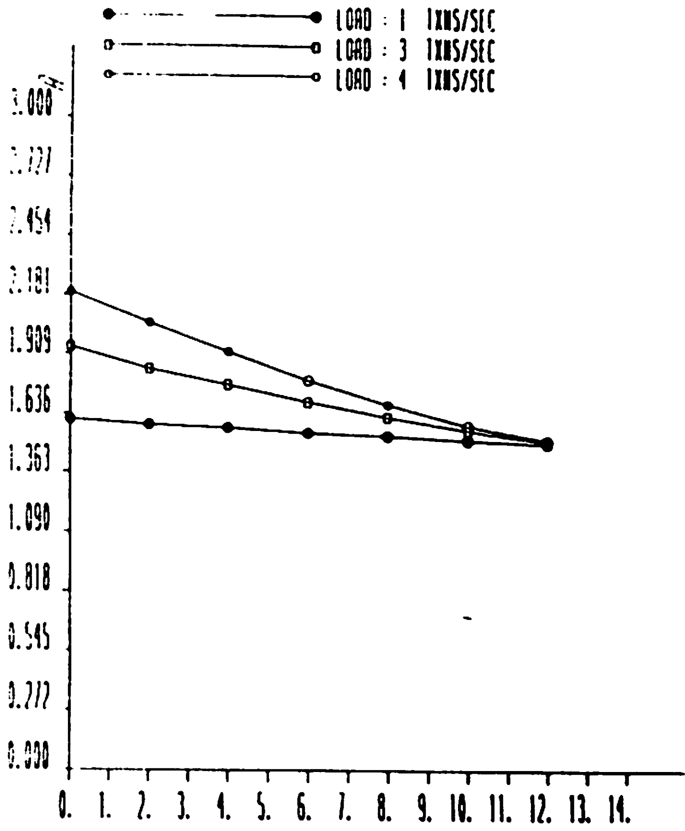

Figure 13a: Effect on response time

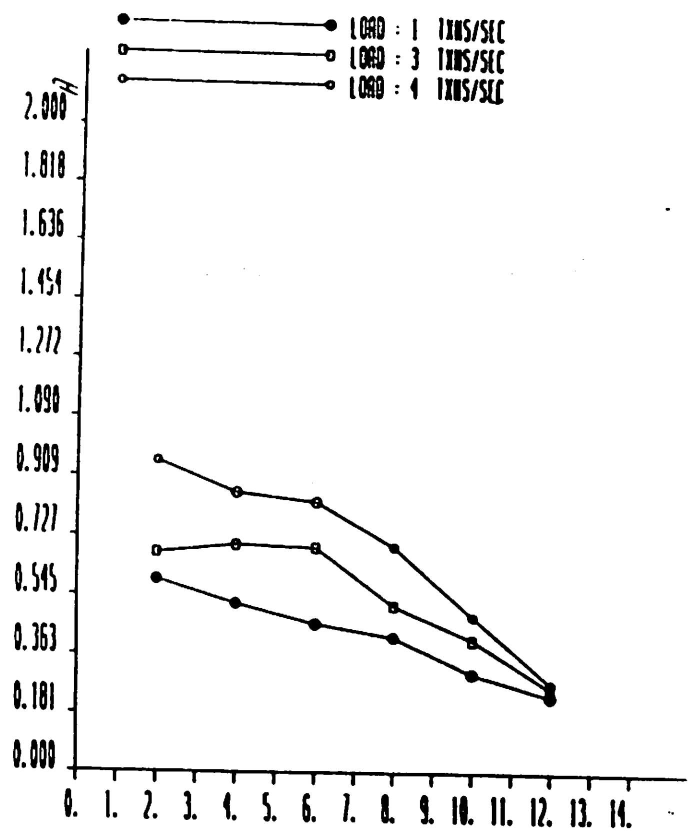

Figure 13b: Effect on pseudo-commit time

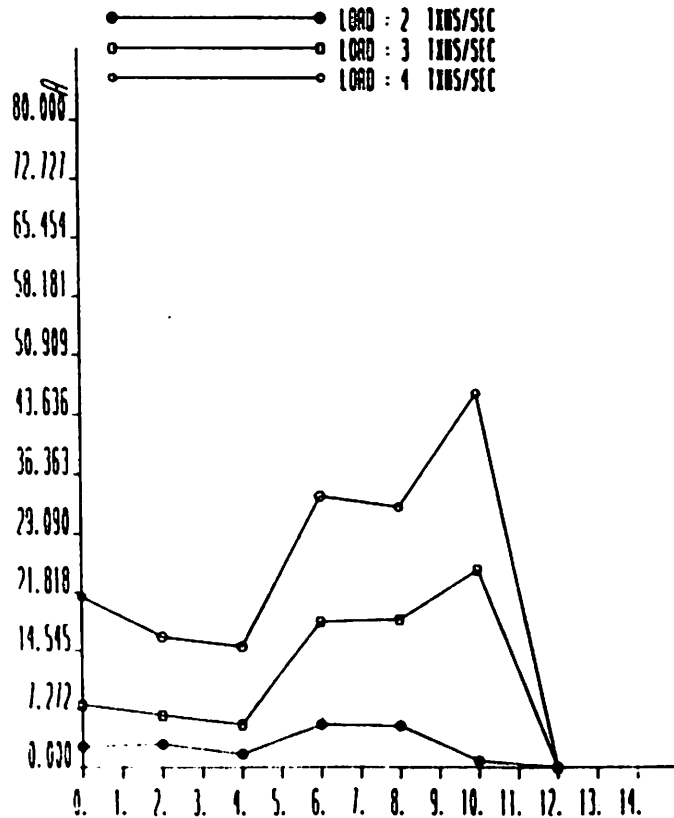

Figure 13c: Effect on aborts

Figure 13: Simulation Results for P_ℓ 4, Transaction length 9 steps