Performance Evaluation of Semantics-based Multilevel Concurrency Control Protocols

B. R. Badrinath*
Dept of Computer Science
Rutgers University
New Brunswick, NJ 08903
badri@cs rutgers edu

Krithi Ramamritham†
Dept of Computer and Information Science
University of Massachusetts
Amherst, MA 01003
krithi@cs.umass edu

Abstract

For next generation information systems, concurrency control mechanisms are required to handle high level abstract operations and to meet high throughput demands The currently available single level concurrency control mechanisms for *reads* and *writes* are inadequate for future complex information systems In this paper, we will present a new *multilevel* concurrency protocol that uses a semantics-based notion of conflict, which is weaker than commutativity, called *recoverability*. Further, operations are scheduled according to *relative conflict*, a conflict notion based on the structure of operations.

Performance evaluation via extensive simulation studies show that with our multilevel concurrency control protocol, the performance improvement is significant when compared to that of a single level two-phase locking based concurrency control scheme or to that of a multilevel concurrency control scheme based on commutativity alone. Further, simulation studies show that our new multilevel concurrency control protocol performs better even with resource contention

1 Introduction

For the next generation information systems, concurrency control mechanisms are required to handle high level abstract operations on complex objects and to meet high throughput demands Examples of applications requiring the support of complex information systems include office information systems (OIS), stock trading databases, Software Engineering, CAD/VLSI, and real-time systems. Most of these applications are complex, require high performance, and are not well supported by existing database management systems where the operations are just *reads* and *writes* on *uniform* sets of data like *tuples* and *records* [20, 21, 22, 23] Traditional approaches to concurrency control are highly inefficient for these types of databases because they ignore much of the semantic information that is available in the emerging information intensive applications The work described in this paper, involves the development of a *high performance* multilevel concurrency control protocol that takes advantage of the *operation semantics*, i.e , the synchronization properties of the operations, and *operation structure*, i e , the hierarchical structure of the operations

In the context of the work described in this paper, we define *complex information systems* as those that need to support one or more of the following features

1 *Abstract operations*— Facilities exist to define data abstractions, hence the operations in complex information systems are not just *reads* and *writes* but are *arbitrary* Conflict between operations depends upon the semantics of the operations.

2. *Complex Operations*— Due to the increasing complexity of applications, the operations issued by transactions are not simple. High level abstract operations are decomposed into sub-operations and these sub-operations are further decomposed into simpler operations.

Thus, in systems supporting complex operations, we have operations at various levels, and at each level, operations have specific synchronization properties. Concurrency control at the leaf level alone, although correct, as we will see, is unnecessarily restrictive Thus, a multilevel (M-L) approach to the problem of concurrency control seems not only *natural* but also *necessary* to meet the high throughput demands in complex information systems

Traditionally, single-level and multi-level concurrency control protocols use conflict notions based on the commutativity of operations. In this paper, we will present a new *multilevel* concurrency protocol that uses a semantics-based notion of conflict, which is weaker than commutativity, called *recoverability* Further, operations are scheduled according to *relative conflict*, a conflict notion based on the structure of operations

The performance of various protocols is evaluated through extensive simulation studies The performance characteristics of these protocols is examined by comparing the performance to a multilevel protocol based on commutativity as well as to that of single level protocol based on locking Recent studies have shown that the performance of concurrency control strategies is sensitive to assumptions about resource contention in the system [1] Hence, the implications of resource contention on multilevel semantics-based concurrency control protocols proposed in this paper are examined In the rest of this section, we provide an introduction to recoverability, multi-level concurrency control, and relative conflicts

*Work supported in part by Henry Rutgers Research Fellowship award

†Work supported in part by NSF under grant DCR-85000332.

Recoverability-based Conflict Resolution 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 In our scheme, non-commuting but *recoverable* operations are allowed to execute concurrently, 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, consider a *write* operation that is invoked by T_j after a *read* operation has been executed on a page by T_i These two operations do not commute, but if the *write* and the *read* operations are forced to commit in the order they were invoked, then the execution of the *read* and *write* operations is serializable in commit order Further, if either of the transactions invoking these operations aborts the other can still commit What makes recoverability an attractive concept is that it permits more concurrency than commutativity while retaining the positive feature of commutativity, namely, avoiding cascading aborts Cascading aborts are avoided because even if one of the transactions involved in a commit dependency aborts, the other can still commit

Transactions can 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 ensure that the commit dependency relationship is acyclic This is similar to the validation phase in optimistic protocols [14] If a transaction wanting to commit is in a commit dependency cycle then that transaction is aborted.

Multilevel Concurrency Control: In complex information systems, high level operations are translated into sequences of lower level operations, where each level has its own set of operations Thus, operations at one level considered as transactions issue suboperations at lower levels, which can invoke further sub-operations and so on Consider the example shown in Figure 1 Each *increment* operation on a counter is implemented as a *read* of the initial value followed by a *write* of the new value In the first computation, a *read* of the second *increment* is interleaved between *read* and *write* of the first *increment* Since a read and a write operation on the same element conflict, w1(x) conflicts with r2(x) and hence a dependency between these sub-operations of the increments is formed Similarly, w2(x) conflicts with r1(x) forming a dependency between r1(x) and w2(x). We are interested in the correctness of the computation resulting from the concurrent execution of *increment* operations Thus concurrent operations should inherit any conflicting dependencies among their sub-operations The increment operations in Computation I inherit cyclic conflict dependencies implying that the sub-operations were not executed atomically In this case, this results in a lost update Hence in this concurrent execution, the specifications of the two *increments* are not met, and so Computation I is unacceptable However, in Computation II, the sub-operations of *increments* are not interleaved, and the increment operations inherit acyclic conflict dependencies Since two *increments* commute, they do not conflict, and either order of execution is equivalent to a serial execution

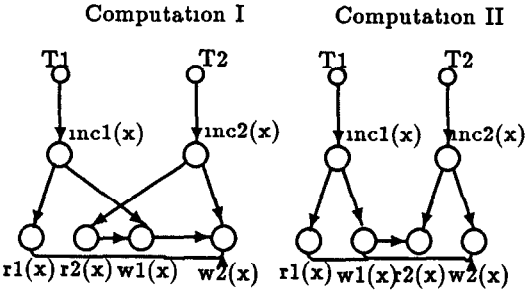


Figure 1 Multilevel computation

In order to avoid mutual dependencies and allow only serializable executions of top level transactions, nested two phase protocols have been proposed [15, 17, 25]. These multilevel concurrency protocols allow more concurrency than standard two phase locking which requires leaf level operations to acquire locks and retain them until the top-level transaction completes However, in a multilevel concurrency control protocol, the leaf level locks are retained only until their parent operation completes Thus, locks are retained over a shorter duration in this scheme than in a single level concurrency control scheme Hence, multilevel protocols potentially provide enhanced concurrency In these nested protocols, concurrency is controlled at each level, and they are based on locking to execute an operation on an object, a transaction must acquire a lock appropriate to that operation at a given level l Since operations are composed of nested sub-operations, in these protocols the execution of each operation results in a number of locks being acquired for the sub-operations at $l-1$ Here we assume that level n corresponds to top-level transaction and level 1 corresponds to leaf-level operations A lock is granted if there no other conflicting lock is held on that object A lock acquired by an operation prevents other conflicting operations from being executed until the lock is released Further, locks at a given level can be released when the parent operation commits without having to wait until the top level transactions complete Thus, the duration of a level l lock acquired by an operation is from the time it is acquired until the completion of its parent at level $l+1$ Once an operation releases a lock, it cannot request further locks Finally, the locks corresponding to the top level operations are released by the transaction that initiated the operations Thus, locks are acquired in a top-down manner and released bottom-up These nested two phase protocols are sufficient to guarantee correctness (serializability) of top-level executions Further, in these protocols, two operations conflict if they are *non-commuting* In this paper, we present a multi-level protocol based on the conflict notion of recoverability

Relative conflict among sub-operations of a complex operation. Consider Computation I in Figure 1 following a nested two phase protocol When r2(x) is requested, a read lock is granted but this read lock forces w1(x) to wait In the conventional sense this can be termed a deadlock but it is interesting to point out that these cyclic waits were generated not by user requests but due to requests stemming from the decomposition of operations into sub-operations within the system Further, the possibility

of cyclic waits exists at every level! Thus, in the context of implicit nested environments, the idea of executing non-conflicting operations immediately may not be a good idea after all

The problem of deadlock among sub-operations of non-conflicting operations has been ignored by most researchers working on multilevel concurrency control protocols A better solution would be to prevent a mutual dependency without getting into deadlocks or cyclic waits Thus, a solution would be to make r2(x) wait until w1(x) completes and then schedule r2(x). This is what happens in Computation II of Figure 1 In this case we say that r2(x) has a *relative conflict* with r1(x), i e , r2(x) and r1(x) conflict relative to the parent of r1(x) Making r2(x) wait even though it does not conflict with r1(x) in the conventional sense has two significant implications· first, once w1(x) is allowed to complete, a lock corresponding to an increment operation can be retained on x thereby releasing read and write locks on x, and secondly, cyclic waits or deadlocks can be prevented Under what circumstances is it useful to make a non-conflicting operation wait? What properties of the operations allows us to make such a decision? These are some of the questions we will answer in Section 4

Broadly speaking, the information about proscribed interleavings of the sub-operations will be used to determine relative conflict between the sub-operations This information can be obtained if the structure of the abstract operation is known so that the look-ahead information can be used to determine the siblings of a given sub-operation. In the past, this view of anticipating future steps in single level systems has been in some sense considered "unnatural" [19]. However, in multilevel operations this sort of information is readily available

The outline of the paper is as follows In Section 2, we review some related work in this area. Section 3 provides further details of recoverability In Section 4, we present a semantics-based multilevel concurrency control protocol that uses recoverability Section 5 describes how the information about the operation structure can be exploited in scheduling operations using relative conflict In Section 6, we describe the simulation studies used to evaluate the performance of multilevel protocols Section 7 summarizes the results and outlines some future work

2 Related Work

A formal model to reason about the correctness of nested computations was proposed in [7, 6] Recently, a number of researchers have proposed simplified models to prove and state correctness criteria for multilevel concurrent computations [25, 17, 15, 10] In [25, 17] it was shown that in order to prove that a top-level computation of a multilevel computation is serializable, it is sufficient to show level by level serializability, and guarantee that the order of execution of operations is preserved in between levels Further, a top down nested two phase locking was proposed to achieve multilevel concurrency control. In [15] a formal model was proposed for nested objects which also corrected some problems that existed in the model of [25] In [16], a bottom up protocol was presented for nested objects All these protocols use commutativity as a basis for defining conflicts

In [3] we introduced the notion of recoverability In [5], we reported the performance of a single level concurrency control algorithm using *recoverability* Recently, a special concurrency control technique based on failure commutative transactions has been proposed for the XPRS system [23] Failure commutativity is an adaptation of our notion of recoverability but applied to transactions Transactions are classified according to whether they failure commute or not

The term recoverability also appears in [11, 9] 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 [18] 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.

Definitions of conflicts based on serial dependency relations have been used in optimistic concurrency control schemes and multiversion timestamp schemes for abstract data types in [12, 13] Two operations conflict if they invalidate each other This definition is weaker than commutativity which requires equivalence of states In order to determine whether a given operation conflicts, a view is constructed from the active operations to determine whether the new operation produces a legal history Construction of a view is necessary because a restrictive but modular condition known as local atomicity is achieved in [12, 13] Further, the concurrency control protocols are single level applied to simple operations and objects

In this paper we extend the concept of recoverability as a notion of conflict to multilevel concurrency control Further, operations are scheduled according to the new notion of relative conflict which uses information about the structure of operations Scheduling operations according to relative conflict prevents some possible deadlocks, thereby avoiding unnecessary restarts.

3 Commutative and Recoverable Operations

We will assume that 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 $S = \{s_1, s_2, \ \}$ is a set of *states* and $V = \{v_1, v_2, . \ \}$ 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[1] 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 in state s is immediately followed by the execution of o_2 We say that operation o_2 is *recoverable relative to operation* o_1, denoted by $(o_2 \ RR_I \ o_1)$, iff for all $s \in S$

$$\text{return}(o_2, state(o_1, s)) = \text{return}(o_2, s)$$

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 the same whether or not 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

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

Definition 2: Two operations o_1 and o_2 *commute* if for all states s, state(o_2, *state*(o_1, s)) = state(o_1, *state*(o_2, s)), return(o_1, s) = return(o_1,state(o_2, s)) and return(o_2, s) = return(o_2,state(o_1, s))

Lemma 1: If o_1 and o_2 commute then (o_2 RR_I o_1) and (o_1 RR_I o_2) □

From the lemma, we can make the following observations First, Commutativity is a symmetric property whereas recoverability is not Secondly, commutativity implies recoverability So in the remaining sections, if we imply recoverability from commutativity, we will explicitly state so.

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 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 $\forall_{1 \le i < n}$ $o_i <_E o_{i+1}$ (o_n RR o_1) if the return value of o_n is the same whether or not 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, $1 \le l < n$, (o_n RR_I o_l) then (o_n RR o_1).

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

Thus, we will consider two operations o_i and o_j to be non-conflicting if either o_i and o_j commute or o_j RR o_i (o_j is recoverable with o_i) Therefore, a give operation o_j can be allowed to execute concurrently if it commutes or is recoverable with all other uncommitted operations

3.1 Example of a set

We will consider an object such as 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 Using a compatibility table we will elucidate the type of dependencies that exist between various operations Our derivation of the dependencies is based on the definitions of commutativity and 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). We use the notation (a, b) to mean an operation a is invoked when operation b has been executed Thus in Table 1, (insert, insert) is commutative and in Table 2, (insert, member) is recoverable 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

The traditional notion of conflict on these objects such as a page with only read and write operations has been that two operations conflict if one of them is write However, with recoverability, this notion of conflict is weakened as the only pair of operations considered conflicting is (read, write) Thus, even for the read/write model of transactions, the potential for parallelism increases under recoverability semantics

We have considered, as examples, most commonly used objects and as we can see from the tables recoverable operations are not uncommon Hence, we intend to exploit this weaker notion in enhancing concurrency

Table 1 Commutativity for Set

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

Table 2 Recoverability for Set

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

4 A Recoverability Based Multilevel Concurrency Control Protocol

Even for systems in which high level operations are decomposed into sub-operations at several levels, traditional concurrency control demands that leaf level operations acquire locks and retain them until the top-level transaction completes However, the multilevel protocols described in Section 1 can be used to provide enhanced concurrency

While previous multilevel concurrency protocols consider operations to conflict if they are not commutative, in our scheme, conflicts between operations are determined based on recoverability The definition of recoverability, in Section 3, assumed that the operations were executed atomically by the system This assumption is valid for single level systems However, in the case of a multilevel system we can make that assumption only for leaf level operations Thus, the multilevel protocols must ensure that intermediate level operations are in effect executed atomically If operations are allowed to execute concurrently only if they commute, then nested two phase protocols guarantee serializability of top-level operations and atomicity of operations Even with *recoverability* we need to ensure that the sub-operations are in effect executed atomically and that the execution order is preserved at all levels Recall that, when recoverable operations are executed by transactions they 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 an operation can *complete* execution even though it can commit (release locks) only after all operations on which it depends terminate, thus respecting the commit dependency relationship In order to keep track of commit dependencies, recoverable operations will have to acquire *lists* that are used to contain the IDs of the uncommitted operations with which a given operation is recoverable Before describing how both atom-

icity of operations can be guaranteed and execution order of operations can be preserved at all levels, some definitions are needed

Definition 4: A non-leaf level operation is said to have *completed* once it releases all the locks and lists acquired for the sub-operations (i e , all sub-operations have completed) An operation is said to be *in-progress* if any of its sub-operations has not been completed

Definition 5: Two leaf level operations o_i and o_j are non-conflicting if o_i and o_j commute or o_i RR o_j.

Definition 6: Two non-leaf level operations are non-conflicting if either 1) o_i and o_j commute or 2) o_i RR o_j and o_j has *completed*

Note that in the above definition, for non-leaf level operations, recoverability is used only with respect to *completed* operations i e , those operations all of whose sub-operations have completed This ensures that when a recoverable operation is executed, its effects are as though the operation executed atomically and further, the state changes and the return values reflect the order of execution. Since this is done at each level, the execution order of operations is preserved at each level

We now present our new nested protocol that uses recoverability as a basis for determining conflicts In this nested protocol, each operation is required to acquire a *lock* or a *list* depending upon the type of operation

Here are the rules of the nested protocol

1. If an operation request at a given level conflicts i.e , is *non-recoverable*, it is delayed until conflicting operations release their *locks* or *lists*.
2. If an operation request is non-conflicting, either
   a) the operation acquires a *lock* if it commutes with all existing operations or
   b) the operation acquires a *list* if the operation either commutes with other operations or it is recoverable with other *completed* operations (if it is a leaf level operation, it should be recoverable with other operations), where the *list* contains the ID's of the operations with which the given operation is recoverable
3. Once an operation releases a *lock* or a *list* it may not acquire any more *locks* or *lists*
4. A *lock* acquired for an operation can be released any time, however, a *list* can be released only when it is empty
5. When a *lock* or a *list* corresponding to an operation is released, its ID is removed from all *lists* in which it is a member

The first 3 rules are the same as in the nested two phase protocol of [15, 17; 8], with embellishments to track dependencies due to recoverability. Rule(4) guarantees that operations complete in the order in which they form dependencies due to recoverable operations Note that in rule (2b), a non-leaf level operation is considered conflicting if there is an in-progress operation with which it does not commute. This is because, the notion of recoverability for non-leaf level operations is applied only with respect to completed operations

Here is a rough sketch of the proof of correctness of our protocol Note that rule(4) guarantees that a given operation will complete (release all locks) only after the completion of any operation with which it is recoverable Since two operations with mutual commit dependencies cannot complete, there can be no cyclic conflict dependencies at any level of the computation resulting from the use of recoverability as a conflict predicate Thus, at each level there are no inherited mutual dependencies due to conflicting sub-operations Further, at non-leaf levels, recoverability, as a notion of conflict, is applied with respect to operations that have completed Thus, suppose two operations are considered as non-conflicting because they are recoverable Then all the sub-operations of one operation are completed before any of the sub-operations of the other is invoked This preserves the order of execution of the operations

In the next section, we will see how we can further enhance the concurrency obtained by our protocol by exploiting information about the operation structure

Table 3: Conflict table for a record

Operation Requested	Operation Executed: Read	Operation Executed: Write
Read	Yes	No
Write	No	No

5 Exploiting Operation Structure in ML Concurrency Control

As indicated in Section 1, in the context of multilevel concurrency control protocols for complex operations, we need to reexamine the conventional wisdom of executing non-conflicting operations immediately We propose scheduling operations within a complex operation based on a new notion called *relative conflict*, i.e , conflict between operations *relative to* the (parent) operations that invoked them For example, in the multilevel computation of Figure 1, the sub-operations of *increment* operations should be non-interleaving to avoid deadlocks.

We define *relative conflict* as follows two non-conflicting operations o_1 and o_2 have relative conflict iff they cannot be interleaved among the sub-operations of parent(o_2) and parent(o_1) respectively, to ensure the correct execution of parent(o_1) and parent(o_2)

Note that multilevel relative conflict information for an operation is dependent only on its parent operation We apply the definition of relative conflict only to operations and sub-operations and not to transactions This is because, assuming that relative conflict is applicable to transactions implies that transactions have statically declared their operation sets, an unnatural assumption to make in most applications we are interested in. Thus, the notion of relative conflict uses information only about *operation structure* and *not transaction structure*

It is important to point out that the notion of *relative conflict* has significant implications not only for scheduling complex operations but also for concurrency control in a *traditional database* such as System R[2] System R uses page level and record level locking Thus we can consider synchronization to be occurring at multiple levels The conflict table for records is given in Table 3 A read operation on a record is implemented by fetching the page to which the record belongs This fetch operation is denoted by fetch-R A write operation is implemented as fetching the page to which the record belongs and storing the new value on that page. These operations are denoted as fetch-W and store-W respectively The relative conflict table for a page is given in Table 4.

Our scheduler, using relative conflict, at the page level

Table 4 Relative conflict for a page

Operation Requested	Operation Executed		
	Fetch-R	Fetch-W	Store
Fetch-R	Yes	Yes	No
Fetch-W	Yes	No	No
Store	No	No	No

will not allow two fetches to be executed concurrently when one of the fetches is the child of a write operation on the record This is because the scheduler knows that fetch for a write will be followed by a store

Based on these ideas, we have developed a nested protocol, called ML-RC, that makes use of *relative conflict* information ML-RC is similar to the nested protocol described earlier in Section 4 In addition to acquiring and releasing locks appropriate to an operation, it exploits a special type of state that can be entered by an operation, namely the *completed state* w r t a particular object

Definition 7 An operation is in a completed state w r t a particular object, if all the sub-operations on that object have been invoked

An operation after entering the completed state w r t a particular object may not request any more sub-operations *on that object* Thus, we not only have a release phase for locks but also an acquire phase for locks on individual objects The reason for requiring an operation to indicate a completed state w r t an object, after which it may not request any more sub-operations on that object will be clear when we discuss the effect of using *relative conflict* as a conflict predicate for scheduling sub-operations The salient rules for ML-RC are

1 Each pair of operations on a given level is classified as conflicting or non-conflicting

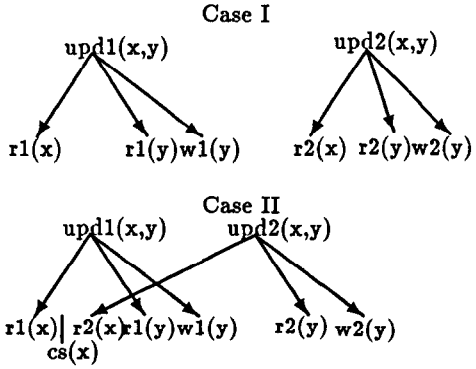
2) At each level, for each requested operation test if the operation conflicts with any of the active operations If it conflicts then delay the request If not, check if it has *relative conflict* with any of the operations If there is no relative conflict, schedule the operation Otherwise, check if the parent of the operation has reached the *completed* state for this object. If so, schedule the operation, else delay the request

3) Once a *completed* state is reached by an operation on an object, the operation may not request any more locks on that object

4) Once an operation at a given level l has *released* a lock it may not obtain any more locks on any object at level $l-1$

5) Once all locks are released at level $l-1$, the operation retains its lock at level l until it is released by the parent at level $l+1$

The reason for (3) is as follows A given operation may have sub-operations on multiple objects Without the notion of a completed state for a particular object, two operations having relative conflict will end up executing all their sub-operations serially However, the sub-operations of one operation can be interleaved with sub-operations of the other on different objects without getting into cyclic waits In order to avoid the situation of having to execute all sub-operations serially (e g , Case I in Figure 2) we have introduced the notion of completed state on objects This is shown in Case II of Figure 2, where once operation upd1(x,y) reaches completed state on object x (denoted by cs(x)), upd2(x,y) can be allowed to execute r2(x)



r2(x) has relative conflict with r1(x)
But r2(x) can proceed after upd1(x) reaches the completed state for object x

Figure 2 Nested protocol ML-RC

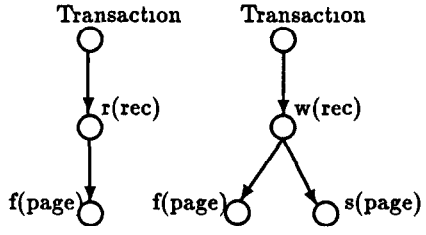


Figure 3 Three level system

6 Simulation Studies

The purpose of our simulation studies is two fold First, compare the amount of concurrency offered by a single level concurrency control to that of a multilevel concurrency scheme Second, compare the amount of concurrency offered when both commutativity and recoverability are used to determine conflicts as opposed to using just commutativity We will evaluate the performance of three protocols the first protocol, SL, is the single level protocol where transactions acquire locks for leaf level operations and retain them until the transaction completes The second protocol, ML-C, is the multilevel protocol based on nested two phase locking that uses only commutativity in determining conflicts The third protocol, ML-RC, is our new multilevel protocol that uses recoverability in addition to commutativity in determining conflicts and relative conflict for scheduling operations

To simplify simulation studies, we examine a three level system shown in Figure 3. The first level constitutes transactions, in the second level the operations are restricted to be reads and writes on records, and in the third (leaf) level the operations are fetches and stores on pages Each operation request by a transaction is either a read or a write operation on a record In determining the performance of concurrency control protocols, we are interested not only in the effect of data contention, but also in the effect of resource (for example, CPU or I/O) contention on the performance of semantics-based concurrency control protocols Hence, we have studied the effect of availability of both infinite re-

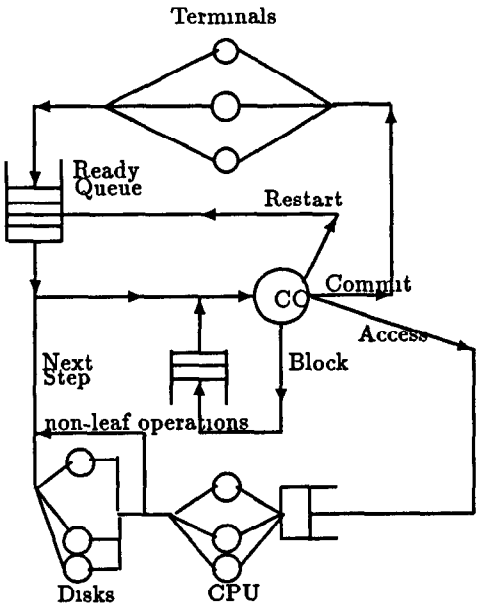


Figure 4 Simulation model

sources and limited resources

6.1 The Simulation Model

The parameters of the model, their meanings, and their nominal values are listed in Table 5 The values of the model parameters are similar to those in previous performance studies of locking protocols [24, 1] The closed queuing model shown in Figure 4 is a modified version of the one used in [1]

There are a fixed number of terminals from which transactions originate The maximum number of active transactions in the system at any given time is the multiprogramming level, the *mpl level* A transaction's length is determined by the number of operations executed by it This parameter, the *transaction length*, is distributed uniformly between min length and max length so that the average transaction length is (*min length*+*max length*)/2 A transaction originates from any of the terminals If the number of active transactions is equal to the *mpl level* then the transaction enters the ready queue, until another transaction commits or aborts The transaction then starts issuing operation requests We have conducted extensive simulation studies for various levels of multiprogramming beginning with 10 all the way up to 200 with the number of terminals chosen to be 200 The transaction length and the level of multiprogramming determine the overall transaction load The transaction load is adjusted by changing the level of multiprogramming For a given level of multiprogramming, different transaction lengths indicate different workloads Instead of running the experiments with fixed transaction sizes, we use a transaction mix consisting of transactions whose length is uniformly distributed random variable between 4 and 12 operations

Table 5 Simulation parameters and their values

Parameter	Value
Num of levels	3
Database size	1500
Num of terminals	200
Mean Transaction length	8 steps
Max length	12 steps
Min length	4 steps
Mpl.level	10, , 200
Step time	0 05 secs
Step time	0 05 + 0 0425 secs
CPU time	0 0075 secs
IO time	0 035 secs
Resource units	1, 5, and ∞
Ext think time	1 secs
Commit delay	0 6 secs
Write probability	0 3

If a requested operation is incompatible with ongoing operations, the invoking transaction is blocked Deadlock detection is initiated every time a transaction blocks A transaction is aborted if a deadlock is discovered or else the transaction is made to wait until the conflict is resolved The deadlock detection algorithm uses both the wait-for graph and the commit-dependency graph maintained as lists to determine a whether a deadlock exists, as deadlocks may occur due to both wait-for dependency and commit dependency Transactions also abort due to cyclic commit dependencies arising from executing recoverable operations If a deadlock is detected, the transaction invoking the algorithm is chosen as the victim and aborted An aborted transaction is restarted immediately A restarted transaction behaves, with respect to operation invocations, like a new, independent transaction A blocked transaction is *retried* every time any transaction that issued a conflicting operation on that object completes

In the case where finite resources are present, each non-leaf step requires a CPU for an interval of length *cpu time* and a leaf-level step requires a CPU for an interval of length *cpu.time* and disk access for an interval of length *io time* The total time for which these resources are used is equal to *step time* The parameter *step time* is the execution time of each operation If an operation request is granted, after the execution time of the operation, the operation invokes sub-operations Under the assumption of *infinite* resources, *step time* represents a constant service time for each operation Each step of a transaction at non-leaf levels requires 0 0075 secs of CPU time and each step at leaf level takes 0 0075 secs of CPU time and 0 035 secs of disk access time Thus, in the case of infinite resources for two levels each read operation takes 0 05 secs and a write operation takes 0 0925 secs (0.0075 + 0 0425 + 0.0425) as it has two sub-operations, a fetch and a store

We consider a CPU and two disks to constitute one resource unit The number of resource units is a model parameter *resource unit* When a transaction needs a CPU, it is assigned a free CPU from a pool of CPU's, otherwise the transaction waits until one becomes free For the I/O part, there is a separate queue associated with each disk When a transaction needs to access a disk, it chooses a disk at random and waits in the queue of the selected disk until it can be served [1] In order to study the effects of resource related assumptions, we have repeated the experiments with different number of resource units For the finite resource case, resource contention manifests itself as waiting for CPU and disks

We assume that the probability that a write operation is requested on an object is determined by the parameter *write probability* chosen to be 0 3 Further, we assume, as in [24], that there is uniform access, that is, the probability

that a transaction chooses an object at each level to execute an operation is a uniformly distributed random variable between 1 and number of objects at that level Thus, when a read operation is executed, a fetch sub-operation is generated The object on which this fetch sub-operation is executed is chosen as explained above When a write operation is executed, two sub-operations fetch and store are generated, which are executed in sequence on the same randomly chosen object In this study, the number of pages (at level 1) was chosen to be 500 and the number of records (at level 2) was chosen to be 1000 yielding a database size of 1500 This database size was chosen to yield good conflict rates so that interesting evaluation of semantics based concurrency control schemes can be obtained

After a transaction completes, the terminal that issued the transaction will initiate a new transaction after a think time given by an exponentially distributed random variable with mean *ext think time*. We do not model the details of the overheads of the commit process; However, we take into account the cost of the commit process by introducing a delay called the *commit delay* for each commit of a transaction This delay is fixed at 0 6 seconds The overheads not considered in our model are the costs involved in maintaining the commit-dependency graph and the overheads of recovery following an abort

6.2 Performance Metrics

The two main performance metrics used in the evaluation are the *throughput rate* and the *response time*(turnaround time) The throughput rate is measured as the number of transactions that successfully complete or commit per second The average throughput 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 higher the average throughput rate The response time in seconds is measured as the difference between when a terminal submits a transaction and when a transaction successfully completes The time includes any time spent in the ready queue and *time spent due to restarts* Since we are simulating a closed queueing system, the response time is the inverse of throughput, hence, we do not explicitly show response time

The other two metrics related to determining the usefulness of semantics in concurrency control are blocking ratio and restart ratio. *Blocking ratio* is the average number of times a transaction blocks per commit This should give a fair indication of the conflict in the system. The *restart ratio* is defined as the average number of times a transaction is restarted per commit

6.3 Simulation Results

We will study the performance of three different protocols, namely, SL, ML-C, and ML-RC The graphs in Figures 5 through 7 show the average results of 10 runs, where each simulation was run till 50000 transactions were completed to obtain sufficiently tight 90 percent confidence intervals. Though, the confidence intervals are omitted from our graphs, the confidence intervals were within the range of a few percentage points of the mean value of the performance metrics shown in the various graphs The maximum value of the confidence interval was ±2% points

First, we determine various performance characteristics when strict two phase locking is used Here the locks for the leaf level operations are retained until the top-level transaction completes Second, to determine the relative performance, we include conflicts defined based on commutativity in multilevel protocols, where locks for sub-operations are retained until the operation completes and not for the duration of the top-level transaction Third, we include conflicts based on recoverability and use relative conflicts for scheduling sub-operations of non-conflicting operations

Infinite Resources: In this part of the simulation, we assume infinite resources Figure 5 shows throughput results for different values of multiprogramming The throughput under all three protocols, namely SL, ML-C, and ML-RC, increases with multiprogramming level and after certain level the throughput begins to drop as multiprogramming increases The drop in throughput is due to thrashing resulting from very high data contention When the data contention increases, so does the blocking due to denial of lock requests and restarts due deadlocks, as a result of which the throughput drops The maximum throughput is obtained with ML-RC at mpl level = 100 As the multiprogramming level is increased beyond mpl level = 50, the throughput begins to drop for both SL and ML-C. However, thrashing begins to occur at mpl.level = 100 for ML-RC Observe that the throughput with ML-RC is higher than both ML-C and SL even with thrashing at high values of multiprogramming Further, the relative improvement in throughput for ML-RC increases as the level of multiprogramming increases

The restart ratio and the blocking ratio for various protocols, due to lack of space, are not shown here, but can be found in[4]

Finite Resources: In this part of the simulation, we conducted experiments for two cases First, for a database with 5 resource units and second with only 1 resource unit The 1 resource unit case models high resource contention and the 5 resource units case simulates a multiprocessor database

Figure 6 and 7 show the throughput results for 5 and 1 resource units respectively Observe that the maximum throughput is obtained with ML-RC in both these cases Further, the throughput obtained with finite resources at a given level of multiprogramming is smaller than the corresponding throughput obtained with infinite resources This is to be expected as transactions will have to wait for resources to become available before they can execute non-conflicting operations In the case where only 1 resource unit is present, the throughput is very low and thrashing begins at mpl level = 25 This is because all the available resources are busy and resource contention becomes the primary bottleneck

6.4 Summary of Simulation Results

We now summarize the simulation results as follows

- The maximum throughput (and hence the minimum response time) occurs with ML-RC for various quantities of available resources
- The magnitude of relative improvement is proportional to the number of available resources The maximum relative improvement occurs for infinite resources case Further, for a given amount of resource contention, the improvement in performance of ML-RC increases as data contention (i e , multiprogramming level) increases
- Both multilevel concurrency control protocols ML-C and ML-RC out perform SL Thus, a more sophisticated semantics-based multilevel approach to concurrency control is justifiable in complex information systems
- The performance of our new ML-RC protocol is much better than ML-C for various quantities of available resource units Thus, the r-aborts, due to cyclic commit

dependencies, do not negate the advantages of exploiting recoverability based semantics.

7 Conclusions

The conceptual contributions of this paper lie in (1) the application of the semantic notion of *recoverability*, based on the concurrency and recovery properties of operations, to *multilevel* concurrency control, and (2) the development of the notion of *relative conflict*, based on the structure of complex operations, to avoid deadlocks In addition, from a practical viewpoint, to our knowledge, this paper is the first to report on performance evaluation of multilevel concurrency control protocols

This paper showed how simple semantics such as the synchronization properties of operations and the structure of operations can be used to enhance concurrency in *complex information systems* As our simulation studies indicate, using semantics in multilevel concurrency control produces appreciable improvements in performance In general, the magnitude of the improvement is dependent upon both data contention and resource contention For a give amount of resource contention, the higher the data contention the better is the improvement due to use use of a sophisticated semantics-based protocol such as ML-RC

The improved performance justifies the increase in the complexity of the protocols The overhead in terms of maintaining commit dependency information via a list for each recoverable operation can be justified by the overall performance gain It is important to note that, even in the case where only commuting operations are allowed to execute, a wait-for-graph will have to be maintained In our multilevel protocol, a wait-for edge may get transformed in to a commit-dependency edge due to a recoverable operation

References

[1] Agrawal, R , Carey, M J , and Livny, M Concurrency control performance modeling Alternatives and implications *ACM Transactions on Database Systems*, 12(4)·609–654, December 1987

[2] Astrahan, M M , Blasgen, M W , Chamberlin, D D , Eswaran, K P , Gray, J N , Griffiths, P P , King, W F , Lorie, R A , Jones, P R , Mehl, J W , Putzolu, G R , Traiger, I L , Wade, B , and Watson, V System R A Relational approach to Database Management *ACM Transactions on Database Systems*, 1(2) 97–137, June 1976

[3] Badrinath, B R and Ramamritham, K Semantics-based concurrency control Beyond Commutativity In *Fourth IEEE Conference on Data Engineering*, pages 132–140, February 1987

[4] Badrinath, B R. and Ramamritham, K Performance evaluation of semantics-based multilevel concurrency control protocols Technical Report DCS-TR-261, Rutgers University, Department of Computer Science, December 1989

[5] Badrinath, B R and Ramamritham, K Semantics-based concurrency control Beyond Commutativity *Submitted to ACM Transactions on Database Systems*, November 1989

[6] Beeri, C , Bernstein, P A , and Goodman, N A Model for Concurrency in Nested Transaction Systems Technical Report CS-86-1, Hebrew University, Jerusalem, Israel, January 1986

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

[8] Beeri, C , Schek, H , and Weikum, G *Multi-level Transaction Management, Theoretical art or practical need?*, volume 303 of *Lecture Notes in Computer science*, pages 134–154 Springer-Verlag, 1988

[9] Bernstein, P A , Hadzilacos, V , and Goodman, N *Concurrency Control and recovery in database systems* Addison-Wesley, Reading, MA , 1987

[10] Hadzilacos, T and Hadzilacos, V Transaction synchronization in object bases In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, March 1988

[11] Hadzilacos, V Issues of fault tolerance in concurrent computations Technical Report 11-84, Harvard University, Aiken Computation laboratory, Cambridge, MA 02138, June 1984

[12] Herlihy, M P Optimistic concurrency control for abstract data types In *Fifth Annual ACM symposium on Principles of Distributed Computing*, pages 206–217, August 1986

[13] Herlihy, M P Extending multiversion timestamping protocols to exploit type information *IEEE Transactions on Computers*, 35(4) 443–449, April 1987

[14] Kung, H and Robinson, J. On optimistic methods for concurrency control *ACM Transactions on Database Systems*, 6(2) 213–226, June 1981

[15] Martin, B E Modeling concurrent activities with nested objects In *Proceedings of the 7th international conference on distributed computing systems*, pages 432–439, Berlin, Germany, September 1987

[16] Martin, B E Scheduling protocols for nested objects Technical Report CS-094, Department of Computer Science and Engineering, University of California, San Diego, California, 1988

[17] Moss, J E B , Griffeth, N , and Graham, M Abstraction in recovery management In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 72–83, May 1986

[18] Papadimitriou, C H The serializability of concurrent database updates *Journal of the ACM*, 26(4) 631–653, October 1979.

[19] Papadimitriou, C H *The theory of database concurrency control* Computer Science Press, 1986

[20] Peinl, P , Reuter, A , and Sammer, H High contention in a stock trading database A case study In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 260–268, Chicago, Illinois, June 1988

[21] Rozen, S and Shasha, D Using a relational system on wall street The good, the bad, the ugly, and the ideal *Communications of the ACM*, 32(8) 988–994, August 1989

[22] Selinger, P , Gawlick, D , Gray, J , Krause, J , and Schrager, P Panel Discussion Issues in building large database systems In *Proceedings of the ACM SIGMOD international conference on management of data*, page 6, Chicago, Illinois, June 1988.

[23] Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. The Design of XPRS. In *Proceedings of the 14th VLDB conference Los Angeles, California*, pages 318–330, September 1988

[24] Tay, Y , Goodman, N , and Suri, R. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4) 415–462, December 1985

[25] Weikum, G A theoretical foundation of Multilevel concurrency control. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 31–42, March 1986

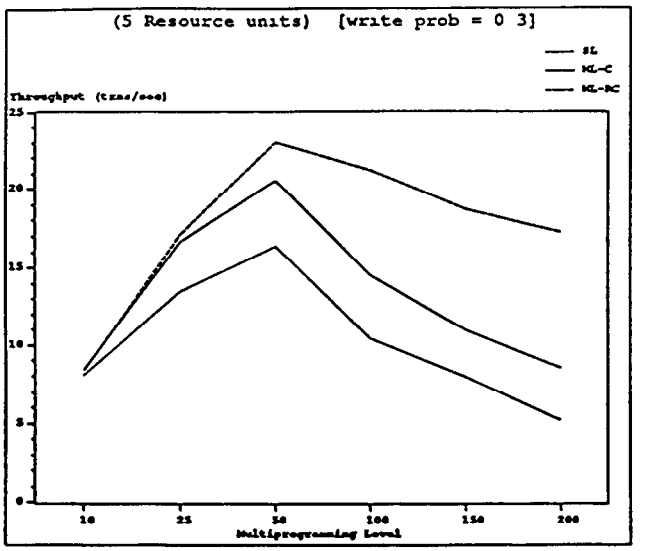


Figure 6 Throughput (5 resource units)

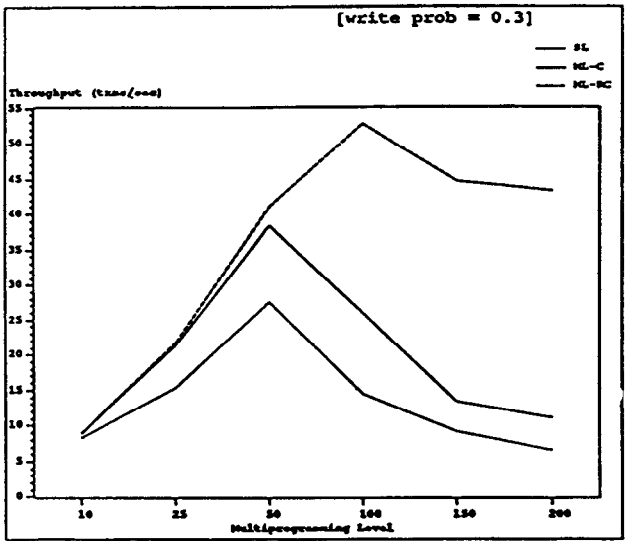


Figure 5 Throughput (infinite resources)

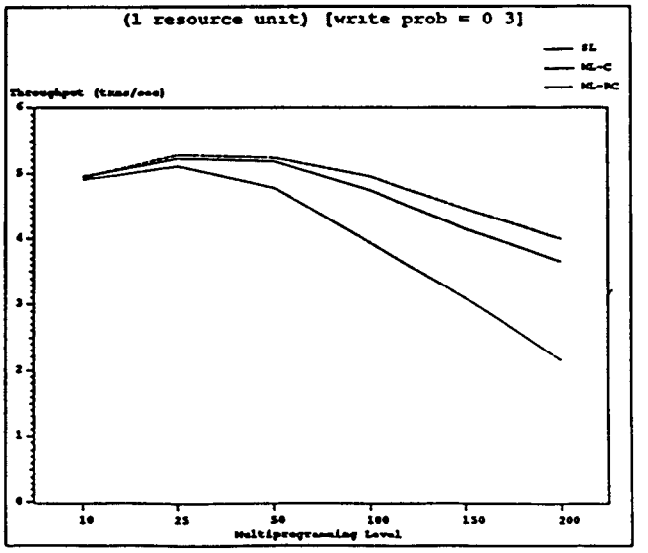


Figure 7 Throughput (1 resource unit)