Semantics-Based Concurrency Control: Beyond Commutativity

B. R. BADRINATH
Rutgers University

and

KRITHI RAMAMRITHAM
University of Massachusetts

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 noncommuting 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, based on the model presented by Agrawal et al. [1], indicate that using recoverability, the turnaround time of transactions can be reduced. Further, our studies show enhancement in concurrency even when *resource constraints* are taken into consideration. The magnitude of enhancement is dependent on the resource contention; the lower the resource contention, the higher the improvement.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*measurements, simulation*; H.2.4 [**Database Management**]: Systems—*transaction processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Concurrency control, semantic information

B. R. Badrinath's work was supported in part by the National Science Foundation under grant IRI-9010174 and K. Ramamritham's work was supported in part by the National Science Foundation under grants DCR-8403097 and DCR-85000332.

Authors' addresses: B. R. Badrinath, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903; K. Ramamritham, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

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* [14, 29]. 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 noncommuting 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 transactions invoking the conflicting operations to commit. Such a feature will be especially useful when long-lived transactions are in progress. In our scheme, noncommuting 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. Furthermore, if either of the transactions aborts, the other can still commit.

Schemes for improving concurrency must be concerned with the problem of transaction rollback, in particular, the possibility of *cascading aborts*. This phenomenon of cascading aborts occurs when aborting one transaction necessitates aborting other transactions that could have read it results. Thus, obliterating the effects of the aborted transaction involves not only undoing the effects of the aborted transactions but also causing the 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 cascading aborts. Cascading 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. The detection of commit dependency cycles is combined with the deadlock detection scheme that uses wait-for graphs. This greatly reduces the overheads involved in providing additional concurrency through the 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 protocol and a 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

Most locking protocols used in semantics-driven concurrency control base conflicts between operations on the notion of commutativity of operations [5, 29, 18]. 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 [31], 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 [7, 24]. These protocols provide more concurrency than protocols using general commutativity [5].

The term recoverability also discussed by Hadzilacos [15] and Bernstein et al. [6]. 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 [23]. 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 optimistic concurrency schemes [19], conflicts are allowed to occur, but at the time of validation, transactions with conflicts are aborted. 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.

Buckley and Silberschatz [9] develop locking protocols using structural information about the data items to permit only noncascading 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.

We introduced [3] the notion of recoverability but without performance evaluation studies. This has since been completed and are reported in the current paper.

Recently, a special purpose concurrency control technique based on failure commutative transactions has been proposed for the XPRS system [26]. Failure commutativity is an adaptation of our notion of recoverability but applied to transactions. Two transactions are said to failure commute if they commute or for any database state S from which they both succeed, there are no cascading aborts.

Herlihy and Weihl [17] define a serial dependency relation between operations. In an history, an operation o_i is serially dependent on o_j, if the execution of o_j before o_i may make the history illegal. Thus, a serial dependency relation captures what happens when operations are inserted into a history. A concurrency control scheme based on serial dependency and using intentions lists is presented by Herlihy [16]. Recoverability, on the other hand, captures what happens when operations are removed from a history. The concurrency control scheme described in this paper uses recoverability and can be used with either intentions lists or undo logs. Further, performance results of using serial dependency relation are not presented by Herlihy and Weihl [17] and Herlihy [16]. However, it would be interesting to compare the performance of various semantics-based notions [3, 16, 30] taking into account the overheads of the particular recovery scheme used.

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. While avoiding cascading aborts, recoverability criterion provides more concurrency than commutativity alone. To ensure serializability, we detect cycles in the transaction commit dependency relation as and when a transaction executes a recoverable operation. The algorithm is based on maintaining commit dependency relationships as part of the wait-for graph that is maintained to detect deadlocks [10].

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 $(\bigcup_i OP_{T_i})$ and $(\bigcup_i <_{T_i}) \subseteq <_E$. $<_E$ 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 $o_i <_E o_j$ [13]. Two operations conflict if they both operate on the same data item and one of them is a write. Here, we 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[1] produced by operation o, and *state*(o, s) which is the state produced after the execution of o. The definition of *state* can be extended to a sequence of operations O. Thus, *state*(O, s) is the state produced after the execution of the operations in O.

Definition 1. Consider two operations o_1 and o_2 such that o_1's execution in state s is immediately followed by the execution of o_2. Operation o_2 is *recoverable relative to operation* o_1, denoted by $(o_2 RR_I o_1)$, iff for all $s \in S$

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

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 commute if the state changes on an object as well as the values returned by the operations are 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* if for all states s, state$(o_2, \textit{state}(o_1, s))$ = state$(o_1, \textit{state}(o_2, s))$, return$(o_1, s)$ = return$(o_1,$ state$(o_2, s))$ and return(o_2, s) = return$(o_2, \textit{state}(o_1, s))$.

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 sequence of operations $O = \{o_1, \ldots, o_{n-1}\}$ and an operation o_n such that $\forall_{1 \le i < n}\ o_i <_E o_{i+1}$. $(o_n RRo_1)$ if return $(o_n,$ state$(O, s))$ = return$(o_n,$ state$(O', s))$ for any subsequence O' of O. Hence $o_n RRo_1 \Rightarrow o_n RR_I o_1$ (Here $O' = o_1$).

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

LEMMA 2. *Given the sequence of operations O as defined above, if* $\forall l$, $1 \leq l < n$, $(o_n RR_I o_l)$ *then* $(o_n RRo_1)$.

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 $O = \{o_3, o_2, o_1\}$. Given that $(o_3 RR_I o_2)$ and since o_2 is executed immediately before o_3, 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_3; Since $(o_n RR_I o_1)$, $(o_n RRo_1)$.

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

Induction step: Let $|\mathrm{F}| = k$ and $O = \{o_n, o_{k+1}, \ldots, o_2, o_1\}$. Now $(o_n RR_I o_{k+1})$ and $(o_n RR_I o_k) \Rightarrow (o_n \mathrm{RR} o_k)$ by using a reasoning similar to the base case. From Definition 3 we have $o_n \mathrm{RR} o_k \Rightarrow o_n \mathrm{RR}_I o_k$, and by induction hypothesis $\forall l\ 1 \leq l \leq k\ o_n RR_I o_l \Rightarrow o_n RRo_1$. □

COROLLARY. $\forall l, 1 \leq l < n\ o_n RR_I o_l \Rightarrow \forall l\ 1 \leq no_n RRo_l$.

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 *Page: A Read/Write Object.* We will first consider an object such as page on which *read* and *write* operations are defined.

Table I. Commutativity for Page

Operation Requested	Operation Executed	
	Read	Write
Read	Yes	No
Write	No	Yes-SP

Table II. Recoverability for Page

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

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 I, (read, read) is commutative and in Table II, (write, read) is recoverable.

The traditional notion of conflict on these objects with read and write operations has been that two operations conflict if one of them is write; as indicated in Table I. 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.

3.2.2 *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 a *push* operation is recoverable relative to another *push*. 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 Tables III and IV. 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.

3.2.3 *Set.* A set object provides three operations: *insert*, *delete*, and *member*. *Insert* adds a specified item to the set object. The parameter to *Delete*

Table III. Commutativity for Stack

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

Table IV. Recoverability for Stack

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

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 (Table V). However, insert is recoverable relative to member, as indicated by the Yes entry (Table VI).

3.2.4 *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 (Table VII). 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 (Table VIII).

Our definitions of commutativity and recoverability were state independent. Clearly, state dependent commutativity or recoverability can be used to extract further concurrency. However, as the following example shows, it will

Table V. 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 VI. 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

Table VII. Commutativity for Table

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

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 we have restricted ourselves to *state-independent*, but *parameter-dependent* notions of commutativity and recoverability.

We find the notation used by Weihl [29] 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

Table VIII. Recoverability for Table

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

into the set and returns "ok" when the operation is completed. Thus, if the integer set object *set 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 *set X* to check for membership of 3 in *set 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 an interleaved operation sequence invoked by transactions T_1 and T_2 on the set object *set X*.

$$\begin{aligned} &X:\langle insert(3), ok, T_1\rangle \\ &X:\langle member(3), yes, T_2\rangle \\ &X:\langle insert(7), ok, T_1\rangle \\ &X:\langle delete(3), ok, T_1\rangle \end{aligned} \tag{1}$$

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:

$$\begin{aligned} &X:\langle member(3), no, T_2\rangle \\ &X:\langle insert(3), ok, T_1\rangle \\ &Y:\langle insert(4), ok, T_2\rangle \\ &Y:\langle delete(5), ok, T_2\rangle \\ &\langle commit, T_1\rangle \\ &\langle abort, T_2\rangle \end{aligned} \tag{2}$$

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

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

$$\begin{aligned} &S:\langle push(4), T_1, ok\rangle \\ &X:\langle member(3), T_1, no\rangle \\ &S:\langle push(2), T_2, ok\rangle \\ &X:\langle insert(3), T_2, ok\rangle \\ &\langle commit, T_1\rangle \\ &\langle commit, T_2\rangle \end{aligned} \tag{3}$$

In concurrency protocols which consider operations to conflict if they are not commutative, the operations invoked by T_2 in the above sequence will

have to wait until T_1 commits. However, in our scheme, since the operations invoked by T_2 are recoverable they can be executed without waiting for T_1 to commit, while avoiding cascading abort of T_2 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 discuss a concurrency control technique and a commit protocol where a transaction can *complete* execution and considered complete from a user's perspective even though the transactions on which it depends have not terminated.

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 the user transaction sees as a system interface. 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.

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 [19]. We have combined the process of checking the dependency-graph for acyclicity with the process of checking deadlocks by maintaining commit-dependency relationships in the wait-for graphs itself.

In Section 4.1 we formally define the correctness requirements of concurrency control. In Section 4.2 we describe the concurrency control technique and discuss how commit-dependency cycles are detected. Section 4.3 explains how to commit transactions that may have commit-dependencies, and Section 4.3 discusses recovery techniques.

4.1 Correctness Requirements

Definition 4. An operation o_i invoked by transaction T_i is *sound* in a log E if for any *extension* $E' = E \| A_j$ for any $j \neq i$ ($\|$ indicates that when A_j, the abort of transaction T_j, is appended to the log, the operations belonging to T_j are undone and deleted from log E), 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 all* $o_i <_E o_j$, *if* o_i *is uncommitted then either* 1) (o_i, o_j) *commute or* 2) $(o_j RR o_i)$. *Then all operations are sound in E.*

PROOF. The proof follows from the definitions of commutativity and recoverability. □

LEMMA 3. *A log* E *is free from cascading aborts if it contains only sound operations.*

PROOF. The proof follows from the definitions of soundness and recoverability. □

Object managers use 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 nonrecoverable operations can be handled using known techniques of deadlock detection and resolution [8, 25].

For each object O_k in the database, the object manager for O_k maintains a *commit dependency graph* G_k. In 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. $G_k = (N, M)$ is a commit dependency graph. N is the set of nodes corresponding to active transactions (that have begun execution but not terminated) and M is a set of edges. An edge e belonging to M 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 $(o_j \mathrm{RR}\, o_i)$. Let $G = \bigcup_k G_k$ (for each object O_k in the database).

Definition 6. A serialization graph $SG = (N, M')$, where N is the set of nodes corresponding to active transactions (that have begun execution but not terminated) 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) $o_i <_E o_j$, and (2) o_i and o_j are not recoverable.

LEMMA 4. *An execution log* E *is serializable if the dependency graph* $DG = G \cup SG$ *is acyclic.*

Definition 7. An execution log E is correct if it is serializable and is free from cascading aborts.

If one were to used commutativity as a basis for defining conflicts, then serializability can be achieved by ensuring that any concurrent execution of

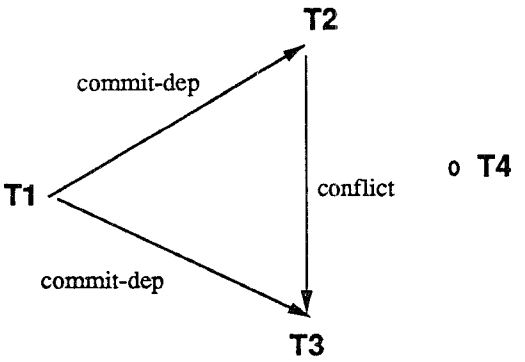

Fig. 1. A dependency graph.

operations results in an acyclic serialization graph. Known methods to do this include two-phase locking and timestamping schemes. Since we are using recoverability as a basis for conflicts, Lemma 4, states that serializability can be ensured by forcing the commit dependency relationship resulting from the recoverability of operations in the log E also to be acyclic. From Lemma 3, cascading aborts can be avoided by ensuring that all operations in the log are *sound.*

Figure 1 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 not 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 by object managers as requests are made to them, i.e., as transactions invokes new operations. The algorithm is given in Figure 2.

4.2 Concurrency Control Strategy

A transaction T_i is a sequence of operations $\{o_1, o_2, \ldots, o_n\}$. An operation o_i on a given object conflicts if it is not recoverable with other operations executed on this object by still-active transactions, i.e., those transactions that have not committed or aborted. However, operation o_i does not conflict if it is recoverable or commutative with other operations. Thus, the notion of conflict is based on recoverability. When an operation conflicts, the transaction requesting that operation is blocked, and deadlock detection needs to be initiated. Further, if the operation does not conflict (i.e., is recoverable) then the operation can be executed provided there are no cyclic commit-dependency relationships. Thus, in either case we need to check for cycles.

The process of checking for deadlocks and commit-dependency cycles can be achieved using a single graph. This graph known as the dependency graph contains both wait-for edges and commit-dependency edges. When a transaction issues a request to execute an operation, the object manager, by using the compatibility matrix, determines whether the operation request conflicts or not. If the request conflicts, the transaction is made to wait. The

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

   add *wait-for* edges from node T_i to other transaction nodes which have invoked operations with whcih o_i is not recoverable.

   Check for a cycle in the dependency graph. If a cycle is found T_i is aborted else T_i is made to wait.

2. If for all operations o_j, o_i and o_j commute then operation o_i is executed.
3. If for all operations o_j, either o_i and o_j commute or o_i is recoverable relative to o_j then

   add *commit-dependency* edges from node T_i to other transactions which have invoked operations with which o_i is recoverable

   Check for a cycle in the dependency graph. If a cycle is found T_i is aborted else operation o_i is executed.

Fig. 2. Algorithm to execute operations.

corresponding wait-for edges are introduced and a cycle detection algorithm is initiated. If a cycle is found, the transaction making the request is aborted. On the other hand, if request o_i is recoverable with respect to one or more operations of still-active transactions, then commit-dependency edges are introduced between the requesting transaction and the transactions with whose operations o_i is recoverable. Again a check for cycle is initiated and if a cycle is found, T_i is aborted. Note that a cycle in the dependency graph may involve both commit-dependency and wait-for edges.

When a transaction terminate successfully or unsuccessfully, the node that corresponds to the terminating transaction together with the edges associated with the node is removed from the dependency graph. The algorithm for executing operations is shown in Figure 2.

4.3 Committing Pseudo-Committed Transactions

Recoverable operations force commit dependencies; commit dependencies imply a commit order. If T_1 has a commit dependency on T_2, T_1 has to commit after T_2. However the observable semantics of T_1 are not affected by the outcome of the still-active transaction T_2. Thus, from a user's perspective of T_1 can be considered to have completed, but from a system's perspective the actual commit of T_1, which makes the changes of T_1 durable, can occur only after T_2 terminates. 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 [20], wherein a transaction that has conditionally committed may be subsequently 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, i.e., commit or abort, thus respecting the commit dependency relationship. A similar notion called *pre-commit* presented by DeWitt et al. [12].

After a transaction pseudo-commits, the operations and the transaction continue to remain in the log and so does the node and the associated edges in dependency graph. 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. This is done as follows: When a transaction T_i terminates (successfully or unsuccessfully), the node corresponding to T_i and all edges to and from this node in the dependency graph are removed. This removal of edges may result in nodes having an out-degree of zero. If such nodes exist and correspond to pseudo committed transactions, then such transactions are committed. Committing a transaction results in all the operations executed by the transaction to be removed from the log and the commit of the transaction is recorded in the log.

4.4 Effecting Recovery

Before we conclude this section we look at the problem of effecting aborts. When a transaction aborts, it is necessary to undo (back out) a transaction. Undo of a transaction involves undo of all operations executed by a transaction. Recovery from transaction aborts can be achieved using two different approaches: *intentions* lists or *undo* logs [22, 21, 30]. Further, the type of undo is dependent upon the type of operation. For example (write, read) is recoverable but there is no undo for a read operation. However (write, write) is recoverable but a write operation needs undo. Similarly (push, top) is recoverable but there is no undo for a top. However, (push, push) is recoverable, and the undo for a push involves removing the pushed element from the stack. If recovery is based on intentions lists, the undo for a push involves dropping the push operation from the transaction's intentions list. Nevertheless, to avoid digression, we do not investigate these strategies in this work; the details on how recovery affects commutativity-based concurrency control schemes are given by Weihl [30]. These schemes can be adapted to effect recovery in our concurrency control scheme.

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. We are not only interested in the effect of data contention but also the effect of resource (for example, CPU or I/O) contention on the performance of semantics-based concurrency control protocols. Hence, we have conducted performance studies under both infinite resources and limited resources conditions. In the case of infinite resources, transactions never have to wait for CPU or I/O service. This case represent just data contention. In the case of finite resources, the model includes a variable

number of CPU or I/O devices, and transactions have to wait until the required resources are available. In this case, the performance results reflect the effect of both data contention and resource contention.

We examine two different data models in this study, the read/write model and the abstract data type model; in the former, the operations are restricted to be reads and writes and in the latter, the operations can be arbitrary. We use a simulator based on a closed queuing model. This is similar to the models that have been used in previous studies [1, 28]. Most of the parameters are the same as those chosen by Agrawal et al. [1].

5.1 The Simulation Model

There are two important aspects to our performance model: the closed queuing model, and the representation of properties of objects in the database via the compatibility table. The model shown in Figure 3 is a modified version of the one used in [1]. The various model parameters and their meanings are listed in Table IX.

There are a fixed number of terminals from which transactions originate. The maximum number of active transactions at any given time in the system is the multiprogramming level, the *mpl.level*. A pseudo-committed transaction is considered active, i.e., is included in determining conflicts until it *commits*. 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. If an operation request is denied, the transaction is blocked and the request entered in the block queue for that object. Every time a transaction is blocked, deadlock detection is initiated. A transaction is aborted if a deadlock is discovered or else the transaction is made to wait until the conflict is resolved. If an operation request is recoverable, cycle detection is initiated. A transaction is aborted if a cycle is detected in the dependency graph made up of commit-dependency and wait-for edges or else the transaction can proceed to request resources to complete the operation. In such a case, the request is entered in the active queue for that object. An aborted transaction is restarted immediately, i.e., placed at the end of ready queue. A restarted transaction behaves, with respect to operation invocations, like the original transaction that was aborted, i.e. reexecuted with the *same* set of operations. Another alternative that we have not considered is the use of *fake* restarts where each restarted transaction behaves as an independent transaction.

The parameter *step.time* is the execution time of each operation. Under the assumption of *infinite* resources, this represents a constant service time for each operation. In the case where finite resources are present, each step requires a CPU (disk access) for an interval of length *cpu.time*(*io.time*). The total time for which these resources are used is equal to *step.time*. We consider a CPU and two disks to constitute one resource unit. The number of

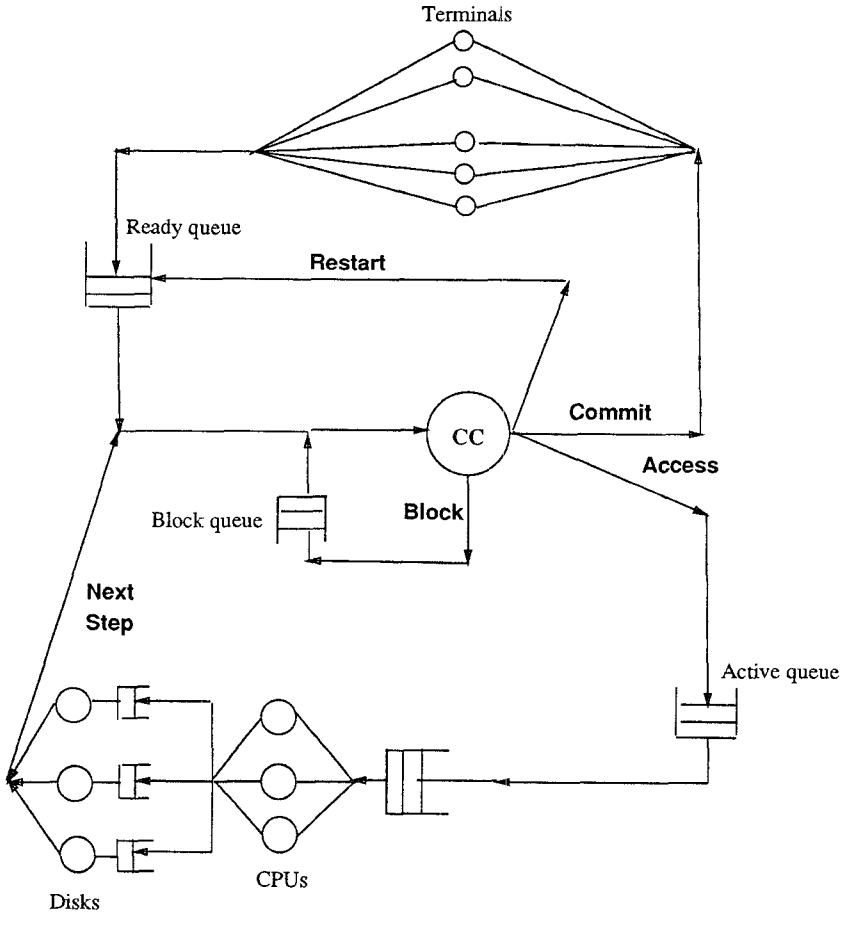

Fig. 3. Simulation model.

resource units is a model parameter *resource.unit*. When a transaction needs a CPU, it is assigned a free CPU from a pool of CPUs: 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 randomly and waits in the queue of the selected disk until it can be served [1].

After a transaction completes (this includes pseudo committed transactions), 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*.

5.2 Experimental Information

The concurrency control strategy we adopt is based on *blocking*. Each transaction T_i makes a sequence of k requests $\{o_1, o_2, \ldots, o_k\}$, where k is

Table IX. Simulation Parameters

Parameter	Meaning
Database size	Number of objects in the database
Num.of.terminals	Number of terminals
Transaction length	Mean transaction length
Max.length	Maximum number of operations in a transaction
Min.length	Minimum number of operations in a transaction
Mpl.level	Level of Multiprogramming
Step.time	Execution time of each operation
CPU.time	CPU time for accessing an object
IO.time	I/O time for accessing an object
Resource units	Number of resource units
Ext.think.time	Mean time between transactions
Write.probability	Probability of a Write operation

the transaction length. A transaction T_i can execute a request on an object if the requested operation does not conflict with requests executed by other active transactions. A request is denied if it conflicts, and the requesting transaction is *blocked*. The decision to honor or deny a request can be made easily by use of the compatibility table maintained for each object. A blocked transaction is *retried* every time any transaction that issued a conflicting operation on that object completes.

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. Hence, a deadlock detection algorithm needs to be invoked. If a deadlock exists, the transaction is aborted. Further, when a transaction executes a recoverable operation, commit-dependencies are introduced. If a cycle exists in the dependency graph then the transaction is aborted.

The algorithm in Figure 2 checks for conflicts with operations of active transactions and executes operations if they do not conflict. This implies that incoming requests have priority over requests in the blocked queue. This type of scheduling favors operations that are nonconflicting over conflicting operations that are blocked. This scheme may result in the starvation of transactions that issue nonrecoverable requests. In order to obtain an unbiased estimate of the performance improvement, we have chosen to use *fair* scheduling: here each incoming request is blocked if it conflicts with a blocked request, even if it is does not conflict with the current set of active requests. Real database systems do this to prevent starvation of writers by readers. Hence, we have chosen to use a fair scheduling scheme and thus, the performance results reported here show no bias towards transactions that execute recoverable operations.

The cost of performing concurrency control is not measured in this study. The cost of concurrency control is the same in cases of commutativity and recoverability except for the additional commit-dependency edges that need to be introduced for each recoverable operation that is executed. Even though we do not measure the cost of detecting cycles, we do measure the number of invocations of the cycle detection algorithm per transaction commit.

As we have mentioned earlier, the operations executed by a pseudo committing transaction are still considered in determining conflicts for other transactions and hence are considered *active* until the transaction commits. However, the results of the execution of a pseudo committed transaction are durable. Thus, a user can invoke a new transaction after the current transaction has pseudo committed. We model this effect by allowing the terminal that issued the pseudo committed transaction to initiate, after its thinking time, a new transaction.

5.3 Performance Settings

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. Since transactions compete for the shared objects, for a given transaction length, as level of multiprogramming increases, i.e., the number of active transactions in the system increases, contentions will increase and hence transactions turnaround time will increase. The transaction load is adjusted by changing the level of multiprogramming. For a given level of multiprogramming, different transaction lengths indicate different workloads. Instead if running the experiments with fixed transaction sizes, we use a transaction mix consisting of transactions whose length is a uniformly distributed random variable between 4 and 12 operations. 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. Each step of the transaction takes 0.015 seconds of CPU time and 0.035 seconds of disk access time. Thus, in the case of infinite resources each step takes 0.05 seconds.

The nominal values of the parameters are listed in Table X. The values of the model parameters have been chosen similar to those in previous performance studies of locking protocols [28, 27, 1] and commutativity-based protocols [11].

5.4 Performance Metrics

The two main performance metrics used in our evaluation are the *throughput* and the *response time* (turnaround time). The throughput is measured as the number of transactions that *complete* per second. This includes committed and pseudo-committed transactions. The response time in seconds is measured as the difference between when a terminal submits a transaction and that transaction completes. The time includes any time spent in the ready

Table X. Parameters and their Nominal Values

Simulation parameters	
Parameter	Value
Database size	1000 objects
Num.of.terminals	200
Transaction length	8 steps
Min.length	4 steps
Max.length	12 steps
Mpl.level	10, 25, 50, 100, 150, 200
Step.time	0.05 seconds
CPU.time	0.015 seconds
IO.time	0.035 seconds
Resource.units	1, 5, and ∞
Ext.think.time	1 second
Write probability	0.3

queue and *time spent due to restarts*. The average 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. 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, when recoverability is considered, the latter time is the same as the time when a transaction pseudo-commits or commits (if it commits without first pseudo-committing).

Given that recoverability is a weaker conflict predicate than commutativity, we expect reductions in the 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 the number of recoverable operations increases we expect a decrease in average response time for transactions.

Three other metrics used to determine the usefulness of semantics in concurrency control are blocking ratio, restart ratio, and cycle check ratio. *Blocking ratio* is the average number of times a transaction blocks per commit (computed as the ratio of the number of transaction blocks to the number of transaction commits). This should give a fair indication of the conflict level in the system. The *restart ratio* is defined as the number of times a transaction has to be restarted before it completes. The lower the restart ratio, the less the work wasted, and hence the better the system utilization. The last metric useful in evaluating recoverability is the *cycle check ratio*). This is defined as the ratio of the number of invocations of the cycle detection algorithm to the number of transaction commits.

Further, we also measure the average length of an aborted transaction denoted by *abort length*. The longer the length of the transaction at the time of abort, the higher the cost of effecting recovery.

5.5 Simulation Results

In this study each simulation is run until 50000 transactions are completed. We measured various factors, including transaction response time, throughput, blocking ratio, restart ratio, cycle check ratio, and abort length. The graphs in Figures 4 through 18 show the average results of 10 runs. Though the confidence intervals are omitted from our graphs the 90 percent confidence intervals lie within $\pm 2\%$ percentage points of the mean value of the performance metrics shown in the various graphs.

5.5.1 *Read Write Model.* In this experiment, we analyzed the impact of using recoverability on the traditional read/write model. Each operation request of a transaction is either a read or a write. 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 do Tay et al. [28], there is uniform access, that is the probability that a transaction chooses an object to execute an operation is a uniformly distributed random variable between 1 and *database size*. In this study, the database size was chosen to be 1000 objects. This database size was chosen to yield good conflict rates so that interesting evaluation of recoverability-based concurrency control scheme can be done.

First, we determine various performance characteristics when conflicts are defined based only on commutativity. The fundamental notion of *conflict*, as applied to the read/write model, is that two operations conflict if one of them is a write. As seen in the compatibility table, Table I, there are three pairs of conflicting operations. Second, to determine the relative performance, we include conflicts defined based on recoverability and commutativity. Thus, with recoverability there is only one pair of conflicting operations in (read, write) as (write, read) and (write, write) are recoverable. These experiments are conducted for different levels of multiprogramming. This study is also aimed at investigating, in the context of the traditional read-write model, the degree to which the positive effects of the decreased conflicts are able to counter the negative effects of aborts due to cycles in the dependency graph. Further, we study the effect of resource contention on the performance of our semantics-based concurrency control scheme by repeating the experiments for various values of available resource units.

Infinite resources. In this part of the simulation, we assume infinite resources. Figure 4 shows throughput as a function of the level of multiprogramming. The throughput under both commutativity and recoverability increases with multiprogramming level and after a certain level drops as the multiprogramming level increases. This is due to thrashing resulting from very high data contention. The peak throughput was obtained with *mpl.level* $= 50$ and at this multiprogramming level, the throughput with recoverability is approximately 67 percent higher than when commutativity alone is used. At high values of multiprogramming, the relative improvement in throughput under recoverability increases with multiprogramming level. Thus, the higher the data contention the better the performance improvement.

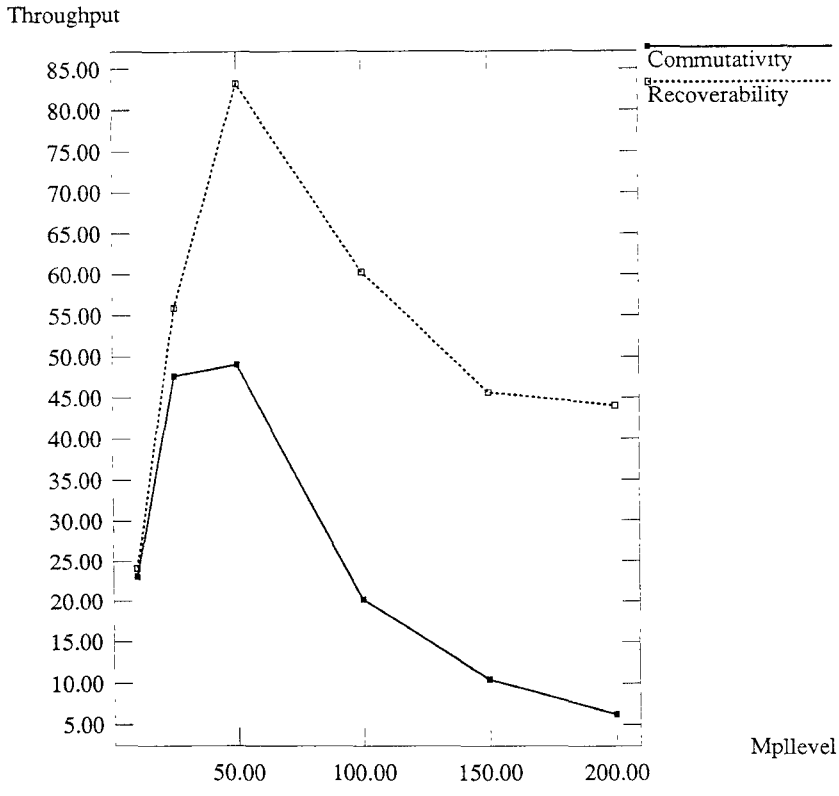

Fig. 4. Throughput (∞ resources).

The effect of using recoverability on response time is shown in Figure 5. The response time initially decreases with multiprogramming level, and at values of multiprogramming greater than *mpl.level* = 50, the average response time increases with multiprogramming level. Note that with just commutativity, the response time is higher than with recoverability at *mpl.level* = 50. As the level of multiprogramming increases, so does the data contention. Hence, more transactions will be restarted which leads to a larger response time and a lower throughput at higher values of multiprogramming.

Figure 6 shows the restart ratio (RR) and blocking ratio (BR) respectively. The blocking ratio is smaller with recoverability than without it for all levels of multiprogramming. The restart ratios with commutativity and with recoverability are approximately the same for lower levels of multiprogramming, and as thrashing increases at higher values of multiprogramming, the restart ratio with recoverability is lower than the restart ratio with just commutativity. Thus, the improvement in concurrency due to reduction in blocking with recoverability does not result in increased restarts due to cyclic dependencies. For all levels of multiprogramming, the restart ratio is smaller than the

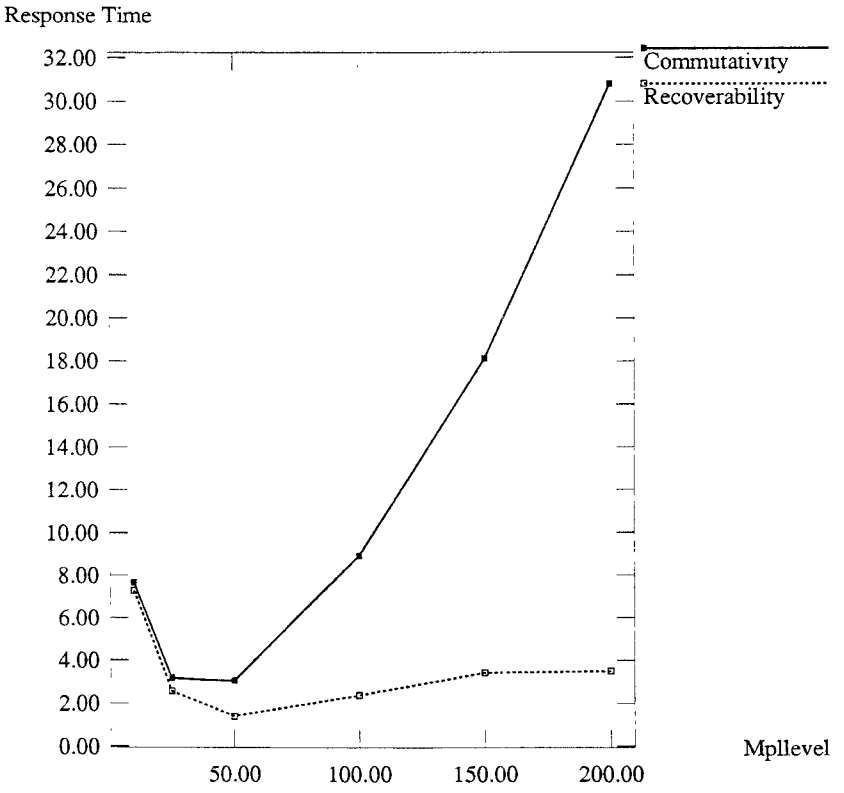

Fig. 5. Response time (∞ resources).

blocking ratio for both commutativity and recoverability. This confirms earlier results of Agrawal et al. [1] that for blocking based concurrency control strategies the number of times that a transaction is blocked is higher than the number of times a transaction is restarted.

Figure 7 shows cycle check ratio (CCR) and abort length (AL). The cycle check ratio with recoverability is higher than with commutativity alone. This is to be expected as we need to check for cycle not only when a transaction blocks but also when a transaction executes a recoverable operation. For instance, when peak throughput occurs, i.e., at *mpl.level* = 200, the cycle check ratio is about 22 percent higher with recoverability. At *mpl.level* = 200, with commutativity, thrashing is very high. Thus, the cycle check ratio with commutativity is higher than the cycle check ratio with recoverability at this multiprogramming level. Once the system begins to thrash, the abort length decreases as a function of *mpl.level*. The decrease in abort length with *mpl.level* is due to the fact that as *mpl.level* increases, so does the data contention, and hence transaction gets aborted earlier.

We also studied the effects of not using fair scheduling, i.e., an incoming request that does not conflict with still-active operations is allowed to execute

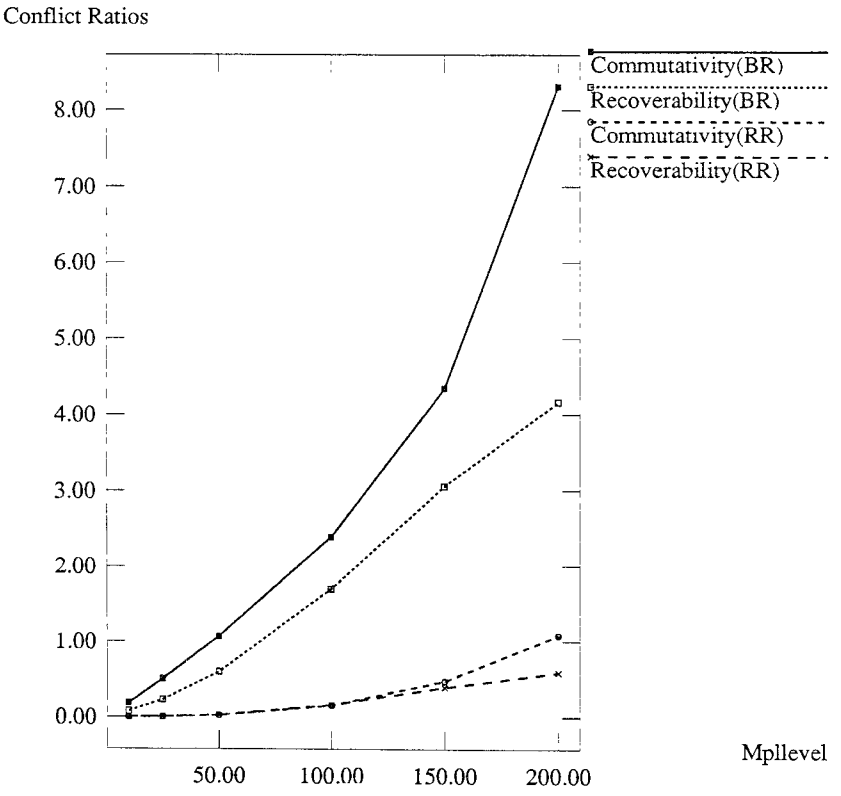

Fig. 6 Conflict ratios (∞ resources).

immediately. Figure 8 shows the throughput when fair scheduling is not used. The peak throughput for both commutativity and recoverability is higher than the corresponding throughput with fair scheduling shown in Figure 4. This is to be expected since in this scheduling scheme, operations that are nonconflicting are favored over operations that are blocked, and both the commutativity table and the recoverability table have one such operation that is given preferential treatment; *read* operation in case of commutativity and *write* operation in case of recoverability. Figure 9 shows the blocking ratio and restart ratio without fair scheduling. The values of these two metrics are lower in comparison with the corresponding values under fair scheduling shown as in Figure 6. This is because under fair scheduling, operations are blocked more often and hence more restarts occur.

Finite resources. In this part of the simulation, we conducted experiments for two cases: First when the database consists of 5 resource units and second with 1 resource unit. With 5 resource units we simulate a multiprocessor database and the 1 resource unit case models high resource contention. For the case of 5 resource units, the throughput first increases with multipro-

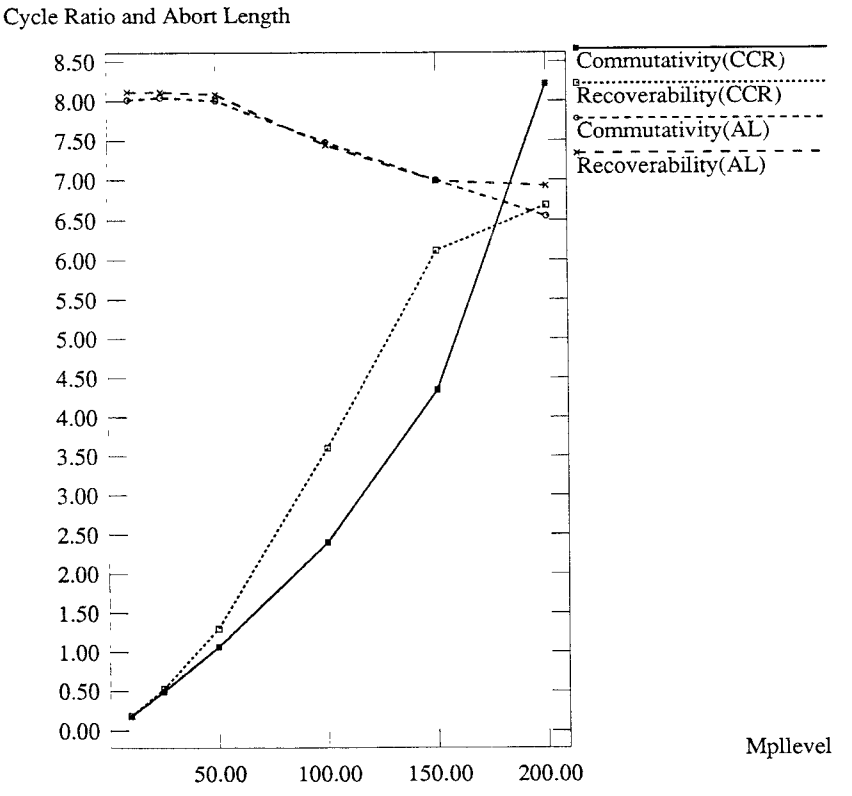

Fig. 7. Cycle check ratio and abort length (∞ resources).

gramming level and then decreases due to thrashing as shown in Figure 10. Further, because of resource contention, the peak throughput is less than the maximum throughput for the case of infinite resources. The peak throughput with recoverability occurs at *mpl.level* = 50 and with commutativity occurs at a *mpl.level* = 25. Thus, with commutativity, thrashing sets in earlier. At *mpl.level* = 50, the throughput with recoverability is approximately 15 percent higher than with commutativity alone.

Figure 11 shows the throughput with 1 resource unit, and the throughput is very low compared to the case of infinite resources. This is to be expected as transactions have to wait for a longer period of time because there is only one resource unit. Further, thrashing starts at *mpl.level* = 25, and as multiprogramming level is increased, the percentage improvement in throughput is larger with recoverability as shown in Figure 11. Thus at higher values of data contention, using recoverability not only improves concurrency but also the improvement is better when very limited resources are present. Observe that the peak throughput is higher, though slightly, with recoverability in the case of 1 resource unit.

The restart ratio and blocking ratio for 5 resource units are shown in Figure 12. Note that the blocking ratio is smaller with recoverability than

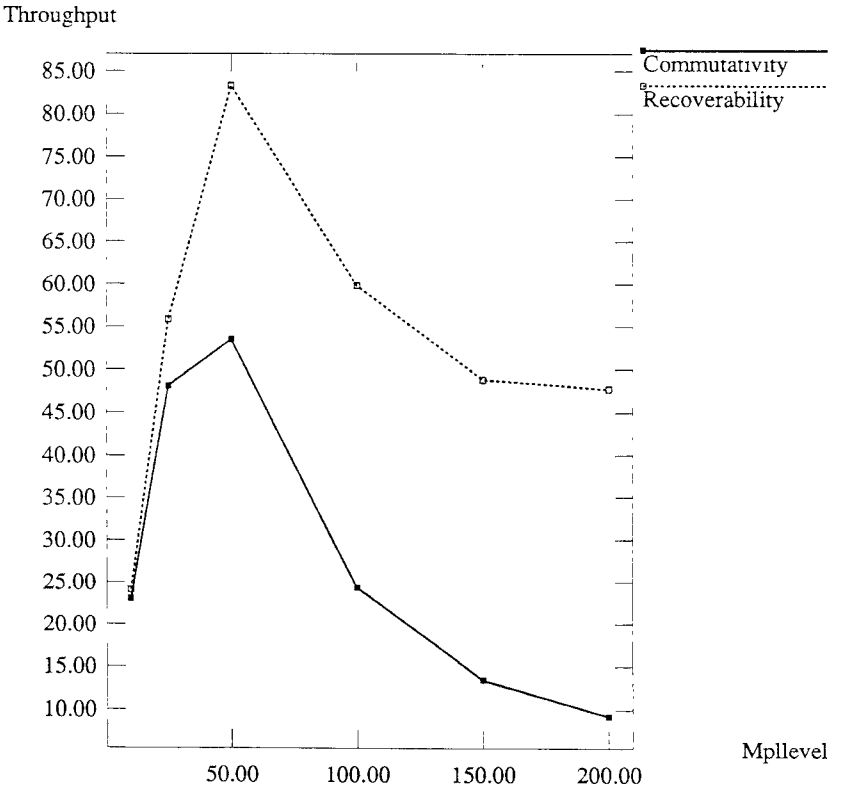

Fig. 8. Throughput (∞ resources).

with only commutativity. Further, the difference becomes larger as the level of multiprogramming is increased. The restart ratio is almost the same for both except at *mpl.level* = 200, where use of commutativity results in higher restart ratio.

The cycle check ratio and abort length for 5 resource units is shown in Figure 13. The cycle check ratio is higher with recoverability, and once thrashing begins to occur, the abort length decreases as *mpl.level* increases. These were also observed in the case of infinite resources.

5.5.2 *Abstract Data Type Model.* In this experiment, the properties of the operations are defined by compatibility tables, and the operations on the objects can be arbitrary. Since the operations are arbitrary, we do not model the cost of recovery, for reasons explained in Section 4.4.

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 indicates whether 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 single *compatibility table*; each entry in this table will be one of *commutative*, *recoverable*, or *non-recoverable*.

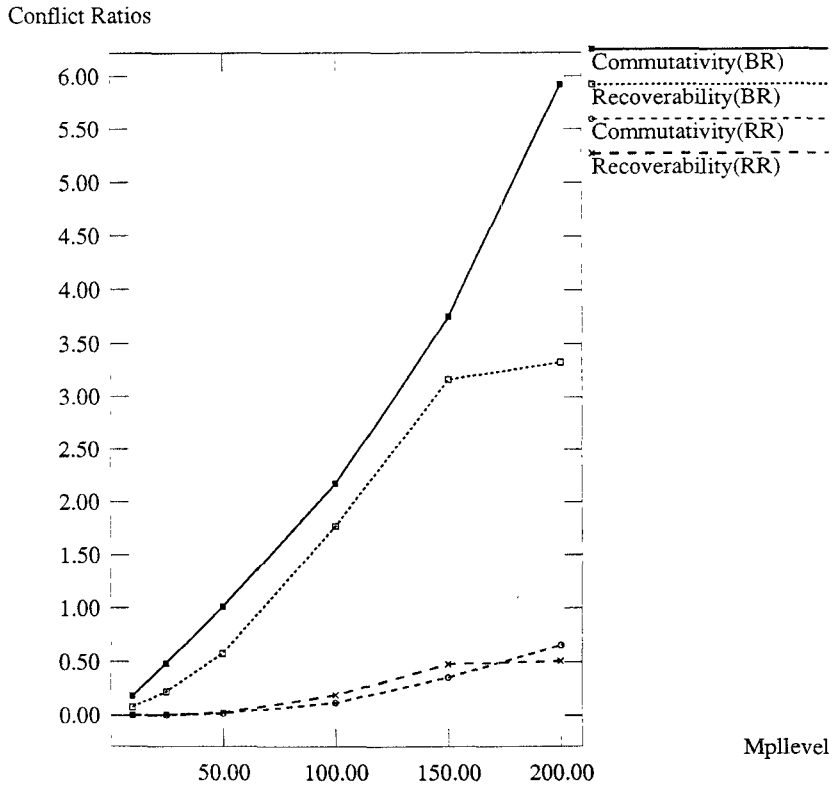

Fig. 9. Conflict ratios (∞ resources).

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 *nonrecoverable* 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 14 through 18, each graph is for a fixed value of P_c (indicated in the graphs as $P_c = 2$, 4, etc.) and varying values of P_r (indicated and $P_r = 0$, 4, and 8). The horizontal axis depicts different values of multiprogramming (mpl.level). At the beginning of a simulation run, given the values of P_c and P_r for an object, $P_c/2$ nondiagonal entries in its compatibility table are *randomly* chosen and set to 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 *non-recoverable*.

In this study, each object has four operations defined on it. Unlike the read/write model where the probability of read was chosen to be 0.7, in the experiment, for any given object, all of the defined operations can be invoked

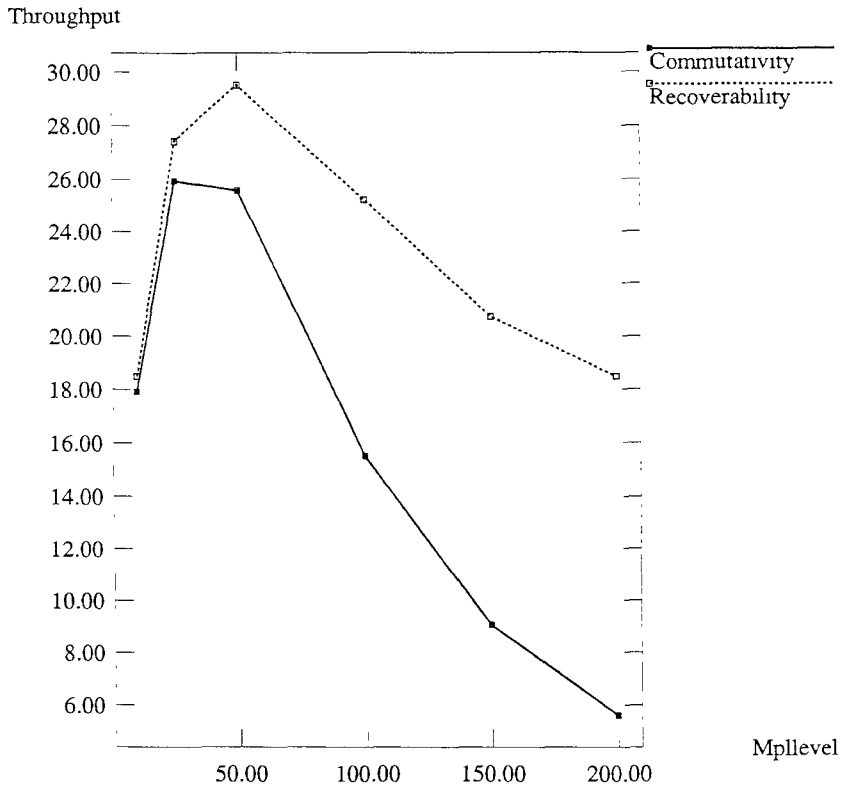

Fig. 10. Throughput (5 resources).

with equal probability. Thus, each operation is selected using a random variable distributed uniformly between 1 and 4. Further, at each step, as in the case of read/write model, the object on which the selected operation it to be executed is selected randomly and independently, being chosen from all the objects in the database (i.e., uniformly distributed between 1 and *database size*).

We examine the performance characteristics for a variety of multiprogramming levels and for different values of recoverability including the case where only commutativity is considered (i.e., where $P_r = 0$). Further, we examine the performance characteristics under varying assumptions about the number of resource units that are available. Results of the experiments conducted for $P_c = 4$ and $P_c = 2$ are presented here.

Infinite resources. In this part of the simulation, we assume infinite resources. Figure 14, depicts increased throughput due to recoverability when $P_c = 4$. The throughput increases as a function of multiprogramming. However, beyond *mpl.level* = 25, for $P_r = 0$ and $P_r = 4$, the throughput falls with multiprogramming level. This phenomenon, as in the case of the

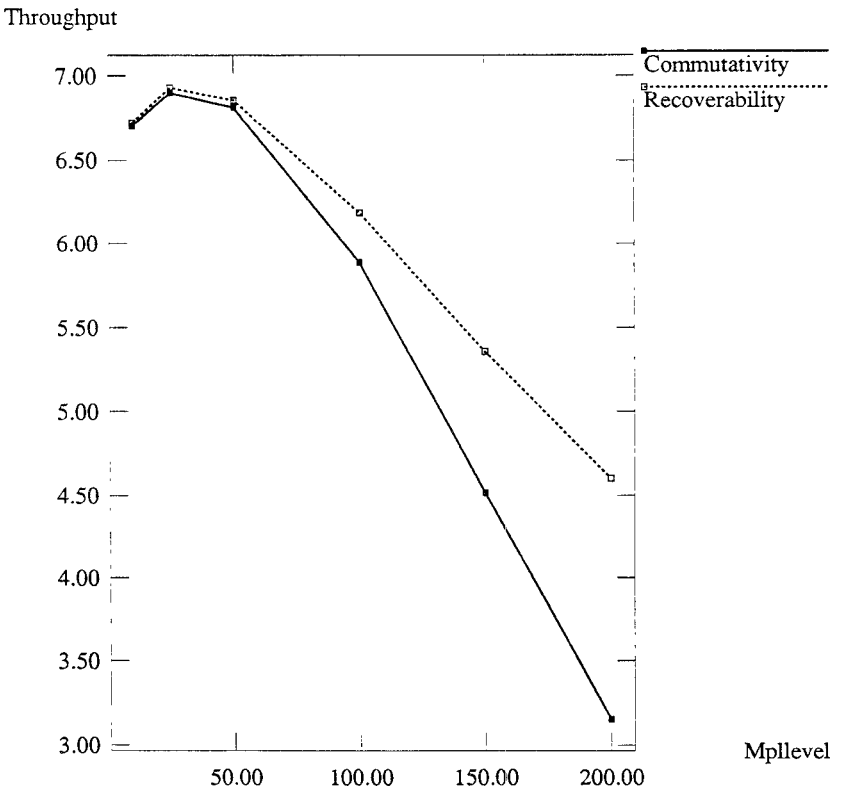

Fig. 11. Throughput (1 resource).

read/write model, is due to thrashing that is induced by high data contention. At *mpl.level* = 25, the throughput for $P_r = 4$ is approximately 15 percent higher than the throughput for $P_r = 0$. Further, for $P_r = 8$, thrashing starts only at *mpl.level* = 50. This reflects an overall reduction in data contention, as a high proportion of the operations is considered nonconflicting. Thus, for higher values of recoverability, not only does the throughput increase but also thrashing sets in at a higher value of *mpl.level*. At *mpl.level* = 50, the throughput for $P_r = 8$ is more than double the throughput for $P_r = 0$.

For different values of recoverability, Figure 15 shows the throughput for $P_c = 2$. The graph for $P_c = 2$, $P_r = 8$ approximates the profile of an object such as stack, and as can be observed from the graph, the improvement in peak throughput for $P_r = 8$ is approximately double the throughput for $P_r = 0$.

Increased multiprogramming level implies increased blocking due to higher data contention. Thus, the blocking ratio (BR) increases with the level of multiprogramming. However, as recoverability increases not only does the blocking ratio decrease but also the rate of increase is slowed down. This can

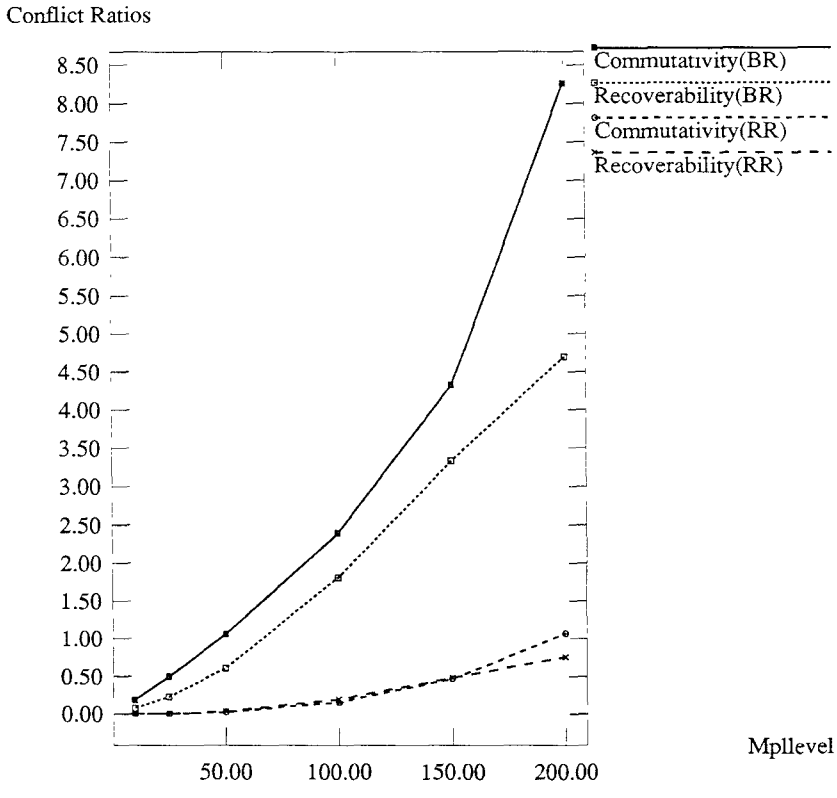

Fig. 12. Conflict ratios (5 resources).

be seen in the decreasing slope of the curves for increasing values of recoverability shown in Figure 16. For various values of recoverability, restart ratios (RR) are approximately the same. However, at *mpl.level* = 200, higher values of recoverability have lower restart ratios. This is because of the reduction in thrashing with recoverability at this multiprogramming level.

Finite resources. In this part of the simulation, we conducted experiments with 5 resource units and 1 resource unit. The throughput results for $P_c = 4$ with 5 resource units are shown in Figure 17. Due to resource contention, the maximum throughput obtained with 5 resource units is smaller than the maximum throughput with infinite resources. Further, for $P_r = 0$ and $P_r = 4$, as multiprogramming level is increased beyond *mpl.level* = 25, throughput begins to drop as a result of thrashing. At *mpl.level* = 25, the throughput at $P_r = 4$ is approximately 6 percent higher than the throughput at $P_r = 0$. However, for $P_r = 8$, thrashing sets in only at *mpl.level* = 50. At *mpl.level* = 50, the increase in throughput for $P_r = 8$ over the throughput for $P_r = 0$ is approximately 35 percent.

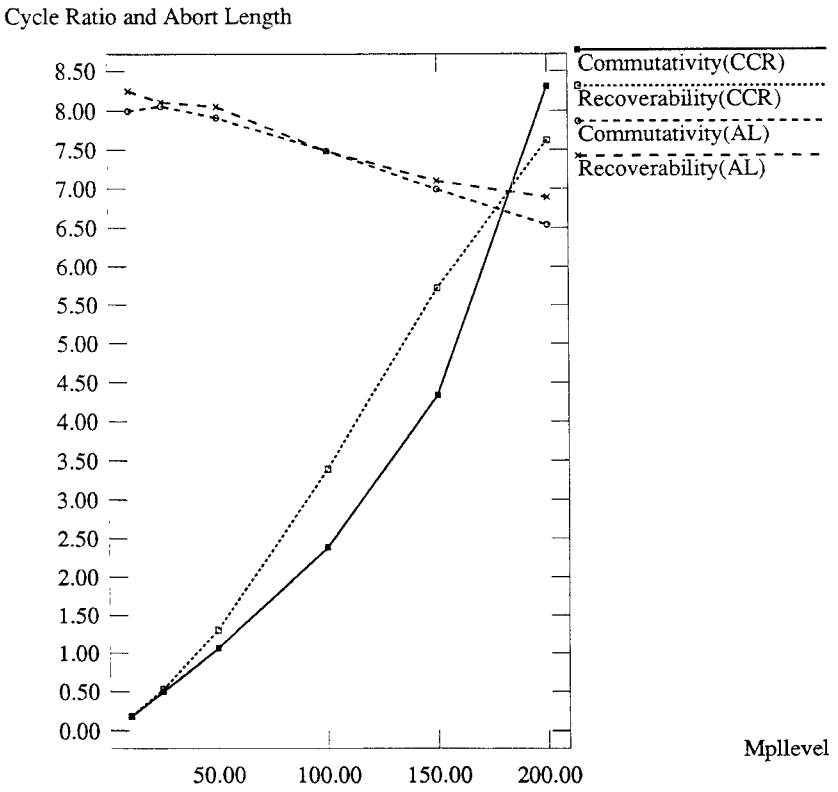

Fig. 13. Cycle check ratio and abort length (5 resources).

Figure 18 shows the throughput with very limited resources, i.e., 1 resource unit. The overall throughput, as in the case of the read/write model, is very low. The throughput begins to drop at *mpl.level* = 25. As multiprogramming level is increased beyond this value, the relative improvement in throughput is appreciable with recoverability, i.e., improvement in performance is observed only after the system begins to thrash heavily.

5.6 Summary of Simulation Results

Based on the studies reported so far, we can make the following observations:

—The use of recoverability does result in better performance (smaller transaction response times and higher throughput in the system). This improvement in performance occurs in spite of transaction aborts due to cyclic commit dependencies. For the read/write model with recoverability, the improvement at the peak throughput value with infinite resources is 67 percent and with 5 resources is 15 percent. For the abstract data type model, with recoverability (P_r = 4), the improvement at the peak

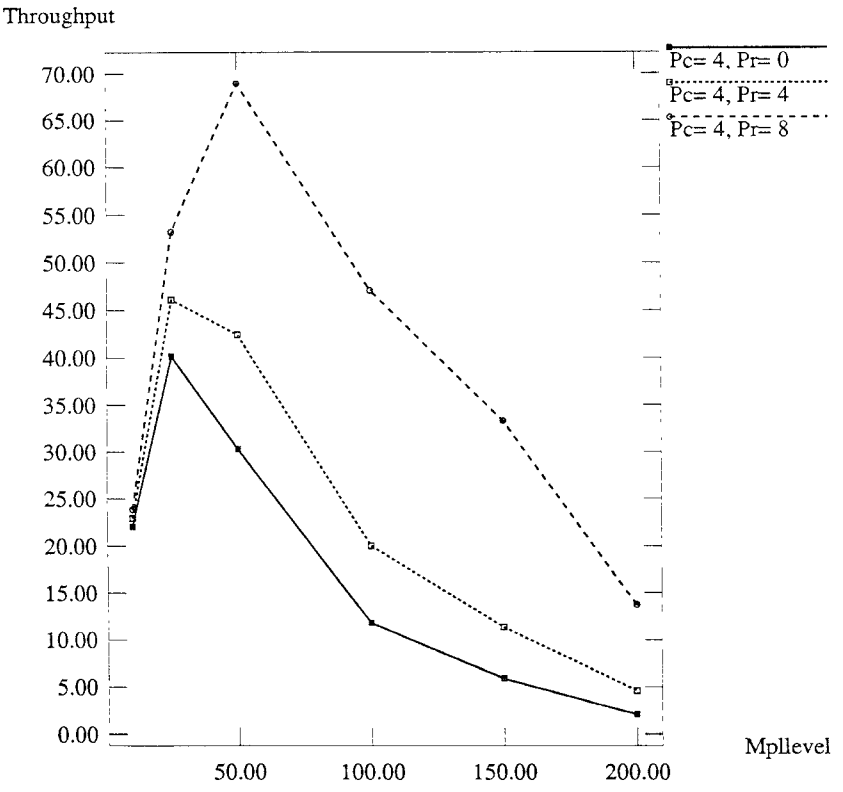

Fig. 14. Throughput (∞ resources).

throughput value with infinite resources is 15 percent and with 5 resources is 6 percent.

—For the abstract data type model, at higher values of recoverability, thrashing occurs at higher values of mpl.level. For $P_r = 8$ thrashing sets in at *mpl.level* = 50 where as for $P_r = 0$ and 4 thrashing occurs at *mpl.level* = 25. This increases the effective range of *mpl.level* over which the system can operate without thrashing. Further, the use of recoverability not only increases the throughput but also decreases the amount of thrashing at higher levels of multiprogramming. This effect is seen by the decrease in the rate of fall of throughput at higher values of multiprogramming for different values of recoverability.

—The relative improvement in performance with recoverability is also a function of resource contention. Unlike an optimistic concurrency control scheme that performs better than blocking scheme only under infinite resources [1], recoverability-based scheme performs better than commutativity both under infinite resources and multiple resources. The lower the resource contention, the better the improvement. However, with very limited resources (when the number of resource units is 1), transactions are

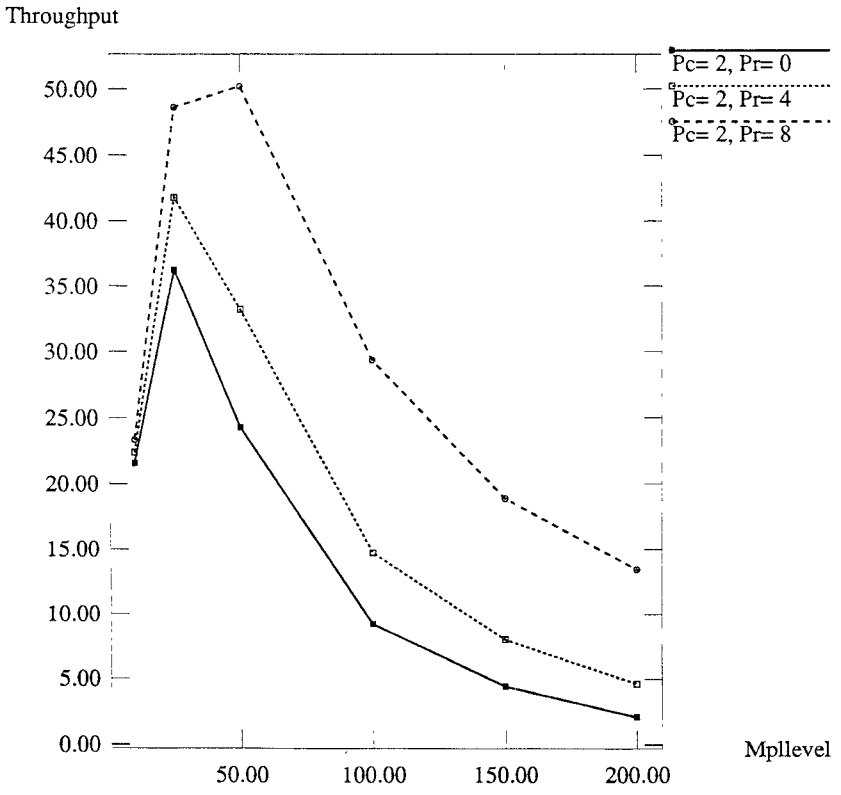

Fig. 15. Throughput (∞ resources).

queued more for resources than they are for data. Hence, only in this case we do not see much improvement in performance when recoverability is used.

Thus, both in the case of the abstract data type model, and the read/write model, use of recoverability results in performance improvement. The improvements in performance suggest that the use of semantics in concurrency control justifies the concomitant sophistication in the scheme employed, *even for transactions performing reads and writes.*

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 predefined 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

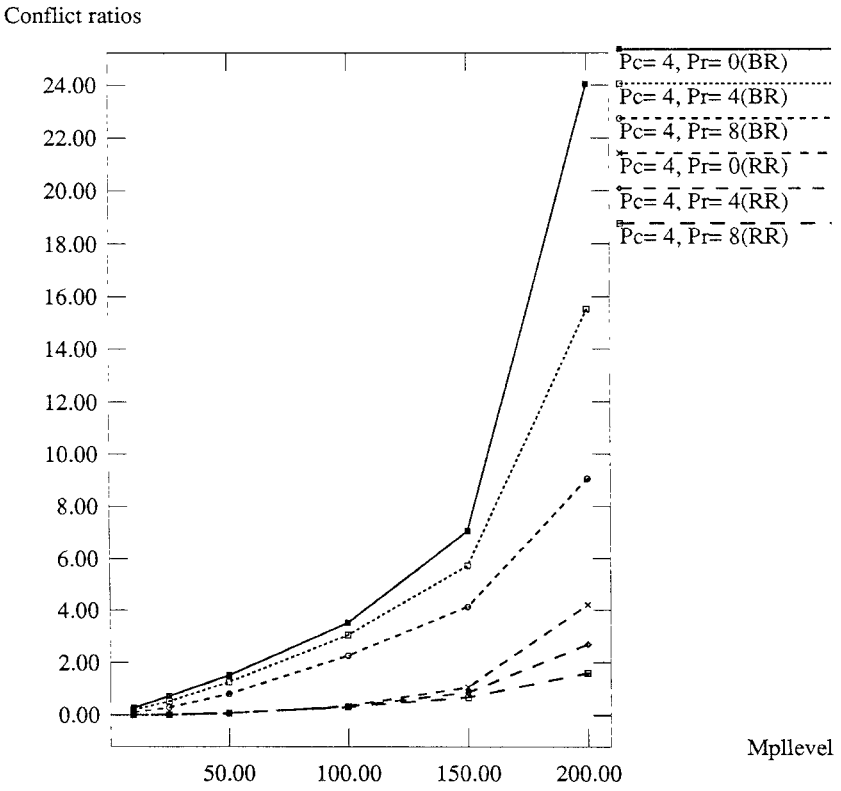

Fig. 16 Conflict ratios (∞ resources)

of recoverability not only reduces the latency involved in processing noncommuting operations but also avoids cascading aborts. As we saw in the examples of Section 3.2, noncommuting but recoverable operations are not uncommon both in the read/write model and in the abstract data type model, and hence we expect the increase in concurrency to be of significant importance.

Since the dynamic commit dependency relationship between transactions can be cyclic, serializability may be violated as transactions execute; thus, transactions may be aborted to maintain serializability. In fact, a cycle may consist of wait-for edges as well as commit-dependency edges. In a distributed system, a distributed cycle checking algorithm has to be employed; but this is needed anyway to check for cycles formed by just the wait-for edges.

From the viewpoint of a user, a transaction completes when it pseudo commits. A pseudo committed transaction can commit after the termination, i.e., commitment or abortion, of all the transactions with which it has commit dependencies. Section 4.3 explained how to commit a pseudo committed transaction. In a distributed system, the overhead involved in achieving this can be reduced by combining the process of commitment of a pseudo committed transaction and the traditional commit protocol [2]; information

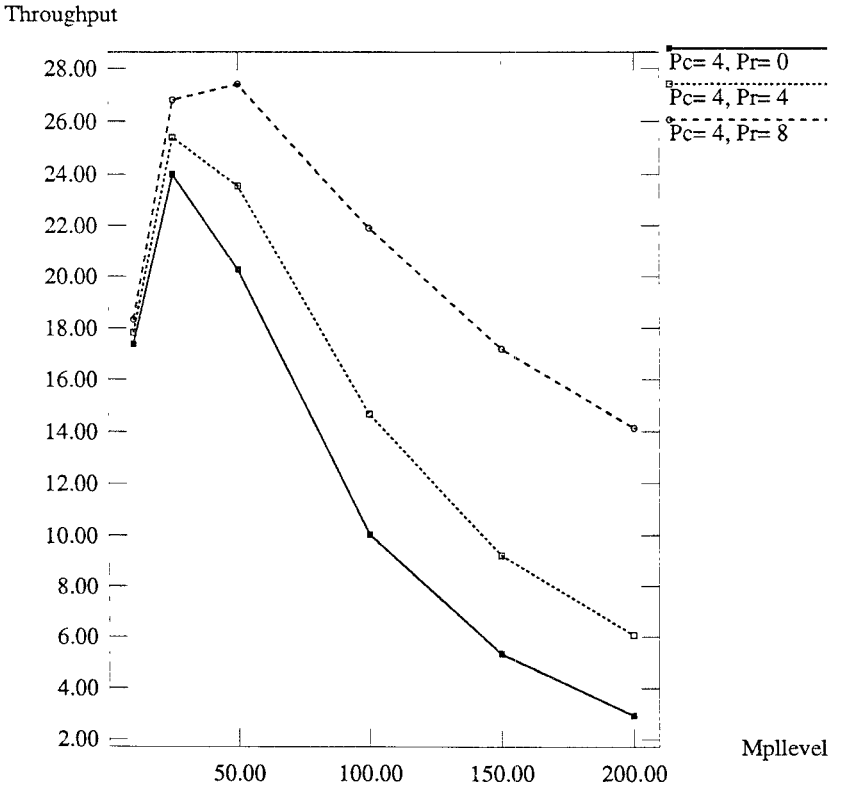

Fig. 17. Throughput (5 resources).

needed for this purpose can be piggybacked on messages used for the commit protocol.

Simulation studies indicate that for objects whose compatibility tables have a reasonable number of recoverable operations, as in the examples of Section 3.2, the improvement in performance is appreciable both under infinite and multiple resource units. The lower the resource contention, the better the improvement in performance. As the performance results indicate, the notion of recoverability is a powerful concept that produces improvement in transaction throughput even for the traditional read/write model. In general, the magnitude of this improvement is dependent on transaction loads as well as the commutativity and recoverability properties of operations on shared objects. As an extension of this work, the notion of recoverability is used in *multilevel* concurrency control protocols for complex information systems [2, 4].

ACKNOWLEDGMENTS

The authors are extremely grateful to the anonymous referees for their detailed comments and for making numerous suggestions for improving the quality of the presentation and the simulation studies.

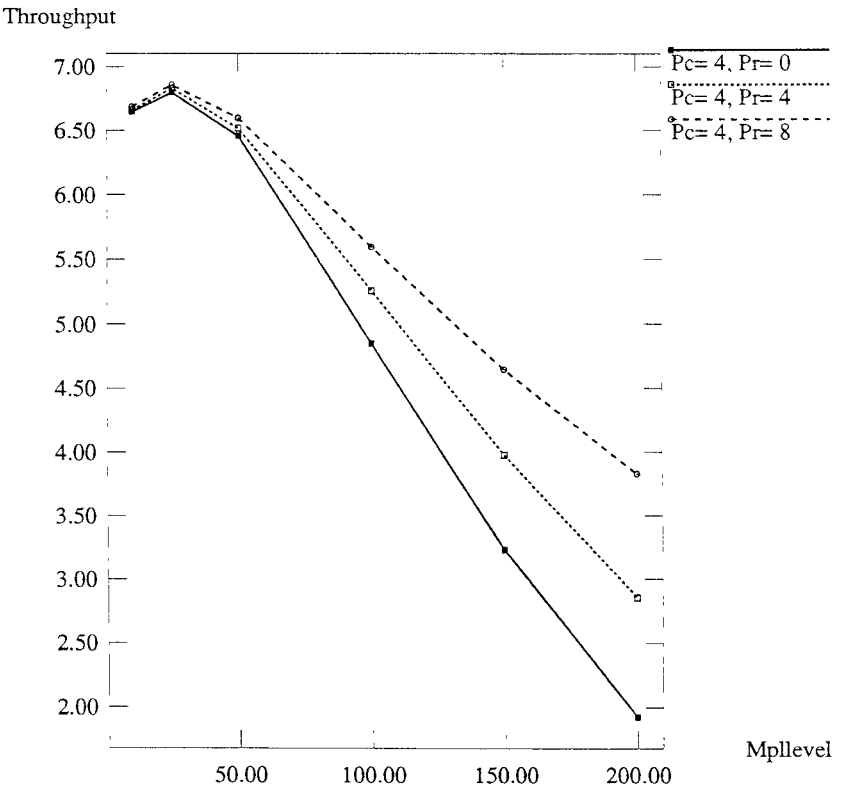

Fig 18. Throughput (1 resource).

REFERENCES

1. Agrawal, R., Carey, M. J., and Livny, M. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 609-654.
2. Badrinath, B R. Concurrency control in complex information systems: A semantics-based approach. Ph.D. dissertation, TR 89-91, Univ. of Massachusetts, Amherst, Mass., 1989.
3. Badrinath, B. R. and Ramamritham, K. Semantics-based concurrency control: Beyond commutativity. In *Fourth IEEE Conference on Data Engineering* (Los Angeles, Feb. 3-5, 1987), pp. 132-140.
4. Badrinath, B. R. and Ramamritham, K. Performance evaluation of semantics-based multilevel concurrency control protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Atlantic City, N.J., May 23-25, 1990), pp. 163-172
5. 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* (Montreal, Aug. 1983), pp. 45-62.
6. Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
7. Birman, K. P., Joseph, T. A. Raeuchle, T., and El-Abbadi, A. Implementing fault-tolerant distributed objects. *IEEE Trans. Softw. Eng. 11*, 6 (June 1985), 520-530
8. Bracha, G. and Toueg, S. Distributed algorithm for generalized deadlock detection. In *Third Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1984), pp. 285-301.

9. BUCKLEY, G. N. AND SILBERSCHATZ, A. Beyond two phase locking. *J. ACM. 31*, 2 (Apr. 1985), 314-326.
10. CASANOVA, M. A. The concurrency control problem for database systems. Vol. 116. In *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
11. CORDON, C. AND GARCIA-MOLINA, H. The performance of a concurrency mechanism that exploits semantic knowledge. In *Fifth International Conference on Distributed Computing Systems* (Denver, Colo., May 13-17, 1985), pp. 350-358.
12. DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, Mass., June 1984), pp. 1-8.
13. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notion of consistency and predicate locks in a database system. *Commun. ACM. 19*, 11 (Nov. 1976), 624-633.
14. GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June 1983), 186-213.
15. HADZILACOS, V. Issues of fault tolerance in concurrent computations. Tech. Rep. 11-84, Harvard University, Aiken Computation Laboratory, Cambridge, Mass., June 1984.
16. HERLIHY, M. P. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst. 15*, 1 (Mar. 1990), 96-124.
17. HERLIHY, M. P. AND WEIHL, W. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems* (Austin, Tex., Mar. 21-23, 1988), pp. 201-210.
18. KORTH, H. F. Locking primitives in a database system. *J. ACM. 30*, 1 (Jan. 1983), 55-79.
19. KUNG, H. T. AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213-226.
20. MOSS, J. E. B. Nested Transactions: An approach to reliable distributed computing. Ph.D. thesis 260, MIT Cambridge, Mass., April 1981.
21. 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* (Washington, D.C., May 28-30, 1986), pp. 72-83.
22. OKI, B. M., LISKOV, B. H., AND SCHEIFLER, R. W. Reliable object storage to support atomic actions. In *Tenth ACM Symposium on Operating Systems Principles* (Washington, Dec. 1-3, 1985), pp. 147-159.
23. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM. 26*, 4 (Oct. 1979), 631-653.
24. SCHWARTZ, P. M. AND SPECTOR, A. Z. Synchronizing shared abstract data types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223-250.
25. SINHA, M. AND NATARAJAN, N. A priority based distributed deadlock detection algorithm. *IEEE Trans. Softw. Eng. 11*, 1 (Jan. 1985), 67-80.
26. STONEBRAKER, M. R., KATZ, R., PATTERSON, D., AND OUSTERHOUT, J. The Design of XPRS. In *Proceedings of the 14th VLDB Conference* (Los Angeles, Calif., Aug. 29-Sept. 1, 1988), pp. 318-330.
27. TAY, Y. C., GOODMAN, N., AND SURI, R. A mean value performance model for locking in databases: The no-waiting case. *J. ACM. 32*, 3 (July 1985), 618-651.
28. TAY, Y. C., GOODMAN, N., AND SURI, R. Locking performance in centralized databases. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 415-462.
29. WEIHL, W. Specification and implementation of atomic data types. Ph.D. Thesis MIT/LCS/TR-314, MIT, 545 Technology Square, Cambridge, Mass., March 1984.
30. WEIHL, W. Commutativity-Based concurrency control for abstract data types. *IEEE Trans. Comput. 37*, 12 (Dec. 1988), 1488-1505.
31. WEIHL, W. AND LISKOV, B. H. Implementation of resilient, atomic data types. *ACM Trans. Program. Lang. Syst. 7*, 1 (Apr. 1985), 244-269.

Received July 1987; revised January 1991; accepted January 1991