SEMANTICS-BASED CONCURRENCY CONTROL: BEYOND COMMUTATIVITY

B.R. Badrinath and Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 89-103
(Replaces COINS TR 86-18)
September, 1989

Semantics-Based Concurrency Control: Beyond Commutativity [1]

B. R. Badrinath
Krithi Ramamritham

Computer and Information Science
University of Massachusetts
Amherst MA. 01003

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

Abstract

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

Contents

1 Introduction

In object-oriented transaction environments it is desirable to attain as high 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* [13, 27]. For example, two insert operations on a set object commute and hence, can be executed in parallel; further, regardless of whether one operation commits, the other can still commit. Applying the same rule, two push operations on a stack object do not commute and hence cannot be executed concurrently. We have identified a property we term *recoverability* to decrease the delay involved in processing non-commuting operations. It turns out that two push operations are recoverable and hence can be executed in parallel.

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

Schemes for improving concurrency 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 its 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. We have developed a centralized and a distributed protocol to detect cyclic dependencies and abort transactions to ensure serializability. We have combined the process of checking for cyclic dependencies with the first phase of the commit protocol. This greatly reduces the overheads involved in providing additional concurrency through 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 and commit protocol designed to utilize recoverability semantics. Results of extensive simulation studies are reported in Section 5. Section 6 concludes with a discussion.

2 Related Work

Most locking protocols used in semantics-driven concurrency control base conflicts between operations on the notion of commutativity of operations [4, 27, 15]. 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 [29], 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 [6, 22]. These protocols provide more concurrency than protocols using general commutativity [4].

The term recoverability also appears in [14, 5]. 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 [21]. 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 [16], 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.

In [8], locking protocols using structural information about the data items are developed to permit only non-cascading rollback. Their model has only read and write operations, and the database is structured as a directed hypergraph. In addition, associated with each transaction is a static set of entities which it must access first.

In [3], we introduced 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 [24]. 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.

In our work we have used the notion of recoverability to define conflicts between operations. We use the semantic information that is available from the specifications of data types to determine recoverability of two operations. Use of the recoverability criterion provides more concurrency than commutativity while avoiding cascading aborts. To ensure serializability, we have developed an algorithm for detecting cycles in the transaction commit dependency relation. The algorithm is based on maintaining dependency graphs [9]. However, we check for acyclicity only when transactions commit by using a new protocol. A node corresponding to a transaction remains in the graph only until the transactions on which it depends commit or abort. We have combined the process of checking for acyclicity of the dependency graph with the first phase of the commit protocol.

3 A Formal Definition of Recoverability

3.1 Operations and Recoverable Operations

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

Concurrent execution of a set of transactions $T_1, T_2, \ldots, T_n$ gives rise to a log $E = (OP_E, <_E)$. OP_E is $(\cup_i OP_{T_i})$ and $(\cup_i <_{T_i}) \subseteq <_E$. $<_E$ is a partial order on the operations in OP_E and the log represents the order in which they are executed by the system. If $o_i <_E o_j$ we say that o_j executed after o_i. The execution log is serializable if there exists a total order $<_s$ called a serialization order on the set $\{T_1, T_2, \ldots, T_n\}$ such that if an operation o_i in transaction T_i *conflicts* with an operation o_j in T_j, and if $T_i <_s T_j$, then $o_i <_E o_j$ [12]. 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 [2] produced by operation o, and $state(o, s)$ which is the state produced after the execution of 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 $s \in S$

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

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

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

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 $\forall_{1 \leq i < n}$ $o_i <_E o_{i+1}$. $(o_n\ \text{RR}\ o_1)$ 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 $\forall\ l,\ 1 \leq l < n$, $(o_n\ RR_I\ o_l)$ then $(o_n\ \text{RR}\ o_1)$.

Proof: Let F denote the operations that execute between between o_n and o_1. The proof is by induction on k where $k = |F|$.
Induction base ($k = 1$ i.e., F contains only one operation): Let $S = \{o_3, o_2, o_1\}$. 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 $(o_n\ RR_I\ o_1)$, $(o_n\ \text{RR}\ o_1)$.
Induction hypothesis (F contains $k-1$ operations): if $\forall\ l,\ 1 \leq l \leq k$, $(o_n\ RR_I\ o_l)$, then $(o_n\ \text{RR}\ o_1)$.
Induction Step: Let $|F| = k$ and $S = \{o_n, o_{k+1}, \ldots, o_2, o_1\}$. Now $(o_n\ RR_I\ o_{k+1})$ and $(o_n\ RR_I\ o_k) \Longrightarrow (o_n\ \text{RR}\ o_k)$ by using a reasoning similar to the base case. From definition 3 we have $o_n\ \text{RR}\ o_k \Longrightarrow o_n\ RR_I\ o_k$, and by induction hypothesis $\forall\ l\ 1 \leq l \leq k\ o_n\ RR_I\ o_l \Longrightarrow o_n\ RR\ o_1$.

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

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.

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, (read, read) is commutative and in Table 2, (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 1. However, with recoverability this notion of conflict is weakened as the only pair of

Table 1: Commutativity for page

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

Table 2: Recoverability for page

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

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 3 and 4. 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* specifies the item to be deleted from the object. If the item is present in the set, it returns *Success,* otherwise, it returns *Failure. Member* determines whether a specified item is an element of the set object. Inserting two elements is commutative; so is deleting different elements. Similarly, insert and member involving different elements commute but do not commute when the specified elements are the same. However, insert is recoverable relative to member, as indicated by the Yes entry.

Table 3: Commutativity for stack

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

Table 4: Recoverability for stack

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

Table 5: 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 6: 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 7: 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

Table 8: 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

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. 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.

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 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 in [27] convenient to describe a sequence of operations invoked on an object. We will consider operations to be events, where an event is a paired operation invocation and response. As an example, consider an object of type *set*. Invoking *insert*(i) inserts the element i into the set and returns "ok" when the operation is completed. Thus, if the integer set object *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.

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

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:

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

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

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

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

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

Since recoverable operations force commit dependencies, a transaction may commit only after other transactions on which it depends commit. However, the semantics of the execution of the transaction are not affected by the commit/abort of other transactions with which it has a commit dependency. Hence a transaction can *complete* execution; with the exception that the operations and the transaction continue to remain in the execution log and commit dependency graph respectively. We call this sort of commit a *pseudo-commit*. Note that this is different from the conditional commit of nested transactions [18], wherein a transaction that has conditionally committed may be forced to abort by its parent. A transaction which has *pseudo-committed* will definitely commit, but only after all transactions on which it depends terminate, i.e., commit or abort, thus respecting the commit dependency relationship. A similar notion called *pre-commit* appears in [11].

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 [16]. We have combined the process of checking the dependency-graph for acyclicity with the first phase of the standard two phase commit protocol.

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

4.1 Correctness Requirements and Commit Dependency Graph

Definition 4: An operation o_i invoked by transaction T_i is *sound* in a log E if for any *extension* $E' = E \parallel A_j$ for any $j \neq i$ ($\parallel$ indicates that 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.

The object manager uses compatibility tables for the objects to determine whether an operation is sound with respect to other uncommitted operations in the log. Once an operation is requested the object manager determines the type of conflict with other uncommitted operations. If the operation is neither recoverable nor commutative with other uncommitted operations, the transaction is made to wait. Deadlocks due to cyclic waits of non-recoverable operations can be handled using known techniques of deadlock avoidance, or deadlock detection and resolution [7, 23].

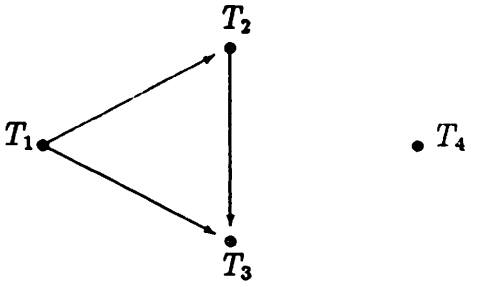

Figure 1: A dependency graph

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

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

The proof follows from the definition of serializability.

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

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

Figure 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 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 it, i.e., as it invokes new operations. The algorithm is given in Figure 2.

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

1. If there is at least one ongoing operation with which o_i is not recoverable then T_i is made to wait.
2. If for all operations o_j, o_i and o_j are commutative or o_i is recoverable relative to o_j then
   - Insert a node corresponding to the transaction T_i. Insert directed edges from node T_i to other transactions which have invoked operations with which o_i is recoverable.

Figure 2: Algorithm to insert commit dependency edges

4.2 A Two Phase Algorithm for Pseudo-committing Transactions

4.2.1 Centralized Algorithm

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

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

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

Table 9: Stepwise execution of pseudo-commit algorithm

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

a time. The two sets of timestamps sent by the object managers are:

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

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

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

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

Having collected these sets the transaction manager then determines whether a transaction can pseudo-commit. The pseudo code for the entire algorithm is shown in Figure 3.

Using Figure 4, which shows dependency graphs at three objects, we illustrate, in Table 9, the execution of the pseudo-commit algorithm in steps. At the beginning none of the transactions have pseudo-committed.

Correctness arguments for the pseudo-commit algorithm: Each pseudo-committed transaction T_i at each object has a pair of sets: $SUCC_{obj}(T_i) = \{\ T_n /\ T_i \stackrel{+}{\Longrightarrow} T_n$, T_n is pseudo-committed and every T_m along the path from T_i to T_n is also pseudo-committed $\}$. $PRED_{obj}(T_i) = \{\ T_j /\ T_j \stackrel{+}{\Longrightarrow} T_i$, T_j is pseudo-committed and every T_k along the path from T_j to T_i is also pseudo-committed $\}$. To determine

```
Begin

    Transaction T_i intends to Pseudo-commit;
    For each Obj visited by T_i do
    {
    Send "prepare to pseudo-commit" message to the
    manager of Obj
    Collect PRED_obj(T_i) and SUCC_obj(T_i);
    }
    PRED(T_i) = ⋃_obj PRED_obj(T_i);
    SUCC(T_i) = ⋃_obj SUCC_obj(T_i);

    If PRED(T_i) ∩ SUCC(T_i) ≠ ∅ then
        Send "Abort T_i" message to all object
        managers visited by T_i
    else
        Send "pseudo-commit T_i" message along with
        PRED(T_i) and SUCC(T_i) to all object
        managers visited by T_i;

End
```

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

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

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

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

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

Figure 3: Pseudo code for the the commit algorithm

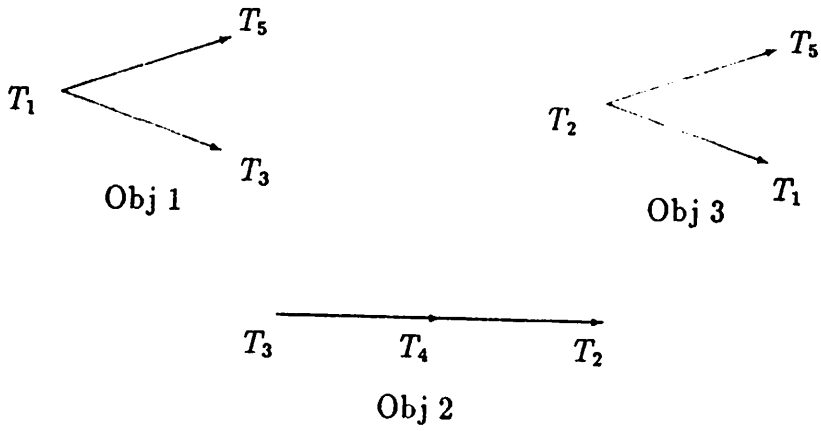

Figure 4: Dependency graphs at three objects

whether T_i can pseudo-commit or not, the transaction manager collects these sets corresponding to *immediate* successors and *immediate* predecessors of T_i from each object that T_i has visited to obtain $PRED(T_i)$ and $SUCC(T_i)$. The proof that a transaction is actually in a commit dependency cycle when the intersection of $PRED(T_i)$ and $SUCC(T_i)$ is non empty follows from the following theorem.

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

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

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

4.2.2 Distributed Algorithm

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

In the distributed case, the local managers of the sites that contain objects visited by a transaction T_i intending to pseudo-commit are the cohorts of the manager of T_i.

Each site has a local manager. Each transaction is assigned a unique timestamp using a system of Lamport clocks [17]. The local managers maintain two sets known as PC and AC. The set AC contains transactions that have started the pseudo-commit process at this site and the set PC contains transactions that have pseudo-committed. The commit dependency graph, and hence the PRED and SUCC sets, are still maintained by the object managers. When a cohort receives a pseudo-commit request, the local manager determines whether there is a cycle locally as in the centralized algorithm. If there is a cycle an abort message is sent to the transaction manager. On the other hand, if there exists no cycle at a site, then the cohort will send AC and PC along with PRED and SUCC sets to the transaction manager. The set AC is then updated to include the transaction initiating the pseudo-commit. Note that the process of checking for a local cycle, sending the sets, and updating AC is assumed to be done in an atomic manner.

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

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

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

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

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

- **Phase I**

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

  - The cohorts then check for a local commit dependency cycle; if there is a local commit dependency cycle then an *abort* message is sent. However, if there is no cycle locally then the sets $PRED(T_i)$ and $SUCC(T_i)$ are sent along with the sets AC and PC. T_i is then added to AC.

- If there was no abort message from any of the cohorts then the transaction manager determines whether there exists a $T_j <_c T_i$ and T_j has not completed the pseudo-commit process. If such a T_j exists then the transaction manager will send a request to cohorts for obtaining the sets PRED and SUCC again. However, this time the cohorts will wait for all T_j such that $T_j <_c T_i$ to complete the pseudo commit process (as we shall see, this happens when T_j is removed from AC) and then send the sets $PRED$ and $SUCC$.

- **Phase II**

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

  - Transaction manager sends *abort* T_i message.

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

  Begin

  If $\bigcup_{cohorts} SUCC_{cohorts}(T_i) = \emptyset$ **then**
  - Transaction manager sends *Commit* T_i message.

  Else
  - Transaction manager sends *Pseudo Commit* T_i message.

  End

- A cohort does the following:

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

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

When T_i completes the first phase of the commit protocol, using AC and PC, the transaction manager determines the set of transactions $B_f = \{T_j / T_j <_c T_i\}$. If this set is non empty then in order to ensure that there is no global cycle, T_1 has to wait until all such T_j complete, at which point the cohorts send PRED and SUCC sets. While T_i is waiting, since T_i is in AC at each site visited by it, any other T_l which starts the pseudo-commit process will find $T_i <_c T_l$ and hence cannot belong to the set B_f, i.e., cannot pseudo-commit before T_i. Thus $<_c$ imposes an ordering on transactions attempting to pseudo-commit thus avoiding race conditions.

Before we conclude this section we look at the problem of effecting aborts. When a transaction aborts, it is necessary to undo (back out) a transaction. Undo of a transaction involves undo of all operations executed by a transaction. Recovery from transaction abort can be achieved using two different approaches: using *intention* lists or using *undo* logs [20, 19, 28]. Further, the type of undo is dependent upon the operation. For example (write, read) is recoverable but there is no need to undo

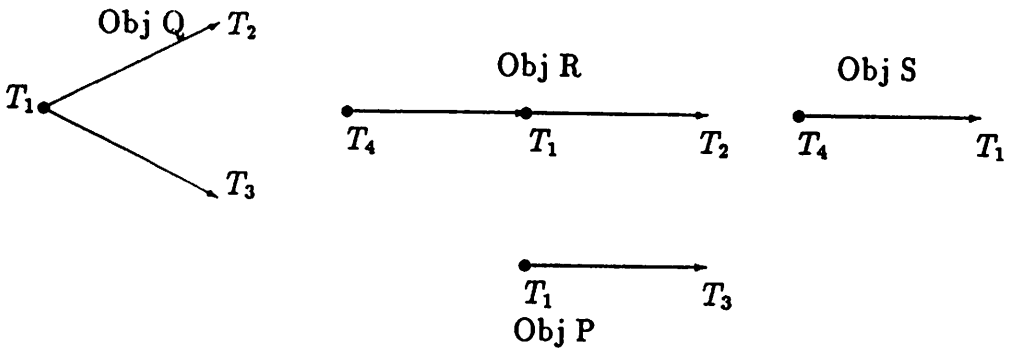

Figure 5: Dependency graphs at objects P, Q, R, and S

a read operation. However (write, write) is recoverable but a write operation needs undo. 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 in [28]. These schemes can be adapted to effect recovery in our concurrency control scheme.

4.3 Committing Pseudo-committed Transactions

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

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

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

The members of CD-SET and VISIT-SET for each transaction can easily be determined by the transaction manager when it collects the PRED and SUCC sets at each object as part of the commit protocol. Hence the transaction manager sends these sets to the object managers involved during the second phase of the commit protocol. Consider the scenario shown in Figure 5. At object P, CD-SET(T_1) = $\{Q, R\}$ and VISIT-SET(T_1) = $\{Q, R, S\}$. At object S, the sets CD-SET(T_1) = $\{P, Q, R\}$ and VISIT-SET(T_1) = $\emptyset$.

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

5 Results of Simulation Studies

We now report on simulation studies designed to evaluate the increased concurrency resulting from the use of recoverability. The purpose of this simulation study is to compare the amount of concurrency offered when both commutativity and recoverability are used to determine conflicts as opposed to using just commutativity. 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 represents only 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 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, 26].

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 6 is a modified version of the one used in [1].

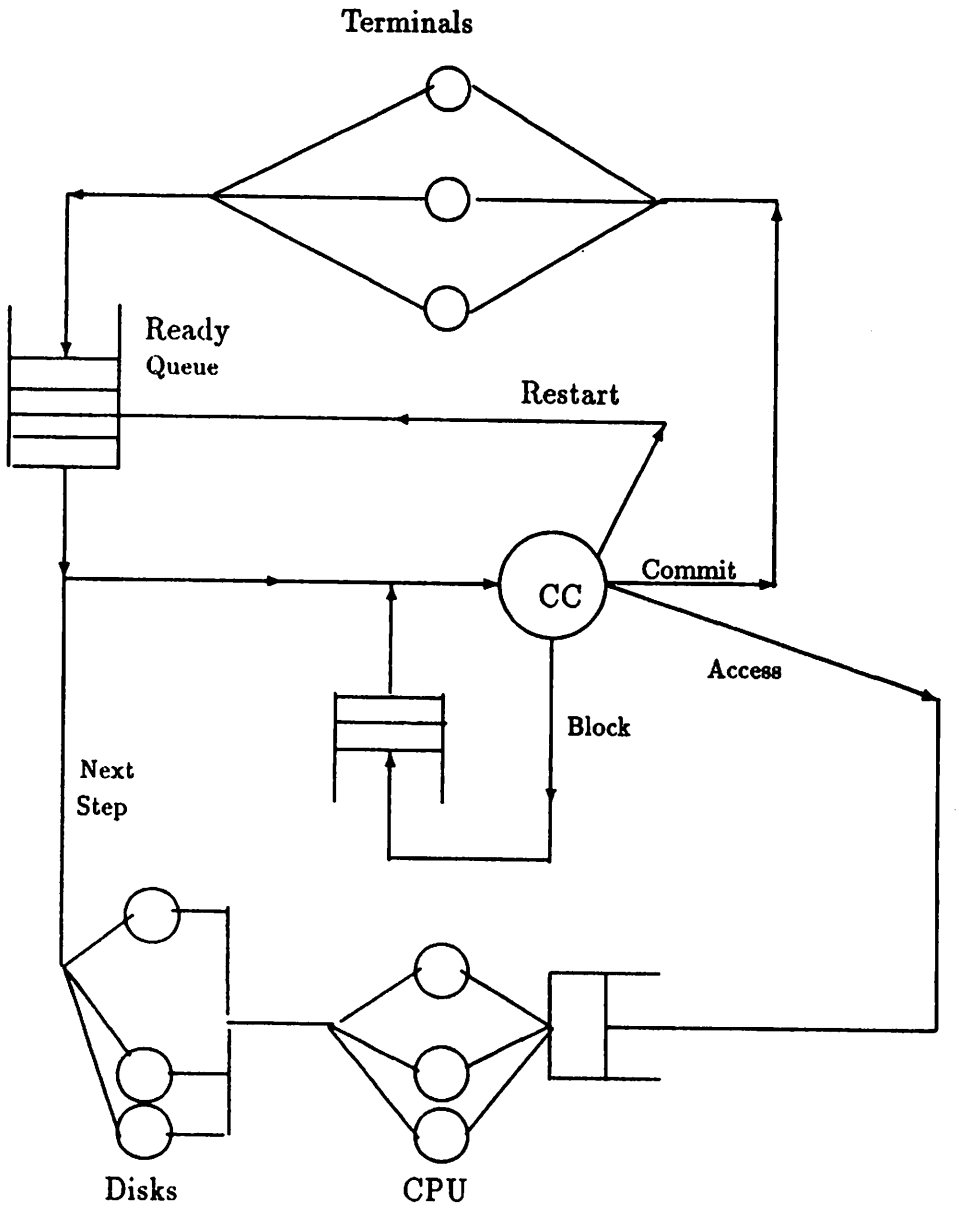

Figure 6: Simulation model

Table 10: 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
Commit.delay	Mean time to commit a transaction
Write.probability	Probability of a Write operation

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 computing the current level of multiprogramming 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 a deadlock detection is initiated every time it blocks. A transaction is aborted if a deadlock is discovered or else the transaction is made to wait until the conflict is resolved. Transactions also abort due to cyclic dependencies arising from executing recoverable operations. An aborted transaction is restarted immediately. A restarted transaction behaves, with respect to operation invocations, like a new, 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 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 randomly and waits in the queue of the selected disk until it can be served [1].

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*. The cost of committing a transaction is modeled by introducing a fixed delay *commit.delay*. This is the delay between when a transaction completes all of its operations and when the managers of objects visited by the transaction are informed by the transaction manager about the (pseudo)commitment or abortion of the transaction. The various model parameters and their meanings are listed in Table 10.

5.2 Experiment 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 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. However, if the transaction is in a deadlock, the transaction is restarted.

As we have mentioned earlier, the operations executed by a pseudo committing transaction are still considered in determining conflicts for other transactions and hence *active* until a pseudo committed transaction commits. However, the results of the execution of a pseudo committed transaction are durable. We model this effect by allowing the terminal that issued the pseudo committed transaction to initiate, after its thinking time, a new transaction, without changing the current level of multiprogramming in the system.

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 transaction 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 of running the experiments with fixed transaction sizes, we use a transaction mix consisting of transactions whose length is

Table 11: 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 secs
CPU.time	0.015 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

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 secs of CPU time and 0.035 secs of disk access time. Thus, in the case of infinite resources each step takes 0.05 secs.

Recall that an operation that is neither commutative with nor recoverable relative to all ongoing operations is made to wait. Such waits may lead to deadlocks. We have made use of time-outs to tackle this problem. If an operation request is not satisfied within 5 seconds, the invoking transaction is aborted. We term such aborts t-aborts.

We do not model the details of the communication between a transaction manager and the object managers; however, we take into account the cost of the commit process by introducing a delay in the commit protocol. This delay is fixed at 0.6 seconds. The overheads not considered in our model are: the cost involved in maintaining the commit-dependency graph, and the communication cost of committing a pseudo committed transaction.

The nominal values of the parameters are listed in Table 11. The values of the model parameters have been chosen similar to those in previous performance studies of locking protocols [26, 25, 1] and commutativity-based protocols [10].

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 commit-

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

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 level in the system. The *restart ratio* is defined as the number of times a transaction has to be restarted before it completes. A transaction is restarted if it blocks and is found to be in deadlock. We call such aborts t-aborts. Recall that one of the transactions in a commit dependency cycle is aborted. We call such aborts r-aborts. The restart ratio includes the restarts due to r-aborts and t-aborts. 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 *r-abort ratio.* This is defined as the percentage of the number of r-aborts to the total number of transactions that complete.

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, and r-abort ratio. The graphs in Figures 7 through 51 show the average results of 10 runs, with sufficiently tight 90 percent confidence intervals. Though, the confidence intervals are omitted from our graphs, the confidence intervals were within the range of $\pm 2\%$ 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 in [26], 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 obtained.

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 1, there are three pairs of conflicting operations. Secondly, 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 the decreased conflicts are able to counter the negative effects of r-aborts due to cyclic commit dependencies. 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 7 shows throughput as a function of the level of multiprogramming. The throughput under both commutativity and recoverability increases with multiprogramming level and after certain level drops as multiprogramming increases. This is due to thrashing resulting from very high data contention. The maximum throughput was obtained with recoverability at mpl.level = 100. 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.

The effect of using recoverability on response time is shown in Figure 8, where the response time initially decreases with multiprogramming level, and at values of multiprogramming greater than mpl.level = 100 the average response time increases with multiprogramming level. However, with just commutativity, the response time begins to increase with multiprogramming level from 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 9 and 10 show the restart ratio and blocking ratio respectively. The restart ratio and the blocking ratio are smaller with recoverability than without it. Thus, the improvement in concurrency due to reduction in blocking when recoverability is used more than compensates for r-aborts due to cyclic commit dependencies. Further, at high levels of multiprogramming, the restart ratio is smaller than the blocking ratio for both commutativity and recoverability. This confirms earlier results in [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.

The percentage number of r-aborts as a function of multiprogramming level is shown in Figure 11. The maximum value of *r-abort ratio* was found to be 7% at the maximum mpl.level = 200.

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. The results are presented in Figures 12 to 21. For the case of 5 resource units, the throughput first increases with multiprogramming level and then decreases due to thrashing as shown in Figure 12. Further, because of resource contention, the maximum throughput is less than the maximum throughput for the case of infinite resources. In the case where only 1 resource unit is present, in Figure 13 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 13. Thus at higher values of data contention, using recoverability not only improves concurrency but also the improvement gets better when very limited resources are present. Observe that the maximum throughput is higher with recoverability in both the cases.

The results of response time for 5 and 1 resource units are shown in Figure 14 and 15 respectively. The response time for 1 resource unit begins to increase rapidly as thrashing begins to occur. With very limited resources restarts are very expensive. This is because every time a transaction is restarted not only does the average response time gets added but also the transactions have to contend for limited resources all over again. Further, the rate of increase in response time is smaller with recoverability.

The restart ratio for 5 resource units and 1 resource unit is shown in Figure 16 and Figure 18 respectively. The blocking ratio for 5 resource units and 1 resource unit is shown in Figure 17 and Figure 19 Note that the restart ratio and the blocking ratio are smaller with recoverability than with only commutativity. Further, the difference gets higher as level of multiprogramming is increased. The maximum value of *r-abort ratio* for 5 resource units and 1 resource unit was found to be 9.4% and 10.2% respectively. The results are shown in Figures 20 and 21.

5.5.2 Abstract Data Type Model

In this experiment, the operations on the objects can be arbitrary, and the properties of the operations are defined by the compatibility table.

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 a single *compatibility table*; each entry in this table will be one of *commutative*, *recoverable*, or *null*.

To model different degrees of commutativity and recoverability, the properties of operations on an object are specified by two integers: P_c determines the number of

commutative entries in an object's compatibility table; P_r determines the number of *recoverable* entries in this table. Thus, $(N^2 - P_c - P_r)$ is the number of *null* entries where N is the number of operations defined on the object. We experimented with even values of P_c and P_r. (In the graphs depicted in Figures 22 through 51, each graph is for a fixed value of P_c (indicated in the graphs as Commutativity = 2, 4, etc.) and varying values of P_r (indicated as recoverability = 4, 6, etc.,). 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, $P_c/2$ non-diagonal entries in its compatibility table are *randomly* chosen and set be *commutative*; their symmetric entries are then made *commutative*. P_r of the remaining entries are then randomly chosen using a uniform distribution and set to be *recoverable*. The rest of the entries are set to *null*.

In this study, each object has four operations defined on it. For any given object, all of the defined operations can be invoked 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, selection of the object on which to execute the selected operation is random and independent, being chosen from all the objects in the database (i.e., uniformly distributed between 1 and *database size*). For this experiment, the database size was chosen to be 1500 objects.

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 recoverability =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 $P_c = 2$ are presented here. The observations made from the results of the experiments, shown in Figures 37 through 51, are consistent with those for $P_c = 2$

Infinite resources: In this part of the simulation, we assume infinite resources. Figure 22, depicts increased throughput due to recoverability, and Figure 23 depicts the reduced average response time when $P_c = 2$. The throughput increases as a function of multiprogramming. However, beyond mpl.level = 50, the throughput falls and the response time begins to increase. This phenomenon, as in the case of the read/write model, is due to thrashing that is induced by high data contention. However, for higher values of $P_r = 8, 10$, thrashing starts at mpl.level 100, reflecting in the overall reduction in data contention as a high proportion of the operations is considered non-conflicting. For a given level of multiprogramming, as recoverability is increased, the throughput increases and the response time decreases. Further, at higher values of multiprogramming, the relative improvement in throughput obtained with recoverability increases with multiprogramming level.

Increased multiprogramming level implies increased blocking due to higher data contention. Thus, the blocking ratio 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 be seen in the decreasing slope of the curves for increasing values of recoverability shown in Figure 25. A similar observation can be made with regards to the restart ratio shown in Figure 24. Further, the number of aborts caused by cyclic commit dependencies and time outs has been found to be

less than the number of aborts due to time outs when recoverability is not considered. This is clear from the lower restart ratios for various values of recoverability than the restart ratio with recoverability = 0. Figure 26 shows the r-abort ratio for different values of recoverability.

Finite resources: In this part of the simulation, we conducted experiments with 5 resource units and 1 resource unit respectively. The throughput results for 5 resource units and 1 resource unit are shown in Figure 27 and 28 respectively. Due to resource contention, the maximum throughput obtained with 5 resource units is smaller than the maximum throughput with infinite resources. Further, as multiprogramming level is increased beyond mpl.level = 50, the throughput begins to drop as a result of thrashing. Figures 29 and 30 show the response time available resource units of 5 and 1.

With very limited resources (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 is increased beyond this value, the relative improvement in throughput and transaction response time is appreciable with recoverability. This is because of fewer restarts with recoverability as shown in Figure 32, and for high values of resource contention, restarts are more expensive as a restarted transaction experiences a a higher response time contending for limited resources.

Figures 33 and 34 show the blocking ratio for 5 resource units and 1 resource unit respectively. In both cases, the value of blocking ratio increases with data contention, but for a given level of multiprogramming, the blocking ratio decreases with recoverability. Figure 35 and 36 depict the r-abort ratio for 5 resource units and 1 resource unit respectively.

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 smaller transaction response times; the larger the value of P_r, the smaller the response time. This decrease occurs in spite of transaction aborts due to cyclic commit dependencies.

- The use of recoverability not only increases the throughput but also decreases the amount of thrashing at high levels of multiprogramming. This effect is seen by the decrease in the rate of fall of throughput at high values of multiprogramming for different values of recoverability.

- For all values of multiprogramming, the blocking ratio and the restart ratio with recoverability are less than without recoverability. Further, in both cases the rate of increase slows down with increased recoverability.

- The notion of recoverability is especially useful under high data contention. In this case the improvement in various performance metrics increases with increased recoverability.

Thus, both in the case of the abstract data type model, and the read/write model, use of recoverability results in performance improvement. The significant improvements in performance suggests 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 pre-defined conflicts between pairs of operations. Conflicts among operations executed by different transactions can be checked by using a compatibility table, and the table can be derived directly from the data type specification. The use of recoverability not only reduces the latency involved in processing non-commuting operations but also avoids cascading aborts. As we saw in the examples of Section 3.2, non-commuting 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 may be cyclic, serializability may be violated as transactions execute; during the commit phase transactions are aborted to maintain serializability. We have described a scheme to achieve this. Using recoverability as a conflict predicate it was possible to combine the process of commitment and validation of a transaction. This reduces some of the overhead involved in providing additional concurrency. This reduction can be achieved by piggybacking the PRED and SUCC sets on the messages used for the commit protocol.

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

Simulation studies indicate that for objects whose compatibility tables have a reasonable number of recoverable operations, as in examples of Section 3.2, the drop in response time is appreciable. Further, significant improvement in performance occurs both under high values of data contention and under resource contention. The number of aborts (including r-aborts) under various values of multiprogramming is lower than the number of aborts when recoverability is not considered. Thus, the r-aborts, due to cyclic commit dependencies, do not negate the advantages of exploiting recoverability semantics.

As our simulation studies indicate, the notion of recoverability is a powerful concept that produces appreciable drops in transaction turnaround times even for the traditional read/write model. In general, the magnitude of this drop 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 the notion of recoverability is used in *multilevel* concurrency control protocols for

complex information systems[2].

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] Badrinath, B. R. Concurrency control in complex information systems: A semantics-based approach. PhD thesis TR 89-91, University of Massachusetts, Amherst, MA., 1989.

[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] 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.

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

[6] Birman, K. P., Joseph, T. A., Raeuchle, T., and El-Abbadi, A. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[7] Bracha, G. and Toueg, S. Distributed algorithm for generalized deadlock detection. In *Third Annual ACM Symposium on Principles of Distributed Computing*, pages 285–301, August 1984.

[8] Buckley, G. N. and Silberschatz, A. Beyond two phase locking. *Journal of the ACM*, 31(2):314–326, April 1985.

[9] Casanova, M. A. *The concurrency control problem for database systems*, volume 116 of *Lecture Notes in Computer science*. Springer-Verlag, 1981.

[10] Cordon, R. and Garcia-Molina, H. The performance of a concurrency mechanism that exploits semantic knowledge. In *Fifth international conference on distributed computing systems*, pages 350–358, 1985.

[11] 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*, pages 1–8, June 1984.

[12] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[13] Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2), June 1983.

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

[15] Korth, H. F. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

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

[17] Lamport, L. Time, Clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] Moss, J. E. B. Nested Transactions: An approach to reliable distributed computing. PhD thesis 260, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[19] 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.

[20] 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*, pages 147–159, December 1985.

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

[22] Schwarz, P. M. and Spector, A. Z. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[23] Sinha, M. and Natarajan, N. A priority based distributed deadlock detection algorithm. *IEEE Transactions on Software Engineering*, 11(1):67–80, January 1985.

[24] 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.

[25] Tay, Y., Goodman, N., and Suri, R. A mean value performance model for locking in databases: The no-waiting case. *Journal of the ACM*, 32(3):618–651, July 1985.

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

[27] Weihl, W. Specification and implementation of atomic data types. PhD Thesis MIT/LCS/TR-314, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA., March 1984.

[28] Weihl, W. Commutativity-Based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[29] Weihl, W. and Liskov, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(1):244–269, April 1985.

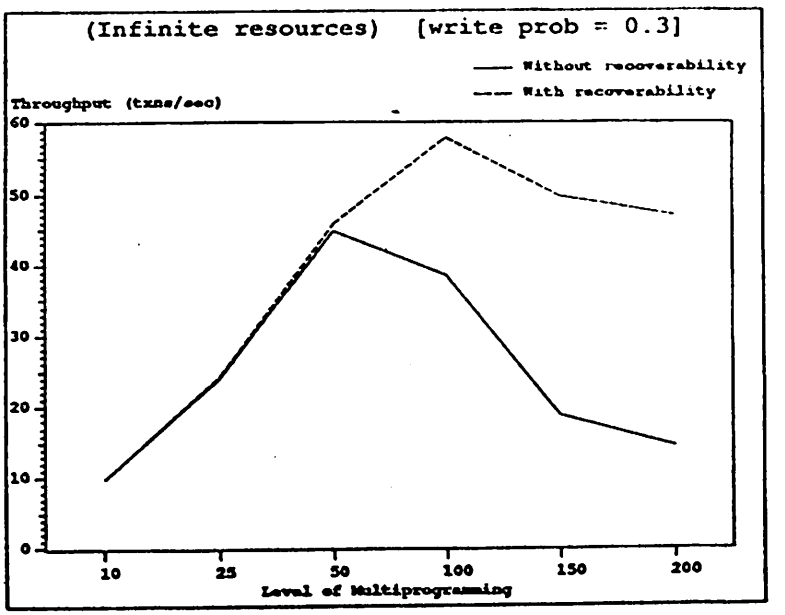

Figure 7: Throughput (infinite resources)

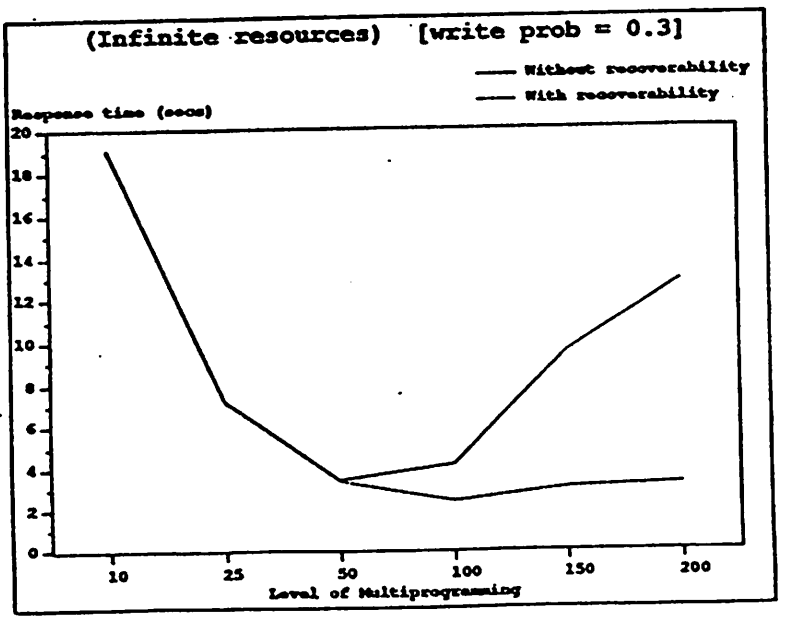

Figure 8: Response time (infinite resources)

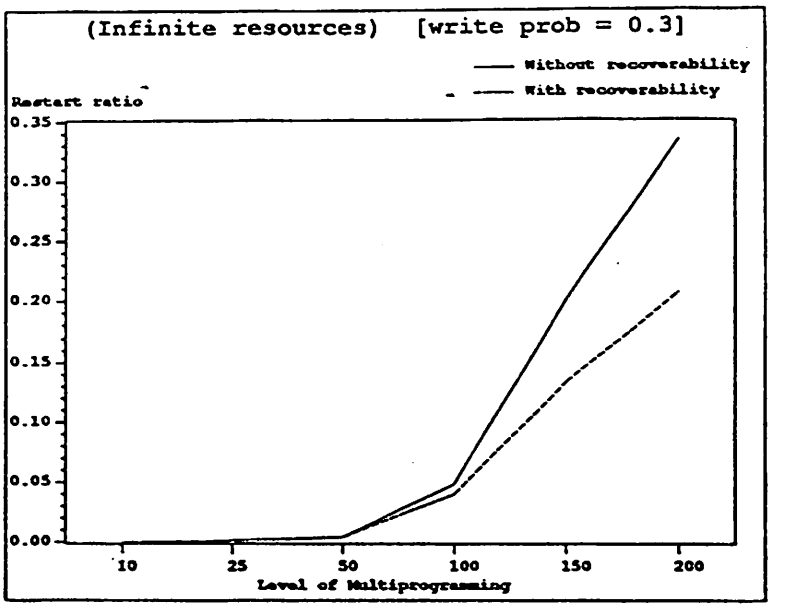

Figure 9: Restart ratio (infinite resources)

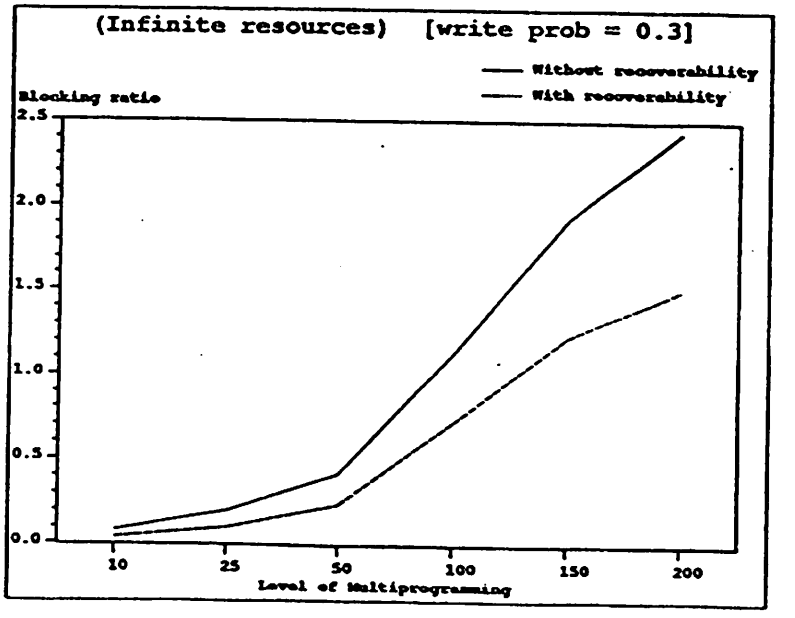

Figure 10: Blocking ratio (infinite resources)

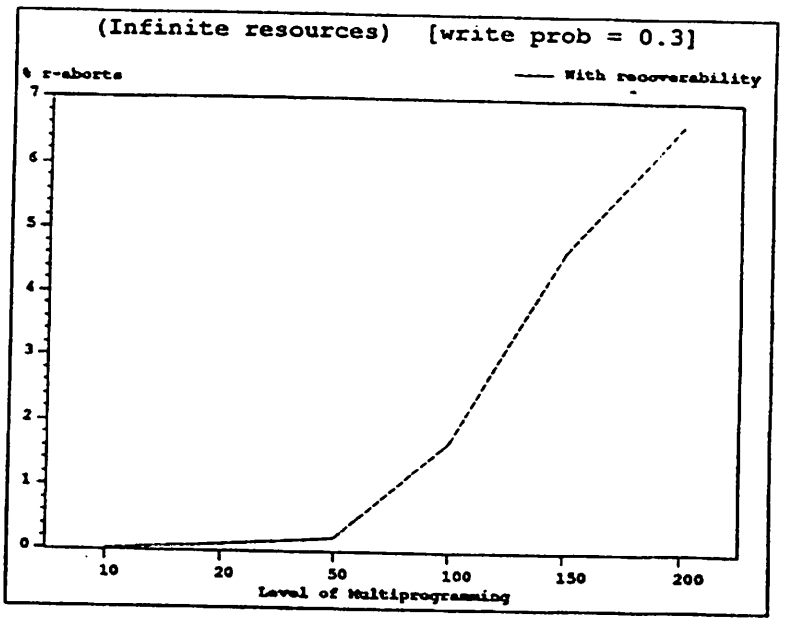

Figure 11: R-aborts (infinite resources)

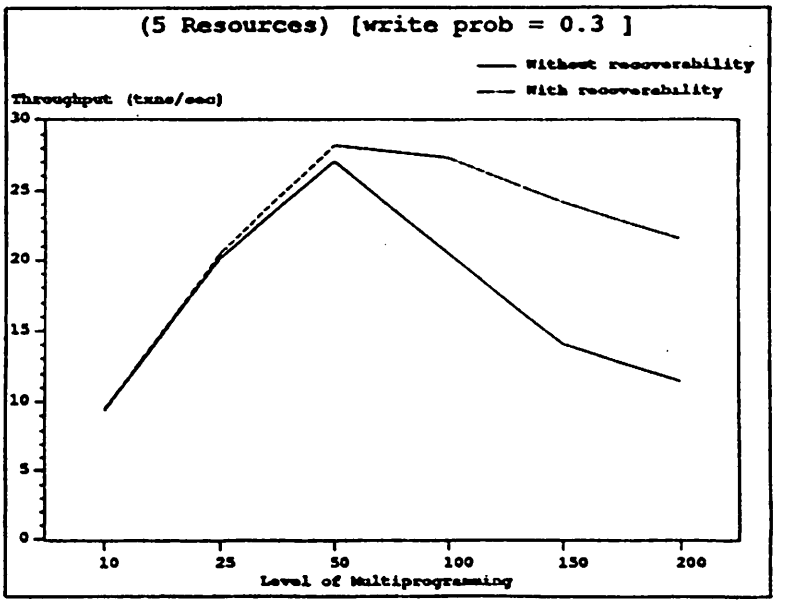

Figure 12: Throughput (5 resource units)

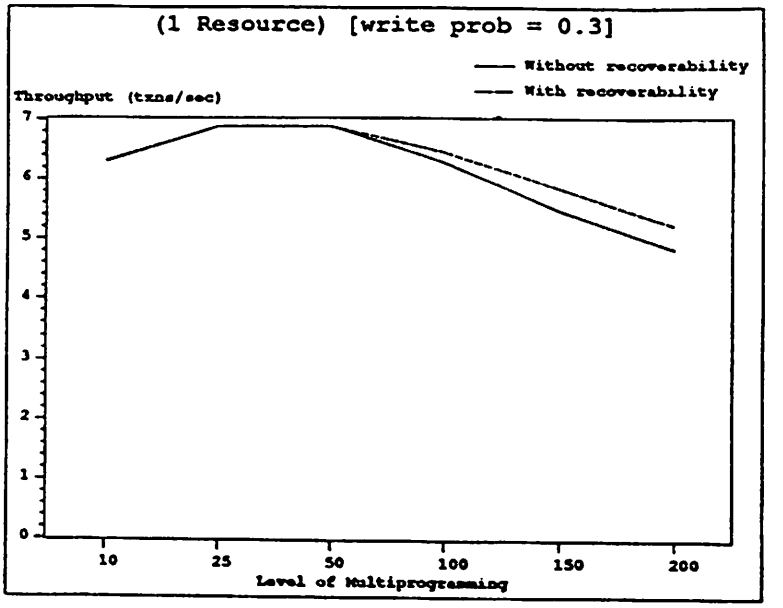

Figure 13: Throughput (1 resource unit)

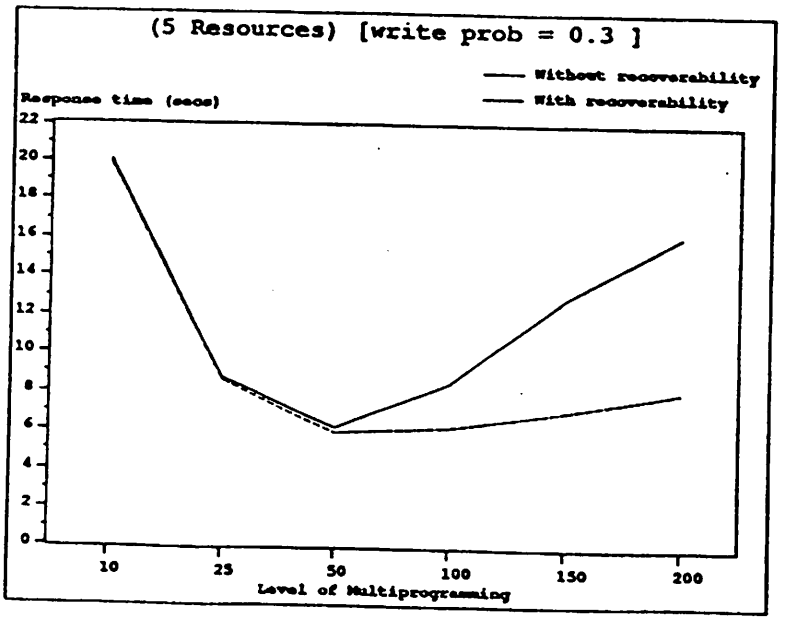

Figure 14: Response time (5 resource units)

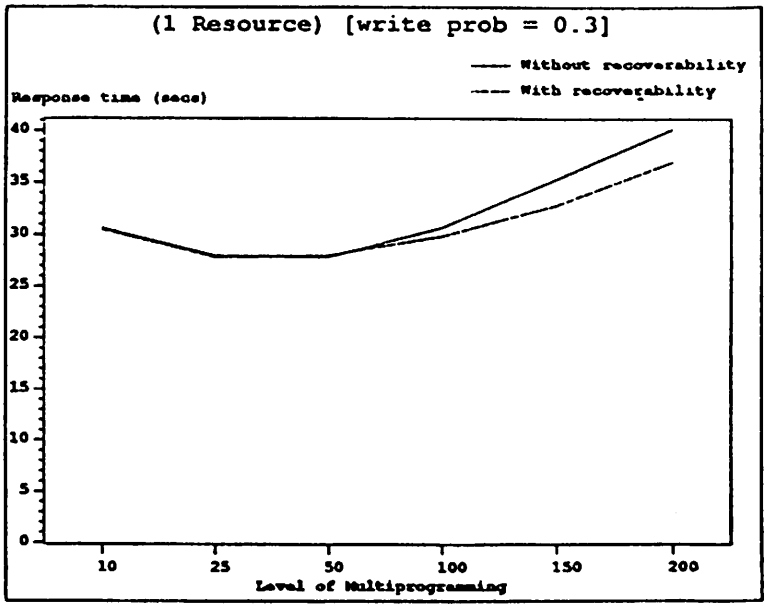

Figure 15: Response time (1 resource unit)

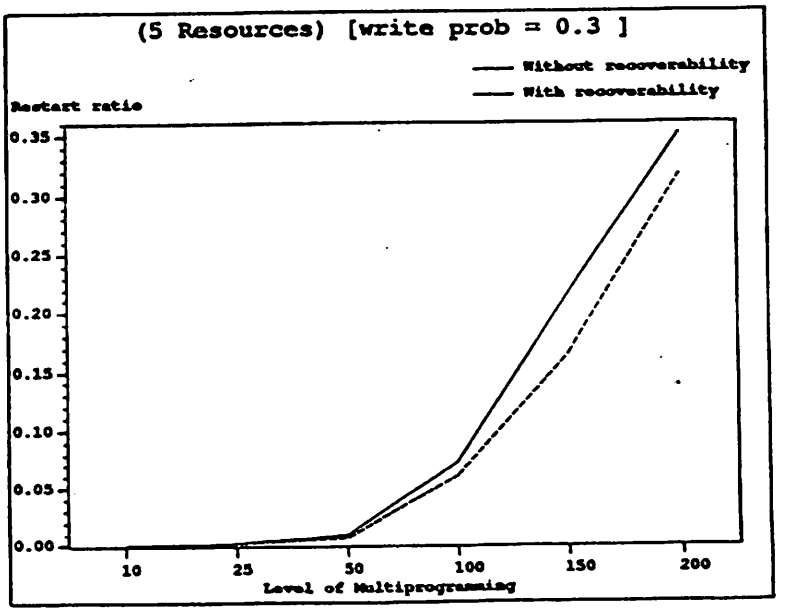

Figure 16: Restart ratio (5 resource units)

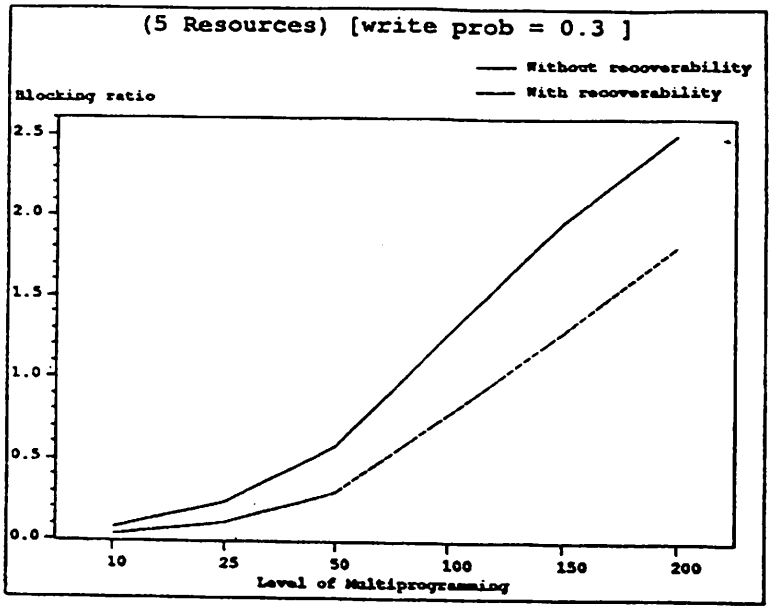

Figure 17: Blocking ratio (5 resource units)

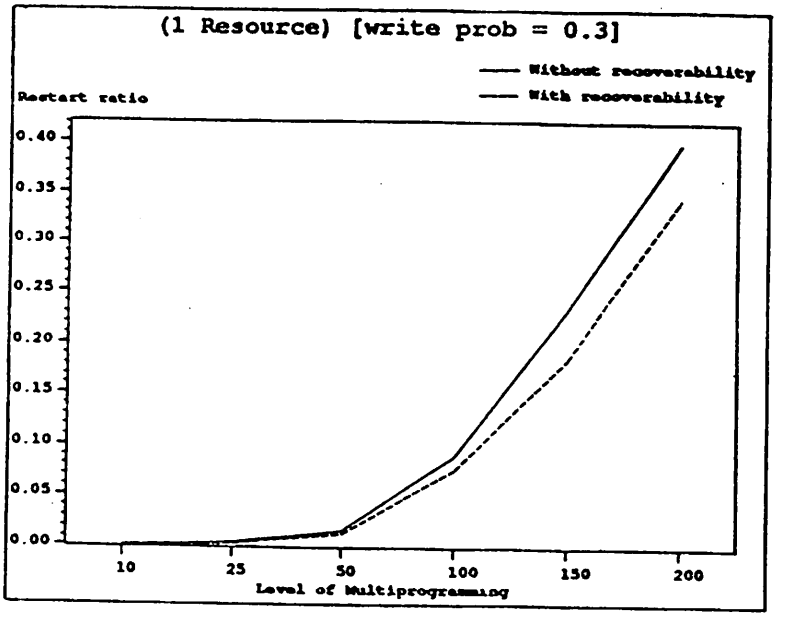

Figure 18: Restart ratio (1 resource unit)

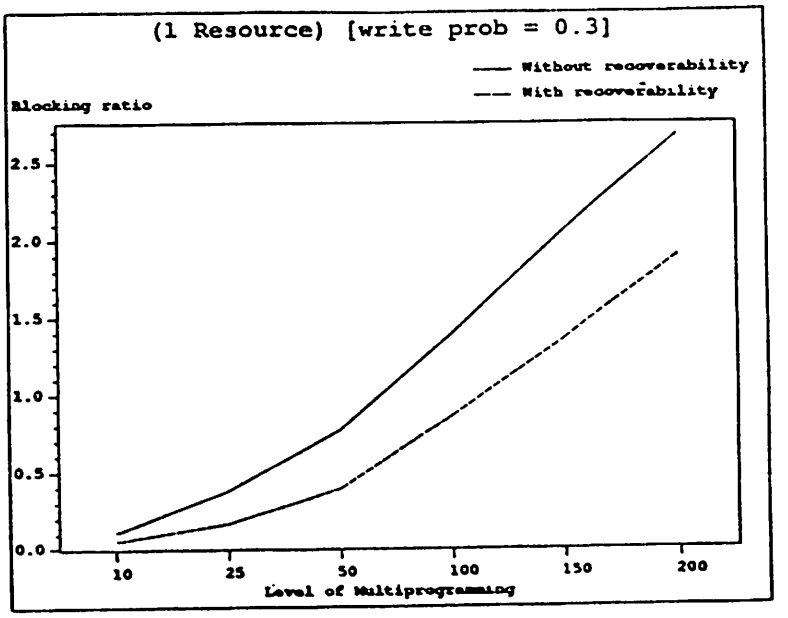

Figure 19: Blocking ratio (1 resource unit)

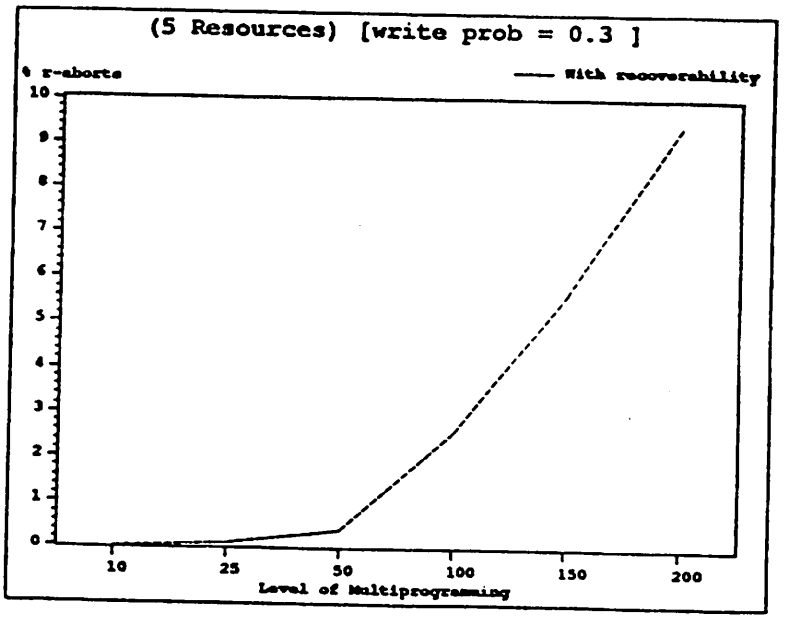

Figure 20: R-aborts (5 resource units)

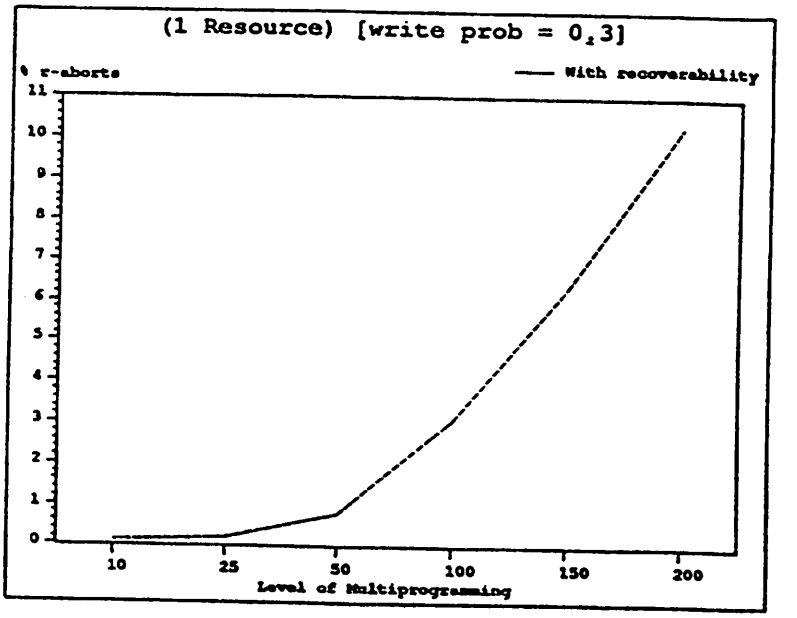

Figure 21: R-aborts (1 resource unit)

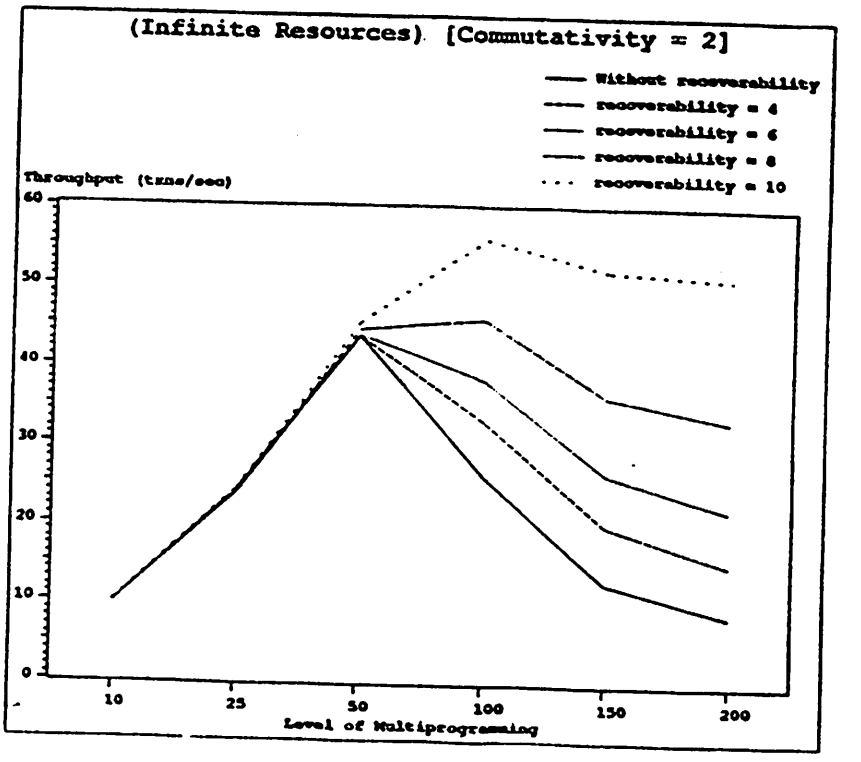

Figure 22: Throughput (infinite resources)

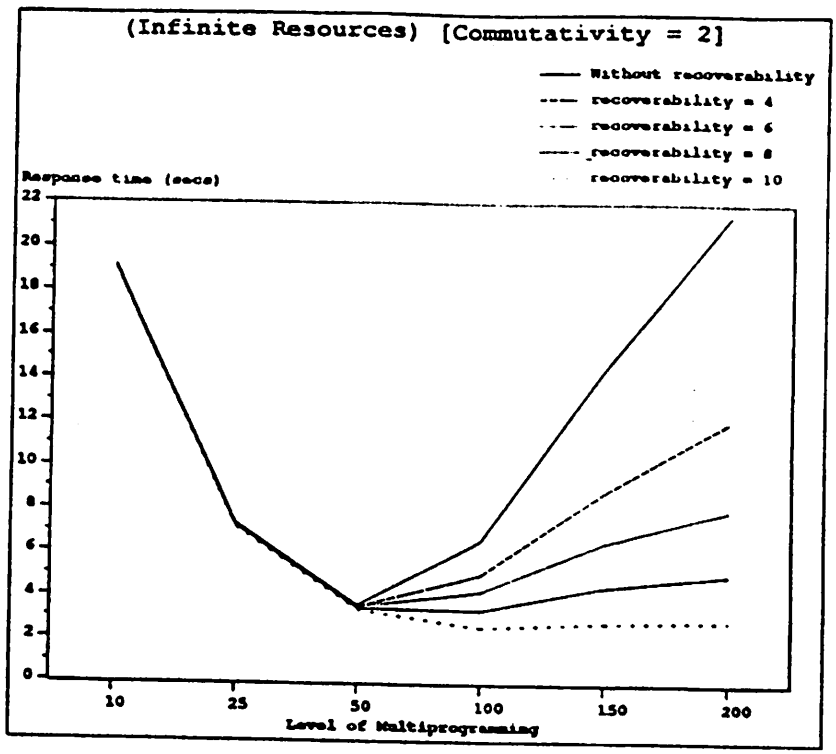

Figure 23: Response time (infinite resources)

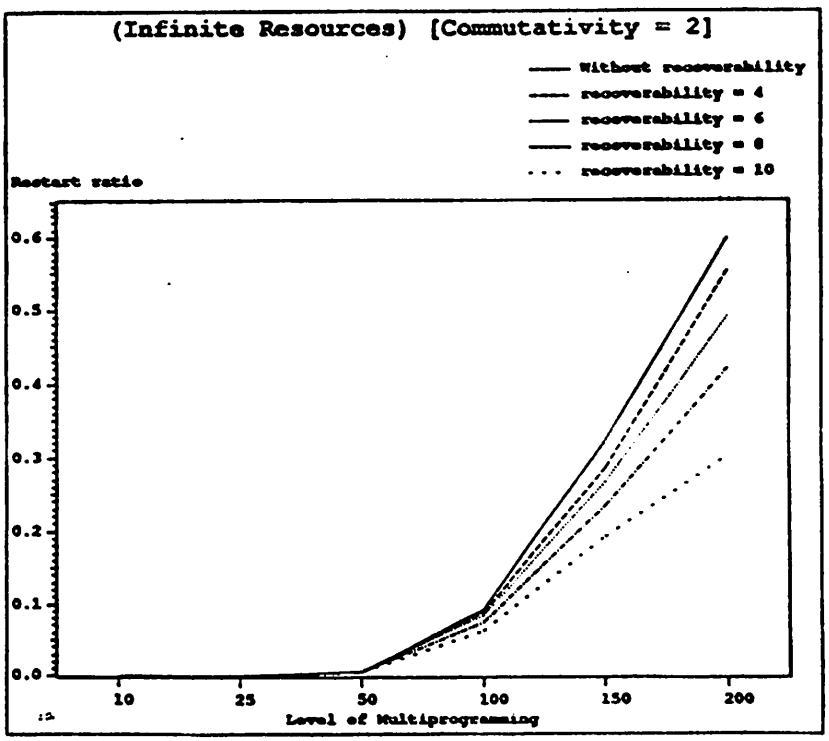

Figure 24: Restart ratio (infinite resources)

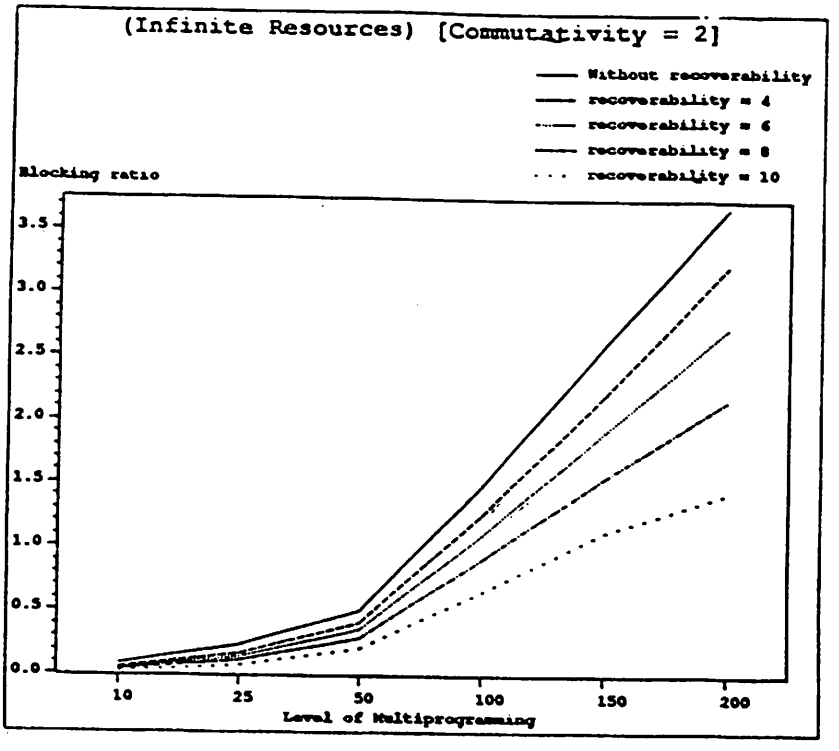

Figure 25: Blocking ratio (infinite resources)

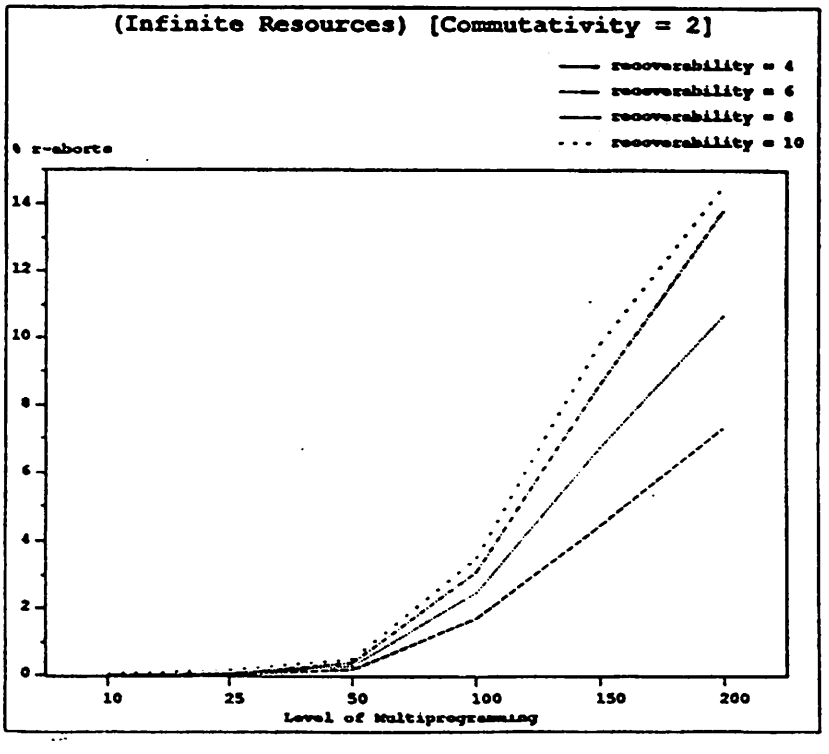

Figure 26: R-aborts (infinite resources)

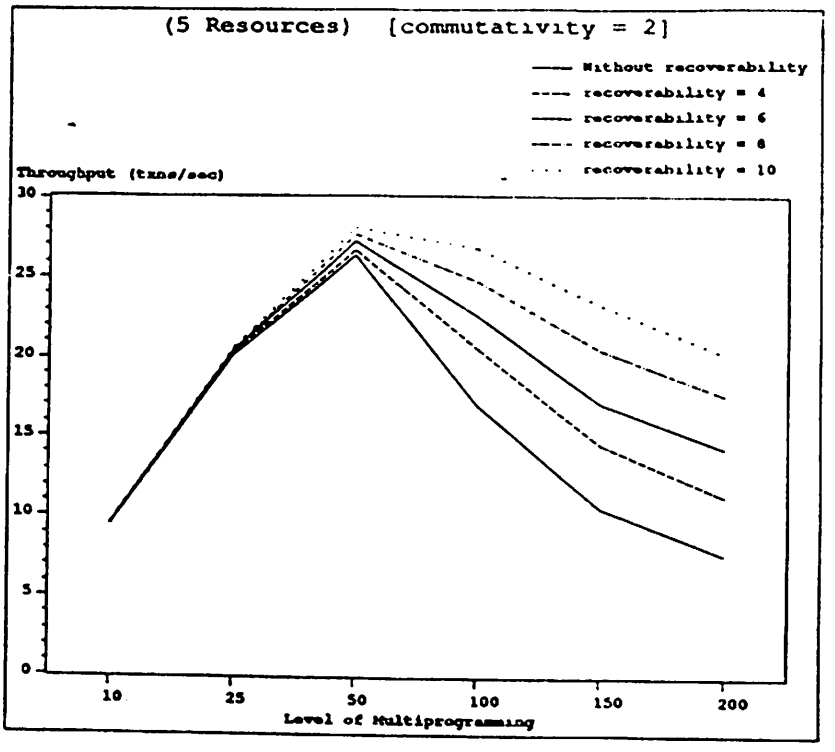

Figure 27: Throughput (5 resource units)

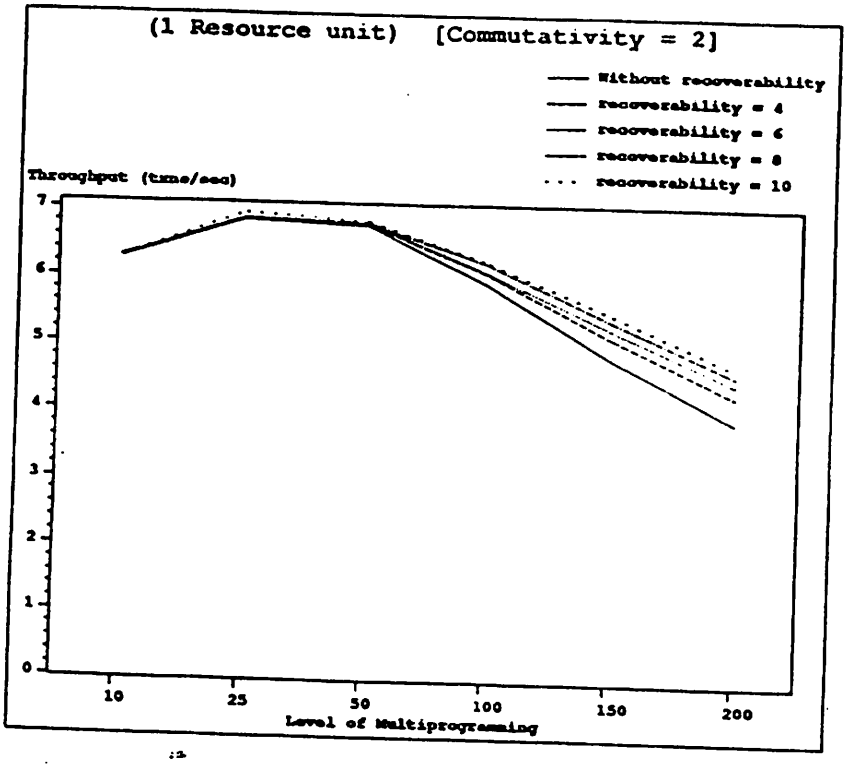

Figure 28: Throughput (1 resource unit)

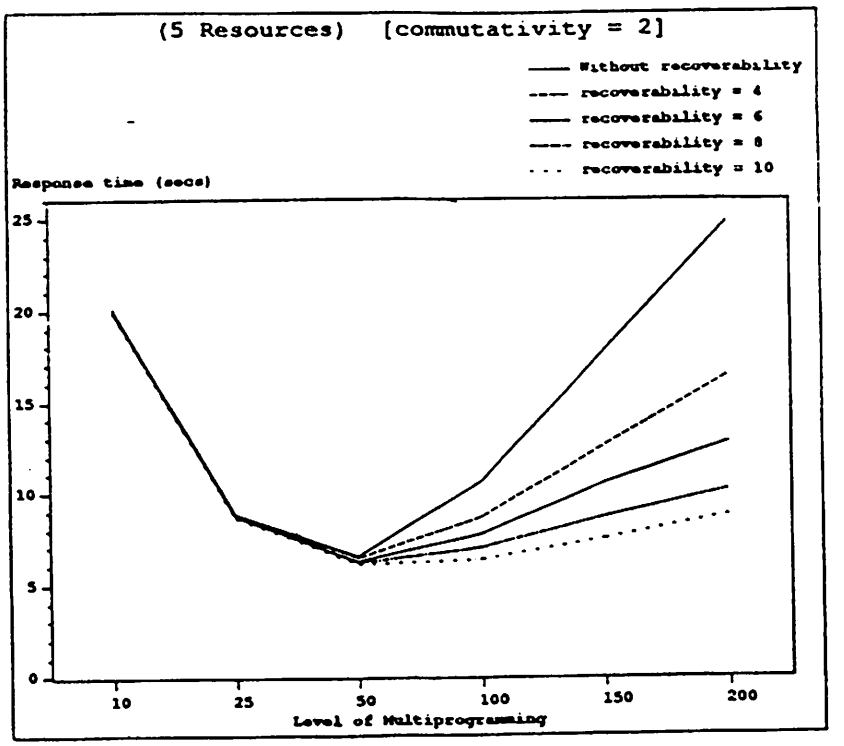

Figure 29: Response time (5 resource units)

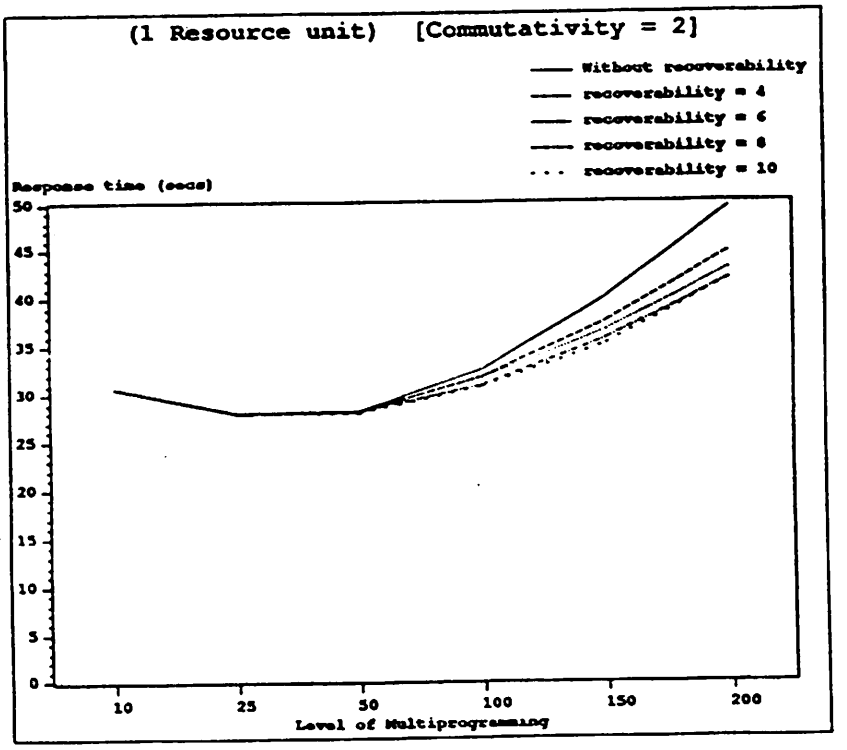

Figure 30: Response time (1 resource unit)

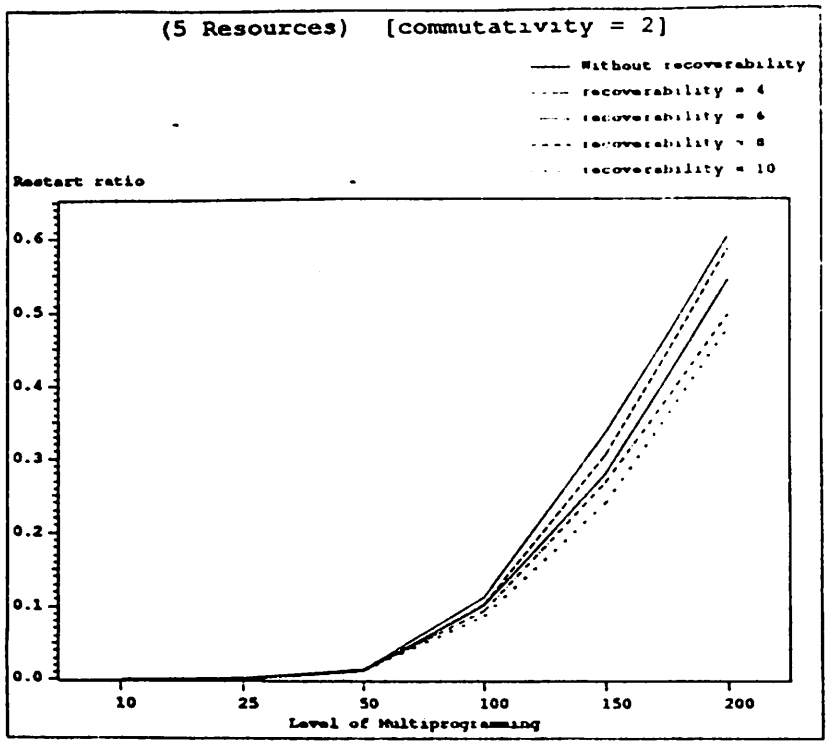

Figure 31: Restart ratio (5 resource units)

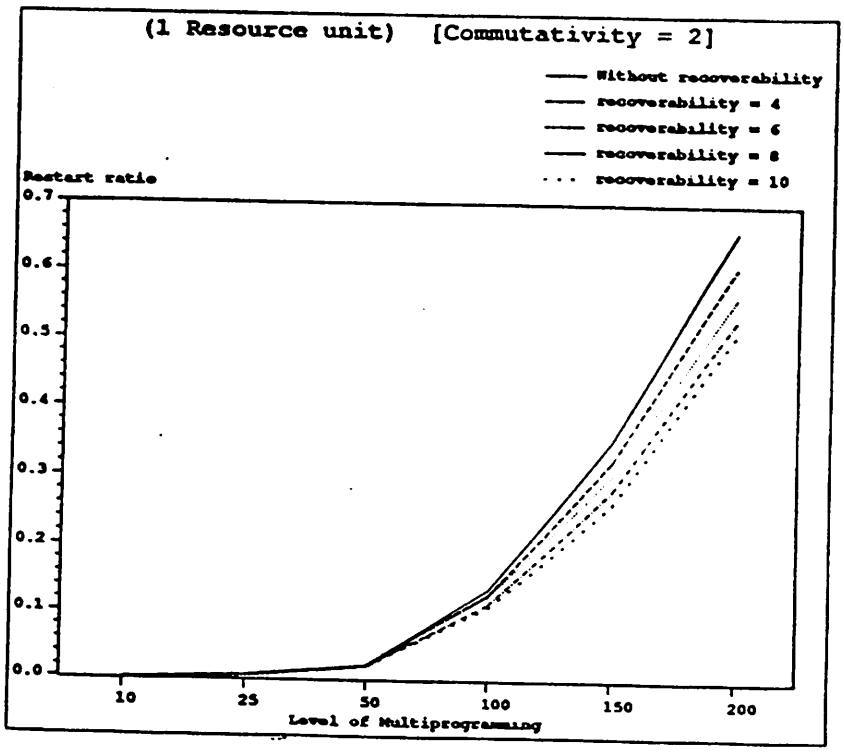

Figure 32: Restart ratio (1 resource unit)

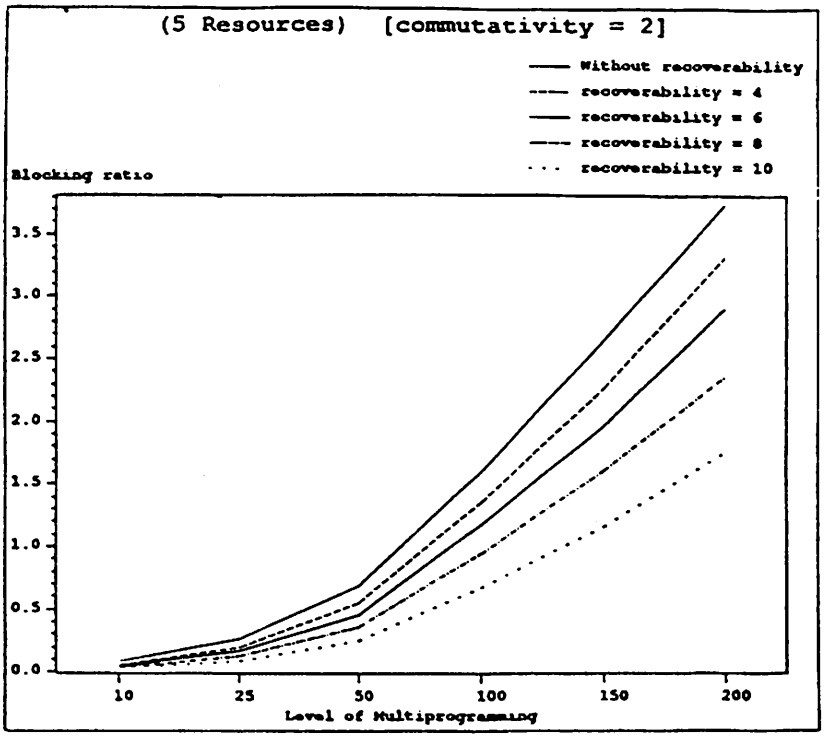

Figure 33: Blocking ratio (5 resource units)

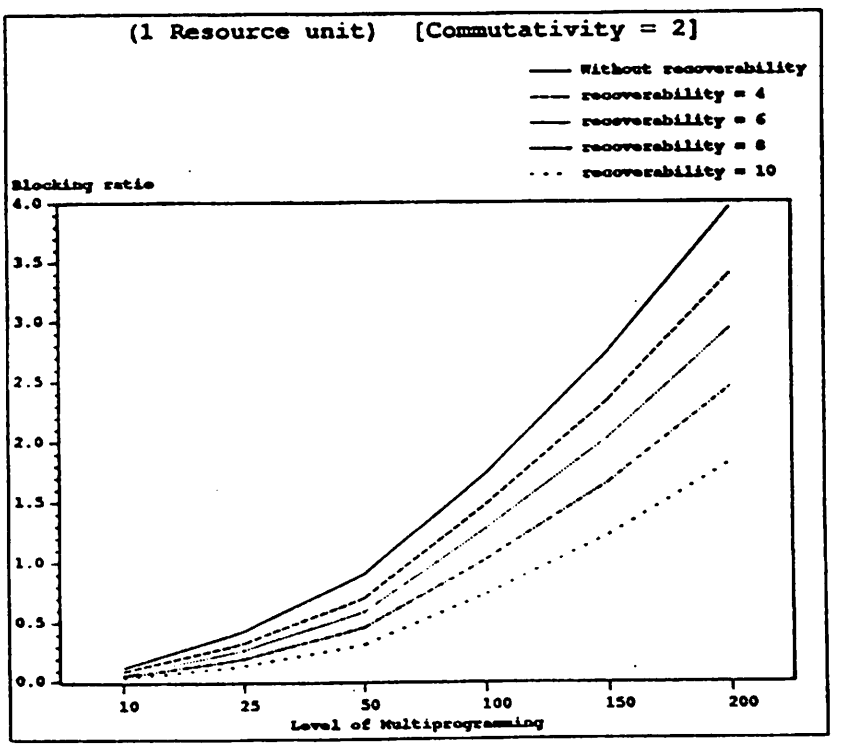

Figure 34: Blocking ratio (1 resource unit)

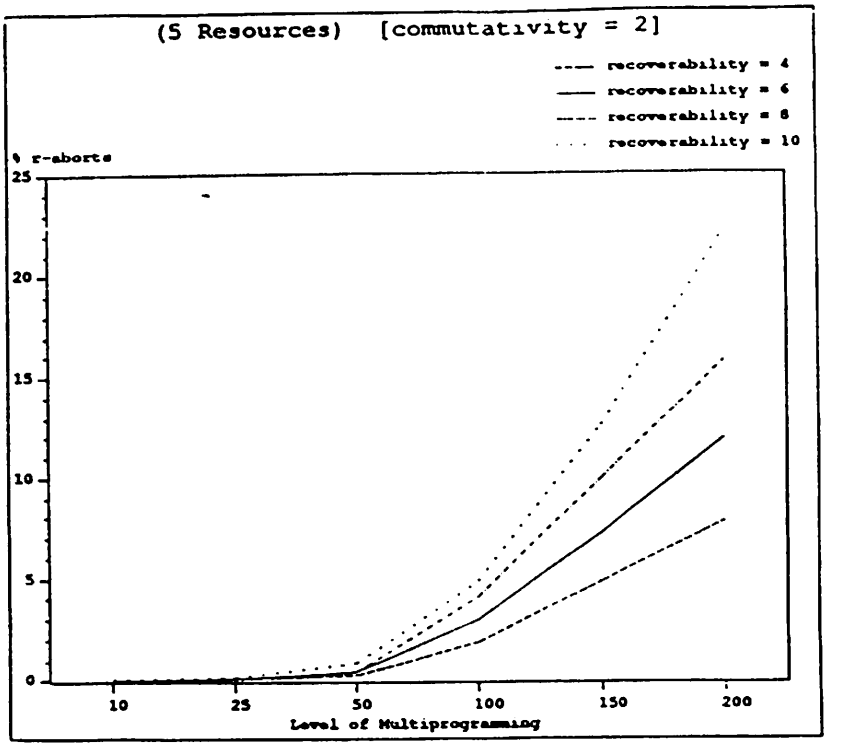

Figure 35: R-aborts (5 resource units)

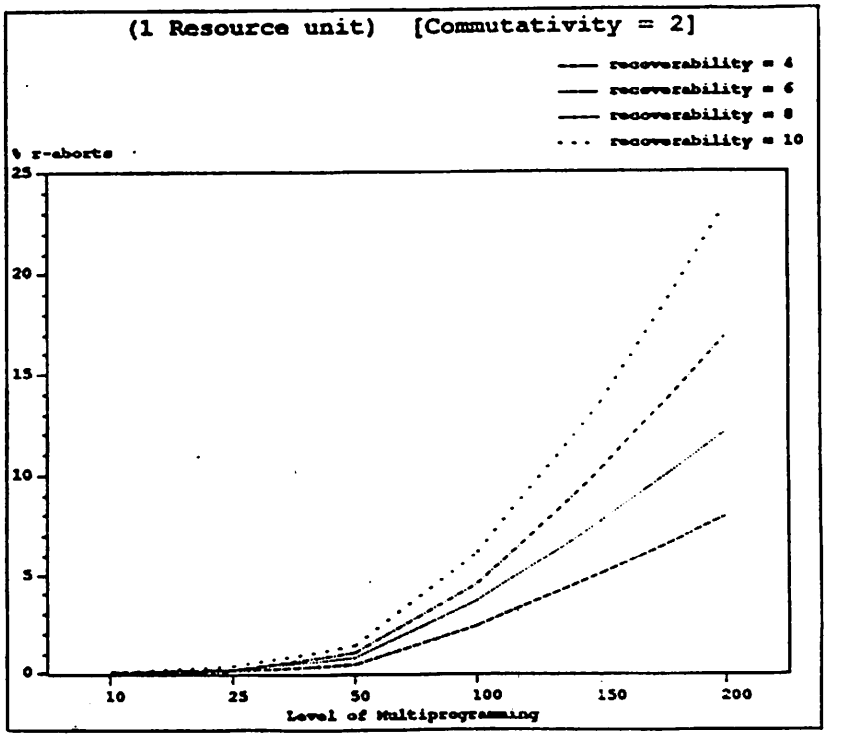

Figure 36: R-aborts (1 resource unit)

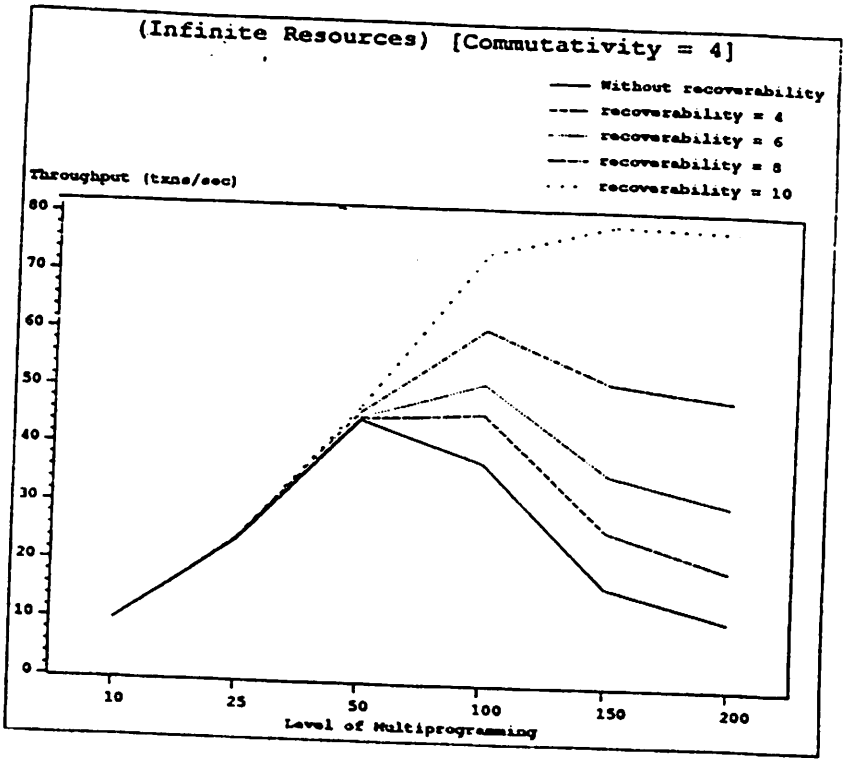

Figure 37: Throughput (infinite resources)

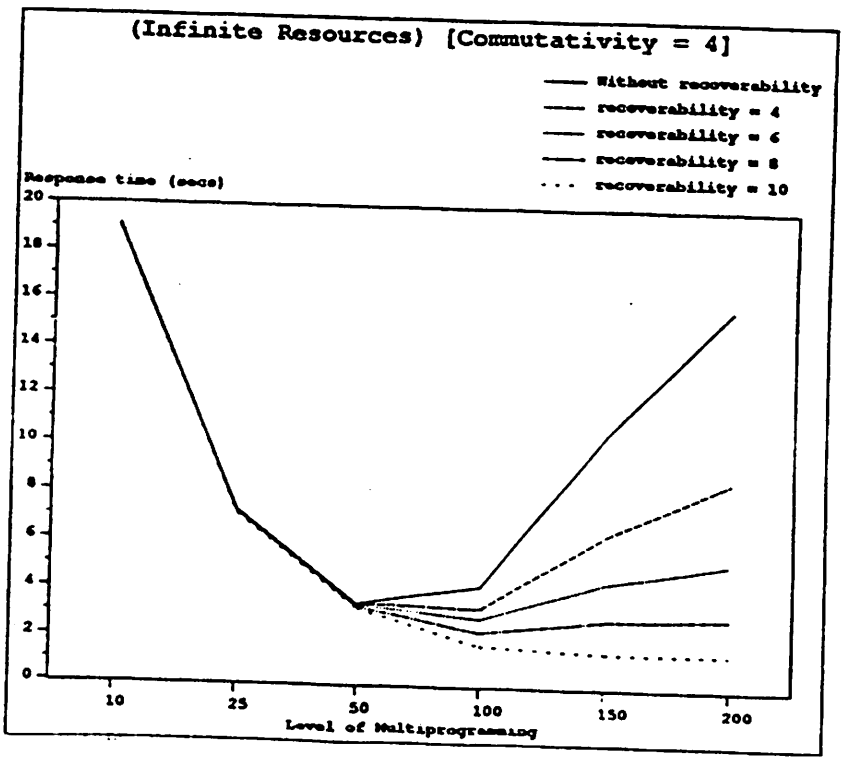

Figure 38: Response time (infinite resources)

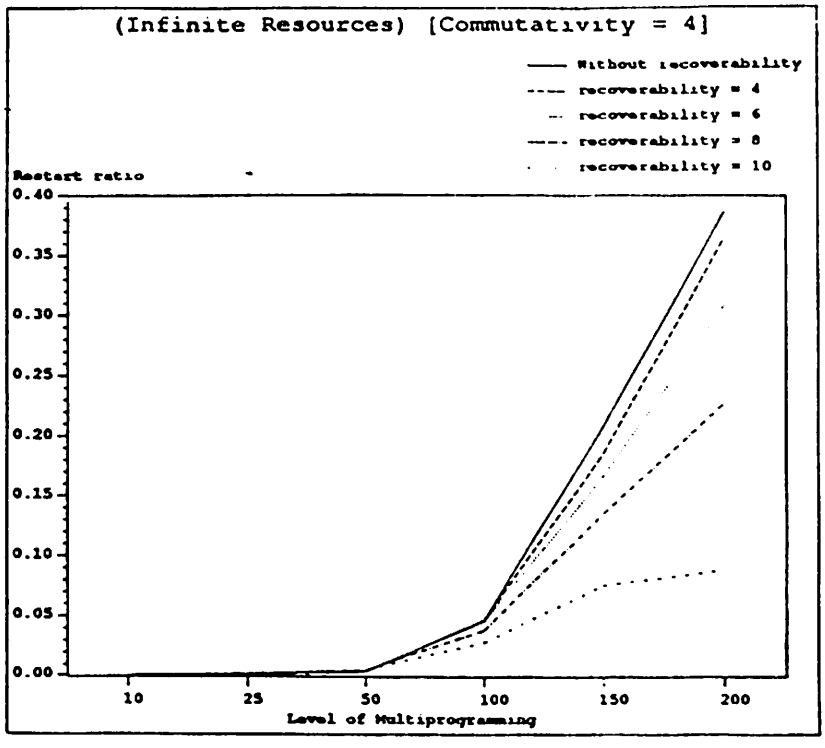

Figure 39: Restart ratio (infinite resources)

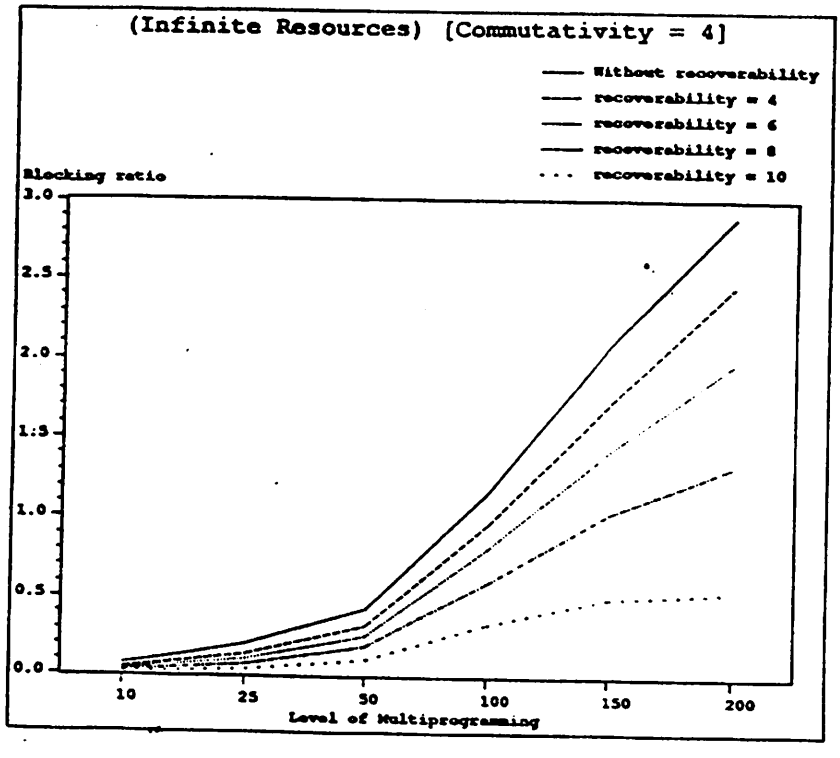

Figure 40: Blocking ratio (infinite resources)

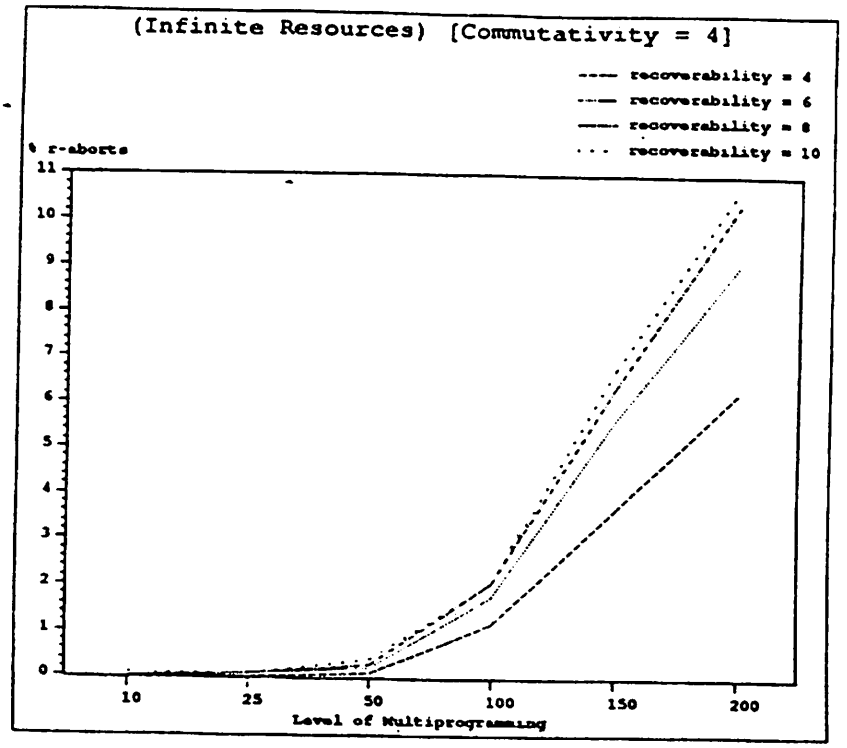

Figure 41: R-aborts (infinite resources)

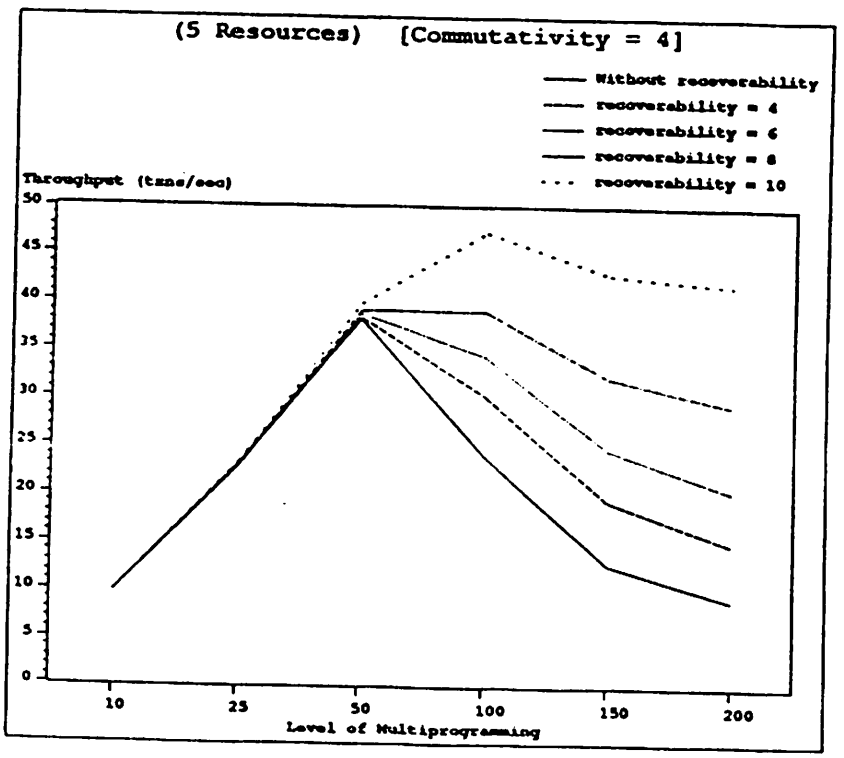

Figure 42: Throughput (5 resource units)

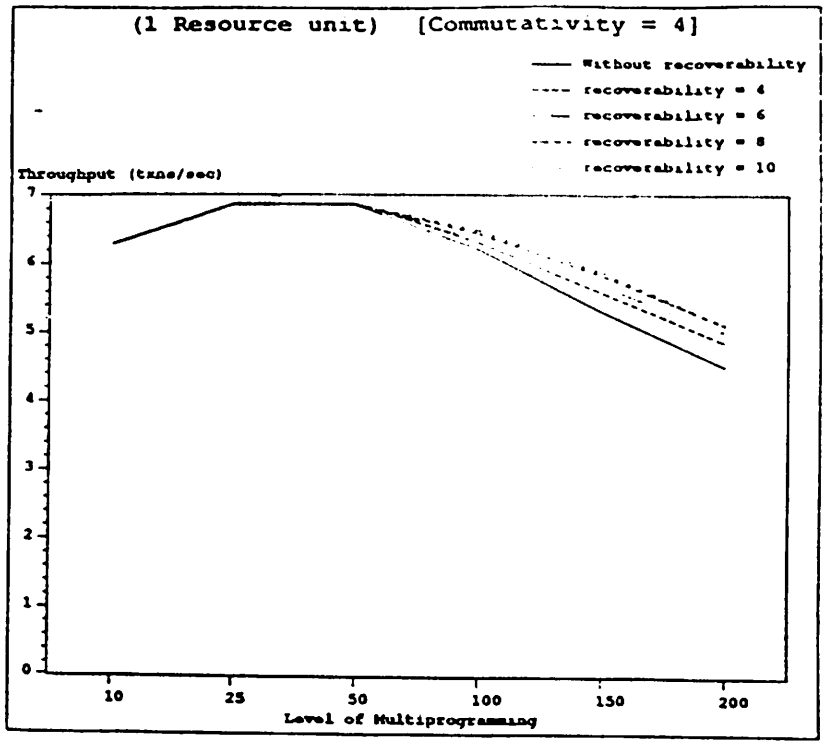

Figure 43: Throughput (1 resource unit)

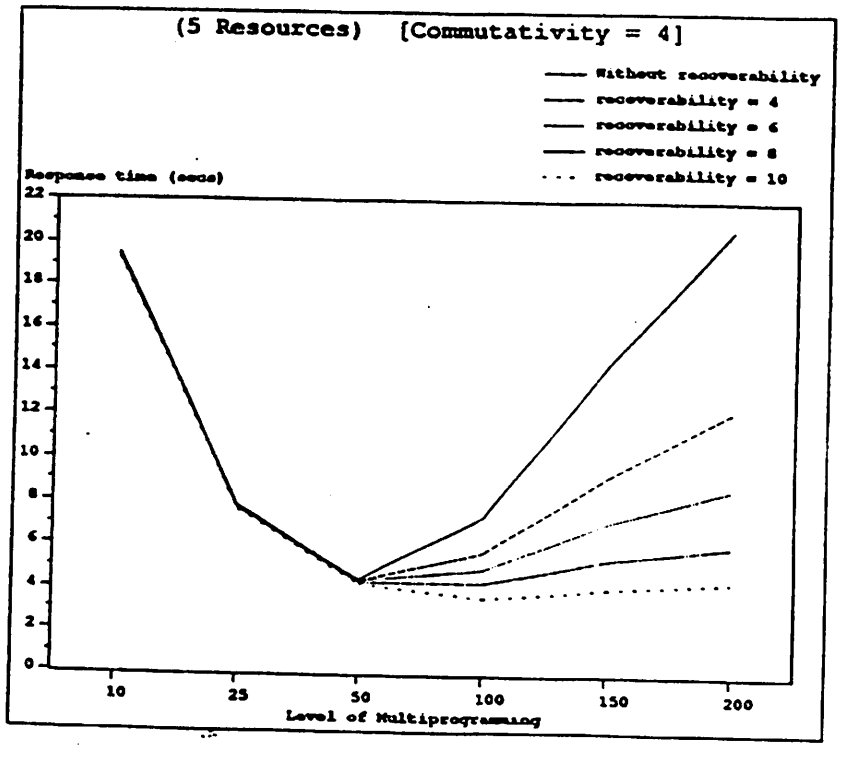

Figure 44: Response time (5 resource units)

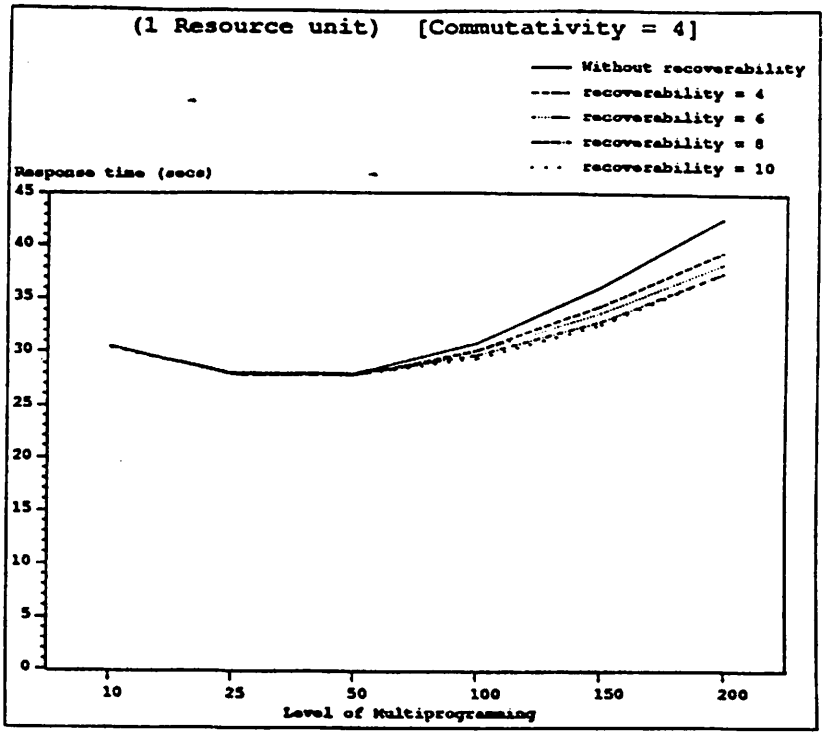

Figure 45: Response time (1 resource unit)

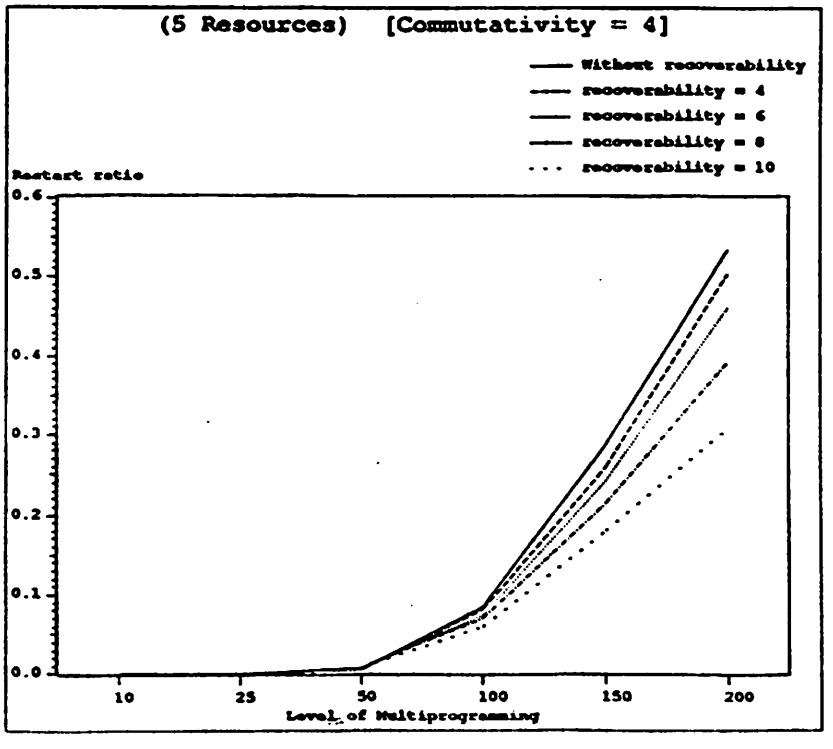

Figure 46: Restart ratio (5 resource units)

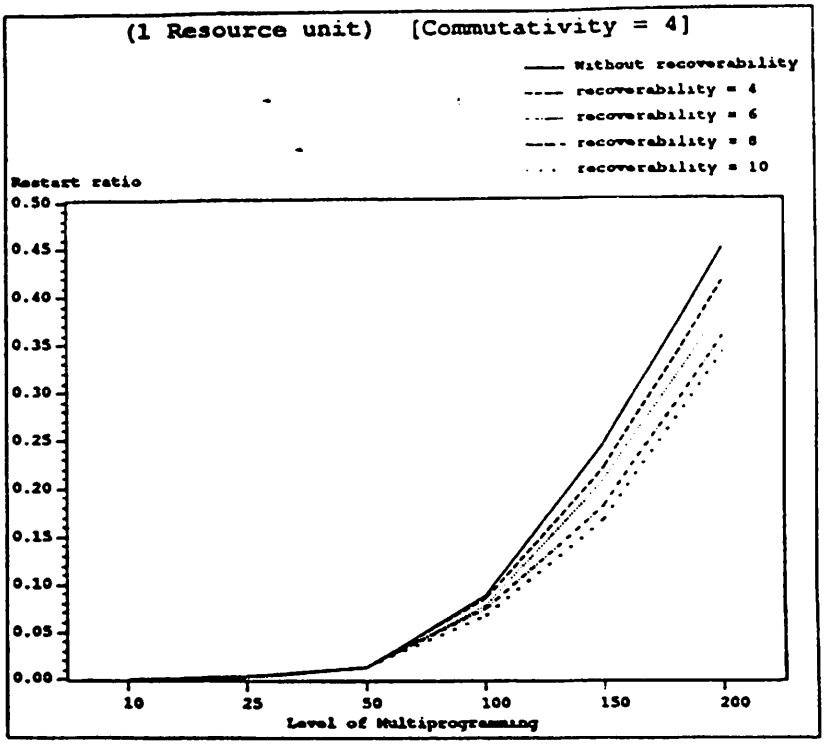

Figure 47: Restart ratio (1 resource unit)

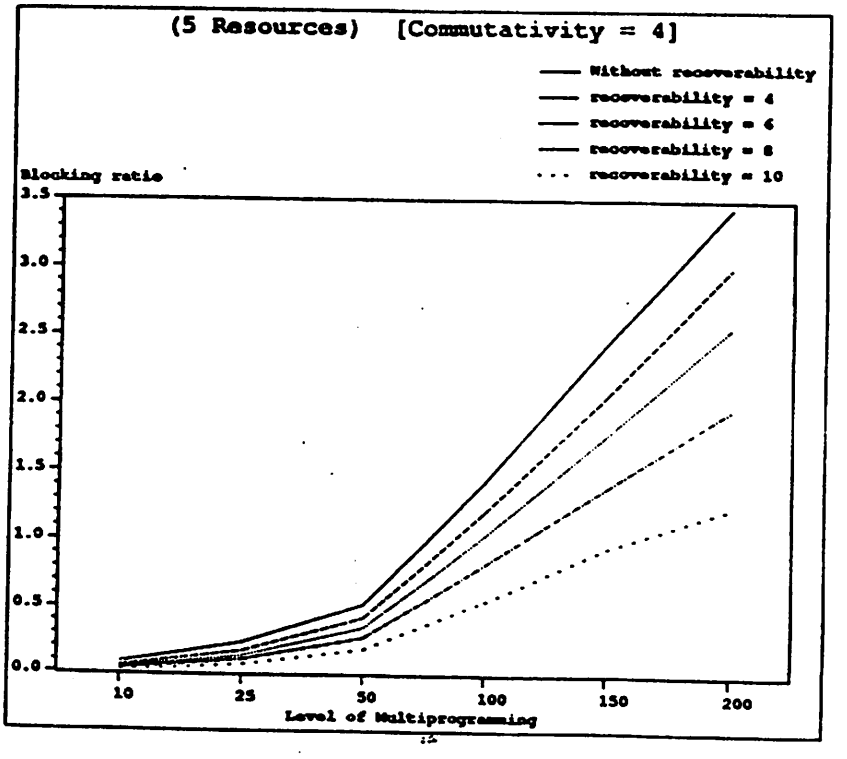

Figure 48: Blocking ratio (5 resource units)

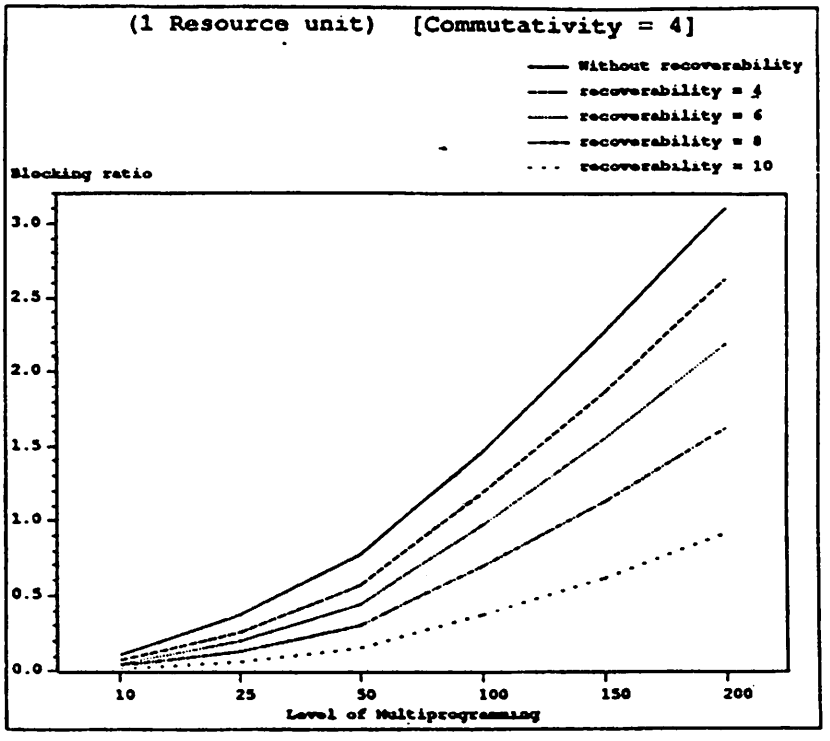

Figure 49: Blocking ratio (1 resource unit)

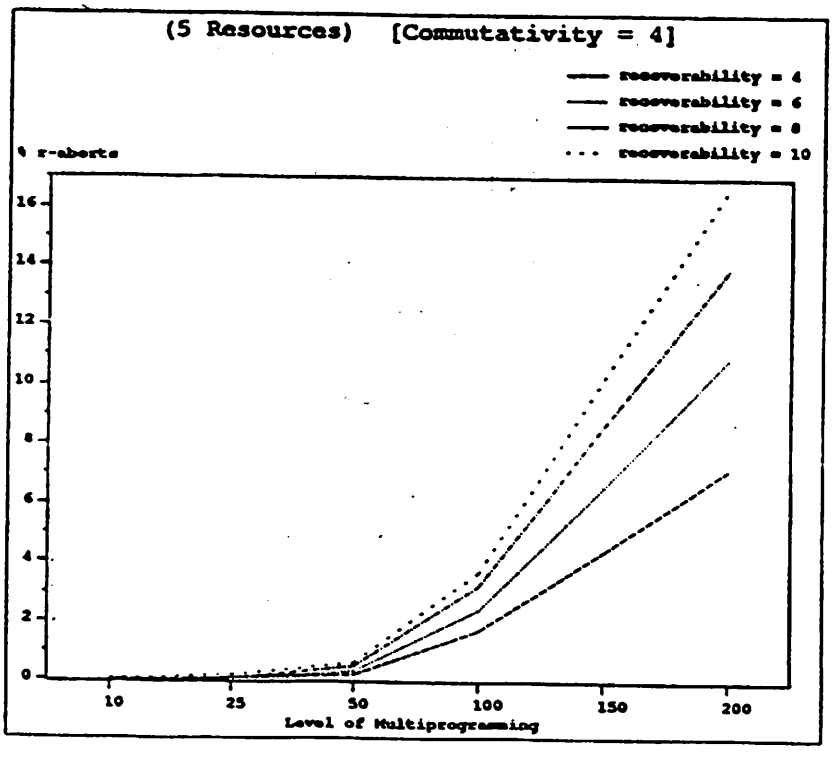

Figure 50: R-aborts (5 resource units)

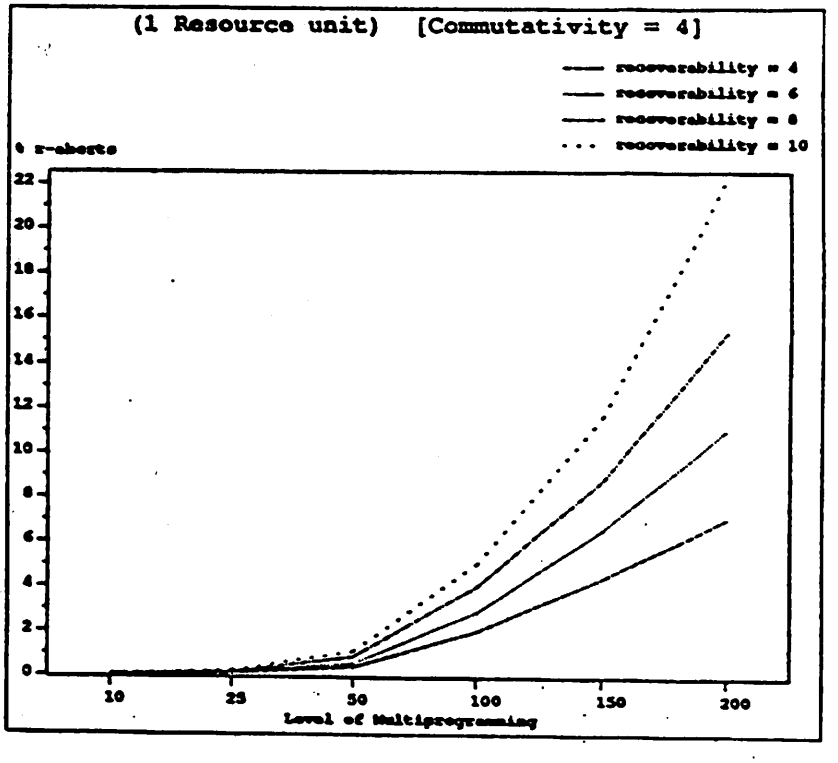

Figure 51: R-aborts (1 resource unit)