CONCURRENCY CONTROL IN COMPLEX INFORMATION SYSTEMS: A SEMANTICS-BASED APPROACH

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

COINS Technical Report 89-91

September 1989

CONCURRENCY CONTROL IN COMPLEX INFORMATION SYSTEMS: A SEMANTICS-BASED APPROACH

A Dissertation Presented

by

B. R. BADRINATH

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1989

Department of Computer and Information Science

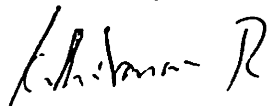
CONCURRENCY CONTROL IN COMPLEX INFORMATION SYSTEMS: A SEMANTICS-BASED APPROACH

A Dissertation Presented

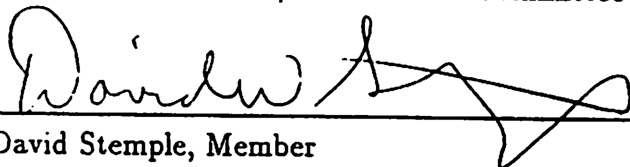
by

B. R. BADRINATH

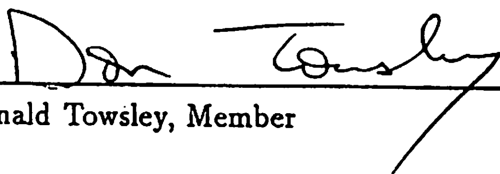
Approved as to style and content:



Krithi Ramamritham, Chairman of Committee

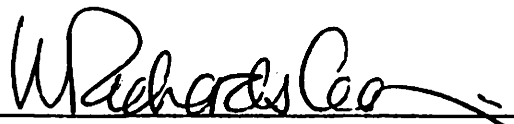


David Stemple, Member



Donald Towsley, Member

C. M. Krishna, Member



W. Richards Adrion, Department Head
Department of Computer and Information Science

ACKNOWLEDGEMENTS

I owe a lot to great many people who helped make this endeavor, which is reflected in this thesis, possible; the words here fail to express my gratitude to them.

My advisor, Krithi Ramamritham, has been a great source of encouragement, advice, and friendship. His keen interest, support under trying circumstances, and the utmost patience to read and reread every written piece I gave him, made this endeavor possible, and whatever clarity and elegance this thesis has is largely due to his efforts.

David Stemple, Don Towsley, and C Mani Krishna have been careful readers of this thesis and have made many helpful suggestions. I am deeply indebted to Jim Kurose and Jan Cuny; they have been an inspiration by being excellent in both teaching and research.

The people in Room 310B — Chia, Panos, Victor, Doug, Goran were always there to absorb my joys, my tribulations, my comments, and my sense of 'humor'. To them I am both sorry and thankful. Zhao Wei set an example by his hard work, and I learnt a great deal from the many discussions we had. Rahul gave me tips for giving presentations, though I never did satisfy his wishes of getting drunk. Lory's ladraw was of great help, and he introduced me to some good music. Duanne, intellectual (sharad), Vj, and many other friends at UMASS helped make graduate student life enjoyable. And to all my friends, who asked "When are you getting done?" and those who never doubted that I would finish, a million thanks!

To Renee for her help in getting much of the paper work needed during graduate studies and for being helpful and caring. To Betty for sending out all the letters and papers and for the help to our group.

Finally, I would like to thank Raghu for the confidence in me and being there when I needed talking out, and my parents for everything.

ABSTRACT

CONCURRENCY CONTROL IN COMPLEX INFORMATION SYSTEMS:
A SEMANTICS-BASED APPROACH

SEPTEMBER 1989

B. R., BADRINATH, B.E., BANGALORE UNIVERSITY
M.E., INDIAN INSTITUTE OF SCIENCE, BANGALORE
Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Krithi Ramamritham

Concurrency control techniques adopted in next generation information banks should be able to handle high level operations on arbitrary objects with complex structure. Unfortunately, current approaches to concurrency control are highly inefficient for these types of objects. This dissertation provides new techniques for high performance semantics-based concurrency control schemes in *complex information systems.* Our techniques make use of the available semantic information in scheduling operations.

Using the semantics of the operations, we define a new notion of conflict, weaker than commutativity, called *recoverability.* We present a concurrency control scheme that uses recoverability, and also present a multilevel concurrency control protocol that uses recoverability. In addition, an analysis of the structure of the high level operations is used define the notion of *relative conflicts*; it is shown that multilevel concurrency protocols using relative conflict as a basis for scheduling operations produce high gains in concurrency. We verify the correctness of the protocols presented using a multilevel graph model to show *semantic serializability.* The performance of the protocols, based on semantic serializability, is determined by conducting extensive simulation studies.

By taking advantage of semantic information that is available in complex information systems, in a general and systematic way, our concurrency control protocols can achieve higher concurrency than currently available methods.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . v

ABSTRACT . . . vi

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

CHAPTER

1. INTRODUCTION . . . 1
   - 1.1 Information Systems: Some Evolving Trends . . . 1
   - 1.2 The Concurrency Control Problem . . . 4
   - 1.3 Contributions of This Work . . . 5
     - 1.3.1 Multilevel Graph Model . . . 5
     - 1.3.2 Recoverability as a Weaker Notion of Conflict . . . 5
     - 1.3.3 Relative Conflict for Scheduling Complex Operations . . . 6
   - 1.4 Outline of the Dissertation . . . 6

2. RELATED WORK . . . 8
   - 2.1 Concurrency Control . . . 8
   - 2.2 Projects Involving Extensible Databases . . . 13

3. MULTILEVEL GRAPH MODEL . . . 16
   - 3.1 Formal Aspects . . . 18
     - 3.1.1 Proving Semantic Serializability . . . 22
     - 3.1.2 Application to Implicit Nested Computations . . . 22

3.2 Main Features 23
3.3 Conclusions 24

4. THE NOTION OF RECOVERABILITY 25
4.1 Introduction 25
4.2 A Formal Definition of Recoverability 26
4.2.1 Operations and Recoverable Operations 26
4.2.2 Examples 29
4.3 A Concurrency Control and Commit Protocol 34
4.3.1 Correctness Requirements and Commit Dependency Graph . . 35
4.3.2 A Two Phase Algorithm for Pseudo-committing Transactions 37
4.3.3 Committing Pseudo-committed Transactions 44
4.4 Results of Simulation Studies 45
4.4.1 The Simulation Model 46
4.4.2 Experiment Information 49
4.4.3 Performance Settings 49
4.4.4 Performance Metrics 50
4.4.5 Simulation Results 51
4.4.6 Summary of Simulation Results 56
4.5 Conclusions 57

5. MULTILEVEL CONCURRENCY CONTROL 82
5.1 Need For Multilevel Concurrency Control Protocols 82
5.2 A Recoverability Based Multilevel Concurrency Control Protocol . 84
5.3 Exploiting Operation Structure in ML Concurrency Control 87
5.4 Simulation Studies 93
5.4.1 The Simulation Model 94
5.4.2 Experiment Information 97

5.4.3 Performance Settings 99

5.4.4 Performance Metrics 100

5.4.5 Simulation Results 101

5.4.6 Summary of Simulation Results 104

5.5 Conclusions 105

6. CORRECTNESS OF MULTILEVEL PROTOCOL 112

6.1 Locks and Lists 112

6.2 Correctness 114

7. CONCLUSIONS AND FUTURE WORK 116

7.1 Contributions of This Research 116

7.1.1 Multilevel Graph Model 116

7.1.2 Recoverability as a Weaker Notion of Conflict 116

7.1.3 Relative Conflict for Scheduling Complex Operations 117

7.2 Beyond the Dissertation 117

BIBLIOGRAPHY 120

LIST OF TABLES

4.1 Commutativity for page 30

4.2 Recoverability for page 30

4.3 Commutativity for stack 31

4.4 Recoverability for stack 31

4.5 Commutativity for set 31

4.6 Recoverability for set 31

4.7 Commutativity for table 32

4.8 Recoverability for table 32

4.9 Stepwise execution of pseudo-commit algorithm 40

4.10 Simulation parameters 48

4.11 Parameters and their nominal values 50

5.1 Conflict table for counter 89

5.2 Relative conflict for counter 89

5.3 Simulation parameters 96

5.4 Conflict table for records 98

5.5 Relative conflict for pages 98

5.6 Parameters and their nominal values 100

5.7 Commutativity for read/write 102

5.8 Recoverability for read/write 102

LIST OF FIGURES

2.1 A taxonomy of semantics-based concurrency control schemes 9
3.1 Condensation of a multilevel computation graph 17
4.1 A dependency graph 37
4.2 Algorithm to insert commit dependency edges 37
4.3 Pseudo code for the the commit algorithm 39
4.4 Dependency graphs at three objects 40
4.5 Dependency graphs at objects P, Q, R, and S 45
4.6 Simulation model 47
4.7 Throughput (infinite resources) 59
4.8 Response time (infinite resources) 59
4.9 Restart ratio (infinite resources) 60
4.10 Blocking ratio (infinite resources) 60
4.11 R-aborts (infinite resources) 61
4.12 Throughput (5 resource units) 61
4.13 Throughput (1 resource unit) 62
4.14 Response time (5 resource units) 62
4.15 Response time (1 resource unit) 63
4.16 Restart ratio (5 resource units) 63
4.17 Blocking ratio (5 resource units) 64
4.18 Restart ratio (1 resource unit) 64
4.19 Blocking ratio (1 resource unit) 65
4.20 R-aborts (5 resource units) 65

4.21 R-aborts (1 resource unit) 66
4.22 Throughput (infinite resources) 66
4.23 Response time (infinite resources) 67
4.24 Restart ratio (infinite resources) 67
4.25 Blocking ratio (infinite resources) 68
4.26 R-aborts (infinite resources) 68
4.27 Throughput (5 resource units) 69
4.28 Throughput (1 resource unit) 69
4.29 Response time (5 resource units) 70
4.30 Response time (1 resource unit) 70
4.31 Restart ratio (5 resource units) 71
4.32 Restart ratio (1 resource unit) 71
4.33 Blocking ratio (5 resource units) 72
4.34 Blocking ratio (1 resource unit) 72
4.35 R-aborts (5 resource units) 73
4.36 R-aborts (1 resource unit) 73
4.37 Throughput (infinite resources) 74
4.38 Response time (infinite resources) 74
4.39 Restart ratio (infinite resources) 75
4.40 Blocking ratio (infinite resources) 75
4.41 R-aborts (infinite resources) 76
4.42 Throughput (5 resource units) 76
4.43 Throughput (1 resource unit) 77
4.44 Response time (5 resource units) 77
4.45 Response time (1 resource unit) 78
4.46 Restart ratio (5 resource units) 78
4.47 Restart ratio (1 resource unit) 79

4.48 Blocking ratio (5 resource units) 79
4.49 Blocking ratio (1 resource unit) 80
4.50 R-aborts (5 resource units) 80
4.51 R-aborts (1 resource unit) 81
5.1 Multilevel computation 83
5.2 Classification of Operations 90
5.3 Nested protocol ML-RC 92
5.4 Three level system 94
5.5 Simulation model 95
5.6 Throughput (infinite resources) 106
5.7 Response time (infinite resources) 106
5.8 Restart ratios (infinite resources) 107
5.9 Blocking ratios (infinite resources) 107
5.10 Throughput (5 resource units) 108
5.11 Throughput (1 resource unit) 108
5.12 Response time (5 resource units) 109
5.13 Response time (1 resource unit) 109
5.14 Restart ratios (5 resource units) 110
5.15 Restart ratios (1 resource unit) 110
5.16 Blocking ratios (5 resource units) 111
5.17 R-aborts (1 resource unit) 111
6.1 Example of multilevel protocol 113

CHAPTER 1

INTRODUCTION

1.1 Information Systems: Some Evolving Trends

Until recently, the role of information systems was confined to data-intensive applications and record-oriented operations. In recent years, the role of databases has expanded to encompass application requirements that go beyond those imposed by typical business applications. Though the relational data model is preferred for data processing applications, the increasing complexity of applications and throughput requirements demand new data models, storage techniques, and access methods. These new features being developed are natural candidates for applying the object-oriented paradigm. Thus, many object-oriented databases are being developed to meet data handling needs of new applications.

These *complex information systems* based on varying object-oriented data models support a rich set of operations and objects. The operations are not necessarily confined to *reads* and *writes*, and the objects are not necessarily confined to *tuples* and *records*. The next generation information systems require concurrency control mechanisms to handle high level operations on arbitrary objects with complex structure. Unfortunately, traditional approaches to concurrency control are highly inefficient for these types of databases [62, 69, 75]. The problem of efficiently performing atomic updates, i.e., transaction processing, in complex information systems given the new characteristics of the data model and increasing demands on throughput, is an important and challenging research problem.

Examples of applications requiring the support of complex information systems include office information systems (OIS), stock trading databases, AI, software engineering, CAD/VLSI, real-time systems and distributed operating systems. Most of these applications are complex, require high performance and are not well supported by existing database management systems where the operations are just *reads* and *writes* on *uniform* sets of data like *tuples* and *records*.

In the context of our research, we define complex information systems as those systems supporting one or more of the following features:

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

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

3) *Compositional Objects* — These systems are typically object bases, i.e., designed in terms of objects, using an object-oriented paradigm. Object bases support two important features:

   a) *Object Hierarchies*— In object bases, objects are instances of a class and these classes are organized as a hierarchy or lattice. Operations on a class structure are translated into operations on classes and sub-classes. Further, updates are not only on instances of objects but also on class objects. A change or retrieval operation may thus propagate to many levels in the class hierarchy.

   b) *Complex objects*— A given object, might consist of components that are themselves objects; a specific relationship such as PART-OF, and SET-OF exists between the components and the object. In object bases, we have the ability to manipulate a complex object as an entity, and thus any operation executed on the object will have operations executing on sub-objects as well.

The distinguishing feature of computations in complex information systems with the above capabilities is their *multilevel* nature. We have operations at multiple levels because: 1) complex operations are decomposed into sub-operations, 2) objects are grouped under classes and sub-classes, and 3) complex objects are composed of objects and sub-objects. In contrast, flat single level computations occur in traditional databases. In all the above cases, nesting of operations is provided internally as a system facility; we call this feature *implicit nesting*. On the other hand, there are systems where nesting is external and provided as a user facility in the form of nested transactions. We call this type of nesting *explicit nesting* [57]. Throughout this work, we will limit our discussion to implicit nesting environments provided as a system facility.

In the rest of this introduction, we motivate the need for new approaches to improving concurrency in complex information systems and how this thesis fills this need. As more applications require the use of databases, the sharing of data will become even more important. If efficient ways of sharing is to be provided to meet throughput requirements with arbitrary operations, efficient concurrency control schemes should be designed for these new object-oriented data models. This dissertation focuses on the development of *high performance* concurrency control algorithms that take advantage of the *semantics* available in the emerging information intensive applications.

The primary questions we answer in this research are:

- How do we approach the problem of concurrency control in complex information systems?

  In implicit nested environments, we have operations at various levels, and operations at each level have specific synchronization properties. Concurrency control at the transaction level alone, as we will see, is unnecessarily restrictive. Thus, a multilevel (ML) approach to the problem of concurrency control seems not only *natural* but also *necessary* to meet the high throughput demands in complex information systems.

- How should we develop high performance concurrency control algorithms ?

  The key to the development of efficient concurrency control algorithms lies in not only *reducing the number of conflicts* but also *the duration over which conflicts are considered*. In the approach we elucidate in this thesis, we exploit *semantics* available from the knowledge about the type of operations and the structure of complex operations in order to

  1) define a new notion of conflict which is weaker than commutativity.
  2) develop novel ways of scheduling concurrent requests in *implicit* nested environments to minimize the duration over which conflicts are considered.

- How can we prove the correctness of the proposed multilevel concurrency control protocols?

  In the protocols we have designed, concurrency is controlled at one or more levels. We need to to prove that computations that follow these protocols are correct. To this end, we have developed a formal model based on representing a computation as a multilevel graph along with a graph condensation procedure.

Using this model we verify the correctness of the proposed semantics-based concurrency control protocols.

- How should we evaluate the performance of the proposed algorithms?
  We have conducted extensive simulation studies in order to evaluate the performance of our semantics-based protocols. For purposes of comparison, we also measure the performance of other traditional (existing) methods under the same assumptions. The relative improvement in performance of semantics-based schemes is a yardstick measuring the success of the approach taken in this dissertation. Through these studies, we demonstrate that our approach dramatically increases the efficiency of transaction processing in complex information systems.

1.2 The Concurrency Control Problem

For concurrent executions of transactions, serializability is a well-accepted correctness criterion in databases. A large body of literature over the past decade has addressed the question of how to achieve serializability. Concurrency control algorithms, based on serializability, introduce latency: if a conflict is believed possible, they avoid interference between concurrent transactions by forcing one of them to wait while the other executes. Serializability provides the highest degree of concurrency when only syntactic information is used for scheduling [42]. Hence, to obtain a higher degree of concurrency, additional information, in addition to syntax information, must be used. Thus, our task will be to minimize the number of conflicts and the duration over which such conflicts are considered by exploiting semantic knowledge available in complex information systems, thereby avoiding excessive concurrency limitations.

Serializability in databases is defined in terms of the equivalent effect of computations. The standard approach is purely syntactic— features of the computation such as number of operations, view seen by the operations, and state changes are considered but the semantics of the operations are not considered. Further, histories are single level and any two conflicting operations are required to be ordered in the same way in equivalent histories. *Semantic serializability* is also based on the notion of equivalence. However, the definition of equivalence regards only some features of the execution as important. It is this ability to abstract certain features of the computation that has allowed various researchers to use semantic serializability for designing provably-correct concurrency control algorithms with enhanced concurrency [11, 12, 30, 54, 58, 82].

We adopt *semantic serializability* as our correctness criterion. Informally, the notion of semantic serializability as a correctness criterion for multilevel concurrent computations implies that a concurrent execution of a multilevel computation should be equivalent to a sequential execution of only top-level operations. The definition of conflicts includes the *semantics* of the operations. For example, two multilevel schedules can be shown equivalent even when the sub-operations have different orders of execution in the two schedules.

This dissertation focuses on the *development* of a formal framework in which semantic serializability can be analyzed, the *development* of semantics-based concurrency control schemes for complex information systems, and the *evaluation* of the proposed protocols. In general, our method uses the semantics available not only in abstract operations but also in the multilevel nature of computations in complex information systems.

1.3 Contributions of This Work

The work presented here describes several semantics-based concurrency control schemes, their correctness, and the overall improvement in concurrency from utilizing available semantics.

1.3.1 Multilevel Graph Model

We have developed a general model to prove the correctness of various semantics-based concurrency control schemes. This model is a natural extension of the single level serialization theory. This model allows the incorporation of operation semantics and levels of abstraction to ignore dependencies. Hence, concurrency control schemes that use these features can be proved correct in a formal setting.

1.3.2 Recoverability as a Weaker Notion of Conflict

In the past, commutativity of operations has been exploited to provide enhanced concurrency while avoiding cascading aborts. We have identified a new 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 significantly. Further, our studies show enhancement in concurrency even when *resource contention* is taken into consideration.

1.3.3 Relative Conflict for Scheduling Complex Operations

Many information systems are structured along levels of abstraction. The currently available single level concurrency control mechanisms for *reads* and *writes* are inadequate for future complex information systems[13, 62]. We present a new *multilevel* concurrency protocol that not only uses recoverability as a basis for determining conflicts but also uses a new way of scheduling operations based on *relative conflict*. Relative conflict uses information about the type of parent of an operation in determining whether two operations can be executed concurrently. In a ML nested protocol, if two operations are executed concurrently then there is a possibility of deadlock among sub-operations. However, scheduling operations according to relative conflict reduces the probability of deadlock among sub-operations. We show that the performance of this new protocol is significantly better than previously proposed protocols by comparing its performance with a multilevel protocol that uses only commutativity as a basis for conflicts. Performance is evaluated by conducting extensive simulation studies. We also study the implications of *resource contention* on this new multilevel concurrency protocol.

1.4 Outline of the Dissertation

Chapter 2 discusses related work in the area of concurrency control, and reviews the current status of databases that support various notions of object-orientation.

In chapter 3, we introduce a formal model to prove correctness of multilevel concurrent computations. This model is based on a graph theoretic notion of *condensation* applied to hierarchically constructed graphs.

Chapter 4 begins the study of operation semantics, defining a new notion of conflict called *recoverability*. This notion of conflict is weaker than commutativity and includes return values of operations in defining conflicts between operations. We describe in detail the results of various simulation studies conducted.

In Chapter 5 , we examine the idea of multilevel concurrency protocols aided by information about the structure of abstract operations. We present a protocol that schedules complex operations using a new concept called *relative conflict.* We enhance concurrency by including recoverability as a conflict predicate. Results of the simulation studies that show the significant improvement in concurrency obtained by using semantic information in multilevel protocols are presented.

Chapter 6 provides the correctness proof for our multilevel concurrency control protocol.

In Chapter 7 we review the results presented in this dissertation, and outline some suggestions for future work.

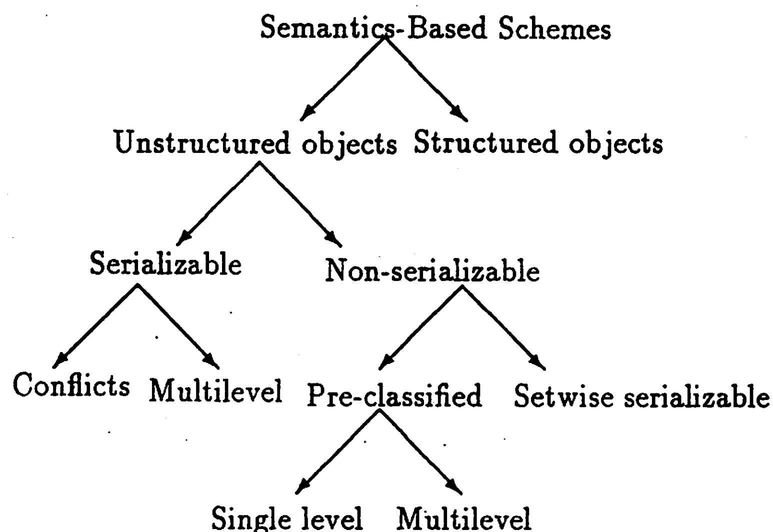


Figure 2.1. A taxonomy of semantics-based concurrency control schemes

correct schedule, one whose effect on the database is the same as that of some serial schedule. The formal notion of *effect* has resulted in various versions of serializability such as state and view serializability[85]. Serializability is an intricate property — in fact, it is NP-complete to test whether a schedule is (view or state) serializable [60, 61]. Most existing or proposed solutions to the concurrency control problem enforce the criterion in [25], known as conflict preserving serializability; known simply as serializability.

Many algorithms have been proposed to guarantee serializability — both in centralized and distributed systems. In [14], a review of techniques underlying most existing algorithms is presented. The majority of these algorithms are based on two-phase locking or timestamp ordering. Two schedules consisting of the same operations are considered equivalent if pairs of conflicting steps appear in the same order in both the schedules. These algorithms ensure that a schedule is equivalent to some serial schedule, thereby achieving conflict serializability. Further, these protocols are based on the assumption that transactions execute only ***single level read and write*** operations.

Conflicts and commutativity: Concurrency control schemes based on various lock modes for different granularities of data were proposed in [31, 32]. This is suitable for transactions accessing data of different sizes like files and records. By having

different lock modes one could optimize the number of locks to be held on finer granules by locking a coarse granule of data. This work has been generalized to arbitrary operations using commutativity as a basis for defining conflicts [40]. Further, deadlock avoidance schemes using granular lock modes have been proposed in [39].

The operations executed by transactions have been extended, from reads and writes, to handle operations on abstract data types. In [79], abstract data types embedded with mechanisms for recovery and serializability are known as atomic data types. To enhance the concurrency of atomic data types, methods that support the use of user specified commutative operations are utilized. Further, several implementations of atomic types are presented using Argus [45, 81] as a vehicle for describing implementations. Similar approaches extending the concept of atomicity to shared abstract data types can be found in [2, 3, 68]. The TABS project is also based on the concept of transactions and abstract data types [74]. Methods for specifying the properties of shared data types, and locking techniques for their implementation have also been developed [79], and in [66] recovery techniques for shared abstract data types are also investigated. ISIS [16] uses data abstractions and transactions and achieves reliability through replication. The main focus of this research has been to provide efficient replication algorithms for abstract data types. In these approaches, commutativity of operations has been utilized for enhancing concurrency.

Definitions of conflicts based on serial dependency relations have been used in optimistic concurrency control schemes and multiversion timestamp schemes for abstract data types in [34, 35]. Two operations conflict if they invalidate each other. This definition is weaker than commutativity which requires equivalence of states. In order to determine whether a given operation conflicts with other active operations, a view is constructed from the active operations to determine whether the new operation produces a legal history. Construction of a view is necessary because a modular condition, which is weaker than global atomicity considered in this thesis, known as local atomicity is achieved in [34, 35]. Concurrency control schemes which use state based information have been proposed in [64, 65]. Here, the concurrency control schemes use information obtained from the return values of the operation and the parameters of the operations. Further, the concurrency control protocols are single level applied to simple operations and objects. We too make use of synchronization properties of operations invoked by different transactions to define weaker notions of conflict. But this forms just the starting point of our work.

Multilevel and nested transactions: In [12], the authors present a theoretical basis for nested computations. Here, nested sub-computations are considered to be equiv-

alent to their parent computation in a very general sense and sufficiency conditions are provided for reducing a nested computation into a single level computation. The model we have developed is much simpler and specially suited for implicit nested environments, though their work has been a prime source of inspiration.

A model for multilevel transactions, a special form of nested transactions, has been proposed in in [58, 82]. This model is appropriate for systems based on several layers of abstraction. Each layer corresponds to an abstract level, and interacts primarily with the layer below. It is common to regard each layer in such a system as a virtual machine whose purpose is to support the next higher-level virtual machine. Suppose user transactions issue operations to *read* and *write* records and these operations are implemented by fetching and storing the disk pages containing the records. It is possible for two transactions executing write operations on different records, stored in the same page, to be judged non-serializable by looking at the sequence of page level operations; nevertheless, the execution may be serializable at the record level. Concurrency control protocols designed for such systems, allow abstract serializability — serializable top-level actions may involve cyclic access of elements by lower-level actions. Further, it is possible to show that by providing conflict-based serializability at each level, abstract-serializability can be obtained for top-level actions. This idea of multilevel serializability has also been used to verify concurrency control protocols for search structures [30, 71].

Recently, in [53, 54], a model for shared nested objects has been proposed. This model is similar to the one in [82] in the sense that both use multilevel relations to define dependencies among sub-operations. Further, a new bottom up nested protocol has been proposed for nested objects. In contrast to the top down nested protocol in [33, 58], this protocol first schedules leaf level requests and then higher level operations.

Non-serializable executions: Concurrency control mechanisms, which do not achieve serializability, using semantic information of transactions to allow higher concurrency and still preserve consistency were proposed in [27]. In this scheme, transactions are classified into different semantic types. Each type of transaction has an associated set of compatible transactions. Compatible transactions are transactions that due to their semantic construction, can run concurrently. Hence, the notion of compatibility is coarse; it is at the transaction level. The only source of semantic information used is that compatible transactions can be interleaved, which in turn is derived from commutativity of operations. The effects of the concurrency desired, i.e., the acceptable non-serializable executions, in [22, 27], appears to have

been achieved by use of abnormal affects or exception conditions in some transactions. Further, there seems to be a scarcity of richer examples than the example of update, withdrawal, and transfer transactions given in [22] where such a classification, based on general commutative steps helps in enhancing the concurrency of transactions.

In [28], the concept of *sagas* has been proposed to deal with the problems of long lived transactions. *Sagas* is a set of transactions with an associated set of transactions called compensating transactions. *Sagas* is similar to a two level nested transaction with undos being implemented by pre-defined compensating transactions. However, unlike the transactions in [27], users cannot specify transaction compatibilities. We would call the type of model employed in *sagas* explicit nesting.

The notion of compatible transactions can be extended to several levels forming a hierarchy [47]. Based on this hierarchy, the concept of multilevel atomicity has been proposed; it too utilizes transaction semantics in concurrency control. In this approach, transactions, with different degrees of interleaving constraints, are grouped into different classes. For example, consider a banking system with transfer transactions consisting of a withdrawal step followed by a deposit step. Transfers might be allowed to interleave arbitrarily with each other. However, an *audit* transaction, which reads all the account balances, should not be allowed to interrupt a transfer transaction as the audit transaction would miss counting the money in transit. That is, the entire transfer transaction is atomic relative to an audit transaction. Hence, a transaction has different sets of breakpoints with respect to each of the remaining transactions. Transactions, which have executed some steps, issue a breakpoint wherein steps of other compatible transactions can be interleaved. The specification of break points, which is hierarchical in nature, is relative to the classes of transactions. It is, however, the responsibility of the user to provide breakpoint specifications and classify the transactions into a hierarchy.

Recently, the notion of compound transactions has been proposed in [70]. Non serializable executions are permitted by using a criterion known as *setwise serializability*. This criterion allows transactions to interleave, producing non-serializable executions while satisfying the individual post-conditions of the transactions. This method analyses the consistency constraints to partition the set of objects in the database into *atomic data sets* having independent consistency constraints. The individual transactions must access data belonging to a particular atomic data set.

Structured databases: For a specific collection of objects the structure of the objects can be a source of semantic information. Based on the access patterns of the transactions it is possible to define the allowed interleavings of various operations.

The database (objects) can have a structure which controls the access pattern of transactions. Many protocols have been developed assuming databases structured as trees, directed acyclic graphs, hypergraphs etc., [18, 37, 72, 84, 86]. These structures help in providing interesting properties like deadlock freedom, lack of cascading rollbacks, and in some cases more concurrency than two-phase locking or timestamp methods. No attempt has been made to use semantic information about the transactions or the operations executed by them. Further the entire set of objects may not exhibit such a structure but some subsets may. Though we assume a nested structure of operations and objects, unlike the protocols cited above, we do not in any way constrain the order of data access by transactions.

2.2 Projects Involving Extensible Databases

In recent years, a number of new database system research projects have been initiated to address the needs of the emerging class of applications using extensible databases: EXODUS at the university of Wisconsin [20, 19], PROBE at CCA [23, 52], POSTGRES at Berkeley[76], GemStone at the Oregon Graduate center [21, 50], STARBURST at IBM [67], ORION project at MCC [8, 29, 38], and GENESIS at the University of Texas-Austin [9, 10]. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, the overall approaches vary widely. We will summarize the main features that are relevant to our work and relate how our transaction management schemes can be used in these databases.

EXODUS provides kernel facilities such as a storage manager and a type manager to support complex objects. It provides a framework for building application specific database systems. The storage manager provides concurrency control and recovery services for storage objects. Two-phase locking of byte ranges is used for concurrency control, with an option to lock an entire object. For recovery, before/after-image logging and in-place updating at the object level is used.

The goal of PROBE, a knowledge-oriented DBMS, is to solve data modeling and query processing problems that arise in non-traditional applications. Research has focused on three areas: data modeling, recursive query processing, and facilities for spatial and temporal data. PROBE provides support for structured objects that allow definition of objects that consist of other objects and capability to operate on these as one unit. Further, the ability to access a structured object at different levels of

abstraction is also provided. The problems of concurrency control and recovery have not been addressed as yet.

POSTGRES is based on the relational data model with features provided to handle non-traditional applications and support for complex objects. Conventional two-phase locking using read, write locks is adapted for concurrency control. Recently, in the system XPRS [75] a special purpose concurrency control technique based on failure commutative transactions has been proposed. This is similar to our definitions of recoverability[5].

The GemStone project supports objects similar to that of Smalltalk-80[51]. Transaction updates are always done on a shadow version of the object. An optimistic concurrency scheme is employed for concurrency control. Validation on commit depends on read-write or write-write conflicts. If there are no conflicts, the modified version is overlaid on the shared object. In case of a conflict the changes in the shadow are discarded.

The goal of the Starburst project is to examine how traditional relational databases may be adapted for new applications and technologies. The research addresses issues of extensibility: external data structures, access methods, abstract data types and complex objects. Complex objects are composed of an arbitrary collection of records from one or more relations. Issues of concurrency control and recovery for transaction support have not been addressed.

Genesis is a project developed to address the needs of non-traditional applications like VLSI CAD, graphics and statistical databases. A functional data model and data language form the front-end to GENESIS. The physical back-end of the DBMS such as storage structures, concurrency control and recovery algorithms are yet to be designed

ORION is an object-oriented database system to support CAD/CAM, AI and OIS domains[8]. ORION provides a number of advanced features such as, version control, support for composite objects, and dynamic changes to the schema[38]. The concurrency control scheme they have adopted is based on conventional granularity based locking protocols[29].

Work on using transactions for CAD databases is reported in [36]. This model has been enhanced further in [41]. This model of CAD transactions allows a group of cooperating designers to collaborate on a design without having to wait over a long duration. Each design transaction is composed of sub-transactions. The sub-transactions are short-duration transactions whose atomicity is achieved by two phase locking. In a complex design project some of the tasks are sub-contracted to other designers. These transactions are provided with a multilevel concurrency control

scheme implemented at the database operation level. The model adopted here uses a nested transaction framework and is suited for computer-supported, cooperative environments.

As is evident from the discussion, very few projects are focusing on the problem of providing efficient transaction management for object-oriented databases; more often modest attempts employing conventional conflict based locking schemes have been employed. Few attempts have been made to consider the semantics of the operations, objects, or transactions, nor the requirements of an application in order to provide new concurrency control protocols. The research work described in this dissertation addresses some of the problems that have not been considered, in particular, support for efficient transaction management. The approach described here, is based on exploiting semantic information, and offers novel schemes for various aspects of transaction management in complex information systems.

CHAPTER 3

MULTILEVEL GRAPH MODEL

We are interested in developing semantics-based concurrency control schemes for implicitly nested environments. To prove that the proposed protocols for implicit nested concurrent computations ensure the serializability of top-level operations, we introduce the *multilevel computation graph* and a concomitant *graph condensation* procedure. In recent years, a number of formal models have been proposed for nested computations [11, 12, 48, 49, 53, 71, 82]. Our formal framework does not replace these, but attempts to provide an arguably simpler and natural model to prove correctness of multilevel semantics-based concurrency control protocols in the specialized setting of *implicit* nested environments.

An Example

We will use a simple example to build intuition and illustrate features of the model. Consider the simple example shown in Figure 3.1. The figure shows two transactions T_1 and T_2 that have executed increment operations inc1(x) and inc2(x) respectively, where x is the variable on which these operations are executed. Each *increment* operation is composed of a *read* of the initial value followed by a *write* of the new value. In Computation I, a *read* operation, r2(x), of the *increment* operation, inc2(x), is interleaved between a *read* operation, r1(x), and a *write* operation, w1(x), of the *increment* operation inc1(x). We are interested in the correctness of the computation resulting from the concurrent execution of *increment* operations. Thus concurrent operations should inherit any conflicting dependencies among their sub-operations. Thus, in order to determine that the increment operations are executed correctly in spite of interleaving of sub-operations, the dependencies between read and write operations should be inherited by the increment operations.

This is achieved by a condensation step which represents all the sub-operations (nodes corresponding to reads and writes) and the parent operation (node corresponding to increment) by a single node, and a dependency edge between sub-operations

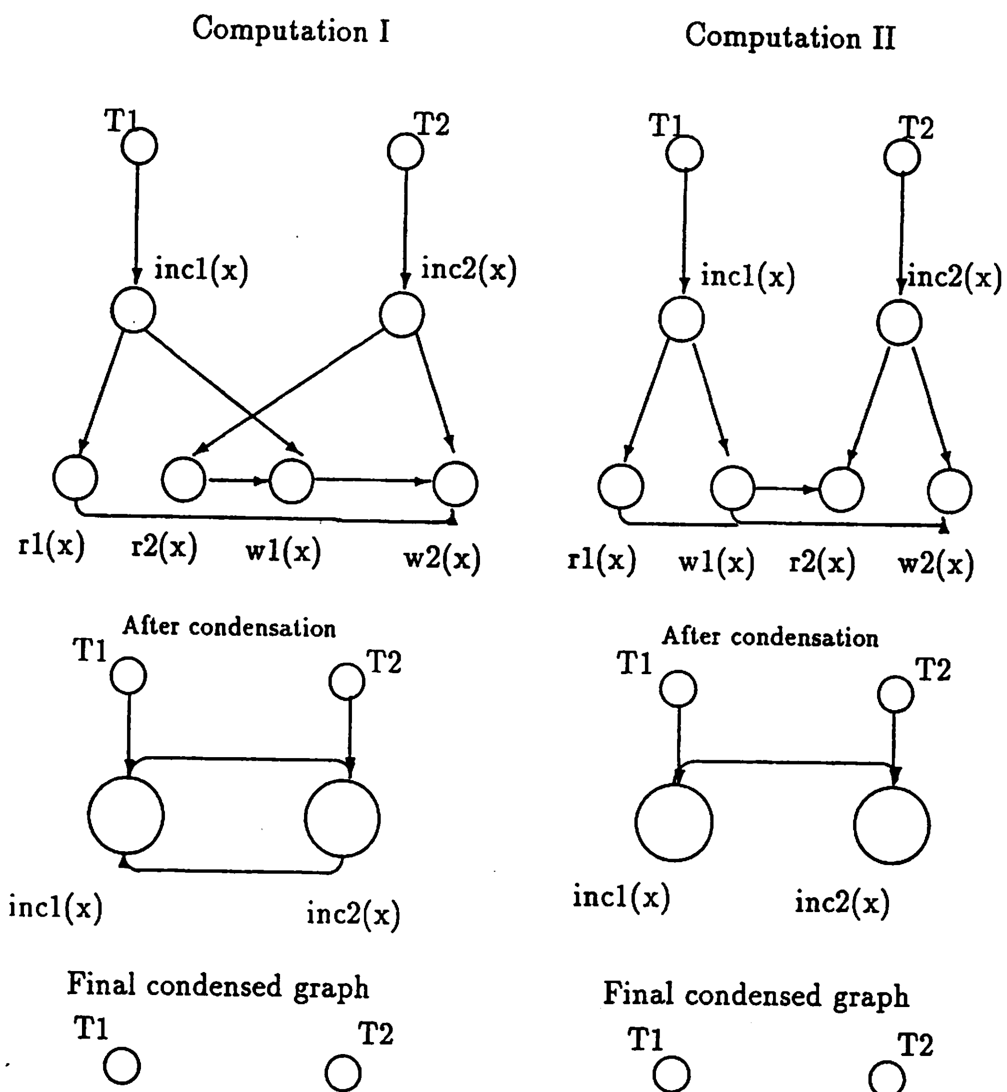


Figure 3.1. Condensation of a multilevel computation graph

of different parent operations is represented by a dependency edge between the condensed nodes. There is a dependency between r2(x) and w1(x), since a read and a write operation on the same element conflict. This dependency when inherited results in a dependency edge between inc2(x) and inc1(x). Similarly, there is a dependency between r1(x) and w2(x), as these two operations conflict, which results in a dependency edge to be inherited between inc2(x) and inc1(x) due to the condensation procedure. Thus, after one step of the condensation procedure, there are cyclic dependency edges between the two increment operations. This implies that the sub-operations were not executed atomically thereby resulting in a lost update. Hence in this concurrent execution, the specifications of the two *increments* are not met. Thus, for Computation I we arrived at an (intermediate) condensed graph with cycles, and hence Computation I is unacceptable.

However, in Computation II, the sub-operations of *increments* are not interleaved, and hence the condensation of the graph shows acyclic conflict dependencies. The dependency edge between the two increment operations denotes the order of completion. Having now established the validity of the computation of *reads* and *writes*, we can concentrate on the computation of *increment* operations. Since two *increments* commute, they do not conflict, and either order of execution is equivalent to a serial execution. Therefore, in the next condensation, the dependency edge between *increments* is dropped. In summary, the serializability of top-level operations can be established by performing a stepwise condensation of the multilevel graph. In each step of the condensation we use semantics of the operations to define conflicts among operations thus making it possible to ignore some edges. The more *dependency edges* we can drop by using *semantics* the more *condensed graphs* we can show as acceptable thus allowing more concurrent executions.

3.1 Formal Aspects

We now present some important formal definitions and theorems that are useful in proving semantic serializability of implicit nested concurrent computations.

The model represents the structure of implicit nested operations as directed rooted trees. The root is the user level transaction and the internal nodes represent the operations and sub-operations of the transaction.

Definition 1: Let $G = (V, E)$ be a forest of directed rooted trees. Each tree in G represents the structure of a transaction. A root is a transaction and each internal vertex is associated with an unique operation. If (a, b) is an edge in E then b is

the sub-operation of a. Vertices which do not have out going edges correspond to the leaf-level operations.

We consider each operation to be at a certain level. The leaves are at level 1, the roots (transactions) are at level n, and top level operations are at level $n-1$. If an operation is at level i, then its children are at level $i-1$ and so on.

Definition 2: SG(H)= $(G, <_c)$ is a multilevel history, where $<_c$ denotes a set of partial orders $(<_1, \ldots, <_n)$, one for each level of G. Each $<_l$ is a partial order on the operations belonging to level l.

The partial order $<_l$ for each level denotes the order of execution of operations. $o_1 <_l o_2$ if o_1 completed before o_2 was invoked. If o_1, o_2 are not related by $<_l$ then o_1, o_2 are said to be concurrent.

A computation can be defined from this history by mappings $m_l = (S_i, S_f, R_i)$ associated with the vertices belonging to each level l $(1 \leq l < n)$ in the multilevel computation. S_i is a initial state, S_f the final state, and R_i the set of return values obtained by executing level l operations according to $<_l$. In other words, a computation induces a mapping at each level.

Definition 3: A multilevel computation $C(H) = (G, <_c, m)$, where m is a set of mappings, one for each level of the multilevel graph.

Definition 4: Consider a single level computation $C(H) = (G, <_1, m_1)$. We say that $C(H)$ is atomic if for any total order $<_1'$, $<_1 \subseteq <_1'$, $(G, <_1')$ also produces the same mapping m_1.

When operations are executed concurrently, atomicity does not imply that the operations are executed indivisibly but only that the operations in the computation appear to have been executed in some serial order.

Definition 5: Two mappings m_l and m_l' are equivalent if $S_i \equiv S_i'$, $S_f \equiv S_f'$, and $R_i = R_i'$, where $\equiv$ is an equivalence relation on states.

Observe that the definition of equivalent mappings for a level requires that the state changes and return values be equivalent only with respect to that level. In fact, the lower level mappings may not be equivalent.

To make the model general, we allow the equivalence relation to be any arbitrary equivalence relation on states. The desired equivalence relation will, however, depend

on the consistency requirement of the application. The most common equivalence relation (e.g. when syntactic serializability is desired) defined on states is the equality relation. In general, when a concurrent set of operations are executed, we are interested in seeing if the top level objects are in equivalent states and the return values of the operations are the same when the same computation is executed serially. This defines the equivalence of a computation with respect to a serial computation.

In addition to the structure of the operations, our model requires the specification of conflict information between operations. The definition of conflict between operations depends upon the state changes and return values obtained by executing the operations. So we will define conflicts in terms of the mappings that result from the execution of the operations.

Definition 6: Consider two operations o and o'. o and o' conflict if $(\{o, o'\}, o < o', m)$ and $(\{o, o'\}, o' < o, m')$ where m and m' are not equivalent.

We need to define certain conditions that these multilevel computations have to satisfy in order to be well-formed.

1) If $o <_l o'$, then for any $x \in child(o)$, $y \in child(o')$, $x <_{l-1} y$.

2) Leaf level computations are atomic.

3) For any two conflicting leaf level operations, o, o', either $o <_1 o'$ or $o' <_1 o$.

Condition 1 ensures that the partial orders of operations at different levels are related by the temporal order of execution of their children. Condition 1 has also been called downward order compatibility by other authors[11, 71]. Condition 2 states that in order to reason about the multilevel computation, we begin with an assumption about the execution of leaf level operations by the system, while condition 3 requires that conflicting leaf level operations be ordered by $<_1$.

Definition 7: Two multilevel computations $C_1(H) = (G, <_c, m^1)$ and $C_2(H) = (G, <'_c, m^2)$ are equivalent if the following hold
- G, G' have the same set of roots (transactions) and the same set of level $n-1$ (top level) operations.
- The execution of $C_1(H)$ and $C_2(H)$ produce mappings $m^1_{n-1} \equiv m^2_{n-1}$.

Note that the definition of equivalence requires that the computation be equal to only top level operations issued by the roots since state changes to objects at the top level contribute to external behavior.

Definition 8: A multilevel computation $C'(H)$ is serial if $<_n$ is total. A computation $C(H)$ is semantically serializable if $C(H) \equiv C'(H)$.

We will now derive conditions for a multilevel computation to be semantically serializable. In order to do this we define a sequence of condensations. Condensations form the foundation for our formal model.

The specification of operations at each level is defined in terms of state changes and return values when operations belonging to that level are executed. The result of this execution is defined by the order of executing various operations. This order of execution is captured in terms of the dependencies between conflicting operations. In order to determine the order of execution of the top level operations, we begin condensing the multilevel graph bottom-up. At each level the dependencies are captured by the dependency edges between vertices at the same level in $SG(H)$. First, we need to determine the execution order of the leaf level operations.

Definition 9: There is a dependency edge from o to o', i.e., $o \rightarrow_d o'$ iff $o <_1 o'$ and o, o' conflict.

Definition 10: A *condensation sequence h* on $SG(H)$ is a totally ordered sequence of condensations $SG_{i+1}(H) = X_i(SG_i(H))$ induced on $SG_i(H)$ by X_i, where $X_i = Y_i \circ Z_i$, maps nodes and edges in $SG_i(H)$ to nodes and edges in $SG_{i+1}(H)$. Y_i and Z_i will be defined shortly.

Thus, if $SG_1(H)$ is the original multilevel graph, the next graph $SG_2(H)$ is obtained by condensing $SG_1(H)$ with X_1 and so on.

Given a SG_i, Z_i determines a new order for the operations by considering conflicts among the leaf level operations in the newly condensed graph. This is done by dropping dependency edges between non-conflicting operations and introducing dependency edges between conflicting operations.

Definition 11: Mapping Z_i ensures the following: Given leaf nodes v and w $\in SG_i(H)$ there is a dependency edge (v, w) *iff* v and w conflict.

Y_i reduces the computation by a level and at the same time captures information about the order in which sub-operations were executed.

Definition 12: Mapping Y_i is defined as follows:

- Let $\{S_1, S_2, \ldots, S_n\}$, be sets of nodes in $SG_i(H)$ where each S_j is the set of leaf nodes of $SG_i(H)$ having the same parent together with that parent.

- Nodes in S_j for all $j \in 1 \dots n$ are mapped onto a single (new) node v_j in $SG_{i+1}(H)$. All other nodes v in $SG_i(H)$ are mapped to v in $SG_{i+1}(H)$
- Edges (including dependency edges) between vertices in $SG_i(H)$ are mapped to edges between the corresponding vertices in $SG_{i+1}(H)$. Thus, there is a dependency edge between v_m and $v_p \in SG_{i+1}(H)$ iff some node of S_m is adjacent (has a dependency edge) to at least one node of S_p.

3.1.1 Proving Semantic Serializability

Since our interest lies in the serializability of transaction level operations we will relate the condensation of a multilevel graph and the correctness of a multilevel computation represented by the graph, to a correctness result by means of the following theorem.

Theorem 1: Let h_{SG} be the *condensation sequence* of a multilevel history SG(H). Then the computation $C(H) = ((SG(H), m)$ is serializable if and only if all graphs in h_{SG} are acyclic. The topological sort of SG_n in h_{SG} gives the serialization order of the multilevel computation represented by $SG(H)$.

The next theorem states that in order to show the serializability of a multilevel computation, we need only show that the computation at each level is serializable. This will enable us to design multilevel schedulers where a scheduler at each level guarantees the serializability of the computation at its level.

Theorem 2: Let c_i represent a two level computation consisting of level $i+1$ and level i operations of the multilevel computation $C(H)$. If all c_i's $(1 \leq i < n-1)$ are serializable then $C(H)$ is serializable [54, 58, 82].

3.1.2 Application to Implicit Nested Computations

Our model requires a description of the structure of the operations. The description should include the sub-operations at various levels, and the specification of the operations so that conflicts among these operations can be defined. Consider our running example shown in Figure 3.1, where each increment operation is composed of two sub-operations; a read followed by a write.

In order to determine the correctness of a given computation, we begin condensing the multilevel graph bottom-up, i.e., by applying the condensation procedure to the leaf level of the current graph. As implied by Theorem 1, the condensation procedure can also be used to infer the imposed order of execution of the top level operations.

Each step of the Condensation involves two phases. In the first phase, the order of execution of conflicting leaf-level operations are captured by the dependency edges between leaf level vertices. Mapping Z realizes this by dropping dependency edges (produced by the previous condensation step if any) between non-conflicting operations and by introducing dependency edges between conflicting operations. The specification of operations at each level is defined in terms of state changes and return values when operations belonging to that level are executed. The result of this execution is defined by the order of executing various operations. This order of execution is captured in terms of the dependencies between conflicting operations.

The second phase realized by mapping Y reduces the graph by a level and at the same time captures information about the order in which (sub-operations of) the new leaf-level operations were executed.

If there are no dependency cycles at the new leaf level, then we can infer that the operations were in effect executed atomically in spite of interleaving sub-operations. Thus, a given multilevel computation is serializable if and only if there is no cycle at the leaf level of any of the graphs produced during the condensation process. This correctness requirement is formally stated in Theorem 1.

The key to proving the correctness of semantics-based protocols lies in two factors: first, the decision to ignore certain orderings among operations will depend on ensuring that the specifications of the operations are met, and hence depends on conflicts among operations. Secondly, deciding which orderings to inherit from the sub-operations depends on the sub-operations that are needed for the specification of the operations to be satisfied. The more the dependency edges dropped and the less the number of dependency edges inherited, the larger the number of concurrent executions that can be shown correct. However, both these factors will have to assure that the computations are semantically serializable.

3.2 Main Features

The salient features of this model are:

- It is a natural generalization of serialization graph theory which has been used to prove serializability in traditional single level r/w models. This is shown by the following: Consider a two-level graph where the roots represent transactions and the leaves are reads and writes issued by these transactions. The condensed graph in this case is exactly the serialization graph given in [15]. The condition

for serializability is that the serialization graph be acyclic; in our formalism this implies that the condensation of the two-level graph should be acyclic.

- Two condensed nodes with the same semantics (state changes and return values) can be considered equivalent even though the vertices from which they were condensed are different. Thus we can include context-dependent translation of operations in our framework. This is useful in verifying concurrency control protocols for objects with structure. In these situations, operations on complex objects may have different sub-operations because of other concurrent operations, than their counterparts in a serial execution. In contrast, in the models of [11, 82] the translation (in our terminology, condensation) of a given operation into sub-operations is assumed to be the same in any equivalent schedule.

- It is easy to incorporate semantics by using specifications of the operations represented by the condensed nodes and to ignore dependencies.

- Only the order of *conflicting* operations are considered in the condensation. In the model of [82] two multilevel computations differing just in the order of non-conflicting leaf operations are considered non-equivalent. This problem has been corrected in the model of [53].

3.3 Conclusions

To summarize, an implicit nested computation is modeled as a multilevel graph, and the vertices of the graph correspond to operations at various levels. The formalism we have developed is particularly useful in showing *semantic serializability*. This is because, in each step of the condensation process, certain features of the computation can be considered as important by retaining the dependencies between condensed nodes and other features can be considered as irrelevant by ignoring the dependency edges between condensed nodes. The model enables us to state and prove correctness results about concurrency control protocols in implicit nested environments. Two key ideas are used to show semantic serializability: one is the use of semantics of the operations to ignore conflict dependencies. The other is the process of hierarchical graph condensation to reduce a multilevel computation into a single level computation. By condensing the multilevel graph to a single level we can determine the result of a concurrent execution of top level or user level operations.

CHAPTER 4

THE NOTION OF RECOVERABILITY

4.1 Introduction

As mentioned in Section 1, 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* [27, 79]. 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.

4.2 A Formal Definition of Recoverability

4.2.1 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 $(\cup_i OP_{Ti})$ and $(\cup_i <_{Ti}) \subseteq <_E$. $<_E$ 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 $o_i <_E o_j$ [25]. Two operations conflict if they both operate on the

same data item and one of them is a write. Here, we generalize the notion of conflict by considering the semantics of the operations. Execution of operations on different objects can be thought of as generating logs E_j for each object O_j such that log E is the union of all these logs.

Each object has a type, which defines a possible set of states of the object, and a set of primitive operations that provide the only means to create and manipulate objects of that type. The specification of an operation indicates the set of possible states and the responses that will be produced by that operation when the operation is begun in a certain state. Formally, the specification is a total function: $S \mapsto S \times V$ where $S = \{s_1, s_2, \ldots\}$ is a set of *states* and $V = \{v_1, v_2, \ldots\}$ is a set of *return values*. For a given state $s \in S$ we define two components for the specification of an operation: $return(o, s)$ which is the return value [1] produced by operation o, and $state(o, s)$ which is the state produced after the execution of o.

Definition 13: Consider two operations o_1 and o_2 such that o_1's execution in state s is immediately followed by the execution of o_2. Operation o_2 is *recoverable relative to operation* o_1, denoted by $(o_2\ RR_I\ o_1)$, iff for all $s \in S$

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

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 14: 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.

---

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

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 15: Consider a set of operations $S = \{o_1, \ldots, o_n\}$ such that $\forall_{1 \leq i < n}$ $o_i <_E o_{i+1}$. $(o_n$ RR $o_1)$ if the return value of o_n is 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 $(o_n$ 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 $(o_n\ RR_I\ o_1)$, $(o_n$ RR $o_1)$.
Induction hypothesis (F contains $k-1$ operations): if $\forall\ l,\ 1 \leq l \leq k,\ (o_n\ RR_I$ $o_l)$, then $(o_n$ RR $o_1)$.
Induction Step: Let $|F| = k$ and $S = \{o_n, o_{k+1}, \ldots, o_2, o_1\}$. Now $(o_n\ RR_I\ o_{k+1})$ and $(o_n\ RR_I\ o_k) \Longrightarrow (o_n$ RR $o_k)$ by using a reasoning similar to the base case. From definition 3 we have o_n RR $o_k \Longrightarrow o_n\ RR_I\ o_k$, and by induction hypothesis $\forall\ l\ 1 \leq l \leq k\ o_n\ RR_I\ o_l \Longrightarrow o_n\ RR\ o_1$.

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

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.

4.2.2 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.

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 4.1, (read, read) is commutative and in Table 4.2, (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 4.1. However, with recoverability this notion of conflict is weakened as the only pair of operations considered conflicting is (read, write). Thus, even for the read/write model of transactions, the potential for parallelism increases under recoverability semantics.

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

Table 4.1. Commutativity for page

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

Table 4.2. Recoverability for page

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

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 4.3 and 4.4. 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.

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

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

Table 4.3. Commutativity for stack

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

Table 4.4. Recoverability for stack

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

Table 4.5. 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 4.6. 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 4.7. 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 4.8. 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

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 [79] 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.

$X : \langle insert(3), ok, T_1\rangle$
$X : \langle member(3), yes, T_2\rangle$
$X : \langle insert(7), ok, T_1\rangle$
$X : \langle delete(3), ok, T_1\rangle$ (1)

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

$X : \langle member(3), no, T_2\rangle$
$X : \langle insert(3), ok, T_1\rangle$
$Y : \langle insert(4), ok, T_2\rangle$
$Y : \langle delete(5), ok, T_2\rangle$
$\langle commit, T_1\rangle$
$\langle abort, T_2\rangle$ (2)

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

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

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

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.

4.3 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 [57], 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 [24].

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 [43]. 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 4.3.1 we formally define the correctness requirements of the concurrency control and commit protocols and introduce the commit dependency graph. In Section 4.3.2 we develop the two-phase protocol for pseudo-committing transactions. An algorithm for committing pseudo-committed transactions is given in Section 4.3.3.

4.3.1 Correctness Requirements and Commit Dependency Graph

Definition 16: 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 [17, 73].

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 17: 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 18: An execution log E is correct if it is serializable and is free from cascading aborts.

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

Figure 4.1 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

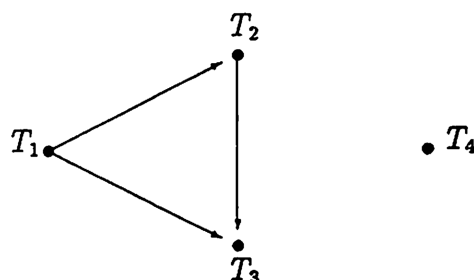


Figure 4.1. A dependency graph

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

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

Figure 4.2. Algorithm to insert commit dependency edges

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 4.2.

4.3.2 A Two Phase Algorithm for Pseudo-committing Transactions

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 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 4.3.

Using Figure 4.4, which shows dependency graphs at three objects, we illustrate, in Table 4.9, 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/

```
Begin

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

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

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 4.3. Pseudo code for the the commit algorithm

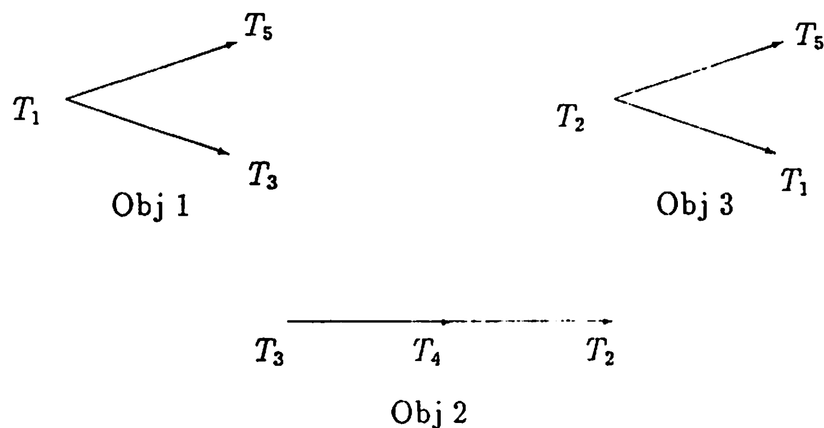


Figure 4.4. Dependency graphs at three objects

Table 4.9. 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

$T_i \stackrel{+}{\Longrightarrow} T_n$, T_n is pseudo-committed and every T_m along the path from T_i to T_n is also pseudo-committed }. $PRED_{obj}(T_l) = \{ T_j / T_j \stackrel{+}{\Longrightarrow} T_l$, T_j is pseudo-committed and every T_k along the path from T_j to T_l is also pseudo-committed }. To determine whether T_i can pseudo-commit or not, the transaction manager collects these sets 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.

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 [44]. 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 19: 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 20: 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 21: 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 [59, 58, 80]. Further, the type of undo is dependent upon the operation. For example (write, read) is recoverable but there is no need to undo 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 [80]. These schemes can be adapted to effect recovery in our concurrency control scheme.

4.3.3 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

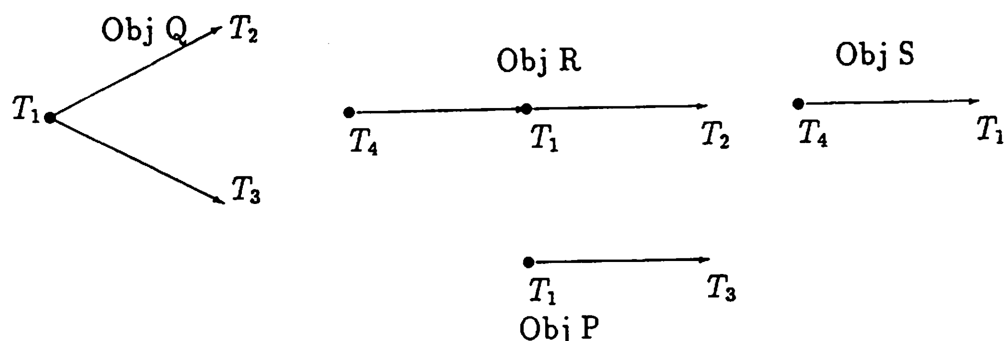


Figure 4.5. Dependency graphs at objects P, Q, R, and S

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 4.5. 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.

4.4 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 concur-

rency 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, 78].

4.4.1 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 4.6 is a modified version of the one used in [1].

There are a fixed number of terminals from which transactions originate. The maximum number of active transactions 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 op-

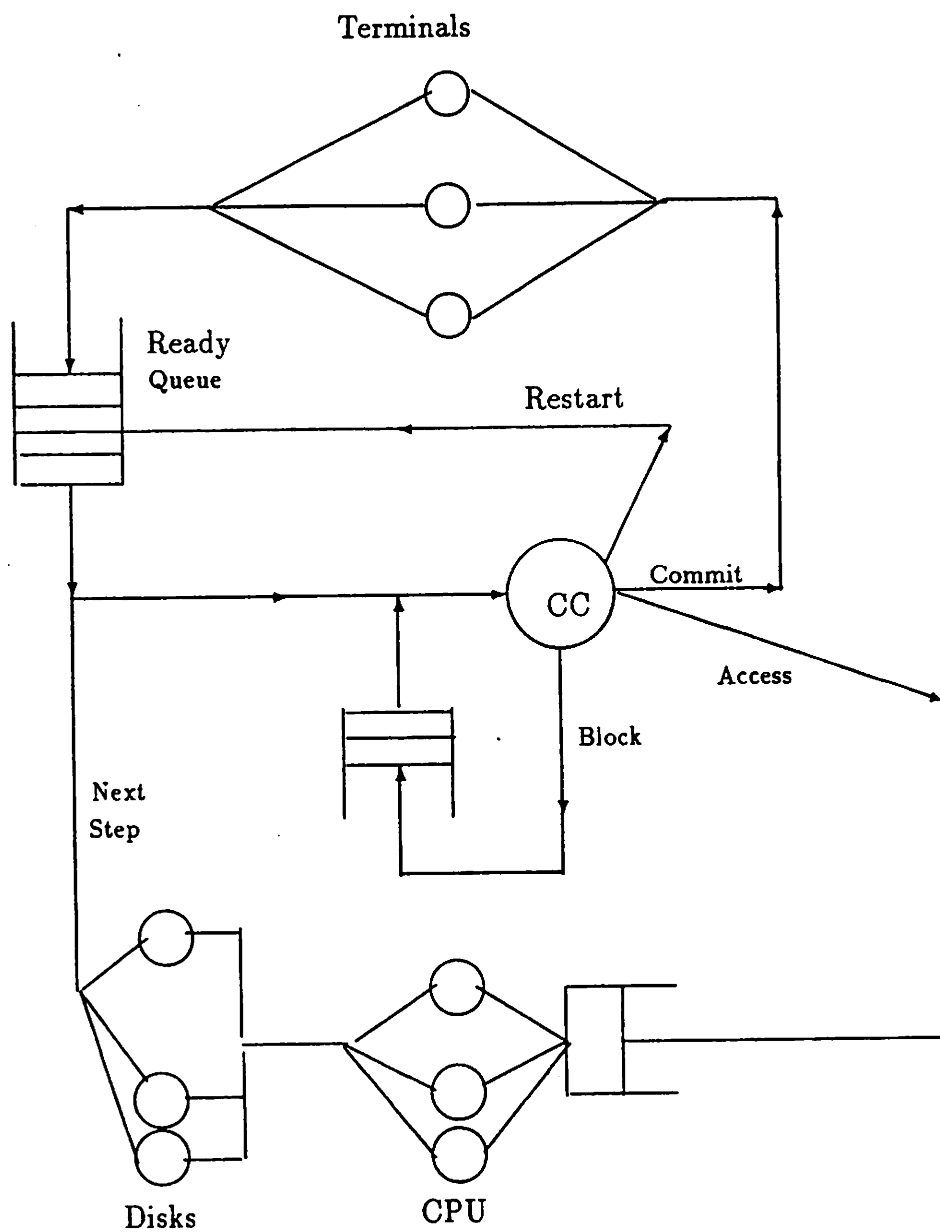


Figure 4.6. Simulation model

Table 4.10. 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

eration. 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 4.10.

4.4.2 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.

4.4.3 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 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.

Table 4.11. 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

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 4.11. The values of the model parameters have been chosen similar to those in previous performance studies of locking protocols [78, 77, 1] and commutativity-based protocols [22].

4.4.4 Performance Metrics

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

includes any time spent in the ready 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.

4.4.5 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 4.7 through 4.51 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 ±2% percentage points of the mean value of the performance metrics shown in the various graphs.

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 [78], 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 4.1, 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 4.7 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 4.8, 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 4.9 and 4.10 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 4.11. 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 4.12 to 4.21. For the case of 5 resource units, the throughput first increases with multiprogramming level and then decreases due to thrashing as shown in Figure 4.12. 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 4.13 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 4.13. 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 4.14 and 4.15 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 4.16 and Figure 4.18 respectively. The blocking ratio for 5 resource units and 1 resource unit is shown in Figure 4.17 and Figure 4.19 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 4.20 and 4.21.

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 4.22 through 4.51, 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 4.37 through 4.51, are consistent with those for $P_c = 2$

Infinite resources: In this part of the simulation, we assume infinite resources. Figure 4.22, depicts increased throughput due to recoverability, and Figure 4.23 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 4.25. A similar observation can be made with regards to the restart ratio shown in Figure 4.24. 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 4.26 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 re-

source units and 1 resource unit are shown in Figure 4.27 and 4.28 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 4.29 and 4.30 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 4.32, 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 4.33 and 4.34 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 4.35 and 4.36 depict the r-abort ratio for 5 resource units and 1 resource unit respectively.

4.4.6 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.*

4.5 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 4.2.2, 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 4.2.2, 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. In the next chapter, we will see how the notion of recoverability can be used in *multilevel* concurrency control protocols for complex information systems.

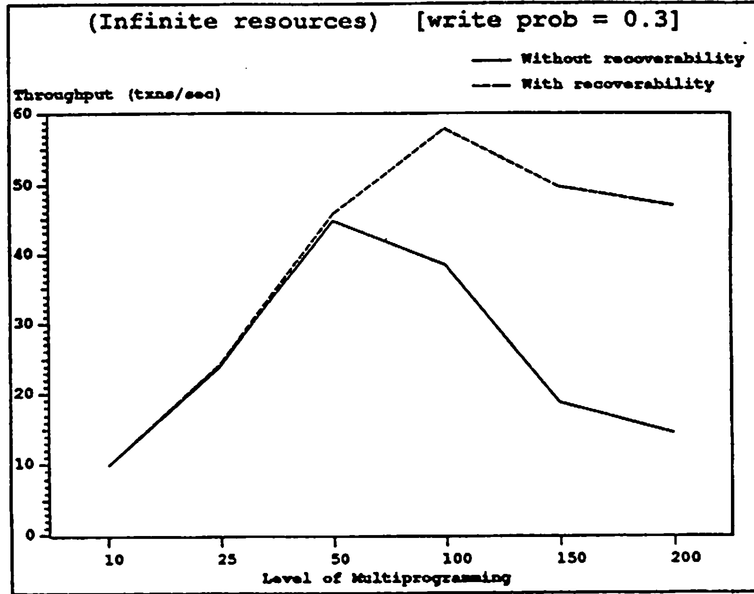


Figure 4.7. Throughput (infinite resources)

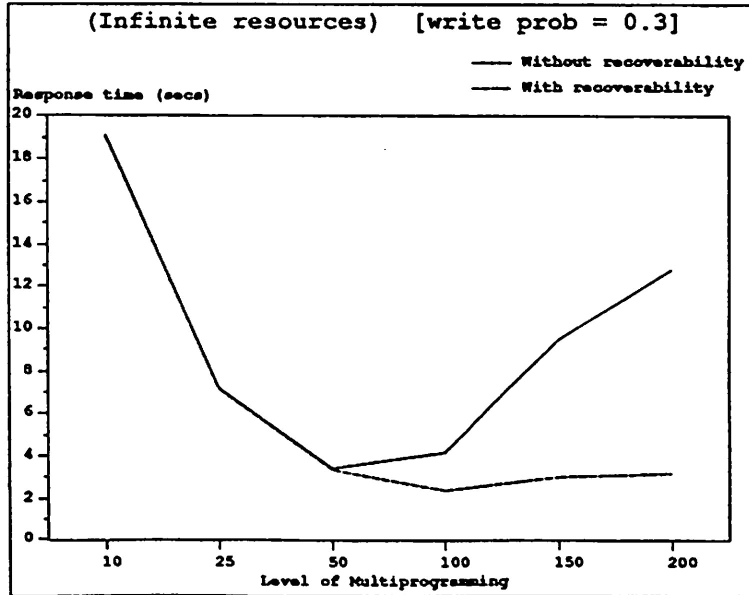


Figure 4.8. Response time (infinite resources)

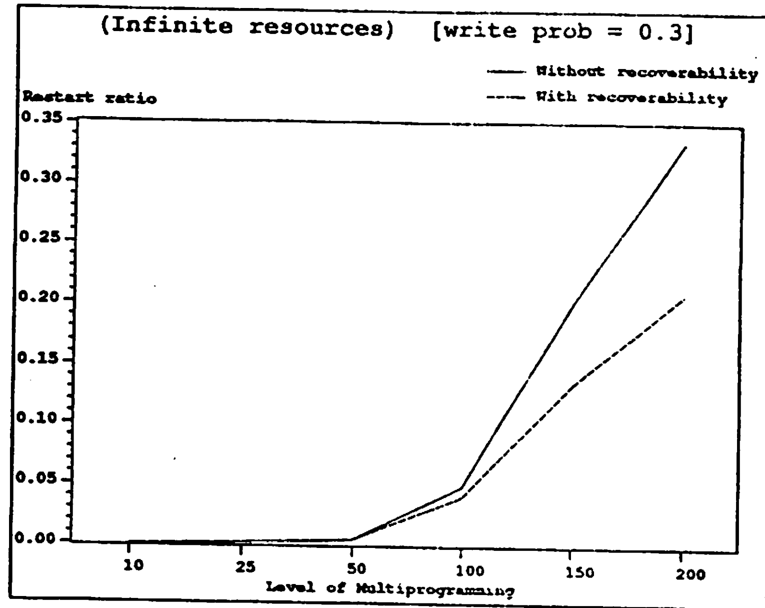


Figure 4.9. Restart ratio (infinite resources)

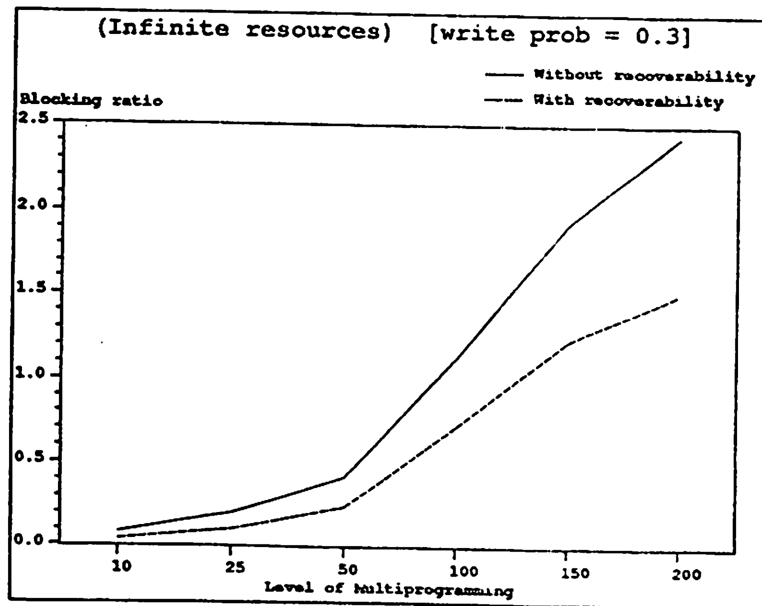


Figure 4.10. Blocking ratio (infinite resources)

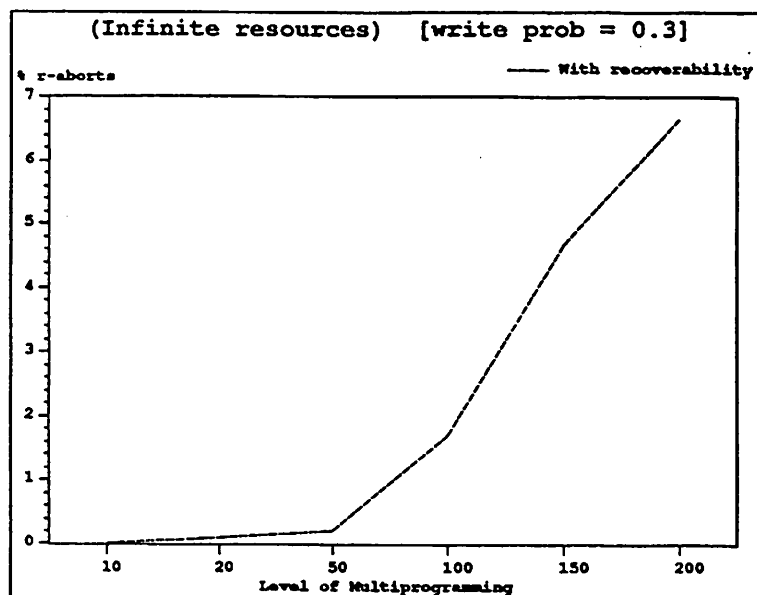


Figure 4.11. R-aborts (infinite resources)

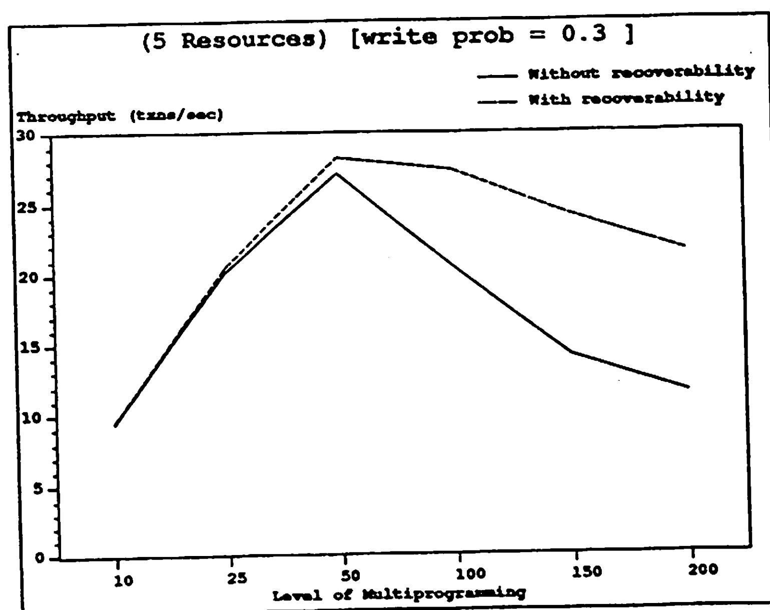


Figure 4.12. Throughput (5 resource units)

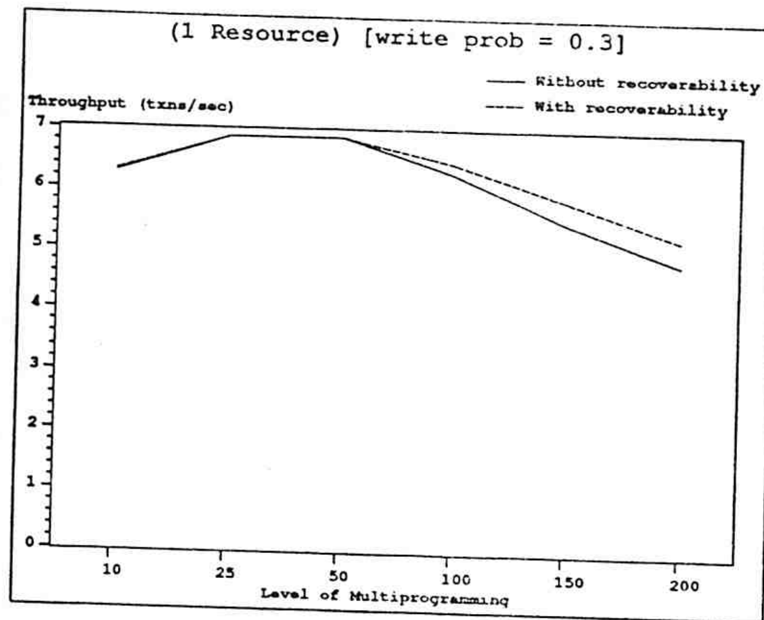


Figure 4.13. Throughput (1 resource unit)

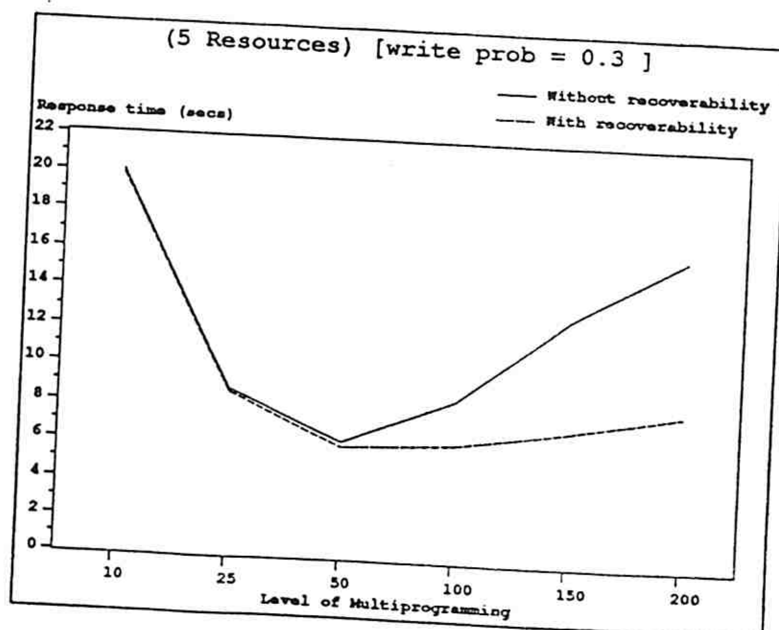


Figure 4.14. Response time (5 resource units)

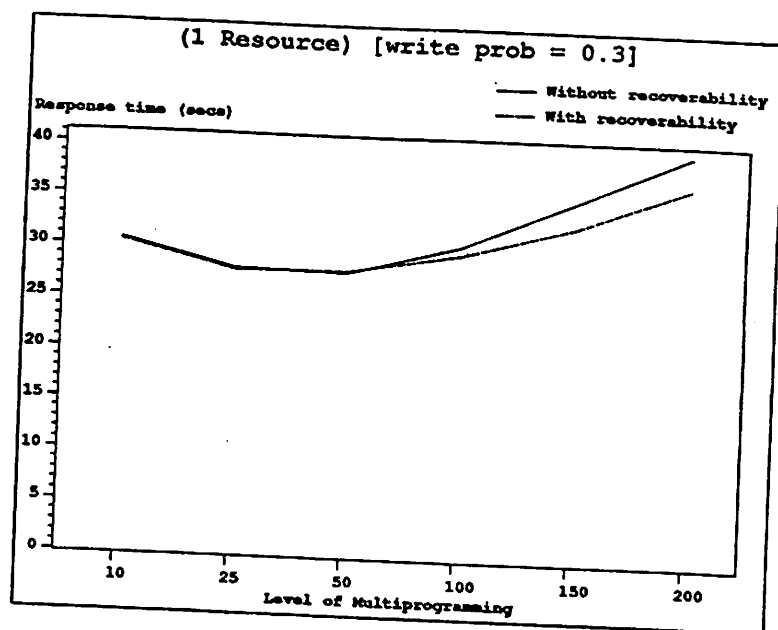


Figure 4.15. Response time (1 resource unit)

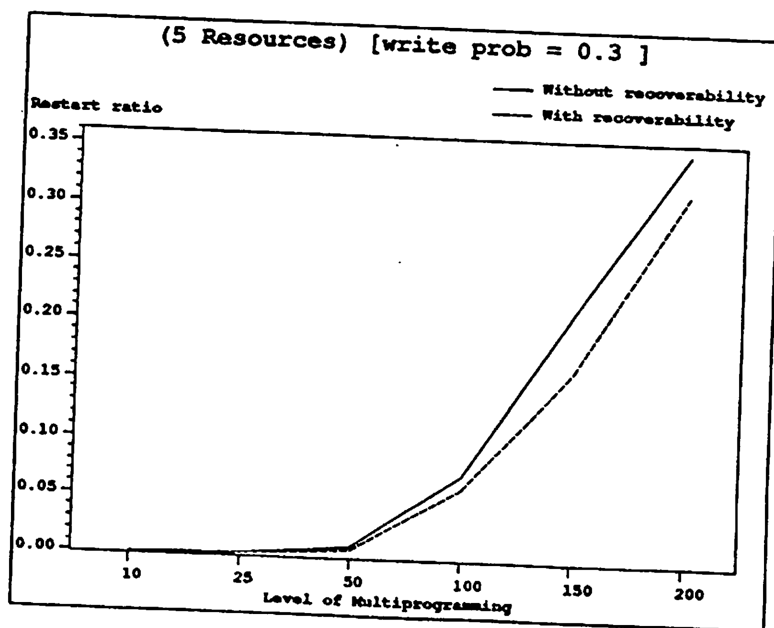


Figure 4.16. Restart ratio (5 resource units)

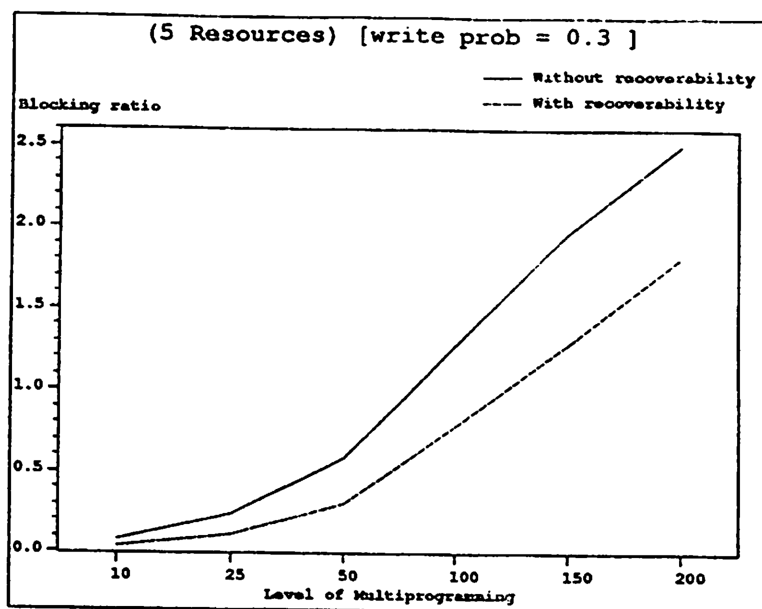


Figure 4.17. Blocking ratio (5 resource units)

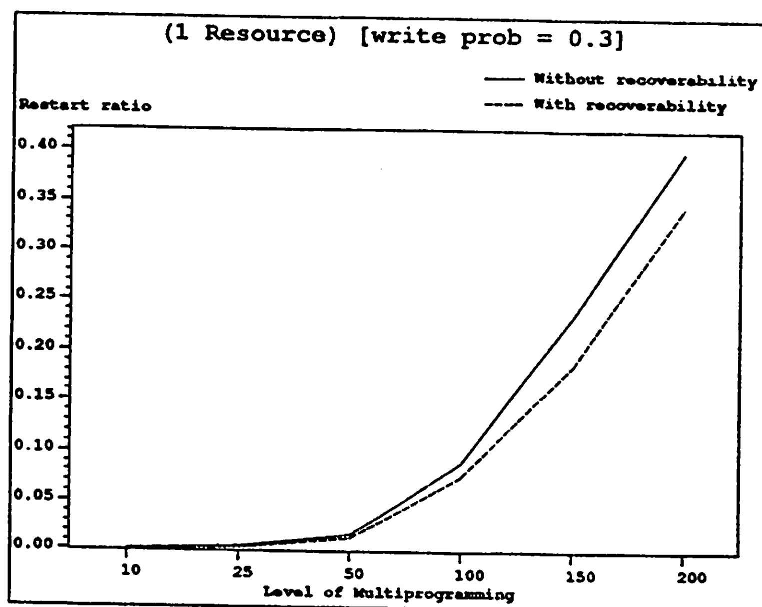


Figure 4.18. Restart ratio (1 resource unit)

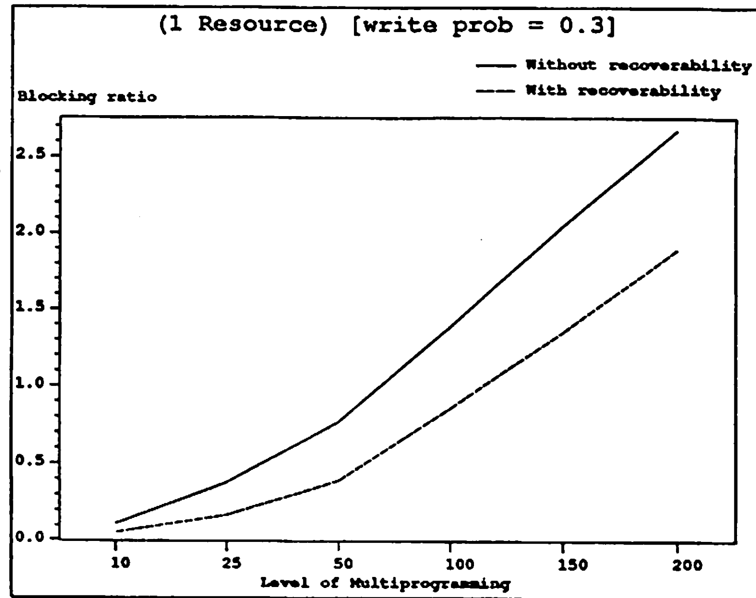


Figure 4.19. Blocking ratio (1 resource unit)

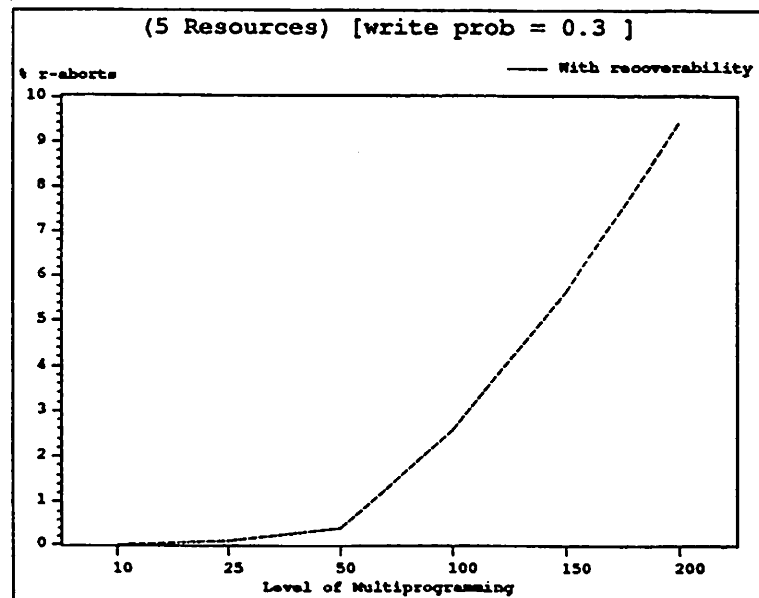


Figure 4.20. R-aborts (5 resource units)

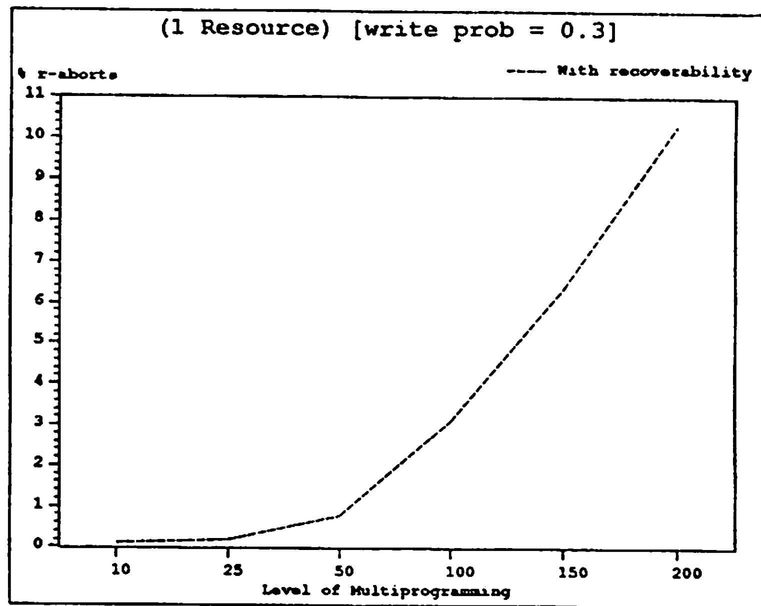


Figure 4.21. R-aborts (1 resource unit)

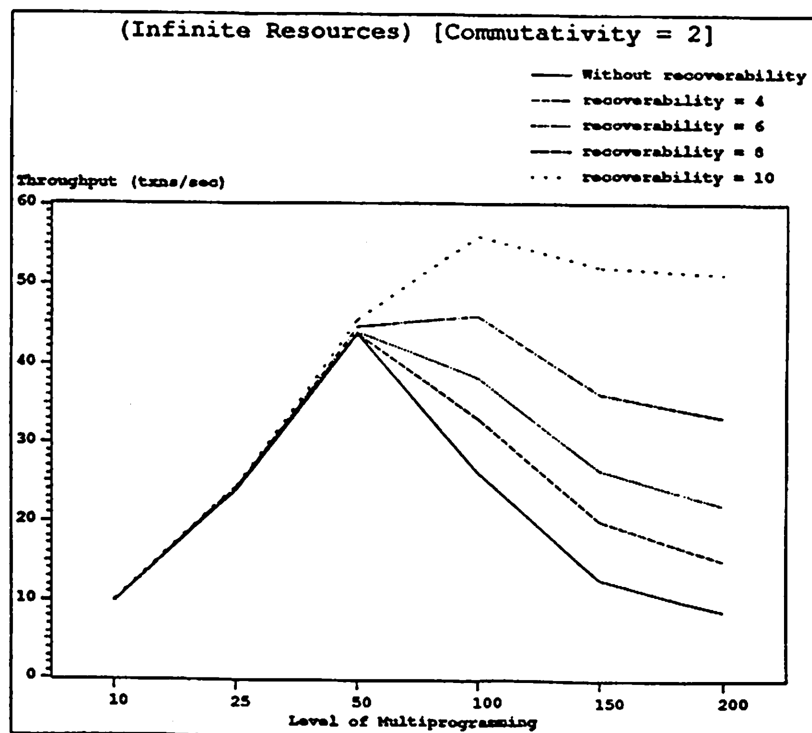


Figure 4.22. Throughput (infinite resources)

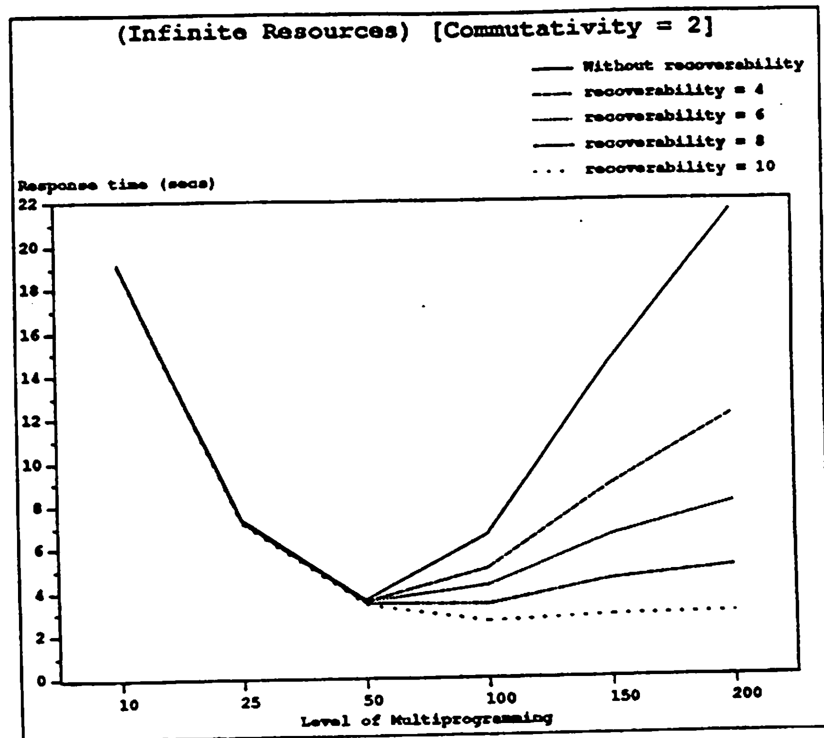


Figure 4.23. Response time (infinite resources)

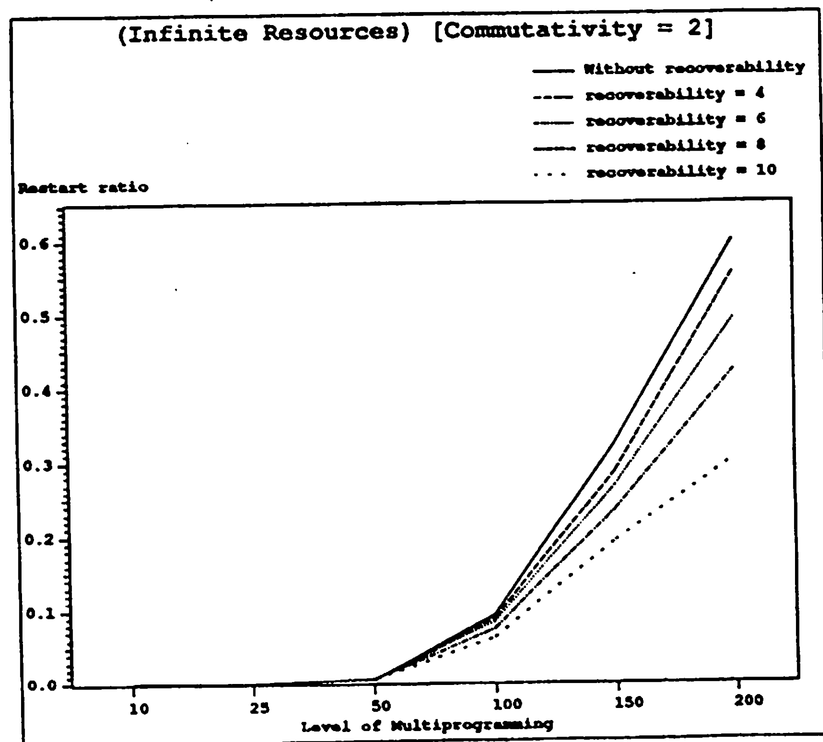


Figure 4.24. Restart ratio (infinite resources)

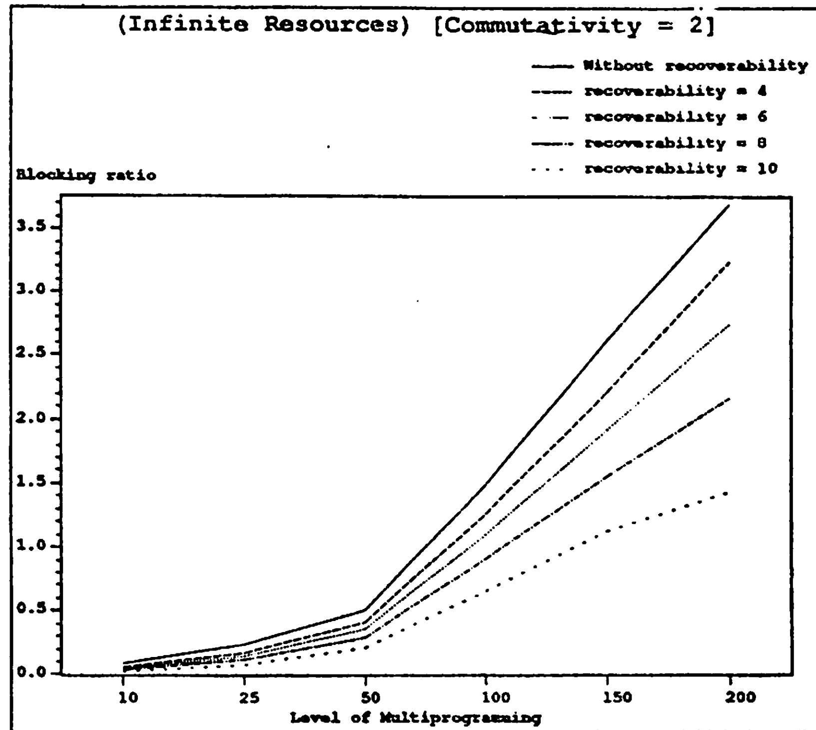


Figure 4.25. Blocking ratio (infinite resources)

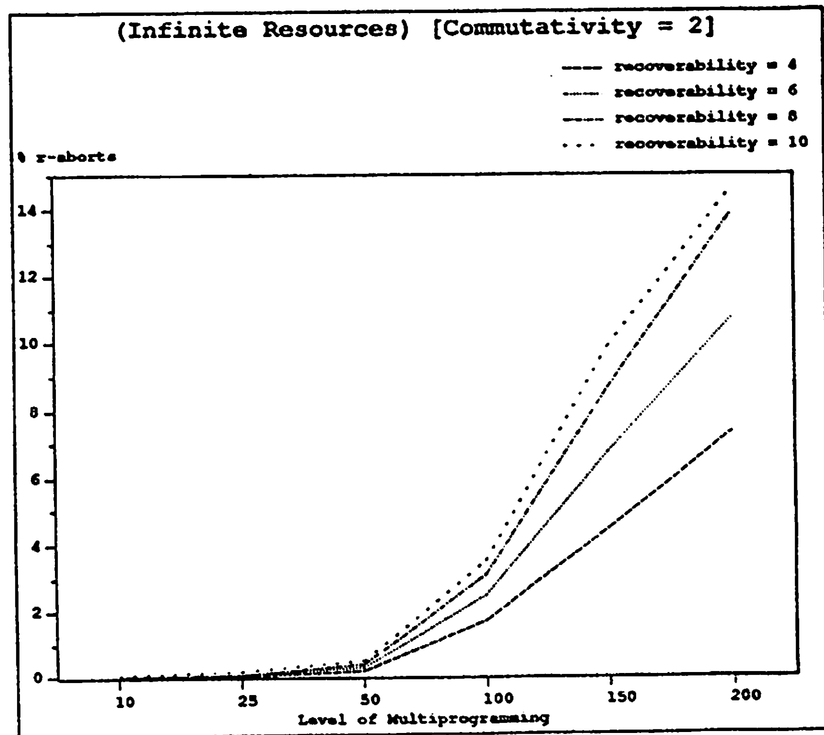


Figure 4.26. R-aborts (infinite resources)

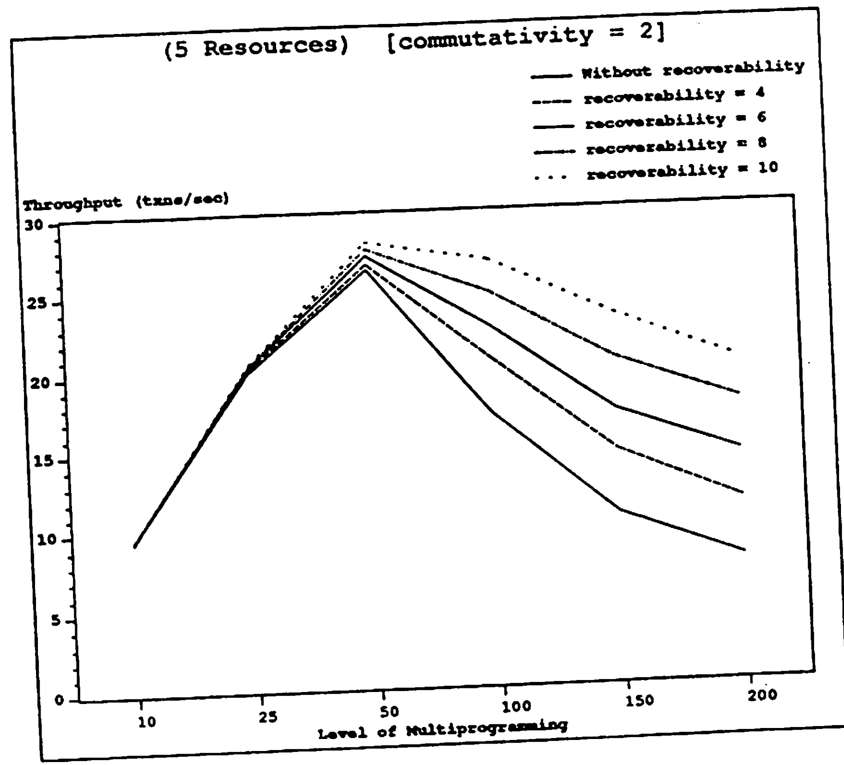


Figure 4.27. Throughput (5 resource units)

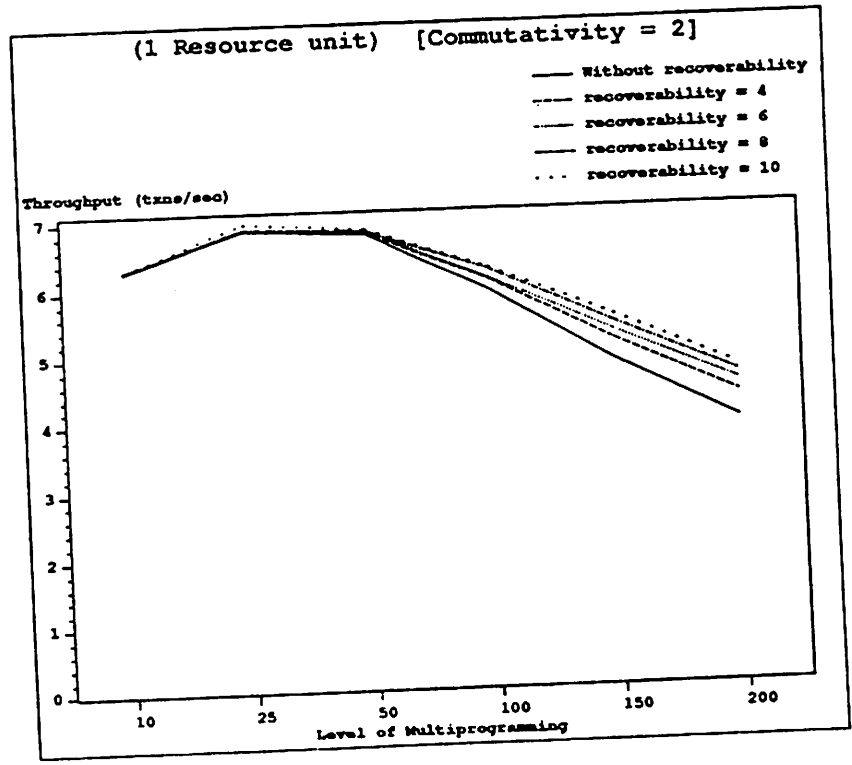


Figure 4.28. Throughput (1 resource unit)

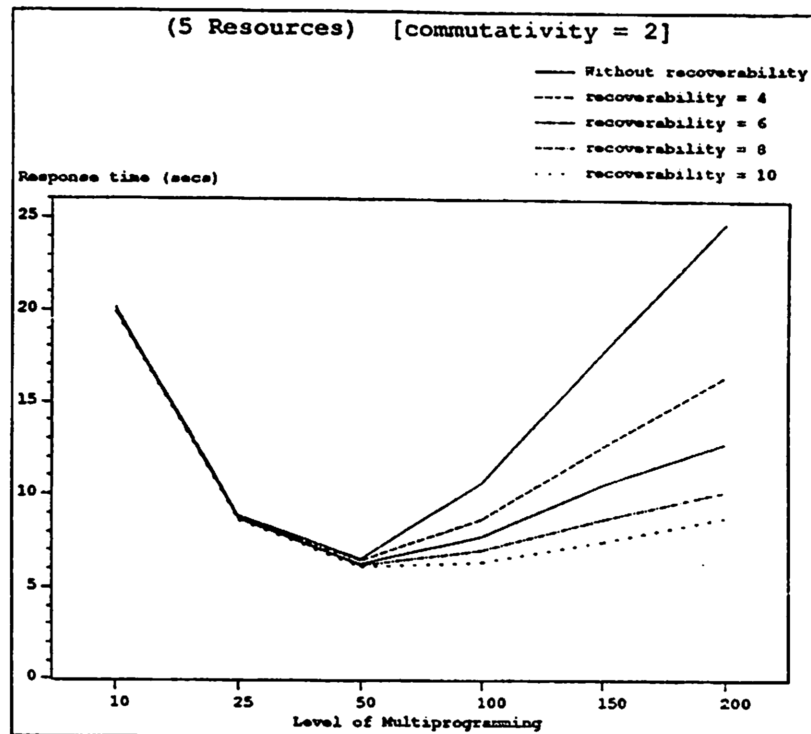


Figure 4.29. Response time (5 resource units)

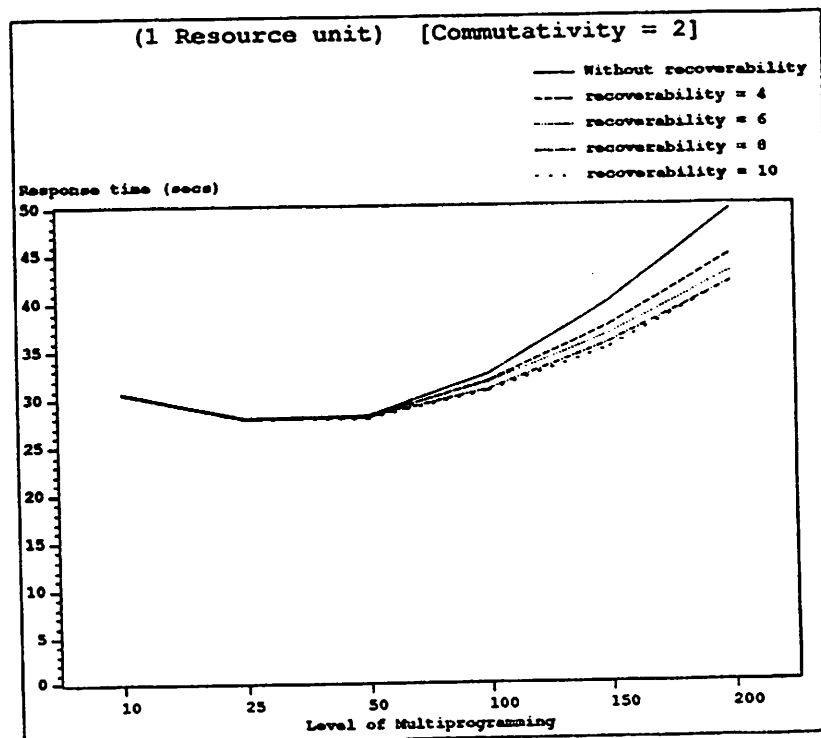


Figure 4.30. Response time (1 resource unit)

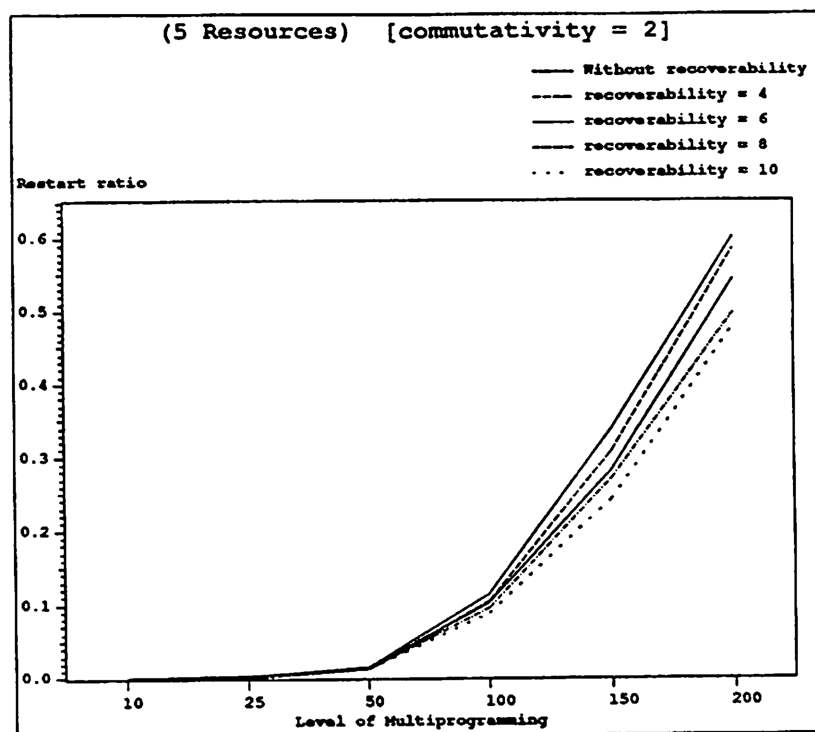


Figure 4.31. Restart ratio (5 resource units)

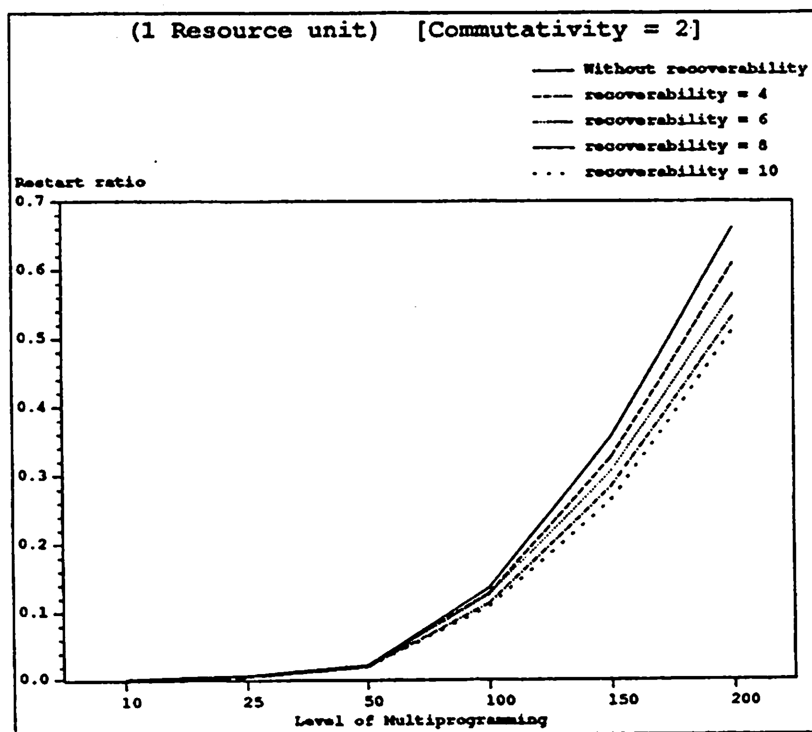


Figure 4.32. Restart ratio (1 resource unit)

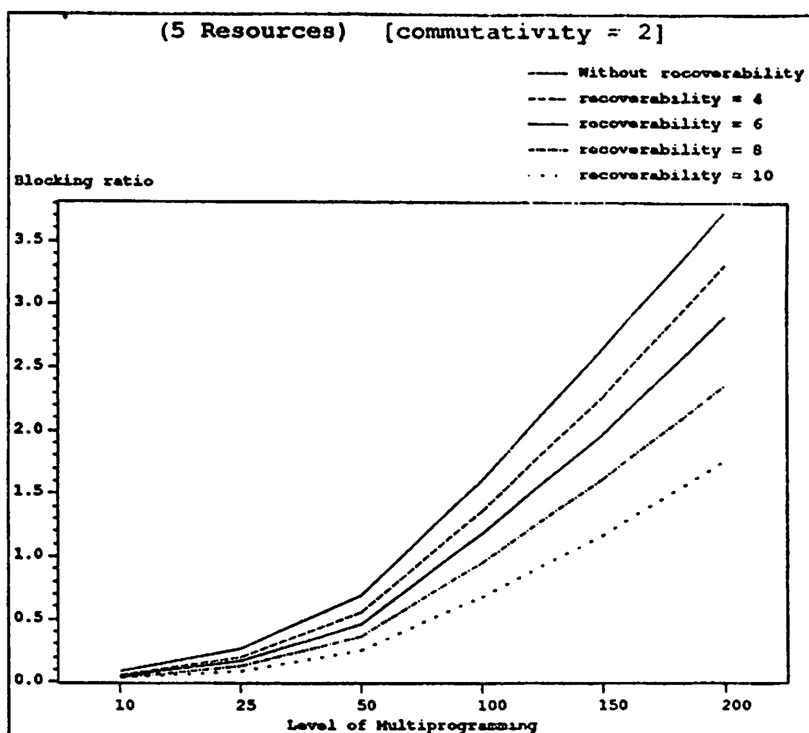


Figure 4.33. Blocking ratio (5 resource units)

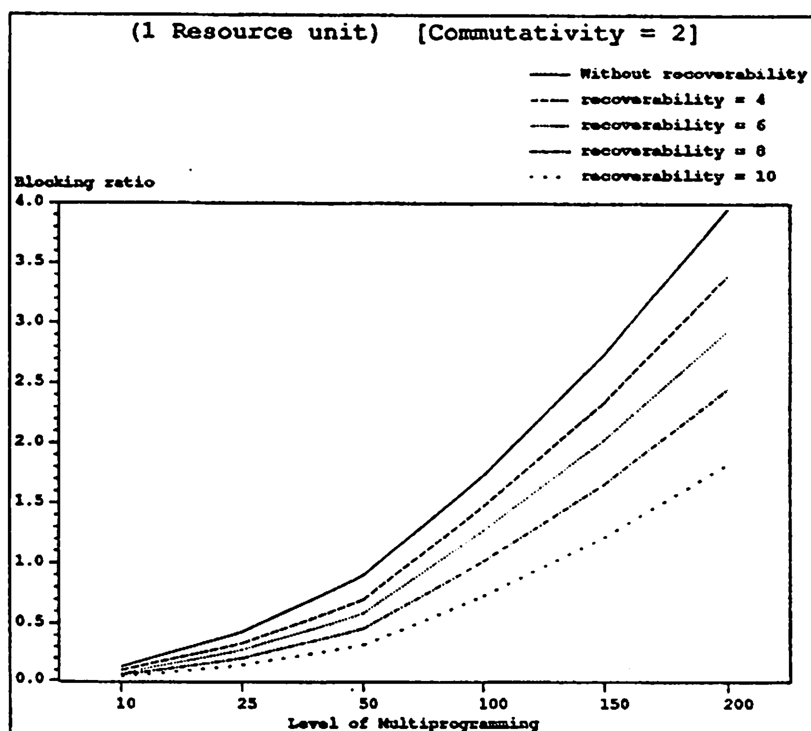


Figure 4.34. Blocking ratio (1 resource unit)

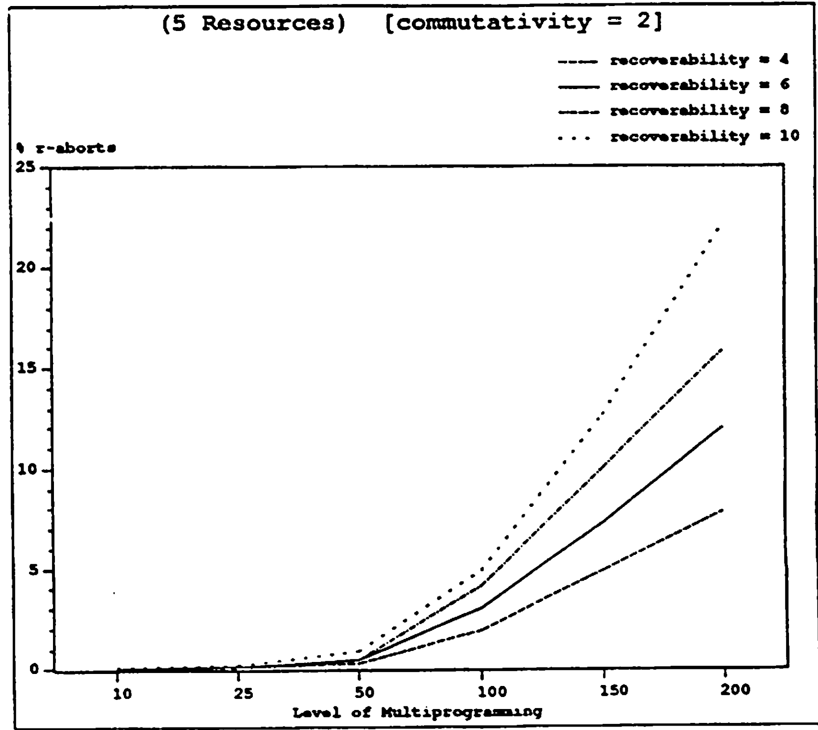


Figure 4.35. R-aborts (5 resource units)

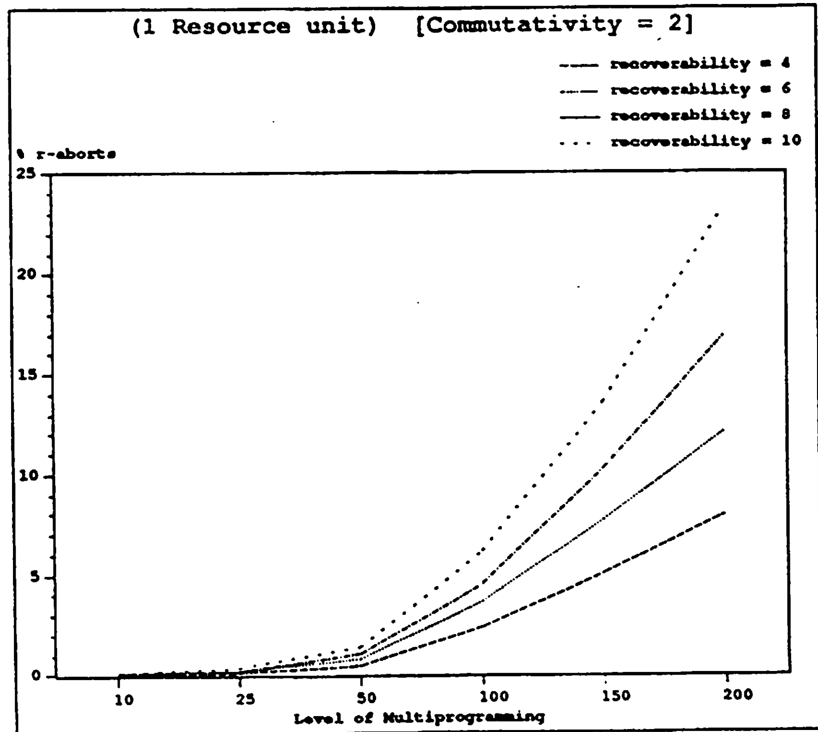


Figure 4.36. R-aborts (1 resource unit)

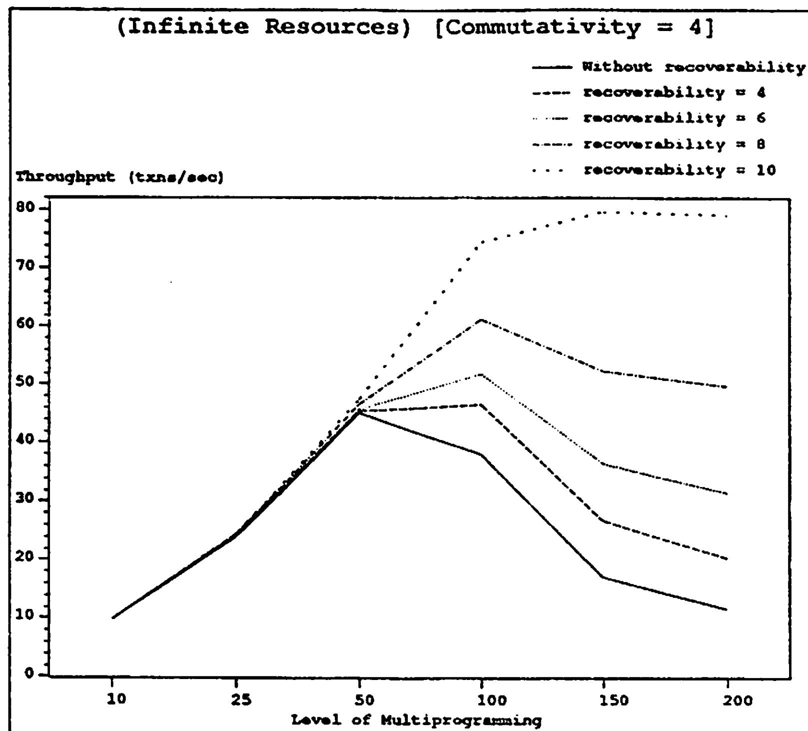


Figure 4.37. Throughput (infinite resources)

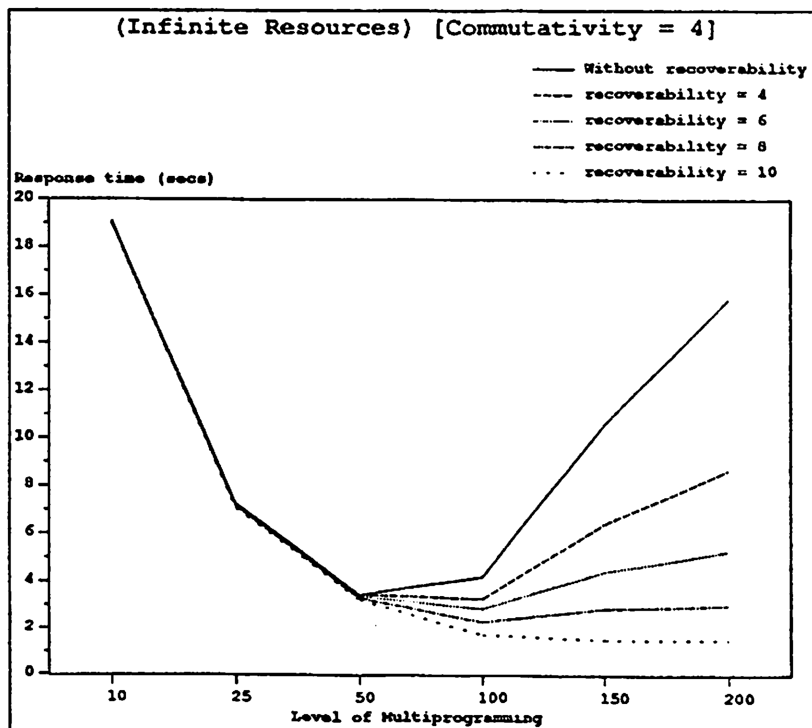


Figure 4.38. Response time (infinite resources)

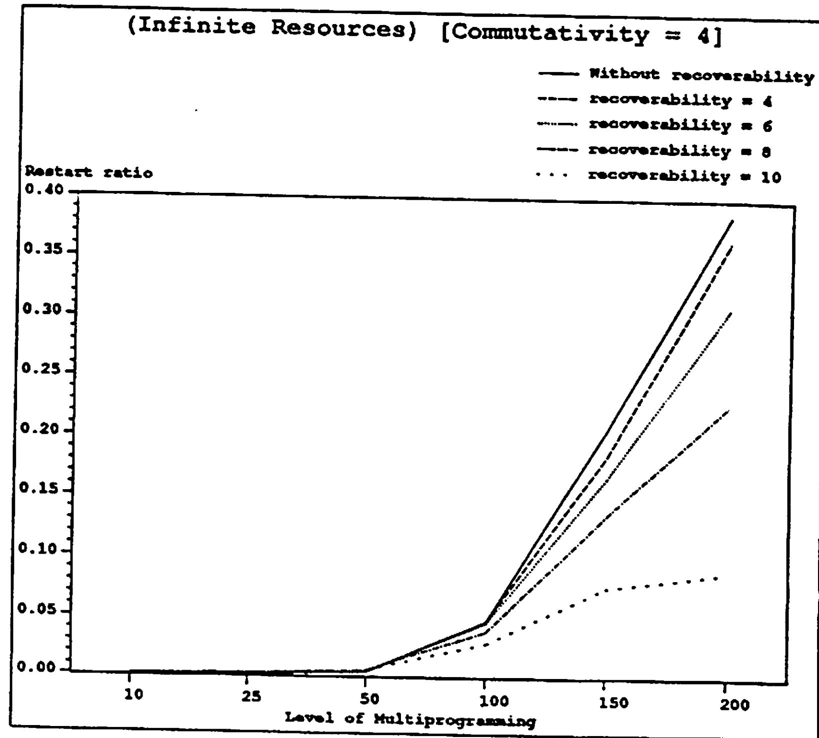


Figure 4.39. Restart ratio (infinite resources)

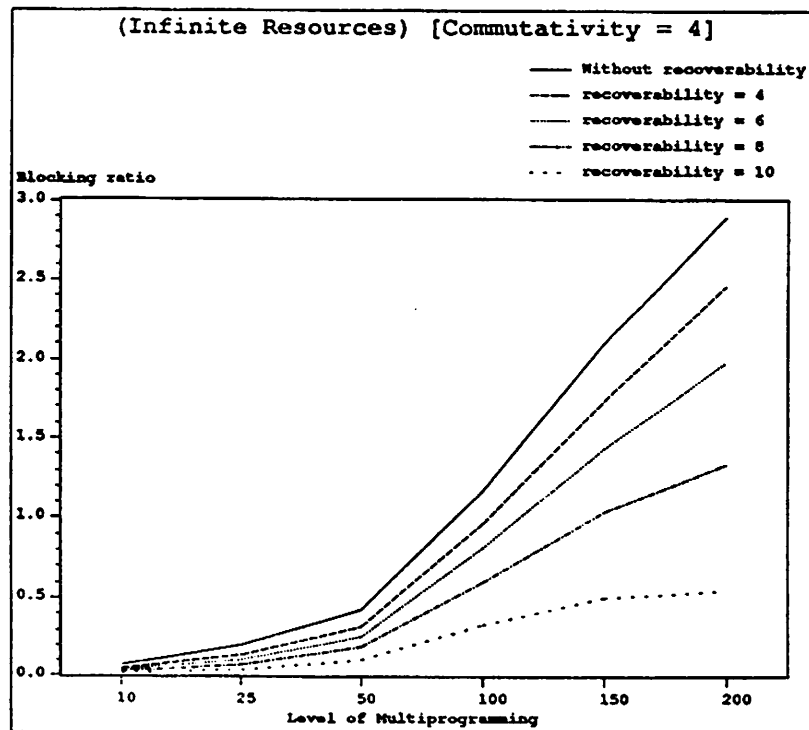


Figure 4.40. Blocking ratio (infinite resources)

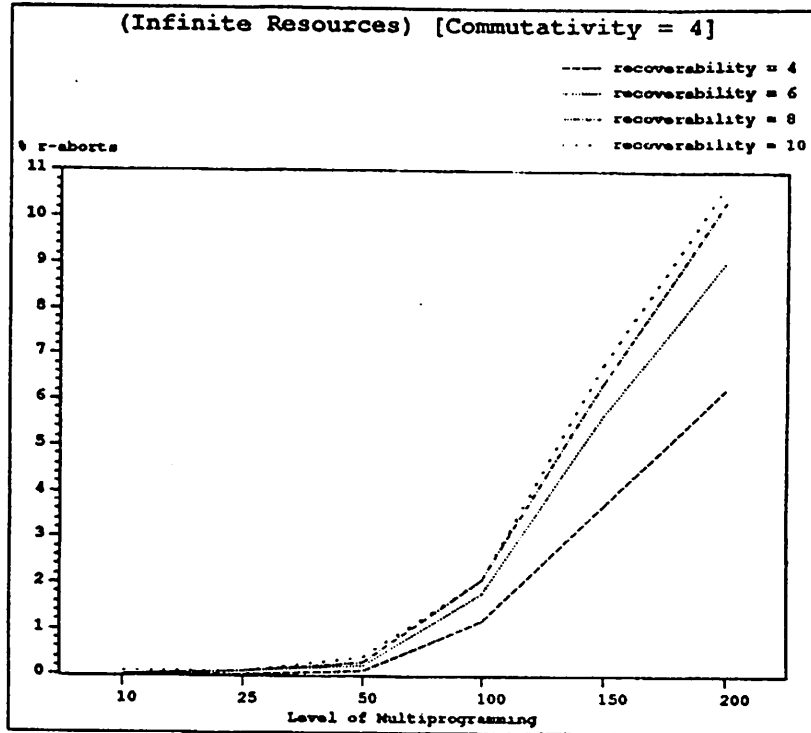


Figure 4.41. R-aborts (infinite resources)

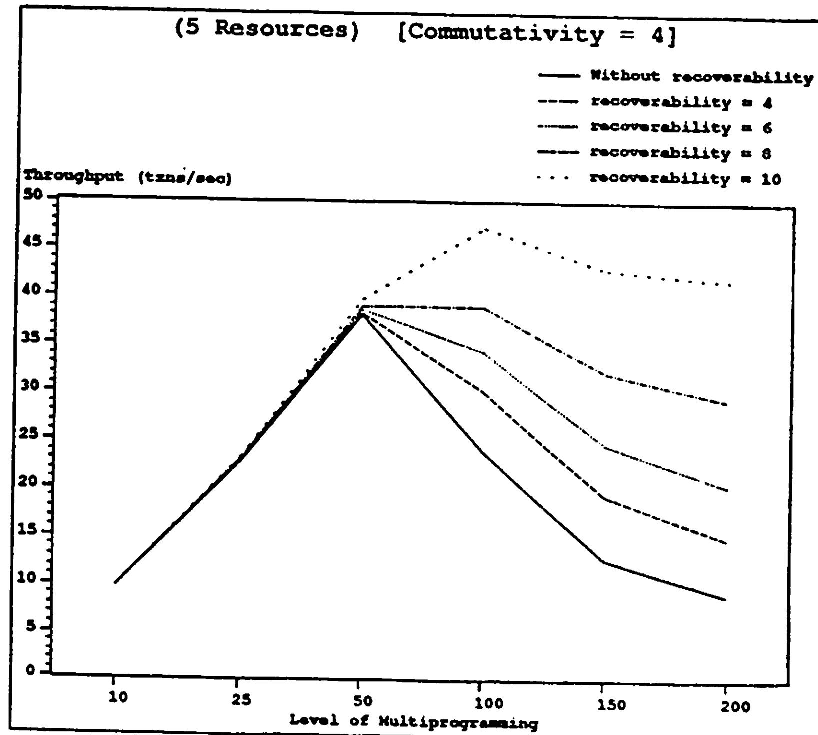


Figure 4.42. Throughput (5 resource units)

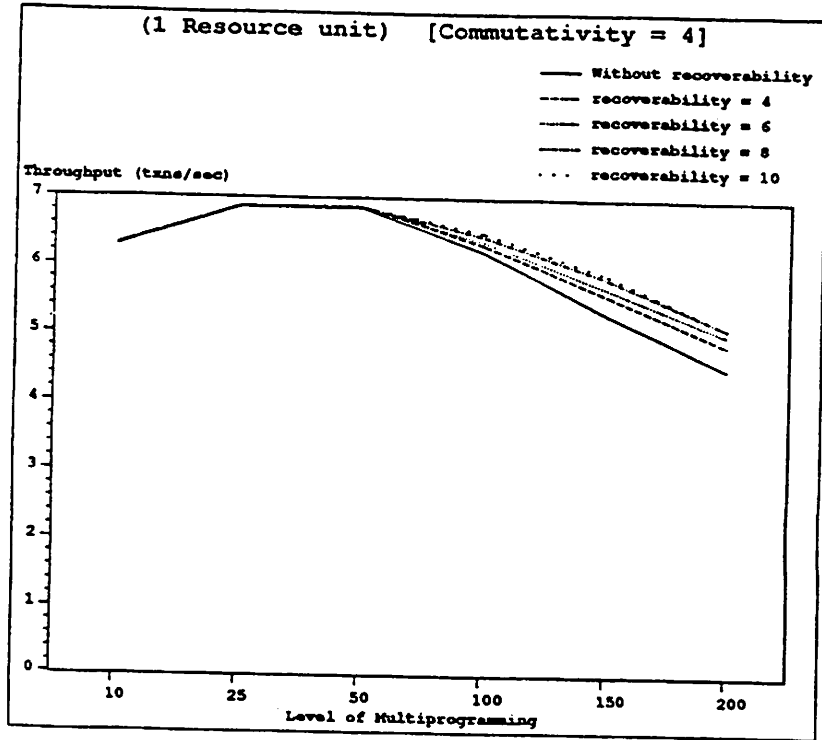


Figure 4.43. Throughput (1 resource unit)

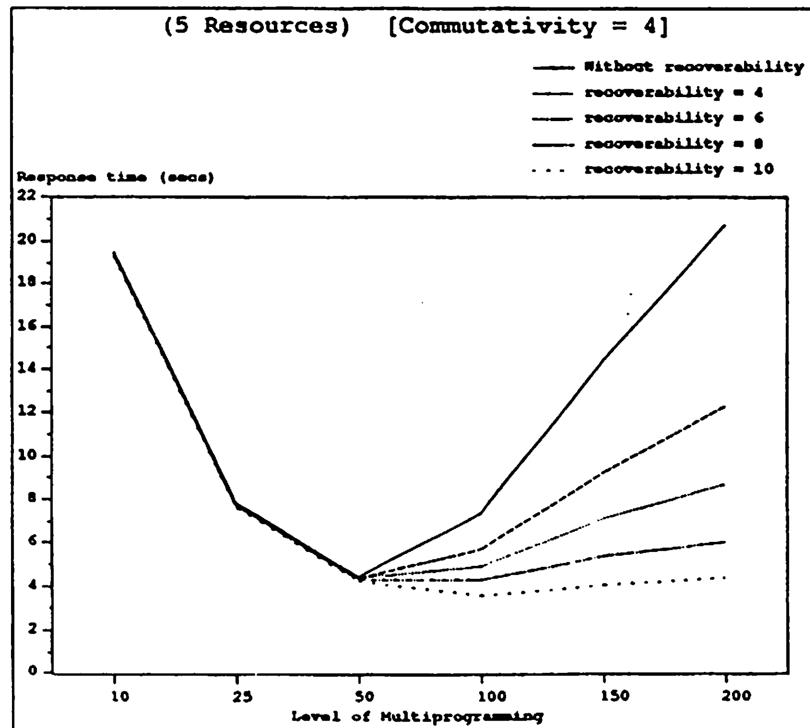


Figure 4.44. Response time (5 resource units)

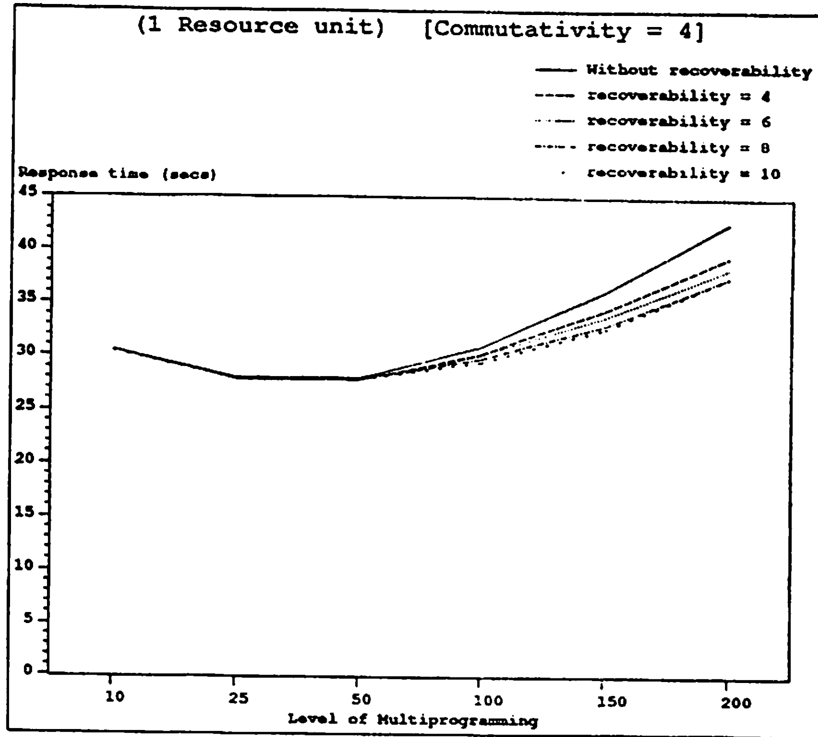


Figure 4.45. Response time (1 resource unit)

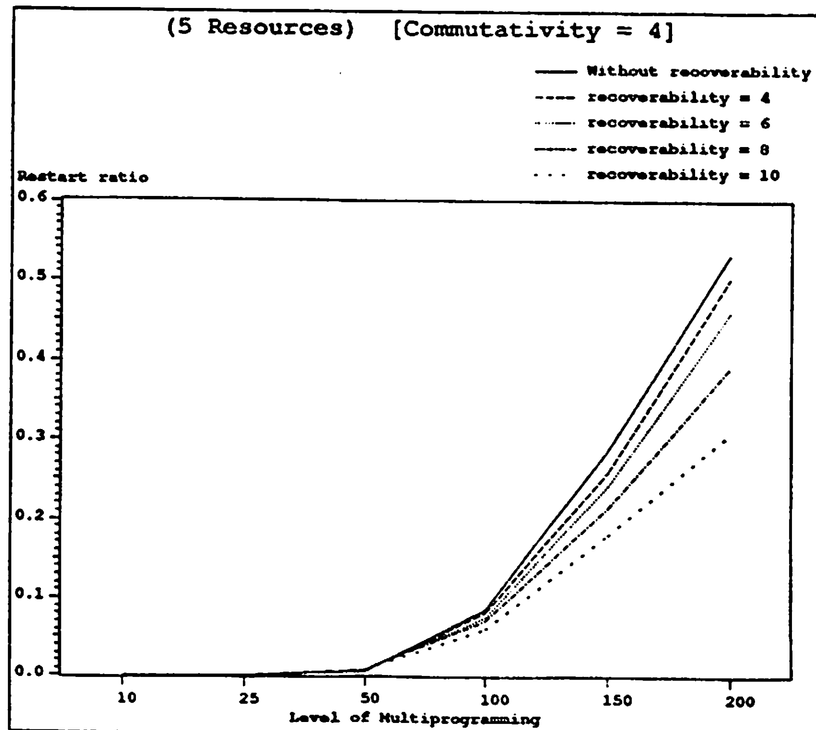


Figure 4.46. Restart ratio (5 resource units)

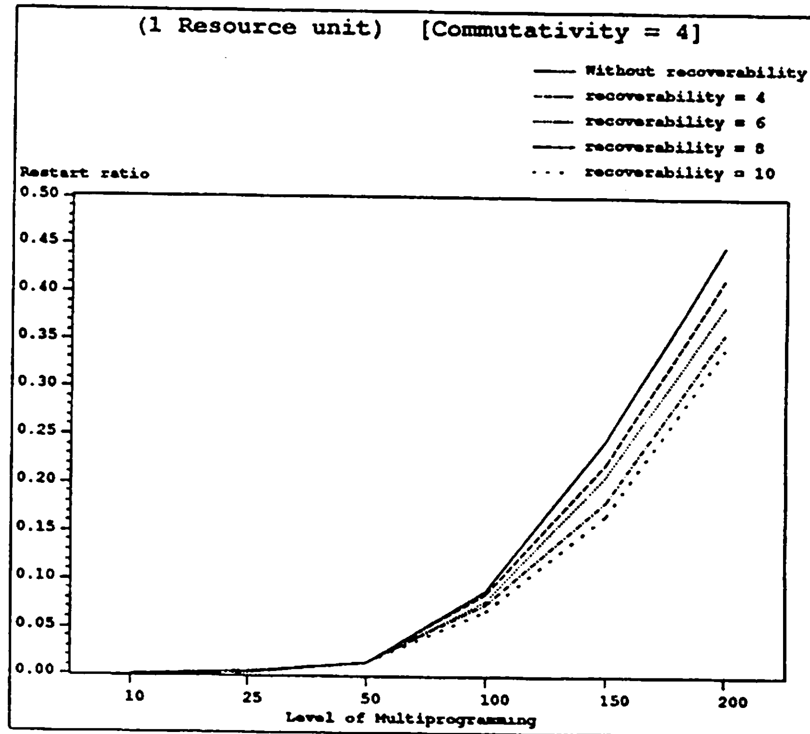


Figure 4.47. Restart ratio (1 resource unit)

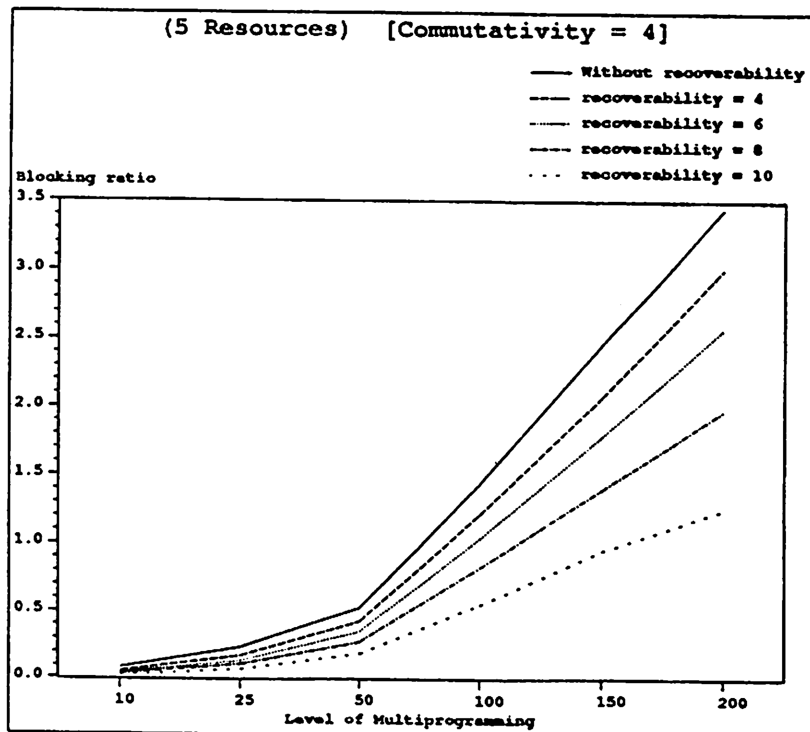


Figure 4.48. Blocking ratio (5 resource units)

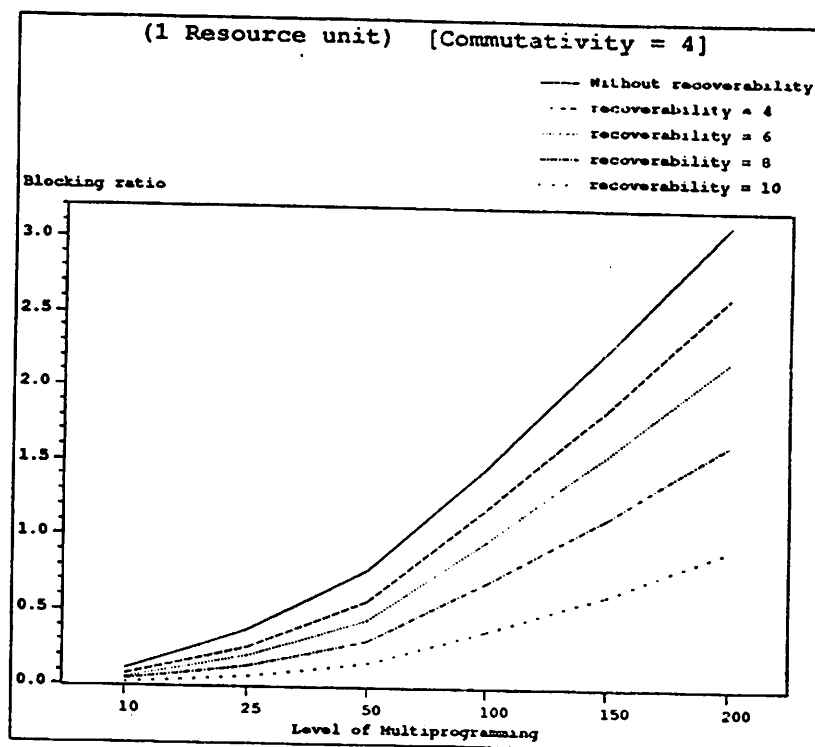


Figure 4.49. Blocking ratio (1 resource unit)

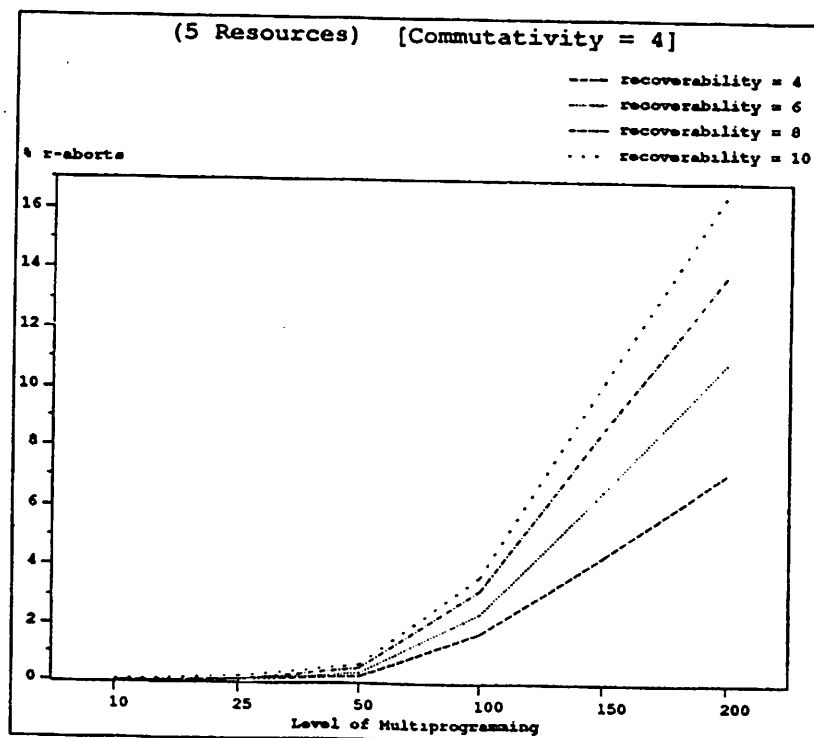


Figure 4.50. R-aborts (5 resource units)

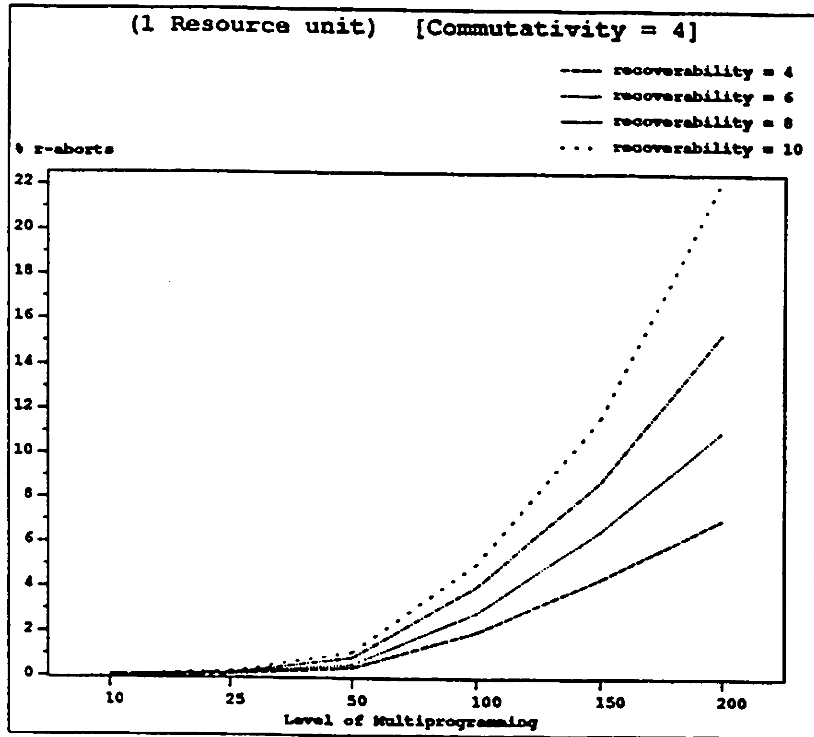


Figure 4.51. R-aborts (1 resource unit)

CHAPTER 5

MULTILEVEL CONCURRENCY CONTROL

In the concurrency control mechanisms described in the last chapter, there was an implicit assumption that the abstract operations were executed atomically as primitives by the underlying system. However, in complex information systems, high level operations are translated into sequences of lower level operations, where each level has its own set of operations. The operations at one level, issue suboperations at lower levels, which can invoke further sub-operations and so on resulting in an *implicitly nested* computation. Thus, in systems supporting implicitly nested operations, we have operations at various levels, and at each level operations have specific synchronization properties. Concurrency control at the leaf level alone (single level), although correct, as we will see, is unnecessarily restrictive. Thus, a multilevel (ML) approach to the problem of concurrency control seems not only *natural* but also *necessary* to meet the high throughput demands in complex information systems. In this chapter, we will see how concurrency can be enhanced when complex operations result in the execution of a set of (lower level) operations.

5.1 Need For Multilevel Concurrency Control Protocols

Consider the example shown in Figure 5.1. Each *increment* operation is composed of a *read* of the initial value followed by a *write* of the new value. In the first computation, a *read* of the second *increment* is interleaved between *read* and *write* of the first *increment*. Since a read and a write operation on the same element conflict, w1(x) conflicts with r2(x) and hence a dependency between these sub-operations of the increments is formed. Similarly, w2(x) conflicts with r1(x) forming a dependency between r1(x) and w2(x). We are interested in the correctness of the computation resulting from the concurrent execution of *increment* operations. Thus concurrent operations should inherit any conflicting dependencies among their sub-operations. The increment operations in Computation I inherit cyclic conflict dependencies implying that the sub-operations were not executed atomically thereby resulting in a lost update. Hence in this concurrent execution, the specifications of the two *increments*

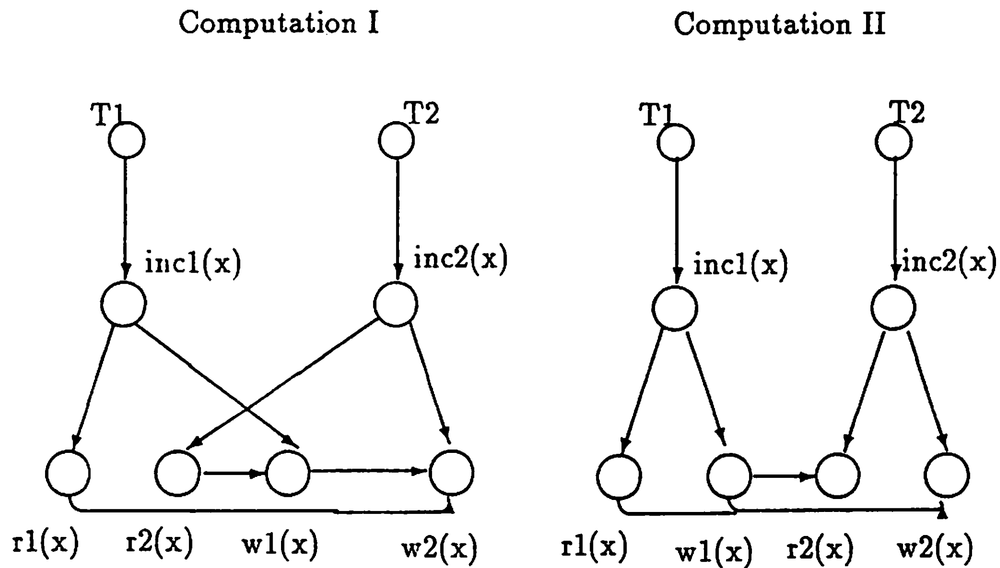


Figure 5.1. Multilevel computation

are not met, and hence Computation I is unacceptable. However, in Computation II, the sub-operations of *increments* are not interleaved, and the increment operations inherit acyclic conflict dependencies. Since two *increments* commute, they do not conflict, and either order of execution is equivalent to a serial execution.

In order to avoid mutual dependencies and allow only serializable executions of top level transactions, nested two phase protocols have been proposed [53, 58, 82]. These multilevel concurrency protocols allow more concurrency than the standard two phase locking applied to the leaf level operations alone. These nested protocols are based on locking: to execute an operation on an object, a transaction must acquire a lock appropriate to that operation. Since operations are composed of nested sub-operations, in these protocols the execution of each operation results in a number of locks being acquired for the sub-operations. A lock is granted if no other conflicting lock is held on that object. A lock acquired by an operation prevents other conflicting operations from being executed until the lock is released. Further, locks at a given level can be released when the parent operation completes without having to wait until the top level transactions complete. Once an operation releases a lock, it cannot request further locks. Finally, the locks corresponding to the top level operations are released by the transaction that initiated the operations. These nested two phase protocols are sufficient to guarantee correctness (serializability) of top-level executions. Further, in these protocols, two operations conflict if they are *non-commuting*.

Because conflict among 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. In Chapter 3, we discussed how the property of recoverability can be exploited to execute non-commuting but *recoverable* operations in parallel. With recoverable operations, 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. In a multilevel system, any given operation may invoke a recoverable sub-operation. Thus, based on the recoverability relationship of an operation with other operations, a parent operation invoking the operation sets up a dynamic commit dependency relation between itself and other operations at its level of execution. If an invoked operation is not recoverable with respect to an uncommitted operation, then the invoking operation is made to wait. In the next section, we will see how we can incorporate recoverability in a multilevel scheme, and automatically maintain dynamic commit dependency relationship.

5.2 A Recoverability Based Multilevel Concurrency Control Protocol

Systems in which high level operations are decomposed into sub-operations at several levels, concurrency control can be achieved by requiring that leaf level operations acquire locks and retain them until the top-level transaction completes. As we have seen in Section 5.1, in these systems a multilevel concurrency control protocol can be followed where the leaf level locks are retained only until the parent operation completes. Thus, locks are retained over a shorter duration in this scheme than in a single level concurrency control scheme. Hence, multilevel protocols potentially provides enhanced concurrency. In these protocols concurrency is controlled at every level. Before executing, an operation at level l has to acquire an appropriate lock. A lock is granted to an operation if it does not conflict with other existing operations. When this operation invokes sub-operations at $l-1$, the sub-operations will also acquire locks appropriate to level $l-1$ and so on. Here we assume that level n corresponds to top-level transactions and level 1 corresponds to leaf-level operations. Similarly, when an operation completes at level l, the locks of suboperations at $l-1$ are released but the lock at level l is retained until it is released by the parent. Thus, the duration of a level l lock acquired by an operation is from the time it is

acquired until the completion of its parent at level $l+1$. Hence, locks are acquired in a top-down manner and released bottom-up.

We will adopt a similar strategy for our top-down nested locking protocol. However, while these concurrency protocols consider operations to conflict if they are not commutative, in our scheme, conflicts between operations are determined based on recoverability. The definition of recoverability, in Section 4.2, assumed that the operations were executed atomically by the system. This assumption is valid for single level systems. However, in the case of a multilevel system we can make that assumption only for leaf level operations. Thus, the multilevel protocols must ensure that intermediate level operations are in effect executed atomically. If operations are allowed to execute concurrently only if they commute, then nested two phase protocols guarantee serializability of top-level operations and atomicity of operations. Even with *recoverability* we need to ensure that the sub-operations are in effect executed atomically and that the execution order is preserved at all levels. However, when recoverable operations are executed by transactions they force commit dependencies; a transaction may commit only after other transactions on which it depends commit. However, the semantics of the execution of the transaction are not affected by the commit/abort of other transactions with which it has a commit dependency. Hence an operation can *complete* execution; with the exception that the operations can commit (release locks) only after all operations on which it depends terminate, thus respecting the commit dependency relationship. In order to keep track of commit dependencies, recoverable operations will have to acquire *lists* that contain the IDs of the operations with which a given operation is recoverable. Before describing how both atomicity of operations can be guaranteed and execution order of operations can be preserved at all levels, some definitions are needed.

Definition 22: An operation is said to have *completed* once it releases all the locks and lists acquired for the sub-operations (i.e., all sub-operations have completed). An operation is said to be *in-progress* if any of its sub-operations has not been completed.

Definition 23: Two leaf level operations o_i and o_j are non-conflicting if o_i and o_j commute or o_i RR o_j.

Definition 24: Two non-leaf level operations are non-conflicting if either 1) o_i and o_j commute or 2) o_i RR o_j and o_j has *completed*.

Note that in the above definition, for intermediate operations, recoverability is used only with respect to *completed* operations i.e., those operations whose all sub-

operations have completed. This ensures that when a recoverable operation is executed, the state changes and the return values are effected in the order of execution i.e., preserving the order of execution.

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

Here are the rules of the nested protocol.

1. If an operation request at a given level conflicts i.e., is *non-recoverable*, it is delayed until conflicting operations release their *locks* or *lists*.

2. If an operation request is non-conflicting, either

   a) the operation acquires a *lock* if it commutes with all existing operations or

   b) the operation acquires a *list* if the operation either commutes with other operations or it is recoverable with other completed operations (or for leaf level it is recoverable with other operations), where the *list* contains the ID's of the operations with which the given operation is recoverable.

3. Once an operation releases a *lock* or a *list* it may not acquire any more *locks* or *lists*.

4. A *lock* acquired for an operation can be released any time, however, a *list* can be released only when it is empty.

5. When a *lock* or a *list* corresponding to an operation is released, its ID is removed from all *lists* in which it is a member.

The first 3 rules are the same as in the nested two phase protocol of [53, 58, 13], with embellishments to track dependencies due to recoverability. Rule(4) guarantees that operations complete in the order in which they form dependencies due to recoverable operations. Note that in rule (2b), a non-leaf level operation is considered conflicting if there is an in-progress operation with which it does not commute. This is because, the notion of recoverability for non-leaf level operations is applied only with respect to completed operations.

We will present a rough sketch of how we would prove the correctness of our protocol. A formal proof appears in Chapter 6. Note that rule(4) guarantees that a given operation will complete (release all locks) only after the completion of any operation with which it is recoverable. Since two operations with mutual commit

dependencies cannot complete, there can be no cyclic conflict dependencies at any level of the computation resulting from the use of recoverability as a conflict predicate. Thus, at each level there are no inherited mutual dependencies due to conflicting sub-operations. Further, at non-leaf levels, recoverability, as a notion of conflict, is applied with respect to operations that have completed. Thus, if two operations were considered as non-conflicting because they were recoverable then all the sub-operations of one operation were completed before any of the sub-operations of the other was invoked, thereby preserving the order of execution of the operations.

As in the case of single level concurrency control, transactions can invoke operations on several objects. This leads to a problem: We must ensure that the executions on different objects agree on at least one serialization order for the committed transactions. To determine whether the execution is serializable we have to ensure that the commit dependency relationship is acyclic. This is similar to the validation phase in optimistic protocols [43]. If a transaction wanting to commit is in a commit dependency cycle then that transaction is aborted. In the next section, we will see how we can further enhance the concurrency obtained by our protocol by exploiting information about the operation structure.

5.3 Exploiting Operation Structure in ML Concurrency Control

The nested protocol, described in the previous section, uses recoverability as a notion of conflict, and the nested two phase protocols proposed for multilevel concurrency control in [58, 83, 54], schedule operations based on conflicts defined between operations at each level. Let us first look at what happens when the notion of conflict defined at each level is used in these protocols.

Consider Computation I in Figure 5.1 following a nested two phase protocol. When r2(x) is requested, a read lock is granted but this read lock forces w1(x) to wait. In the conventional sense this can be termed a deadlock but it is interesting to point out that these cyclic waits were generated not by user requests but due to requests stemming from the decomposition of operations into sub-operations within the system. Further, the possibility of cyclic waits exists at every level! Thus, in the context of implicit nested environments, the idea of executing non-conflicting operations immediately may not be a good idea after all.

The problem of deadlock among sub-operations of non-conflicting operations has been ignored by most researchers. A better solution would be to prevent a mutual

dependency without getting into deadlocks or cyclic waits. Thus, a solution would be to make r2(x) wait until w1(x) completes and then schedule r2(x). This is what happens in Computation II of Figure 3.1. In this case we say that r2(x) has a *relative conflict* with r1(x), i.e., r2(x) and r1(x) conflict relative to the parent of r1(x). Making r2(x) wait even though it does not conflict with r1(x) in the conventional sense has two significant implications: first, once w1(x) is allowed to complete, a lock corresponding to an increment operation can be retained on x thereby releasing read and write locks on x, and secondly, cyclic waits or deadlocks can be prevented. Under what circumstances is it useful to make a non-conflicting operation wait? What properties of the operations allows us to make such a decision? These are some of the questions we will answer below.

As the above example illustrates, in the context of complex operations, we need to reexamine the conventional wisdom of executing non-conflicting operations immediately. Instead of using conflicts between operations as a basis for scheduling operations, we will schedule operations based on a new notion called *relative conflict*, i.e., conflict between suboperations *relative to* the operations that invoked them. We will examine the impact of this strategy on concurrency in implicitly nested systems. The question now is "how do we define *relative conflict*?" The information about proscribed interleavings of the sub-operations will be used to define the notion of *relative conflict*. This information can be obtained if the structure of the abstract operation is known so that the look-ahead information can be used to determine the siblings of a given sub-operation. In the past, this view of anticipating future steps in single level systems has been in some sense considered "unnatural" [61]. However, in multilevel operations this sort of information is readily available.

In the multilevel computation of Figure 3.1, the sub-operations of *increment* operations should be non-interleaving for correctness. We call such operations, whose sub-operations cannot be interleaved arbitrarily, as having *inprogress* $\times$ *invoke* conflict. This implies that for such operations it is better to wait for all sub-operations of the *inprogress* operation to complete before executing any sub-operations of the *invoking* operation. Thus, the scheduler at the level of *reads* and *writes* will not allow another *read invoked by an increment operation* to be executed concurrently until the *write* of the first *increment* is completed. If the sub-operations of two non-conflicting operations can be interleaved in any order possible, then the operations are said to have no *inprogress* $\times$ *invoke* conflict. Thus we can define *relative conflict* as follows: two non-conflicting operations o_1 and o_2 have relative conflict iff parent(o_1), and parent(o_2) have *inprogress* $\times$ *invoke* conflict.

Table 5.1. Conflict table for counter

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

Table 5.2. Relative conflict for counter

Operation Requested	Operation Executed		
	Read-V	Read-I/D	Write
Read-V	Yes	Yes	No
Read-I/D	Yes	No	No
Write	No	No	No

By means of a compatibility tables, we will illustrate the type of relative conflict that exist between various operations of an object such as counter. A counter object provides three operations: *increment, decrement, and value.* Increment adds one to the value of the counter. The decrement operation decreases the value of the counter by one, and the operation value returns the current value of the counter. Value is implemented as reading the value of the variable representing the counter. An increment or a decrement operation is implemented as a read operation followed by a write operation on the variable representing the counter. The conflict based compatibility table for *reads* and *writes* is shown in Table 5.1. According to this table, two read operations do not conflict. However, two read operations of increment or decrement operations (indicated as Read-I/D) are in relative conflict in Table 5.2, as executing them concurrently will inevitably lead to deadlock because write operations will be executed subsequently.

When synchronization is done at several levels, at each level the scheduler uses look ahead information encoded in terms of the class to which the parents of the operations belong. Based on this idea, we have developed a nested protocol, called ML-RC, that makes use of *relative conflict* information obtained from classification of non-conflicting operations. Scheduling decisions at level i are not only made on the basis of conflicts between level i operations but also on the basis of the class to which the parents (at level $i+1$) belong. Let us first consider commutativity-based conflicts. In this case, pairs of operations which do not conflict, i.e., which commute, are further classified as operations having or not having $inprogress \times invoke$

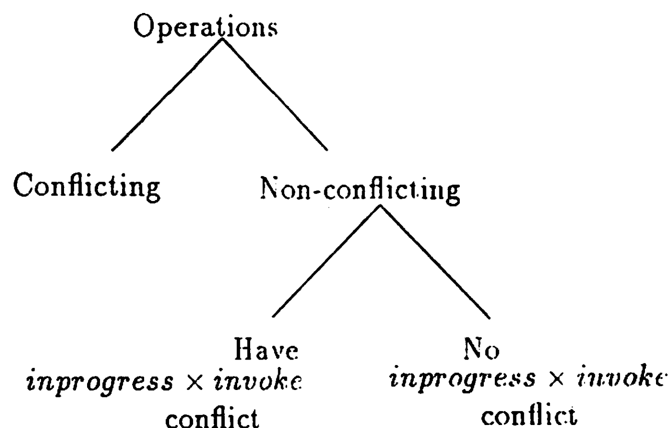


Figure 5.2. Classification of Operations

conflict as shown in Figure 5.2. Note that the classification of operations based on *inprogress* × *invoke* conflict is dependent only on its *immediate* sub-operations. This *level by level* classification of operations is then used to schedule multilevel operations. Further, we will apply the definition of relative conflict only to operations and sub-operations and not to transactions. Assuming that relative conflict is applicable to transactions implies that transactions have statically declared their operation sets, an unnatural assumption to make in most applications we are interested in. Thus, the notion of relative conflict uses information only about *operation structure* and *not transaction structure*.

The nested protocol, ML-RC, is similar to the nested protocol described earlier in Section 5.2. Operations issue sub-operations and locks are granted for these sub-operations depending upon the type of conflicts. Once an operation releases a lock it may not request any more locks (i.e., may not issue further sub-operations). This is the standard two phase algorithm applied to nested operations. In addition to acquiring and releasing locks appropriate to an operation we introduce a special type of state that can be entered by an operation, namely the *completed state*.

Definition 25: An operation is in a completed state w.r.t a particular object, if all the sub-operations on that object have been invoked.

An operation after entering the completed state w.r.t a particular object may not request any more sub-operations *on that object*. Thus, we not only have a release phase for locks but also an acquire phase for locks on individual objects. The reason for requiring an operation to indicate a completed state w.r.t an object, after which it may not request any more sub-operations on that object will be clear when we discuss

the effect of using *relative conflict* as a conflict predicate for scheduling sub-operations. The salient rules for our nested protocol ML-RC are as follows:

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

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

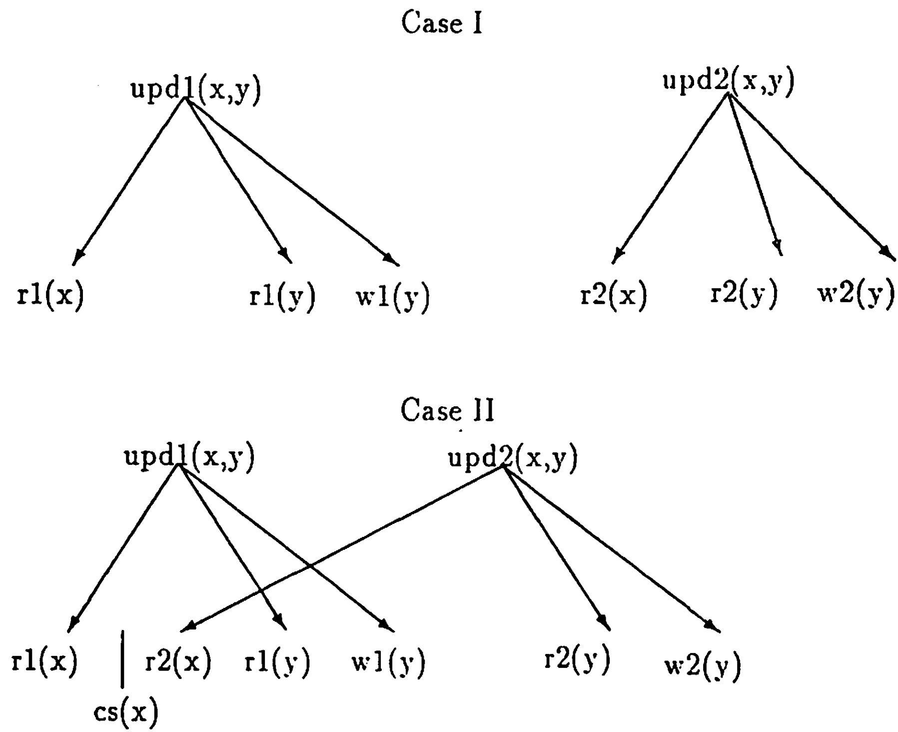
3) Once a *completed* state is reached by an operation on an object, the operation may not request any more locks on that object.

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

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

In the nested protocol ML-RC, operations are scheduled on the basis of relative conflict, and each operation after entering a completed state w.r.t an object cannot request any more locks on that object. The reason for this is additional requirement is as follows: A given operation may have sub-operations on multiple objects. Without the notion of a completed state for a particular object, two operations having relative conflict will end up executing all their sub-operations serially. However, the sub-operations of one operation can be interleaved between sub-operations of the other on different objects without getting into cyclic waits. In order to avoid the situation of having to execute all sub-operations serially (e.g., Case I in Figure 5.3) we have introduced the notion of completed state on objects. Once a completed state is reached for an object by an operation, according to the rules of the protocol, no more sub-operations can be issued by that operation on that object; thus, non-conflicting sub-operations can be allowed to execute without getting into cyclic waits. This is shown in Case II of Figure 5.3, where operation upd1(x,y) has reached completed state on object x, cs(x), and hence r2(x) can be allowed to execute.

We would like to note that, the nested protocol, ML-RC, based on *relative conflict* has significant implications not only for scheduling complex operations but also for



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

Figure 5.3. Nested protocol ML-RC

concurrency control in a *traditional database* such as System R[4]. System R uses page level and record level locking. Thus we can consider synchronization to be occurring at multiple levels. For example a read of a record is implemented as fetching the page to which the record belongs. The write of a record is implemented by fetching the page in which the record is stored and then storing the updated value in the page. Thus, our scheduler, using relative conflict, at the page level will not allow two fetches to be executed concurrently when one of the fetches is the child of a write operation on the record. This is because the scheduler knows that fetch for a write will be followed by a store.

5.4 Simulation Studies

We now report on simulation studies designed to evaluate the increased concurrency resulting not only from the use of of recoverability but also from adopting a multilevel approach to concurrency control. The purpose of this simulation study is two fold: First, compare the amount of concurrency offered by a single level concurrency control to that of a multilevel concurrency scheme. Second, compare the amount of concurrency offered when both commutativity and recoverability are used to determine conflicts as opposed to using just commutativity. Thus, we will evaluate the performance of three protocols: the first protocol, SL, is the single level protocol where transactions acquire locks for leaf level operations and retain them until the transaction completes. The second protocol, ML-C, is the multilevel protocol based on nested two phase locking that uses only commutativity in determining conflicts. The third protocol, ML-RC, is our new multilevel protocol that uses recoverability in addition to commutativity in determining conflicts and relative conflict for scheduling operations.

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

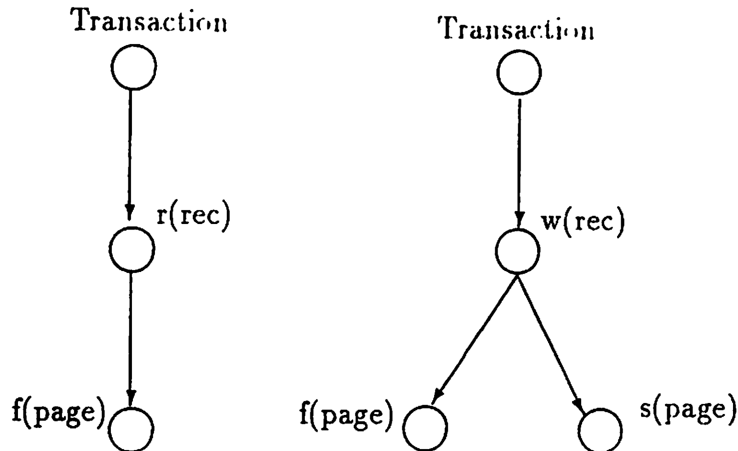


Figure 5.4. Three level system

To simplify simulation studies, we examine a three level system shown in Figure 5.4. The first level constitutes transactions, in the second level the operations are restricted to be reads and writes on records, and in the third (leaf) level the operations are fetches and stores on pages. Each operation request by a transaction is either a read or a write operation on a record. A read operation on a record is implemented as fetching the page to which the record belongs. A write operation is implemented as fetching the page to which the record belongs and storing the new value on that page.

5.4.1 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 5.5 is a modified version of the one used in [1].

There are a fixed number of terminals from which transactions originate. The maximum number of active transactions at any given time in the system is the multiprogramming level, the *mpl.level*. A transaction's length is determined by the number

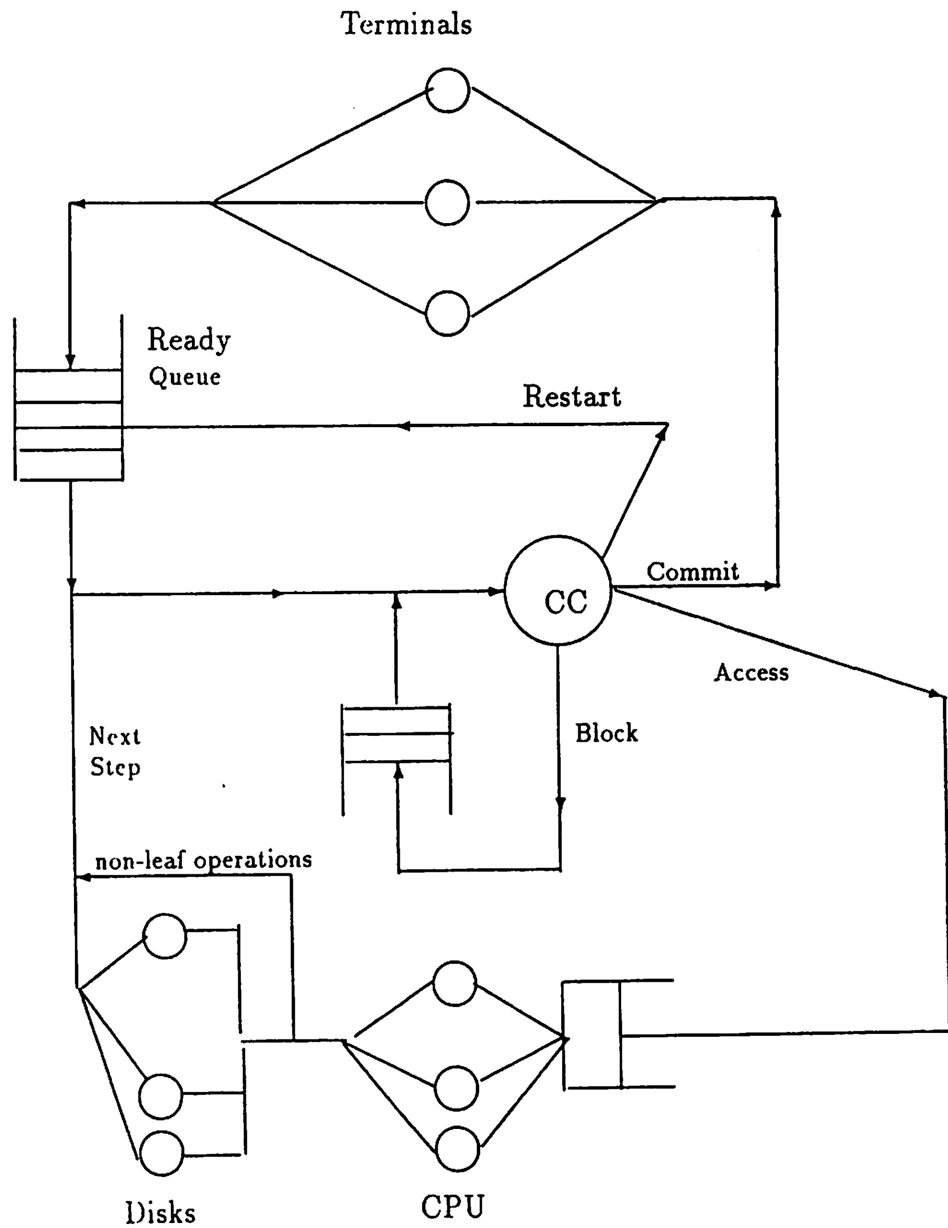


Figure 5.5. Simulation model

Table 5.3. Simulation parameters

Parameter	Meaning
Num.of.levels	Number of levels in the system
Database size	Number of objects in the database
Num.of.terminals	Number of terminals
Transaction length	Mean transaction length
Max.length	Maximum operations in a transaction
Min.length	Minimum 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

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. The deadlock detection algorithm uses both the wait-for graph and the commit-dependency graph maintained as lists to determine a whether a deadlock exists, as deadlocks may occur due to both wait-for dependency and commit dependency. Transactions also abort due to cyclic commit dependencies arising from executing recoverable operations. An aborted transaction is restarted immediately. A restarted transaction behaves, with respect to operation invocations, like a new, independent transaction.

In the case where finite resources are present, each non-leaf step requires a CPU for an interval of length *cpu.time* and a leaf-level step requires a CPU for an interval of length *cpu.time* and a disk access for an interval of length *io.time*. The total time for which these resources are used is equal to *step.time*. The parameter *step.time* is the execution time of each operation. Under the assumption of *infinite* resources, this represents a constant service time for each operation. We consider a CPU and two disks to constitute one resource unit. The number of resource units is a model parameter *resource.unit*. When a transaction needs a CPU, it is assigned a free CPU from a pool of CPU's; otherwise the transaction waits until one becomes free. For the I/O part, there is a separate queue associated with each disk. When a transaction needs to access a disk, it chooses a disk at random and waits in the queue of the selected disk until it can be served [1].

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 fixed delay given by the parameter *commit.delay*. The various model parameters and their meanings are listed in Table 5.3.

5.4.2 Experiment Information

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

Table 5.4. Conflict table for records

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

Table 5.5. Relative conflict for pages

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

nested protocols described in Section 5.1. 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 accepted if it is non-conflicting. 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. The compatibility table at the level of records is shown in Table 5.4. Further, the scheduling of operations at the level of pages is based on relative conflict.The relative conflict for operations on pages is shown in Table 5.5, where Fetch-R indicates it is a fetch operation invoked by a read operation and Fetch-W indicates that it is invoked by a write operation on a record.

A blocked transaction is *retried* every time any transaction that issued a conflicting operation on that object completes. If an operation request is granted, after the execution time of the operation, the operation invokes sub-operations. At each level the same procedure is followed as suboperations are treated as transactions executing at that level. At commit time, using the lists associated with each operation, the scheduler will abort a transaction if it is found to be in a cycle of commit dependencies.

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.

5.4.3 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 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 a transaction at non-leaf levels requires 0.0075 secs of CPU time and each step at leaf level takes 0.0075 secs of CPU time and 0.035 secs of disk access time. Thus, in the case of infinite resources for two levels each read operation takes 0.05 secs and a write operation takes 0.0925 secs (0.05 + 0.0425) as it has two sub-operations; namely, a fetch and a store.

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. A waits-for graph is maintained by an deadlock detection algorithm within the simulator. A deadlock detection is performed each time a transaction blocks. If a deadlock is detected, the transaction invoking the algorithm is chosen as the victim and restarted. We term such aborts t-aborts.

We do not model the details of the overhead in the commit process; However, we take into account the cost of the commit process by introducing a delay for each commit of a transaction. 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.

The nominal values of the parameters are listed in Table 5.6. The values of the model parameters have been chosen similar to those in previous performance studies of locking protocols [78, 77, 1].

Table 5.6. Parameters and their nominal values

Simulation parameters	
Parameter	Value
Num.of.levels	3
Database size	1500 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
Time.out	5 secs
Step.time for read	0.05 secs
Step.time for write	0.05 secs + 0.0425 secs
CPU.time	0.0075 secs
IO.time	0.035 secs
Resource.units	1, 5, and ∞
Ext.think.time	1 secs
Commit.delay	0.6 secs
Write probability	0.3

5.4.4 Performance Metrics

The two main performance metrics used in our evaluation are the *throughput rate* and the *response time* (turnaround time). The throughput rate is measured as the number of transactions that *complete* per second. The response time in seconds is measured as the difference between when a terminal submits a transaction and when a 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.

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 in the system. The *restart ratio* is defined as the average number of times a transaction is restarted per commit. A transaction is restarted if it blocks and is in a deadlock. We call such aborts as t-aborts. Recall that one of the transactions in a commit dependency cycle is aborted and then restarted. We call such aborts as r-aborts. Further, restarts may occur due to the above conditions occurring at any level. A restart at any level always involves restarting the top level transaction. The restart ratio includes the restarts due to r-aborts and t-aborts. Lower the restart ratio, less the work wasted, and hence better utilized is the system.

5.4.5 Simulation Results

We will study the performance of three different protocols; namely, SL, ML-C, and ML-RC. In this study, we measured various factors including transaction response time, throughput, blocking ratio, and restart ratio. The graphs in Figures 5.6 through 5.13 show the average results of 10 runs, where each simulation was run till 50000 transactions were completed to obtain sufficiently tight 90 percent confidence intervals. Though, the confidence intervals are omitted from our graphs, the confidence intervals were within the range of a few percentage points of the mean value of the performance metrics shown in the various graphs. The maximum value of the confidence interval was ±2% points.

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 [78], there is uniform access, that is the probability that a transaction chooses an object at each level to execute an operation is a uniformly distributed random variable between 1 and number of objects at that level. In this study, the number of pages (at level 1) was chosen to be 500 and the number of records (at level 2) was chosen to be 1000 yielding a database size of 1500. This database size was chosen to yield good conflict rates so that interesting evaluation of recoverability based concurrency control scheme can be obtained.

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 5.7. Commutativity for read/write

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

Table 5.8. Recoverability for read/write

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

table will be one of *commutative, recoverable*, or *null*. Further, within commutative entries, each entry is either a No or a Yes depending upon whether the operation has a relative conflict or has no relative conflict.

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

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 5.7, there are three pairs of conflicting operations.

However, with recoverability there is only one pair of conflicting operations in (read, write) as (write, read) and (write, write) are recoverable, as shown in Table 5.8. Further, the *fetch* sub-operation of a write operation has a relative conflict with fetch operation of another write as shown in Table 5.5, and hence considered as conflicting in our protocol ML-RC.

Infinite Resources: In this part of the simulation, we assume infinite resources. Figure 5.6 shows throughput results for different values of multiprogramming. The throughput under all three protocols, namely SL, ML-C, and ML-RC, increases with multiprogramming level and after certain level the throughput begins to drop as

multiprogramming increases. The drop in throughput is due to thrashing resulting from very high data contention. When the data contention increases, so does the blocking due to denial of lock requests and restarts due deadlocks, and as a result of which the throughput drops. The maximum throughput was obtained with ML-RC at mpl.level = 100. As the multiprogramming level is increased beyond mpl.level = 50, the throughput begins to drop for both SL and ML-C. However, thrashing begins to occur at mpl.level = 100 for ML-RC. Observe that the throughput with ML-RC is higher than both ML-C and SL even with thrashing at high values of multiprogramming. Further, the relative improvement in throughput for ML-RC increases as the level of multiprogramming increases.

The effect of using recoverability on response time is shown in Figure 5.7. The response time initially decreases as the level of multiprogramming increases because more number of transactions are allowed to access the database. However, the response time begins to increases beyond mpl.level = 50 for SL and ML-C when thrashing begins to occur. Due to thrashing, there are more number of restarts, and the response time of a restarted transaction is increased by a mean of one entire response time and the transaction must be executed all over again. Thus, more number of restarts implies larger response time for transactions. However, in the case of ML-RC, the response time begins to increase only at mpl.level =100. The minimum response time was obtained with ML-RC. At higher values of multiprogramming the average response time is smaller with ML-RC than with ML-C or SL. As the level of multiprogramming increases, so does the data contention. Hence, more transactions will be restarted which leads to a larger response time for transactions and a lower throughput at higher values of multiprogramming.

Figure 5.8 shows the restart ratio and Figure 5.9 shows the blocking ratio for various protocols. The restart ratio and the blocking ratio are smaller with ML-RC than with ML-C or SL. 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, for all the three protocols, the restart ratio is smaller than than the blocking ratio. 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.

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 only 1 resource unit. The 1 resource unit case models high resource contention and the 5 resource units case simulates a multiprocessor database.

Figure 5.10 and 5.11 show the throughput results for 5 and 1 resource units respectively. Observe that the maximum throughput is obtained with ML-RC in both these cases. Further, the throughput obtained with finite resources at a given level of multiprogramming is smaller than the corresponding throughput obtained with infinite resources. This is to be expected as transactions will have to wait for resources to become available before they can execute non-conflicting operations.

In the case where only 1 resource unit is present, the throughput is very low and thrashing begins at mpl.level = 25. This is because all the available resources are busy and resource contention becomes the primary bottle neck. The response time results for 5 resource units is shown in Figure 5.12 and the response time results for 1 resource unit is shown in Figure 5.13. The response time for 1 resource unit first decreases with multiprogramming level and then begins to increase rapidly as thrashing begins to occur. With 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. Thus, the response time for transactions is very high and results in a low throughput.

The restart ratio for 5 resource units and 1 resource units are shown in Figures 5.14 and 5.15 respectively. The blocking ratio for 5 resource units and 1 resource units are shown in Figures 5.16 and 5.17 respectively.

5.4.6 Summary of Simulation Results

We now summarize the simulation results as follows:

- The maximum throughput (minimum response time) occurs with ML-RC for various quantities of available resources.

- The magnitude of relative improvement is proportional to the number of available resources. The maximum relative improvement occurs for infinite resources case. Further, for a given amount of resource contention, the improvement in performance of ML-RC increases as data contention (i.e., multiprogramming level) increases.

- Both multilevel concurrency control protocols ML-C and ML-RC out perform SL. Thus, a more sophisticated semantics-based multilevel approach to concurrency control is justifiable in complex information systems.

- The performance of our new ML-RC protocol is much better than ML-C for various quantities of available resource units. Thus, the r-aborts, due to cyclic commit dependencies, do not negate the advantages of exploiting recoverability based semantics.

5.5 Conclusions

In this chapter, we presented and evaluated a new semantics-based multilevel concurrency control protocol. We have shown how simple semantics such as the synchronization properties of operations and the structure of operations, can be used to enhance concurrency in *complex information systems*. As our simulation studies indicate, using semantics in multilevel concurrency control produces appreciable improvements in the performance thereby justifying the concomitant overheads that are incurred such as having to maintain dependency information and detect commit dependency cycles at commit time. The overhead in terms of maintaining a list for each recoverable operation can be justified by the overall performance gain. Furthermore, a wait-for graph will have to be maintained even in the case where only commuting operations are allowed to execute. Thus, in our multilevel protocol, a wait-for edge may get transformed to a commit-dependency edge due to a recoverable operation. In general, the magnitude of the improvement is dependent upon both data contention and resource contention. For a given amount of resource contention, higher the data contention better is the improvement.

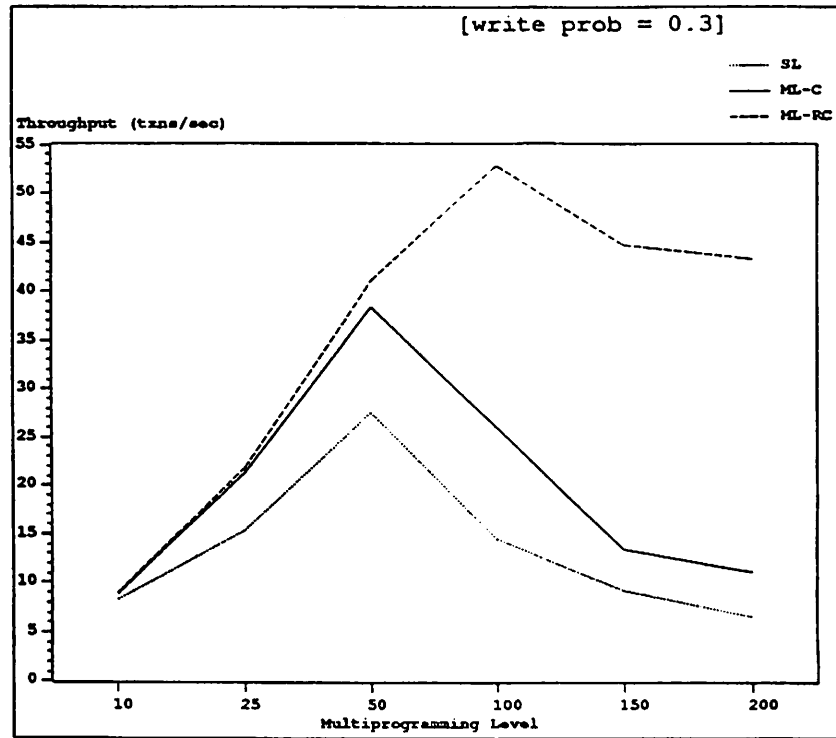


Figure 5.6. Throughput (infinite resources)

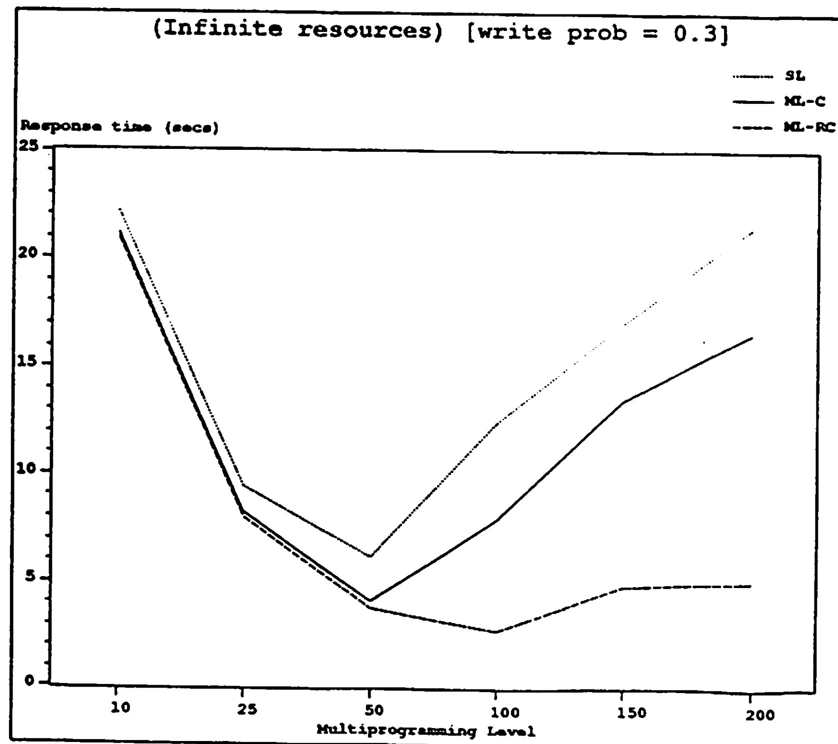


Figure 5.7. Response time (infinite resources)

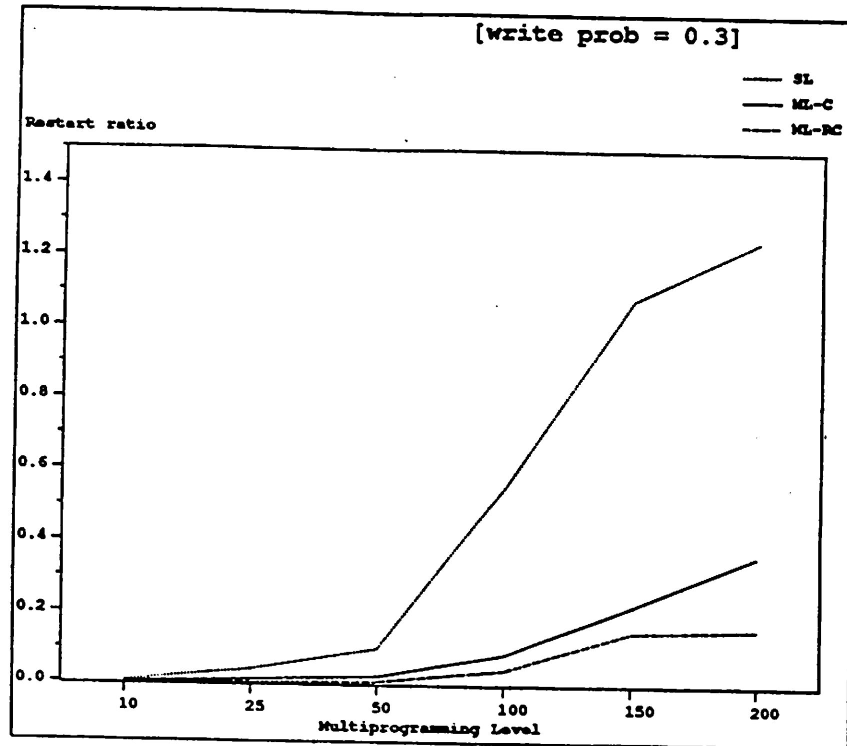


Figure 5.8. Restart ratios (infinite resources)

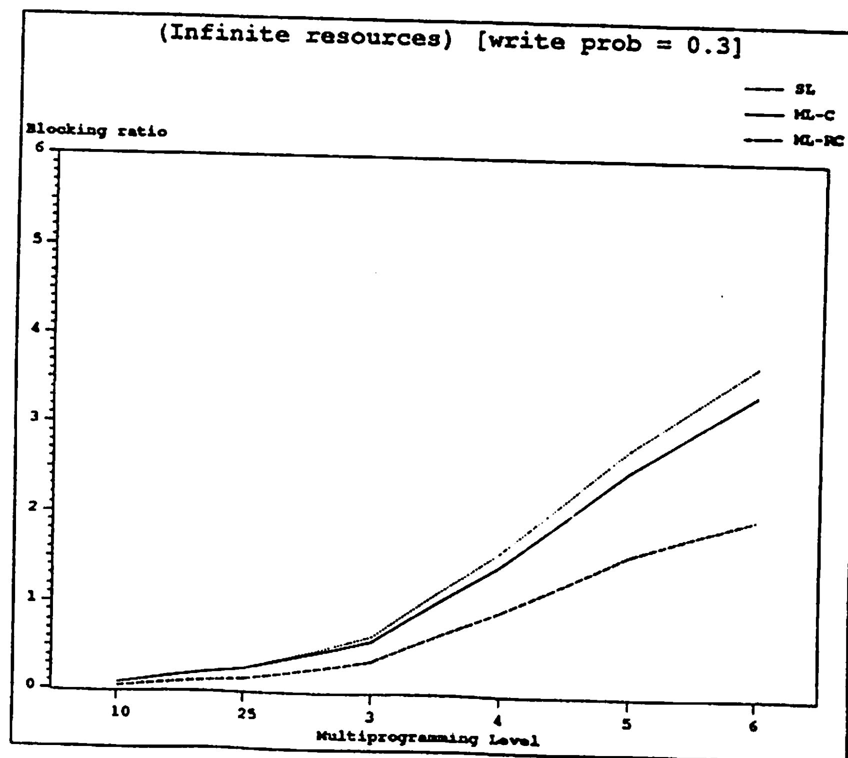


Figure 5.9. Blocking ratios (infinite resources)

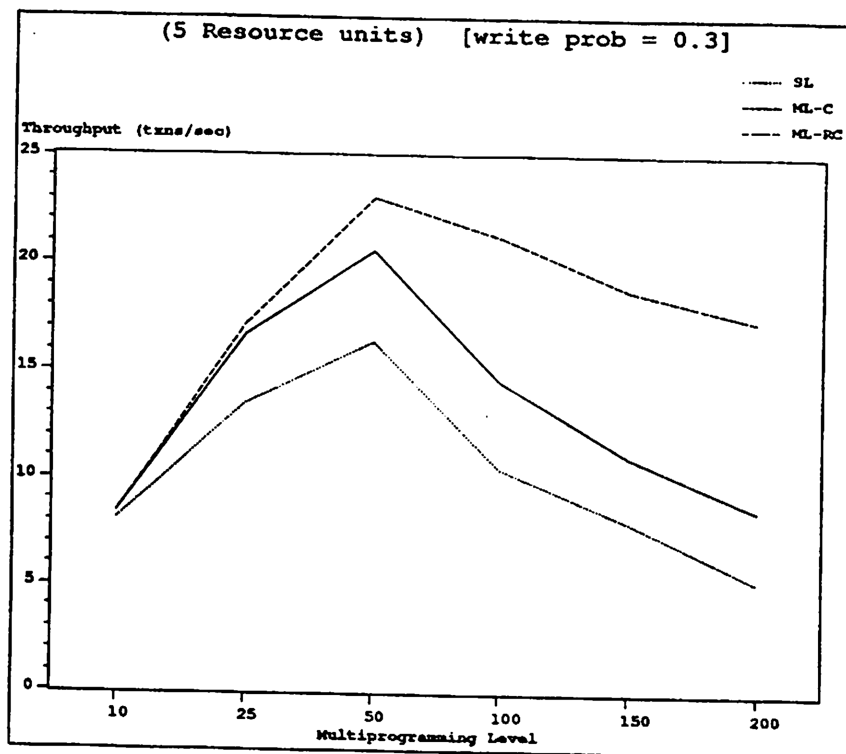


Figure 5.10. Throughput (5 resource units)

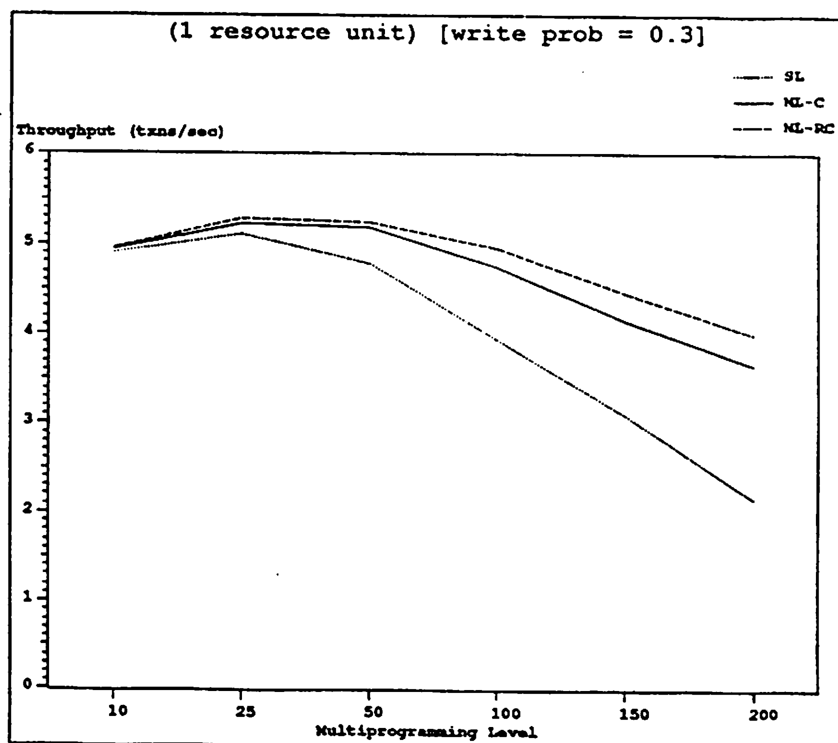


Figure 5.11. Throughput (1 resource unit)

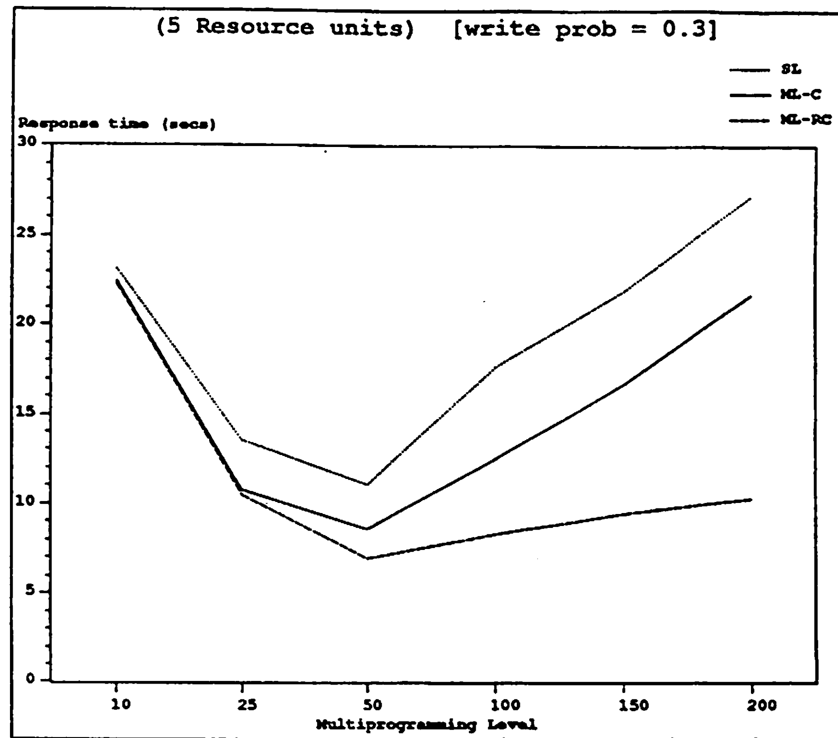


Figure 5.12. Response time (5 resource units)

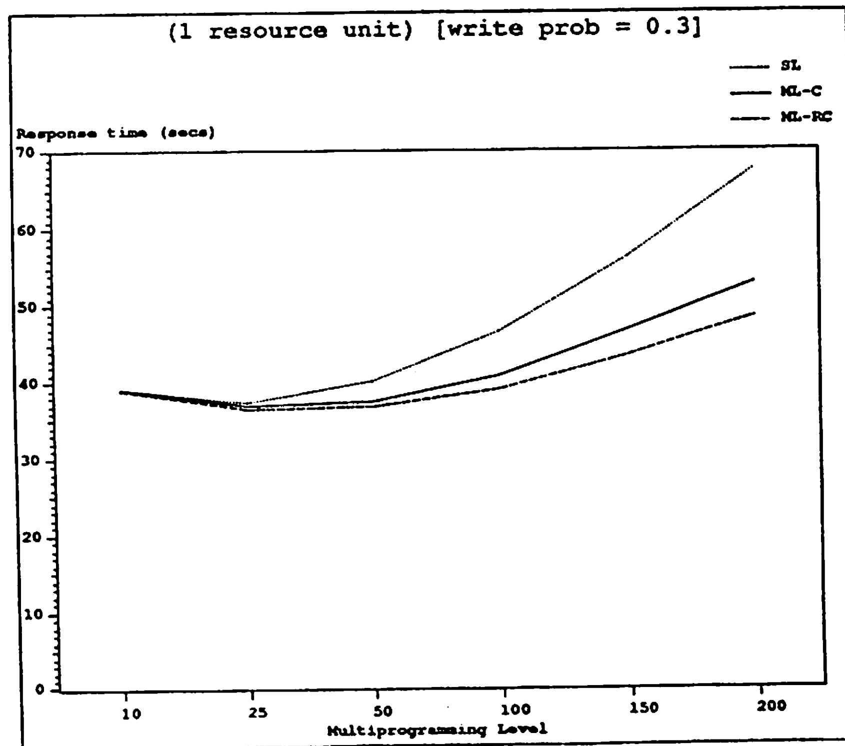


Figure 5.13. Response time (1 resource unit)

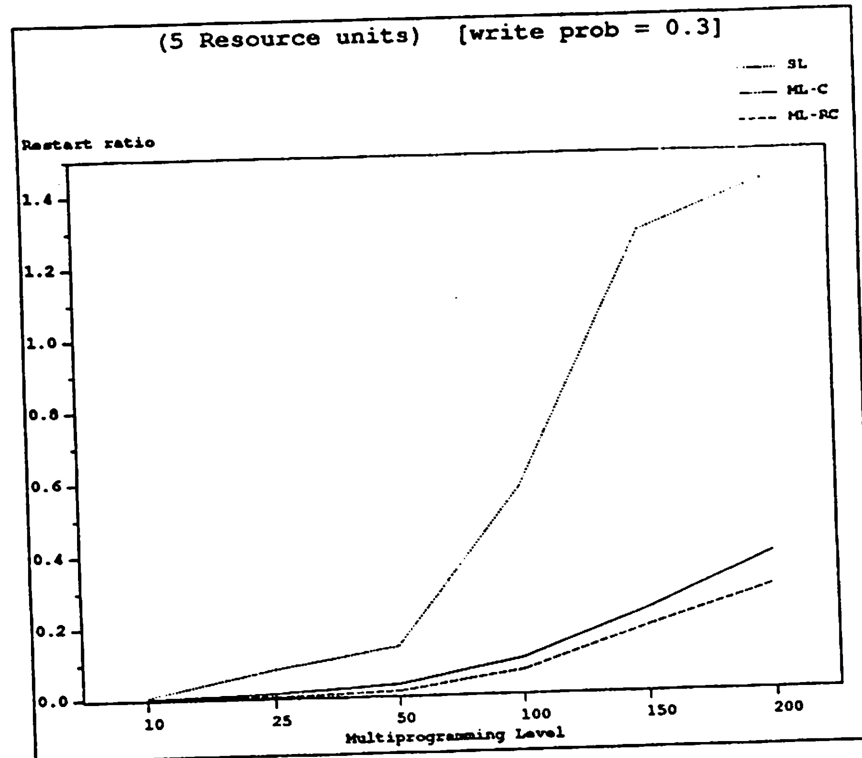


Figure 5.14. Restart ratios (5 resource units)

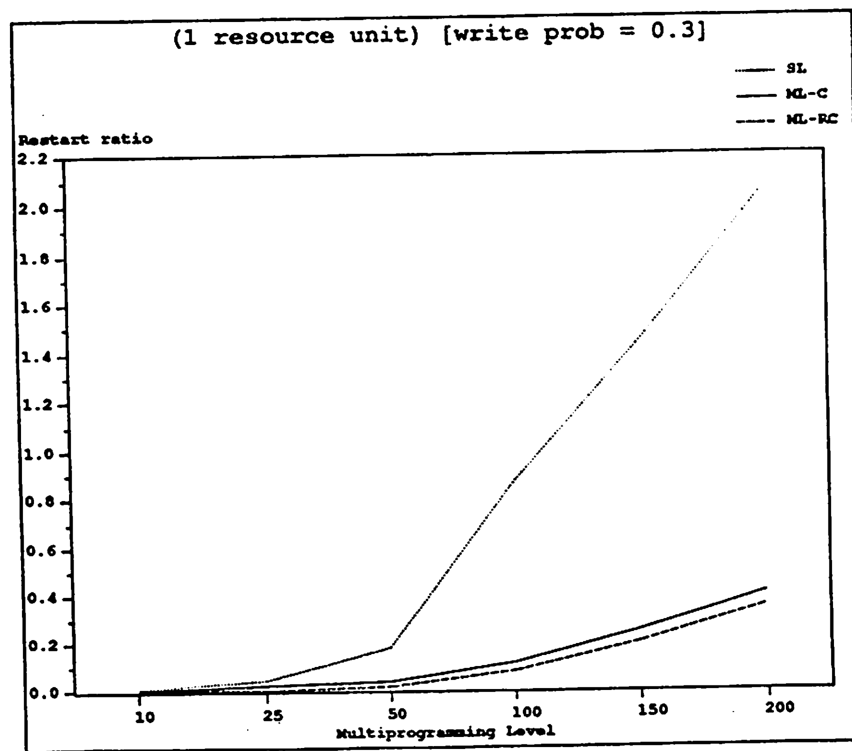


Figure 5.15. Restart ratios (1 resource unit)

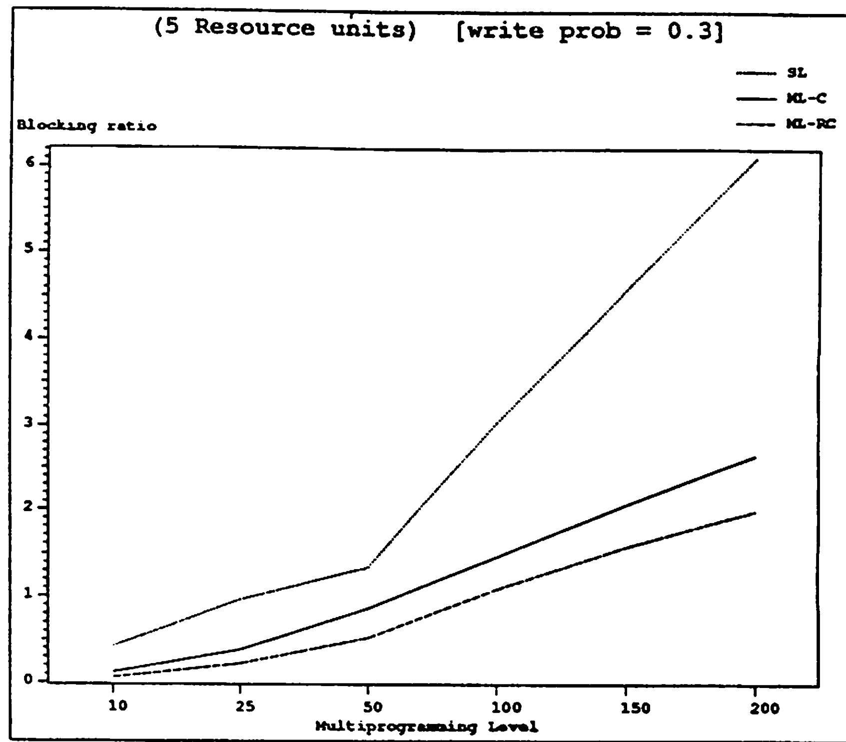


Figure 5.16. Blocking ratios (5 resource units)

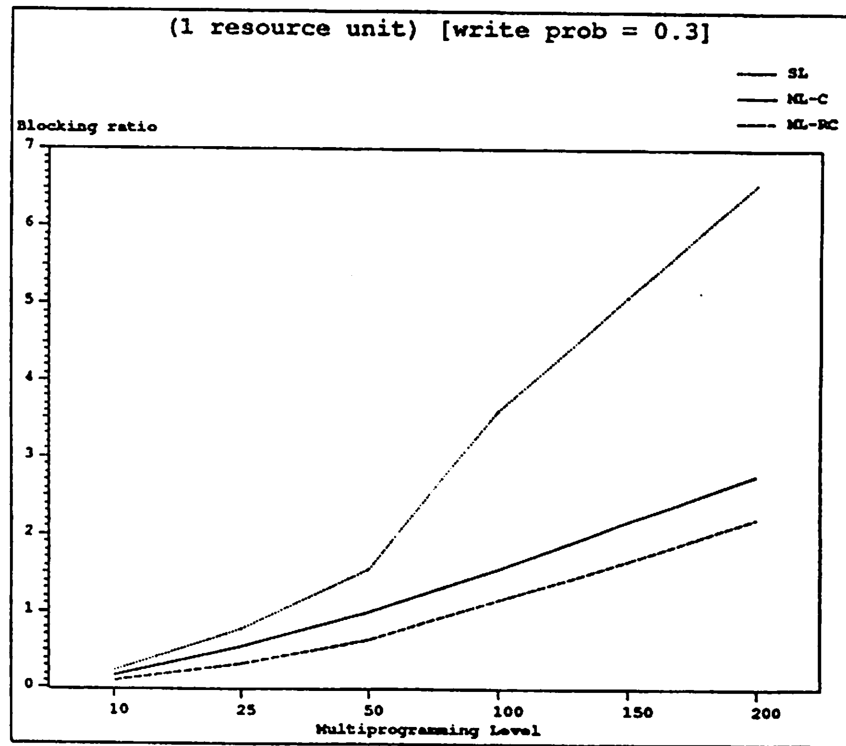


Figure 5.17. R-aborts (1 resource unit)

CHAPTER 6

CORRECTNESS OF MULTILEVEL PROTOCOL

The multilevel protocol ML-R described in Section 5.1 ensures that operations are executed atomically and at the same time the serializability of top level operations. However, instead of blocking non-commuting operations, we block only non-recoverable operations. Recall, from Section 4, that when a given operation o_j is recoverable with respect to o_i, the transaction that issued o_j should commit after o_i. In a multilevel system, the commit dependencies have to be enforced at each level. In order to track commit dependencies at each level we introduced lists. Thus, if an operation o_i is to be executed, the ancestor of o_i acquires locks or lists depending upon the type of conflict that o_i encounters with other active operations on that object. Lists are a mechanism to track commit dependencies, in that they contain the adjacency list of the commit dependency graph. Hence, lists can be used to detect cyclic commit dependencies.

6.1 Locks and Lists

The multilevel protocol is described in terms of *locks* and *lists*. These are acquired by operations at all levels in the system, except the leaf level operations.

A lock is obtained if the operation commutes with other active operations on that object.

A list is obtained if the operation is recoverable with other active operations on that object. A list contains the ids of the transactions that issued the operations with which the given operation is recoverable.

Figure 6.1 shows a three level system in which a list has been acquired by transaction T_2. The list contains the id of transaction T_1, as T_2 has executed an operation *inc*2 which is recoverable with respect to operation *val*1 issued by T_1. Once T_1 commits or aborts, the id of the transaction is removed from the lists to which it may belong. Thus, once T_1 commits or aborts, the list acquired by T_2 is now empty and can be released by T_1. Thus, the order of completion follows the commit dependencies that get formed.

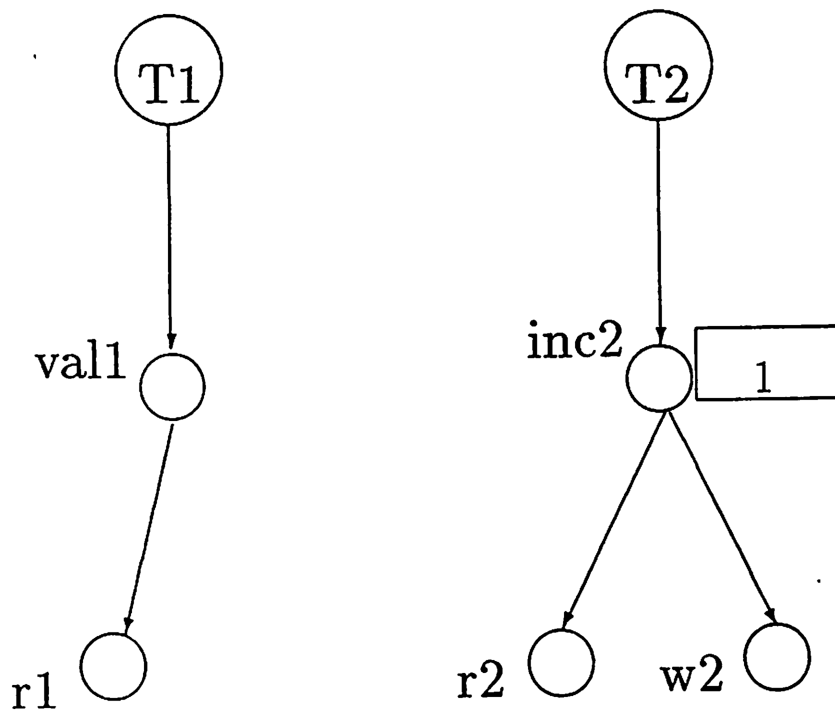


Figure 6.1. Example of multilevel protocol

6.2 Correctness

The multilevel protocol ML-R satisfies the correctness notions given by Theorem 1 and Theorem 2 in Section 3.1.1.

Theorem 3: The multilevel nested protocol ML-R ensures that computations produce acyclic graphs at all levels when the condensation X_i is applied to the graph representing the computation.

Proof: In order to show that there are no cycles in the condensed graph at any level, we show that there cannot be cycles of length two. This argument can be easily extended to dependency cycles of length greater than two by straight forward induction.

Consider two level i operations o_m and o_n represented by leaf nodes S_m and S_n of the condensed graph at level i. Dependency edges between nodes S_m and S_n may exist due to inherited dependencies between sub-operations of o_m and o_n at level $i-1$, due to dependencies at level i, as o_m and o_n are conflicting operations, or both. We need to show that there are no cycles in the condensed graph at any level i. We will show this by deriving a contradiction, and for that purpose assume that there is a cycle, i.e., edges $S_m \rightarrow_d S_n$ and $S_n \rightarrow_d S_m$.

Case i) Let $S_m \rightarrow_d S_n$ and $S_n \rightarrow_d S_m$ be due to dependencies inherited from interleaving sub-operations of o_m and o_n, such that $child(o_m) \rightarrow_d^{i-1} child(o_n)$ and $child'(o_n) \rightarrow_d^{i-1} child'(o_m)$. The protocol allows only commutative or recoverable operations to execute. Thus, the dependencies formed are due to a) the commit dependencies, and the operations o_m and o_n would never be able to commit as the lists acquired by them will never become empty or b) wait-for dependencies due to operations that are non-recoverable and hence the operations are deadlocked. In either case one of the transaction will be restarted.

Case ii) There are two sub-cases to consider; a) $S_m \rightarrow_d S_n$ is due to recoverable operations o_m and o_n at level i. b) $S_m \rightarrow_d S_n$ is due to wait-for dependency between operations o_m and o_n. Without loss of generality, assume that the dependency edge $S_m \rightarrow_d S_n$ is due to recoverable operations o_m and o_n at level i and $S_n \rightarrow_d S_m$ due to dependencies inherited from conflicting sub-operations at level $i-1$. If o_n is recoverable w.r.t o_m, according to the rules of the protocol, o_n will be allowed to execute provided o_m is not in_progress. This implies that $\forall child(o_m) <_{i-1} child(o_n)$. Hence, there cannot be a dependency edge $S_n \rightarrow S_m$ inherited due to conflicting sub-operations at level $i-1$.

The dependency edge $S_m \rightarrow_d S_n$ is due to wait-for dependency, i.e., o_m and o_n are non-recoverable. Since it is a top-down protocol, o_n will be blocked at level i until all conflicting operations terminate. Thus, $\forall child(o_m) <_{i-1} child(o_n)$. Hence, a dependency edge $S_n \rightarrow S_m$ inherited due to conflicting sub-operations at level $i-1$ is not possible.

Case iii) Both dependency edges are due to conflicting operations at level i The proof is similar to case i) except that the $parent(o_m)$ and $parent(o_n)$ would not be able to commit unless one of them is restarted either because of deadlock or because of a cyclic commit dependency.

Thus, whenever two operations are executed according to the protocol at any level i, mutual dependencies cannot be formed.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis has proposed new methods for improving concurrency in complex information systems. Our techniques make use of available semantic information to improve concurrency when transactions issue abstract operations on complex data. We summarize specific research contributions of this thesis and, in the last section, discuss further work.

7.1 Contributions of This Research

The important contributions of this dissertation are:

7.1.1 Multilevel Graph Model

In this dissertation, we have developed a general model for proving correctness of various semantics-based concurrency control schemes. This model is a natural extension of the single level serialization theory and facilitates the incorporation of operation semantics and levels of abstraction to ignore dependencies. Hence, concurrency control schemes that use these features can be proved correct in a formal setting.

7.1.2 Recoverability as a Weaker Notion of Conflict

In the past, commutativity of operations has been exploited to provide enhanced concurrency while avoiding cascading aborts. We identified a new 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. By conducting extensive simulation studies we have shown that using recoverability, turnaround time of transactions can indeed be reduced significantly. Further, our studies show enhancement in concurrency even when *resource contention* is taken into consideration.

7.1.3 Relative Conflict for Scheduling Complex Operations

Many information systems are structured along levels of abstraction. The currently available single level concurrency control mechanisms for *reads* and *writes* are inadequate for future complex information systems[13, 62]. We presented a new *multi-level* concurrency protocol that not only uses recoverability as a basis for determining conflicts but also uses a new way of scheduling operations based on *relative conflict*. Relative conflict uses information about the type of parent of an operation in determining whether two operations can be executed concurrently. In a ML nested protocol, if two operations are executed concurrently then there is a possibility of deadlock among sub-operations. However, scheduling operations according to relative conflict avoids deadlock among sub-operations. We have shown that the performance of this new protocol is significantly better than previously proposed protocols by comparing its performance with 1) a multilevel protocol that uses only commutativity as a basis for conflicts and 2) a single level two phase protocol. The performance was evaluated by conducting extensive simulation studies. We also studied the implications of *resource contention* on this new multilevel concurrency protocol.

7.2 Beyond the Dissertation

One area for future work is to study specific applications such as office information systems(OIS), stock trading databases, and applications requiring large databases[69]. We can use specific semantics available by analyzing the properties of typical objects used and the type of transactions that are issued by the users of such systems to develop specialized concurrency control schemes.

- The issue of transaction response time and throughput is becoming critical in a large number of applications[69]. We plan to exploit other sources of semantics to achieve high concurrency. For example, a data frequently accessed, so called "hot spot" may become a bottleneck for concurrency. The fact that certain data is accessed frequently is in itself a semantic information that is

available. So using this information we can design a composite concurrency control scheme; one for "hot spot" data and another for normal data. To this end we can use a multilevel semantics-based approach incorporating our conceptual ideas of *relative* conflicts for "hot spot" data and a more simple and traditional concurrency control for normal data.

- In the future, we will have multiprocessor object bases. Hence, resource limitations will be less severe and the issue of data contention will be all the more prominent. Reducing data contention will not only be important from a performance point of view but also from the point of view of system utilization. So we need to look at the role of semantics-based concurrency control schemes in not only enhancing concurrency but improving resource utilization as well. In a more general sense, there are major issues to be addressed regarding the nature of interactions a particular concurrency control scheme has when resource utilization is taken into consideration in multiprocessor object bases.

- Another plan of study is to apply the ideas presented in this thesis to other transaction models as well. These models are provided as a user facility. These include sagas[28], split-transactions[63] check-in and check-out models[46], multilevel models for CAD transactions[41], and Transaction Groups [26]. These transaction models have varying consistency requirements based on non-serializable semantics and are mostly ad-hoc as they have been proposed with very specific applications in mind. In these models, we can further weaken the properties that the condensed graphs have to obey. It will be very interesting to see if we can relate other graph theoretic properties besides acyclicity and associate them with correct concurrent executions in these models.

- We need to understand the Complexity versus Performance issue. We have outlined protocols that use scheduling at multiple levels; one for each level of nesting. It may not be necessary to control concurrency at every level and concurrency control at two or three levels may be enough to obtain the desired performance. Further, additional levels of scheduling complicate the scheduler but may not result in significant performance benefits. The decision about the number of levels and at which level to provide concurrency control is an interesting problem in itself.

- Experience has shown that to achieve reasonable performance in distributed databases, the database has to be replicated. We will have to address issues in

designing high performance semantics-based schemes for replicated systems as well as systems supporting multiple versions of data. One source of semantics that can be used in the design of protocols for replicated systems is the answer to the question: how recent a view does a given operation require? This kind of information will not only improve concurrency but also reduce the the number of copies one has to access.

- Recovery is another issue that needs to be addressed. Just as concurrency is an important issue, so is the problem of transaction failures. Semantics-based Undos is a very interesting line of research and properties of operations can be used to determine the type of Undo operation that needs to be executed to restore the consistency of the data[80, 55, 56].

In conclusion, we have outlined an approach to efficient transaction processing based on using available semantics of the application. We have developed a semantics-based approach to concurrency control that we believe will be useful in object bases. In defining recoverability, we used operation semantics to provide a weaker conflict predicate, than commutativity. In defining relative conflict, we have also described a new multilevel protocol that use semantic information concerning operation structure, to provide enhanced concurrency. In addition, we outlined a new model to reason about concurrent computations in implicit nested environments and presented detailed evaluation of the performance of the protocols. The major contribution of this thesis has been the development of systematic and general techniques for taking advantage of semantic information to enhance concurrency in complex information systems. This work should make it easier to handle complex, information intensive applications efficiently.

BIBLIOGRAPHY

[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] Allchin, J. E. *An architecture for reliable decentralized systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, September 1983.

[3] Allchin, J. E. and McKendry, M. S. Synchronization and recovery of actions. In *Second Annual ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

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

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

[6] Badrinath, B. R. and Ramamritham, K. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5):541-547, May 1988.

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

[8] Banerjee, J., Chou, H.-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim., H.-J. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[9] Batory, D. S. GENESIS: A project to develop an extensible database management system. In Dittrich, K. and Dayal, U., editors, *Proceedings of the international workshop on Object-Oriented Database Systems*, pages 207–208. IEEE Computer Society, September 1986.

[10] Batory, D. S. and Kim, W. Modeling concepts for VLSI CAD objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

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

[12] 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.

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

[14] Bernstein, P. A. and Goodman, N. Concurrency control in distributed database systems. *Computing Surveys*, 13(2):185–221, June 1981.

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

[16] 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.

[17] 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.

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

[19] Carey, M. J., DeWitt, D. J., Frank, D., Grafe, G., Richardson, J. E., Shekita, E. J., and Muralikrishna, M. The architecture of the EXODUS extensible DBMS. In Stonebraker, M., editor, *readings in Database Systems*, pages 488–501. Morgan Kaufmann, 1988.

[20] Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th international Conference on VLDB, Kyoto, Japan*, August 1986.

[21] Copeland, G. and Maier, D. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 316-325, Boston, MA., June 1984.

[22] 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.

[23] Dayal, U. and Smith, J. PROBE: A knowledge-oriented database management system. In *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[24] 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.

[25] 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.

[26] Fernandez, M. F. and Zdonik, S. B. Transaction Groups: A Model for Controlling Cooperative Transactions . In *Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Newcastle, Australia, January 1989.

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

[28] Garcia-Molina, H. and Salem, K. SAGAS. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 249–259, May 1987.

[29] Garza, J. F. and Kim, W. Transaction management in an Object-Oriented database system. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 37–45, Chicago, Illinois, June 1988.

[30] Goodman, N. and Shasha, D. E. Semantically-based concurrency control for search structures. In *Proceedings of the Fourth Annual Symposium on Principles of Database Systems*, pages 8–19, March 1985.

[31] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the 1st international conference on VLDB*, pages 25–33, Framingham, MA., September 1975.

[32] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared database. In Stonebraker, M., editor, *readings in Database Systems*, pages 94-121. Morgan Kaufmann, 1988.

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

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

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

[36] Katz, R. H. and Weiss, S. Design transaction management. In *Proceedings of the 21st Design Automation Conference*, pages 692–693, 1984.

[37] Kedem, Z. M. and Silberschatz, A. Locking protocols: From exclusive to shared locks. *Journal of the ACM*, 30(4):787-804, October 1983.

[38] Kim, W., Banerjee, J., Chou, H.-T., jorge F. Garza, and Woelk, D. Composite object support in an Object-Oriented database system. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 118–125, Orlando, Florida, October 1987.

[39] Korth, H. F. Deadlock freedom using edge locks. *ACM Transactions on Database Systems*, 7(4):632–652, December 1982.

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

[41] Korth, H. F., Bancilhon, F., and Kim, W. A model of CAD Transactions. In *Proceedings of the 11th international conference on VLDB*, pages 25–33, Stockholm, August 1985.

[42] Kung, H. and Papadimitriou, C. An optimality theory of database concurrency control. *Acta Informatica*, 19(1):1–13, January 1984.

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

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

[45] Liskov, B. and Scheifler, R. Guardians and Actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 8(4):484–502, December 1983.

[46] Lorie, R. and Plouffe, W. Complex objects and their use in design transactions. In *Proceedings of Databases for Engineering Applications*, pages 115–121, Database Week 1983 (ACM), May 1983.

[47] Lynch, N. A. Multilevel atomicity — A new correctness for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[48] Lynch, N. A. Concurrency control for resilient nested transactions. *Advances in Computing Research*, 6:335-373, 1986.

[49] Lynch, N. A. and Merritt, M. Introduction to the theory of nested transactions. Technical report, Massachusetts Institute of Technology and AT and T Bell Laboratories, June 1986.

[50] Maier, D. and Stein, J. Indexing in an Object-Oriented DBMS. In *Proceedings of the international workshop on Object-Oriented Database Systems*, pages 85–92. IEEE Computer Society, September 1986.

[51] Maier, D., Stein, J., Ottis, A., and Purdy, A. Development of an Object-Oriented DBMS. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, Portland, Oregon, October 1986.

[52] Manola, F. and Dayal, U. PDM: An object-oriented data model. In Dittrich, K. and Dayal, U., editors, *Proceedings of the international workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Computer Society, September 1986.

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

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

[55] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. M. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. Research Report RJ 6649, IBM Research Division, Almaden Research Center, San Jose, CA 95120, January 1989.

[56] Mohan, C. and Rothermel, K. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. Research Report RJ 6650, IBM Research Division, Almaden Research Center, San Jose, CA 95120, January 1989.

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

[58] 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.

[59] 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.

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

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

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

[63] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended activities. In *Proceedings of the 14th international conference on VLDB*, pages 26–37, Los Angeles, California, September 1988.

[64] Roesler, M. *Sharing objects in distributed systems.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, California, June 1988.

[65] Roesler, M. and Burkhard, W. A. Concurrency control scheme for shared objects: A peephole approach based on semantics. In *Proceedings of 7th International Conference on distributed computing information systems*, pages 224–231, September 1987.

[66] Schwarz, P. M. Transactions on Typed Objects. PhD Thesis CMU-CS-84-166, Carnegie-Mellon University, Pittsburgh, PA, December 1984.

[67] Schwarz, P. M., Chang, W., Freytag, J. C., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H. Extensibility in the starburst system. In *Proceedings of the international workshop on Object-Oriented database systems*, pages 85–92. IEEE Computer Society, September 1986.

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

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

[70] Sha, L. *Modular concurrency control and failure recovery-- Consistency, Correctness and Optimality.* PhD thesis, Department of Computer and Electrical Engineering, Carnegie-Mellon University, 1985.

[71] Shasha, D. E. Concurrent algorithms for search structures. PhD Thesis 12-84, Harvard University, Aiken Computation Laboratory, 33 oxford street Cambridge, MA. 02138, June 1984.

[72] Silberschatz, A. and Kedem, Z. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, January 1980.

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

[74] Spector, A. Z., Butcher, J., Daniels, D. S., Duchamp, D. J., Eppinger, J. L., Fineman, C. E., Heddaya, A., and Schwarz, P. M. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[75] 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.

[76] Stonebraker, M. and Rowe, L. The design of POSTGRES. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 340–355, May 1986.

[77] 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.

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

[79] 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.

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

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

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

[83] Weikum, G. and Schek, H. Architectural issues of transaction management in multi-layered systems. In *Proceedings of the 10th VLDB conference*, pages 454–465, Singapore, August 1984.

[84] Yannakakis, M. Freedom from deadlock of safe locking policies. *SIAM Journal of Computing*, 11(2):391–408, May 1982.

[85] Yannakakis, M. Serializability by locking. *Journal of the ACM*, 31(2):227–244, April 1984.

[86] Zhao, W. and Ramamritham, K. Use of transaction structure for improving concurrency. In *12th Australian Computer Science Conference*, February 1989.