MODELING AND EVALUATION OF DATABASE CONCURRENCY CONTROL ALGORITHMS

by

M. J. Carey

Memorandum No. UCB/ERL M83/56

7 September 1983

Modeling and Evaluation of Database Concurrency Control Algorithms

by

Michael James Carey

MODELING AND EVALUATION OF DATABASE CONCURRENCY CONTROL ALGORITHMS

Ph.D. Michael James Carey Electrical Engineering and Computer Science

Sponsors:
Air Force Office of Scientific Research
Naval Electronic Systems Command
California Fellowship in Microelectronics

[illegible]
Michael R. Stonebraker
Chairman of Committee

ABSTRACT

In database management systems, *transactions* are provided for constructing programs which appear to execute atomically. If more than one transaction is allowed to run at once, a *concurrency control algorithm* must be employed to properly synchronize their execution. Many concurrency control algorithms have been proposed, and this thesis examines the costs and performance characteristics associated with a number of these algorithms.

Two models of concurrency control algorithms are described. The first is an abstract model which is used to evaluate and compare the relative storage and CPU costs of concurrency control algorithms. Three algorithms, two-phase locking, basic timestamp ordering, and serial validation, are evaluated using this model. It is found that the costs associated with two-phase locking are at least as low as those for the other two algorithms.

The second model is a simulation model which is used to investigate the performance characteristics of concurrency control algorithms. Results are presented for seven different algorithms, including four locking algorithms, two timestamp algorithms, and serial validation. All performed about equally well in situations where conflicts between transactions were rare. When conflicts were more frequent, the algorithms which minimized the number of transaction restarts were generally found to be superior. In situations where several algorithms each restarted the same number of transactions, those which restarted transactions which had done less work tended to perform the best.

Two previously proposed schemes for improving the performance of concurrency control algorithms, multiple versions and granularity hierarchies, are also examined. A new multiple version algorithm based on serial validation is presented, and performance results are given for this algorithm, the CCA version pool algorithm, and multiversion timestamp ordering. Unlike their single version counterparts, all three algorithms performed comparably under the workloads considered. Three new hierarchical concurrency control algorithms, based on serial validation, basic timestamp ordering, and multiversion timestamp ordering, are presented. Performance results are given for these algorithms and a hierarchical locking algorithm. All were found to improve performance in situations where the cost of concurrency control was high, but were of little use otherwise.

To Carol

ACKNOWLEDGEMENTS

I would like to thank my advisor, Mike Stonebraker, for introducing me to the area of database systems and for supervising this thesis. His support, guidance, and encouragement have been invaluable these past few years. I would also like to thank the other members of my thesis committee: Mike Powell taught me a great deal about large software systems during my stay at Berkeley, and Ron Wolff played a major role in the statistical aspects of this work. All three members of my thesis committee were extremely generous with their time in helping me to finish.

I would like to thank all the students, faculty, and staff of Project INGRES for providing an excellent research environment. Toni and Margie were the source of many helpful discussions and ideas, and Joe was always willing to help with system questions and problems.

I am grateful for all of my friends at Berkeley. They believed in me, kept me entertained, and made the last three years memorable. Although there are many, I would especially like to thank Fred and Pat, Clem, the OSMOSIS group, and Paul and Kathleen.

Lastly, I would like to thank the people most responsible for making this all possible: My wife, Carol, for all her love, support, patience, and understanding, and my family, for their love and support throughout the years.

This thesis was supported by the Air Force Office of Scientific Research Grant AFOSR-78-3596, the Naval Electronic Systems Command Contract NESC-N00039-81-C-0569, and a California Fellowship in Microelectronics.

TABLE OF CONTENTS

Chapter 1. Introduction 1

1.1 Background 1

1.1.1 An Example 1

1.1.2 Transactions and Concurrency Control 5

1.2 Concurrency Control Research 8

1.3 Performance Issues 11

1.4 Thesis Overview 14

Chapter 2. Concurrency Control Costs 17

2.1 Overview 17

2.2 Transaction Requests 19

2.3 The Concurrency Control Database 19

2.4 Algorithm Descriptions 21

2.5 Using the Model 22

2.5.1 Two-Phase Locking 23

2.5.2 Basic Timestamp Ordering 27

2.5.3 Serial Validation 31

2.6 Algorithm Cost Comparisons 35

2.6.1 Storage Cost 38

2.6.2 CPU Cost 42

2.6.3 Cost Comparison Summary 45

2.7 Future Model Extensions 48

2.7.1 Multiple Versions 48

2.7.2 Granularity Hierarchies 48

2.7.3 Distributed Databases 49

2.8 Summary 51

Chapter 3. Concurrency Control Performance 52

3.1 Background Considerations 52

3.2 Model Description 54

3.2.1 The Workload Model 54

3.2.2 The Queuing Model 58

3.2.3 Algorithm Descriptions 65

3.3 Performance Experiments 67

3.3.1 Algorithms Studied 68

3.3.2 Concurrency Control Costs 71

3.3.3 The Statistical Approach 74

3.3.4 Experiments and Results 77

3.3.4.1 Experiment 1: Transaction Size 77

3.3.4.2 Experiment 2: Access Patterns 92

3.3.4.3 Experiment 3: Mixed Workload 96

3.3.4.4 Experiment 4: Multiprogramming Level 100

3.3.4.5 Experiment 5: System Balance 103

3.3.4.6 Experiment 6: Concurrency Control Cost 104

3.3.4.7 Algorithm Anomalies 107

3.4 Conclusions 110

3.4.1 Performance in Under Low Conflicts 110

3.4.2 Blocking versus Restarts 111

3.4.3 Transaction Size 113

3.4.4 Concurrency Control Overhead 114

3.4.5 Granularity and Performance 115

3.4.6 System Balance 115

3.4.7 Algorithm Anomalies 115

Chapter 4. Performance Enhancement Schemes 117

4.1 Multiple Versions 118

4.1.1 Multiversion Timestamp Ordering 118

4.1.2 Multiversion Locking 121

4.1.3 Multiversion Serial Validation 123

4.1.4 Multiple Versions and Performance 125

4.1.4.1 Concurrency Control Costs 125

4.1.4.2 The Simulation Approach 127

4.1.4.3 Experiments and Results 128

4.1.4.3.1 Experiment 1: Low Conflict Performance 128

4.1.4.3.2 Experiment 2: Read-Only Transaction Size 132

4.1.4.3.3 Experiment 3: Read-Only Transaction Fraction 136

4.2 Granularity Hierarchies 138

4.2.1 Hierarchical Algorithms 139

4.2.1.1 Hierarchical Locking 139

4.2.1.1.1 Hierarchical Validation 141

4.2.1.1.2 Hierarchical Timestamps 144

4.2.1.1.3 Hierarchical Multiple Version Algorithms 149

4.2.2 Hierarchies and Performance 153

4.2.2.1 Modeling a Hierarchy 153

4.2.2.2 Concurrency Control Costs 154

4.2.2.3 Experiments and Results 155

4.2.2.3.1 Experiment 1: Low Concurrency Control Cost 156

4.2.2.3.2 Experiment 2: Large Transaction Size 159

4.3 Summary 163

Chapter 5. Conclusions 165

5.1 Summary of Results 165

5.2 Comparison With Other Work 168

5.3 Implications of the Results 174

5.4 Limitations of Results 176

5.5 Future Research Directions 179

References 182

Appendix 1. Timestamp-Based Serial Validation 193

Appendix 2. The "No Blind Writes" Assumption 195

Thomas Write Rule 195

Multiversion Timestamp Ordering 196

Appendix 3. Simulation Output Analysis 199

A Statistical Model 199

Variance Estimation 200

Confidence Intervals 202

Experimental Results 204

CHAPTER 1

INTRODUCTION

1.1. BACKGROUND

An important component of any shared database system is the *transaction manager*. This portion of the system is responsible for dealing with the recovery and synchronization aspects of database management; it does so by providing applications programmers with an abstraction known as a *transaction* with which to construct programs which access the database. A small example will illustrate the usefulness of the transaction abstraction.

1.1.1. An Example

Consider the database of three bank accounts shown in Figure 1.1 and the three sample banking transactions shown in Figure 1.2. For the moment, no distinction will be made between a transaction and a program. The read and write operations in Figure 1.2 respectively represent reading values from the database into local variables and writing values from local variables into the database. Transaction T_1 transfers twenty dollars from account X to account Y, transaction T_2 computes interest for account X, and transaction T_3 deposits thirty-five dollars into account Z. If these three transactions are all allowed to run concurrently without synchronization, the operations comprising the transactions may be arbitrarily interleaved. As this example will illustrate, this may produce undesirable results.

Since transaction T_3 accesses no data in common with either transaction T_1 or T_2, the manner in which the operations of T_3 are interleaved with the operations of the other transactions will have no effect on the outcome of their concurrent execution. However, since T_1 and T_2 read and write common data, certain interleavings of the operations of these two transactions may produce undesirable results. If T_1 and T_2 execute *serially*, i.e., one of the transactions completes before the other begins, only two outcomes may result from executing the three transactions on the database of Figure 1.1. Figure 1.3 depicts these two possibilities. In execution A, T_1 executes before T_2, so the funds transfer of transaction T_1 completes before the interest com-

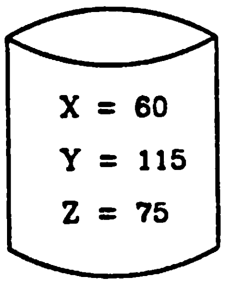

Figure 1.1: Small example database.

putation of transaction T_2 takes place. In execution B, T_1 executes after T_2, so the interest computation of T_2 completes before the funds transfer of T_1 takes place.

Outcomes that are the result of executing transactions serially are known as *serial* outcomes. Executions in which transaction operations may be interleaved, but which produce the same results as some serial outcome, are called *serializable* executions [Eswa76, Papa79, Bern81b]. An example of a non-serializable interleaving of the operations of transactions T_1 and T_2 is given in Figure 1.4. In this example, T_1's subtraction of twenty dollars from

```
transaction T1:
begin
   read x_value from X;
   read y_value from Y;
   x_value := x_value - 20;
   y_value := y_value + 20;
   write x_value into X;
   write y_value into Y;
end;

transaction T2:
begin
   read x_value from X;
   x_value := x_value * 1.10;
   write x_value into X;
end;

transaction T3:
begin
   read z_value from Z;
   z_value := z_value + 35;
   write z_value into Z;
end;
```

Figure 1.2: Three example transactions.

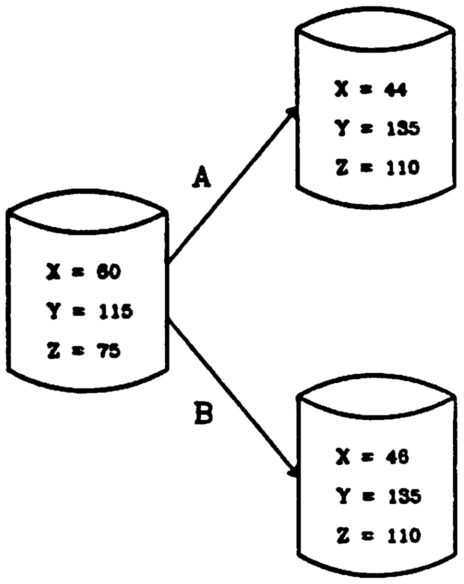

Figure 1.3: Possible serial outcomes.

account X is lost when T_2 writes the results of its interest computation into the database.

There are other problems besides lost updates that can occur when transactions are permitted to execute concurrently with no controls. For example, if a monthly report-generating transaction T_4 were to read the values in

```
T2: read x_value from X;
T2: x_value := x_value * 1.10;
T1: read x_value from X;
T1: read y_value from Y;
T1: x_value := x_value - 20;
T1: y_value := y_value + 20;
T1: write x_value into X;
T1: write y_value into Y;
T2: write x_value into X;
```

Figure 1.4: A non-serializable interleaving.

the database of Figure 1.1 while transaction T_1 was executing, it could read inconsistent data. In particular, if T_4 reads the balances from both accounts X and Y after T_1 has written its new value for X but before it has written its new value for Y, T_4 will see twenty dollars missing from X but not yet added to Y. T_4 could not see such partial results in any serializable execution of T_1 and T_4. In a serializable execution, T_4 would see the database either as it appeared before T_1 ran or as it appears after the completion of T_1. This example illustrates another of the possible ways in which problems can arise from concurrently executing transactions in the absence of some type of synchronization mechanism.

1.1.2. Transactions and Concurrency Control

The transaction abstraction was introduced to provide applications programmers with a solution to the synchronization problems illustrated by the previous example. Transactions are also intended to simplify the construction of reliable applications programs. As summarized in [Spec83], the key properties of transactions are:

Serializability. If several transactions access the database concurrently, their effects are equivalent to the effects that would result from running the same transactions in some serial order.

Failure atomicity. When a transaction executes, either all of its effects or none of its effects will be reflected in the database. This guarantees that only transactions which execute to completion will have an effect on the database.

Permanence. Once a transaction executes successfully, its effects on the database will never be lost due to subsequent software or hardware failures.

To make use of the transaction abstraction, an applications programmer is provided with three primitives: *BEGIN*, *END*, and *ABORT* transaction. The *BEGIN* statement tells the transaction manager that the operations following it in the execution of a program are to be grouped together as a transaction. The *END* statement tells the transaction manager that the transaction is finished and that it should be *committed*. Committing a transaction makes its effects on the database both visible to other transactions and permanent. The *ABORT* statement tells the transaction manager that execution of the transaction is to be stopped and that its effects on the database are to be discarded. Thus, enclosing a collection of database operations between *BEGIN* and *END* allows the applications programmer to make the operations appear as a single, atomic operation.

The aspect of transaction management that is relevant to this thesis is providing the property of serializability. The problem of guaranteeing serializability for concurrent transactions is known as the *concurrency control problem*, and a number of solutions to this problem are known (a comprehensive survey is given in [Bern81b]). This thesis will examine many of these solutions, or *concurrency control algorithms*, with the objective of determining which ones are superior according to a set of criteria to be introduced later in this chapter.

The dual problem of providing the recovery properties of the transaction abstraction will not be addressed in this study. It will be assumed throughout this thesis that a recovery algorithm known as *deferred updates* [Gray79] is employed. In this algorithm, data written by a transaction is not actually changed in the database until the transaction successfully commits. This recovery assumption is made so that all concurrency control algorithms can be evaluated in a common framework. Also, it reduces the number of parameters which might otherwise have to be varied in the performance studies. Other researchers are investigating the problem of jointly evaluating solutions to the concurrency control and recovery problems [Agra83b, Agra83c, Grif83].

In discussing concurrency control algorithms, the terms *object*, *data item*, and *granule* will be used in referring to parts of the database. The term *object* will be used to mean the smallest logical unit of data which is of interest. Typical examples of objects in a database are records or pages. The terms *data item* and *granule* will be used interchangeably to refer to units of the database upon which concurrency control decisions are based. The term *granule* in particular is used to mean a group of one or more objects which is treated as a single unit for concurrency control purposes.

1.2. CONCURRENCY CONTROL RESEARCH

Much research on algorithm construction has been done in the area of concurrency control for both single-site and distributed database systems. Some of this research has focused on the theory involved in proving the correctness of concurrency control algorithms, called *serializability theory* [Eswa76, Bern79, Papa79]. Other research has led to the development of new concurrency control algorithms, most of which are based on one of three mechanisms: *locking* [Mena78, Rose78, Gray79, Lind79, Ston79], *timestamps* [Reed78, Thom79, Bern80, Bern81b], and commit-time *validation* (also called certification) [Bada79, Casa79, Baye80, Kung81, Ceri82]. Bernstein and Goodman [Bern81b] survey many of these algorithms and describe how new algorithms can be created by combining these mechanisms.

Most concurrency control algorithm proposals deal solely with the two operations *read* and *write*. Recently, however, concurrency control algorithms have been proposed that use more information about transactions or recognize additional operations [Bern78, Garc83, Hsu83, Bern81a, Allc82, Schw82, Spec83]. Examples of the kinds of additional information which may be used are the order in which data items are accessed, the particular set of items which the transaction might access, and the manner in which writes are computed from reads. Examples of additional operations which might be recognized are an *insert* operation to create a new data item and a *delete* operation to eliminate a data item. The number of possible proposals is great, limited only by the amount and type of semantic information that applications programmers are willing or able to cope with.

When only read and write operations are allowed, serializability theory is concerned with two types of *dependencies* which can arise between transactions, *read/write* and *write/write* dependencies. Consider the execution of two transactions T_i and T_j. A *read/write* dependency holds from transaction T_i to transaction T_j if either T_i reads some data item which is later written by T_j or T_i writes some data item which is later read by T_j. A *write/write* dependency holds from T_i to T_j if T_i writes some data item which is later written by T_j. The existence of a *read/write* or *write/write* dependency from T_i to T_j implies that T_i must precede T_j in any serial execution of transactions which produces the same results as their concurrent execution. One can construct a graph, called a *dependency graph*, with transactions as nodes and an arc from T_i to T_j if there is a dependency from T_i to T_j. It can be shown that the concurrent execution of a collection of transactions is serializable if and only if the dependency graph for the execution is acyclic [Papa79]. For concurrency control algorithms based on other types of operations, other types of dependencies are defined between operations to provide the basis for serializability [Bern81b, Schw82].

As mentioned above, the most common types of concurrency control algorithms are locking, timestamps, and validation algorithms. Each type of algorithm seeks to prevent non-serializable executions of transactions. In locking algorithms, transactions are required to lock objects which they read or write. Locks can be set either at transaction startup time or dynamically as reads and writes are performed. When a transaction cannot set a lock because a conflicting transaction has the object locked, it must wait until the object is unlocked. All locking algorithms examined in this thesis are variations of a scheme known as *two-phase locking* [Eswa76, Gray79], where transactions must lock each object before accessing it and may not release any locks until all needed locks have been obtained. Thus, locks serve to serialize conflicting accesses to objects by blocking transactions wishing to make such accesses.

In timestamp algorithms, each transaction is assigned a timestamp when it starts running. Timestamps are sequence numbers, guaranteed to be unique, which provide a total ordering for transactions based on their startup order. Typical of timestamp algorithms is *basic timestamp ordering* [Bern81b]. In addition to transaction timestamps, each object has a read timestamp and a write timestamp in basic timestamp ordering. These are the timestamps of the youngest reader and the youngest writer of the object, respectively. (A transaction T_i is said to be younger than another transaction T_j if T_i has a larger timestamp.) These timestamps are used to force transactions which access a common object in a conflicting manner to do so in their startup order. Transactions attempting to violate the timestamp ordering are restarted (aborted and started over).

In validation algorithms, transactions are permitted to run freely until they reach their commit point. Upon reaching this point, each transaction is subjected to a test which ensures that committing it will not lead to non-serializable results. Transactions which fail this test are restarted. Typical of algorithms of this type is *serial validation*, an algorithm in which transaction readsets and writesets are maintained and tested for conflicting intersections at commit time [Kung81].

1.3. PERFORMANCE ISSUES

All concurrency control algorithms have a cost associated with the controls which they provide. Since it would be easy to simply require transactions to execute serially, one might question the decision not to achieve serializability in this simple manner. Several factors make concurrent transaction execution desirable. First, to achieve the best possible transaction throughput, it is necessary to keep the various hardware components busy. The more parallelism (such as CPU-I/O overlap) that can be achieved, the better the overall system performance will be. Running one transaction at a time makes achieving such overlap extremely difficult, leading to poor resource utilization. This problem is even more severe if transactions can pause for thinking in the middle of their execution. Second, system users always want fast response for their transactions. Serial transaction scheduling has the undesirable property of making short transactions wait for any long transactions which precede them regardless of whether or not they actually conflict. This leads to poor average response times. Allowing concurrent database accesses by non-conflicting transactions solves these potential problems.

Given that a concurrency control algorithm is needed, and that many algorithms have been proposed, the database system designer is faced with a difficult decision: Which concurrency control algorithm should be chosen? Several recent studies have evaluated concurrency control algorithm performance using qualitative, analytical, and simulation techniques. Bernstein and Goodman performed a comprehensive qualitative study which discussed performance issues for several distributed locking and timestamp algorithms [Bern80]. Results of analytical studies of locking performance have been reported by Irani and Lin [Iran79] and Potier and Leblanc [Poti80]. Simulation studies of locking done by Ries and Stonebraker provide insight into granularity versus concurrency tradeoffs [Ries77, Ries79a, Ries79b]. Analytical and simulation studies by Garcia-Molina [Garc79] provide some insight into the relative performance of several variants of locking as well as a voting algorithm [Thom79] and a ring algorithm [Elli77]. Simulation studies by Lin and Nolte [Lin82, Lin83] provide some comparative performance results for locking and several timestamp algorithms. A recent thesis by Galler [Gall82] provides a new analytical technique for locking, some qualitative techniques for comparing algorithms, and some simulation results for locking versus timestamps which contradict those of Lin and Nolte. A recent thesis by

Robinson [Robi82a] includes some experimental studies of locking versus serial validation (see also [Robi82b]).

These performance studies are informative, but they fail to offer definitive results regarding the choice of a concurrency control algorithm for several reasons. First of all, little attention has been given to the relative storage and CPU costs of the various algorithms. Second, the analytical and simulation studies have mostly examined either one or a few alternative algorithms, and they are based on a variety of system models and assumptions. Examples of modeling details which vary from study to study are whether transaction sizes are fixed or random, whether there is one or several classes of transactions, which system resources are modeled and which are omitted, and what level of detail is used in representing resources which are included in the models. This makes it difficult to arrive at general conclusions about the alternative algorithms. Third, the models used in many cases are insufficiently detailed to reveal certain important effects. For example, some models group the I/O, CPU, and message delay times for transactions into a single random delay [Lin82, Lin83], in which case the performance benefit of achieving CPU-I/O overlap cannot be revealed. Finally, the few comprehensive studies of alternative algorithms which have been performed were of a non-quantitative nature.

1.4. THESIS OVERVIEW

This thesis reports on a study of concurrency control alternatives which is both more comprehensive and more conclusive than previous studies. Two models of concurrency control algorithms are developed and used to obtain information about how various algorithms compare with one another. The first model is an abstract model which provides a uniform framework for describing concurrency control algorithms in terms of the information which they store, when they require transactions to block or restart, and the way in which they process concurrency control requests. Descriptions of alternative algorithms in this framework are used to perform simple analyses of the costs associated with the algorithms.

The second model presented is a performance model, a closed queuing model of a database system from the perspective of a concurrency control algorithm. This model has been implemented in the form of a substantial simulation program, and it serves as a general framework for studying the performance of concurrency control algorithms. Most of the simulator is algorithm-independent, allowing various algorithms to be described in terms of a small amount of code. Once described for the simulator, all algorithms can be subjected to the same system and transaction workload characteristics. Thus, the simulation model facilitates fair comparisons of the performance of alternative concurrency control algorithms.

This thesis concentrates entirely on the single-site concurrency control problem in which the only operations performed by transactions are read and write. Little is known about concurrency control costs and performance in this environment, so it seems appropriate to study alternative algorithms in hopes of identifying some general principles in this environment. Where appropriate, the implications of these findings on other types of concurrency control algorithms will be discussed, and extensions to the cost and performance models will be proposed for future investigations of distributed concurrency control algorithms.

Chapter 2 of the thesis presents the techniques used for comparing the storage and CPU costs of concurrency control algorithms. The abstract model is used to describe three different algorithms, one based on locking, one based on timestamps, and one based on validation. Storage and CPU cost results are obtained for the three algorithms described, and model extensions are suggested for dealing with both multiple version and distributed concurrency control schemes.

Chapter 3 of the thesis describes the model used to study the performance of alternative concurrency control algorithms. The simulation model of a database system, the transaction workload model, and a set of benchmark workloads for performance studies are all presented in this chapter. Using the simulation model, the performance of seven variants of concurrency control algorithms are studied, and conclusions are drawn about the relative

CHAPTER 2

CONCURRENCY CONTROL COSTS

In this chapter, a model for evaluating the costs associated with alternative concurrency control algorithms is described. The model, first reported in [Care83a], is intended to facilitate descriptions and analyses of single-site concurrency control algorithms. Descriptions of a two-phase locking algorithm, a timestamp algorithm, and a validation algorithm are formulated using the model, and these descriptions are analyzed in order to compare the relative costs associated with these algorithms. At the end of the chapter, some preliminary ideas are described for future extensions of the model for evaluating the costs associated with multiple version, hierarchical, and distributed concurrency control algorithms.

2.1. OVERVIEW

The cost analysis techniques for single-site concurrency control algorithms are based on an abstract model. This model contains a single *concurrency control scheduler*, which makes scheduling decisions based on information that it maintains about the history of requests received to date. This information is referred to as the *concurrency control database*, and is treated conceptually as a simple, relational database, ignoring the many data strucmerits of blocking and restarts as tactics for enforcing serializability.

Chapter 4 examines two approaches which have been suggested for improving the performance of existing concurrency control algorithms, multiple versions of objects and granularity hierarchies. Some of the existing proposals for concurrency control algorithms based on multiple versions and granularity hierarchies are reviewed, and several new algorithms based on these ideas are developed. In particular, the use of granularity hierarchies, previously proposed only for use in conjunction with locking algorithms, is generalized for use with other types of concurrency control mechanisms. The performance of several algorithms based on each of these two types of performance improvements are studied.

Finally, Chapter 5 summarizes the key results obtained in the previous chapters and presents some general concurrency control principles that have been identified in the course of this study. The conclusions of the thesis are presented, and topics for future work in concurrency control are identified.

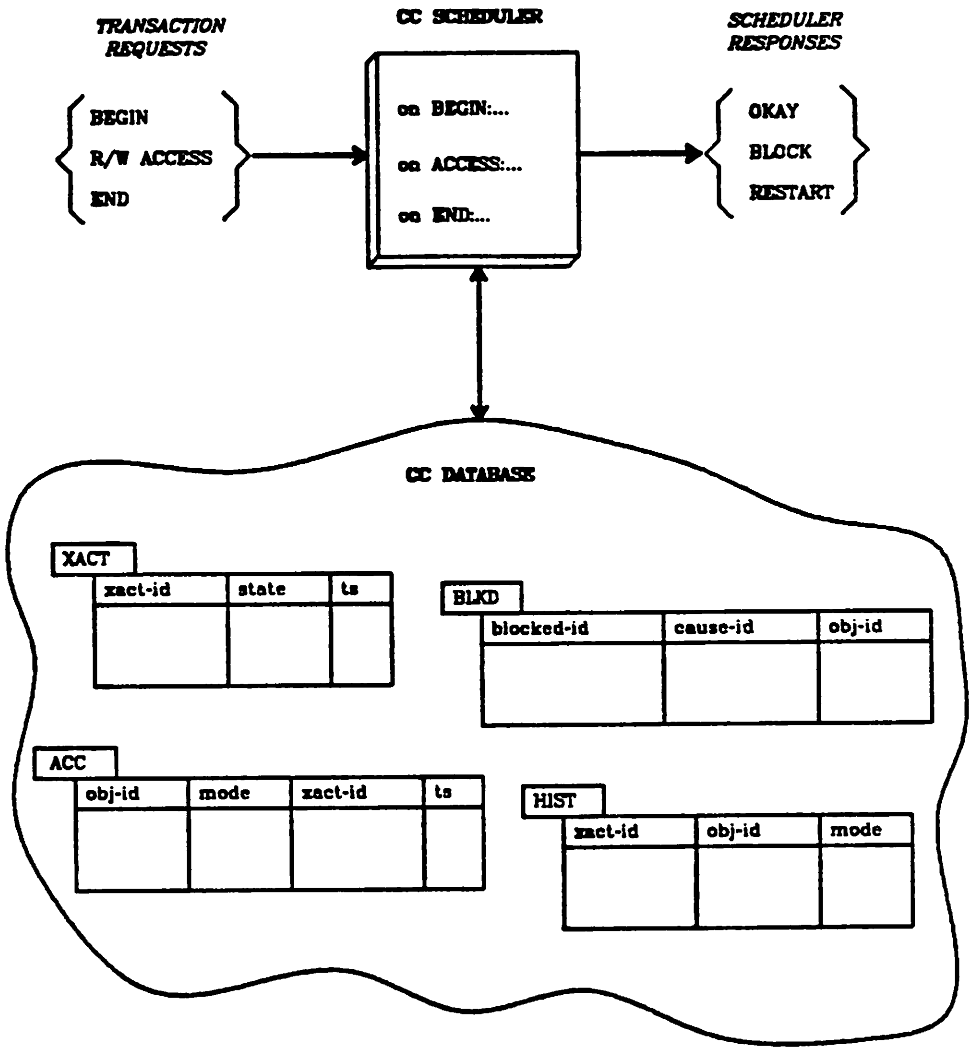

Figure 2.1: Abstract concurrency control model.

tures which might be used in an actual implementation. For a particular concurrency control algorithm, the scheduler obeys a well-defined set of rules which describe how it should respond to incoming requests, based both on the requests themselves and on the contents of the concurrency control database. For reasons of simplicity, conciseness, and implementation independence, these rules are formulated as relational database queries. The abstract model is summarized in Figure 2.1.

2.2. TRANSACTION REQUESTS

The abstract model recognizes three types of requests from transactions: *BEGIN*, *END*, and *ACCESS*. The first two mark the beginning and the end of transaction execution, and the latter indicates that the requesting transaction wishes to access one or more objects. A given transaction may make any number of *ACCESS* requests during its execution. When the scheduler receives an *ACCESS* request, it also receives a collection of (*obj–id*, *mode*) pairs indicating the objects and modes (read or write), associated with the current request. This collection is referred to as the *REQ* relation for the purpose of formulating concurrency control algorithms in relational terms. It is assumed in the model that transactions abide by the responses received from the scheduler, accessing data objects accordingly. It is also assumed that writes go to a list of deferred updates [Gray79] to be installed as new data values at transaction commit time.

2.3. THE CONCURRENCY CONTROL DATABASE

The concurrency control database, shown in Figure 2.2, consists of four relations. The *XACT* relation contains transaction state information, specifying the transaction identifier, state (ready, blocked, committed, aborted), and timestamp of each current transaction. The *ACC* relation contains

XACT(xact-id,state,ts)
ACC(obj-id,mode,xact-id,ts)
BLKD(blocked-id,cause-id,obj-id)
HIST(xact-id,obj-id,mode)

Figure 2.2: Concurrency control database.

information about accesses to objects, specifying the object identifier, access mode (read or write), transaction identifier, and timestamp for each current or recent access. This relation plays the role of a concurrency control table in algorithm descriptions. For locking, the *ACC* relation will store current access information in the form of lock table entries, and it will store information about current and recent accesses in the form of timestamp entries for basic timestamp ordering. The *BLKD* relation contains information about any blocked transactions, containing the transaction identifiers of these transactions, the transaction identifiers of the transactions which they are waiting for, and the identifier of the object which is the source of the conflict which led to the blocking action. It is assumed that deleting a *BLKD* relation entry implicitly unblocks the corresponding transaction, allowing it to continue processing where it left off. The *HIST* relation stores histories of *ACCESS* requests which are conditionally granted until a concurrency control decision is made at transaction commit time (such as in serial validation). Entries in this relation specify the transaction identifiers, object identifiers, and access modes associated with such requests.

Not all concurrency control algorithms use all of the relations in the concurrency control database, as this set of relations is intended to represent the collection of all possible information which algorithms might require. For the same reason, not all concurrency control algorithms use all of the fields of these relations. Thus, the portion of the concurrency control database used by an algorithm is specified as part of its description.

2.4. ALGORITHM DESCRIPTIONS

Concurrency control algorithm descriptions in the abstract model have three parts. These are:

(1) A list of the concurrency control database relations and fields used by the algorithm.

(2) A pair of views, *BLKCFL* and *RSTCFL*, which define the situations where blocking or restarting are called for, respectively. Each of these is a view in the relational database sense [Ullm83], a query which is dynamically evaluated upon reference.

(3) Three query sets, describing the actions to be taken on receipt of *BEGIN*, *ACCESS*, and *END* requests. These query sets access the concurrency control database and *REQ* relation associated with the current request and are assumed to execute atomically when invoked. The syntax for the query sets is based on the syntax of the QUEL query language [Ston76], with deviations or additional high-level macro-

operations introduced in cases where a QUEL description is difficult or impossible (such as checking for cycles in the *BLKD* relation in the upcoming description of locking).

2.5. USING THE MODEL

In this section, the descriptive power of the single-site abstract model is demonstrated by using the model to describe the two-phase locking [Gray79], basic timestamp ordering [Bern81b], and serial validation [Kung81] algorithms. Several liberties are taken with the QUEL syntax in the process. First, range statements are omitted. Second, the macro-operations shown in Figures 2.3 through 2.5 are defined. The *BLOCK* operation blocks a specified transaction, recording its transaction identifier and the identifier of the transaction which it is waiting for in the *BLKD* relation. The *EXPUNGE* operation deletes all of the information associated with a specified transaction, and is used at transaction commit or restart time. The *RESTART* operation restarts a specified transaction. A fourth macro-operation, *CYCLE*(*xact−id*), is also used in the locking description. This macro-operation searches for cycles of blocked transactions in the *BLKD* relation involving a specified transaction and returns true if and only if a cycle is found. (This last operation cannot be conveniently specified in QUEL.) Finally, the existence of several global variables, such as *req−xact−id*, the identifier for the transaction making the current request, is assumed. Other such variables will be assumed and commented upon as they seem reasonable and convenient.

```
BLOCK(xact-id1,xact-id2) =
{
  replace XACT(state = "blocked")
    where XACT.xact-id = xact-id1
  append to BLKD(xact-id1,xact-id2)
}
```

Figure 2.3: Definition of *BLOCK* macro-operation.

```
EXPUNGE(xact-id) =
{
  delete XACT
    where XACT.xact-id = xact-id
  delete ACC
    where ACC.xact-id = xact-id
  delete BLKD
    where BLKD.blocked-id = xact-id
    or BLKD.cause-id = xact-id
  delete HIST
    where HIST.xact-id = xact-id
}
```

Figure 2.4: Definition of *EXPUNGE* macro-operation.

```
RESTART(xact-id) =
{
  replace XACT(state = "aborted")
    where XACT.xact-id = xact-id
  EXPUNGE(xact-id)
}
```

Figure 2.5: Definition of *RESTART* macro-operation.

2.5.1. Two-Phase Locking

In *two-phase locking* (2PL) [Gray79], the concurrency control scheduler maintains a lock table. Transactions set read and write locks on objects before accessing them, and they release their locks at commit time. A transaction may set a read lock on an object as long as no other transaction has a write lock set on the object, and a transaction may set a write lock an object if no other transaction has a read or write lock set on the object. When a transaction tries to set a lock and fails, it must wait until the lock is released and then try again. Deadlocks are a possibility, and must either be prevented or detected and broken by restarting one of the transactions involved. An informal description of two-phase locking is given in Figure 2.6.

The linear-time deadlock detection algorithm of Agrawal, Carey, and DeWitt [Agra83a] is used for this example. In this algorithm, when a transac-

```
procedure readReq($T$,$x$);
begin
  if writeLocked($x$) then
    block($T$);
    if cycle($T$) then
      restart($T$);
    fi;
  else
    grant readReq;
    readLock($T$,$x$);
  fi;
end;

procedure writeReq($T$,$x$);
begin
  if readLocked($x$) or writeLocked($x$) then
    block($T$);
    if cycle($T$) then
      restart($T$);
    fi;
  else
    grant writeReq;
    writeLock($T$,$x$);
  fi;
end;
```

Figure 2.6: Informal description of 2PL algorithm.

tion T_i is forced to wait for a lock on some object X, it blocks on exactly one of the transactions T_j which holds a lock on X. If more than one transaction holds a lock on X, one is chosen arbitrarily. As shown in [Agra83a], if deadlocks are checked each time a transaction must wait, the cycle-checking operation (i.e., the deadlock detector) can operate in a very efficient manner. Figures 2.7 through 2.9 give a description of this variation of 2PL using the abstract model.

The subset of the concurrency control database needed for 2PL is specified in Figure 2.7. In Figure 2.8, the conditions under which blocking and restarts are required are defined as views. The *BLKCFL* view says that a block conflict has occurred if there is an *ACC* relation entry for one of the

```
XACT(xact-id,state)
ACC(xact-id,mode,obj-id)
BLKD(blocked-id,cause-id)
```

Figure 2.7: Concurrency control database for 2PL.

```
define view BLKCFL(xact-id = ACC.xact-id)
  where REQ.obj-id = ACC.obj-id
  and ACC.xact-id != req-xact-id
  and ((REQ.mode = "read"
    and ACC.mode = "write")
    or (REQ.mode = "write"))

define view RSTCFL(xact-id = BLKD.xact-id)
  where CYCLE(BLKD.blocked-id)
  and BLK.blocked-id = req-xact-id
```

Figure 2.8: Block and restart conflict views for 2PL.

```
on BEGIN:

  append to XACT(req-xact-id,"ready")

on ACCESS:

  replace ACC(mode = REQ.mode)
    where not any(BLKCFL)
    and ACC.obj-id = REQ.obj-id
    and ACC.xact-id = req-xact-id

  append to ACC
    (req-xact-id,REQ.mode,REQ.obj-id)
    where not any(BLKCFL)
    and not any(ACC.obj-id
      where ACC.obj-id = REQ.obj-id
      and ACC.xact-id = req-xact-id)

  BLOCK(req-xact-id,BLKCFL.xact-id)
    where any(BLKCFL)
    and BLKCFL.xact-id =
      min(BLKCFL.xact-id)

  RESTART(req-xact-id)
    where any(BLKCFL) and any(RSTCFL)

on END:

  replace XACT(state = "committed")
    where XACT.xact-id = req-xact-id
  EXPUNGE(req-xact-id)
```

Figure 2.9: Request processing queries for 2PL.

current requests, and either the current request is a read request and the *ACC* entry is a write entry, or else the current request is a write request (in which case the mode of the *ACC* entry does not matter). In other words, the *ACC* relation serves as a lock table, and a transaction must block if an incompatible lock is already set on an object that it wants to access. The *RSTCFL* view says that a restart conflict has occurred if there is a cycle in the *BLKD* relation involving the current requesting transaction. In other words, a transaction must restart if it is the cause of a deadlock. (This is not necessarily the best victim to select from a performance standpoint.)

Figure 2.9 gives the query sets for processing requests under 2PL. When a *BEGIN* request arrives, the state of the requesting transaction is set to indicate that it is ready to run. When an *ACCESS* request arrives, the *BLKCFL* view is materialized. If no block conflicts exist (i.e., the *BLKCFL* view is empty), then the *ACC* relation is updated to indicate that locks have been granted on all requested objects. If a block conflict does exist (i.e., the *BLKCFL* view is not empty), the requesting transaction is blocked on one of the conflicting transactions (the one with the smallest transaction identifier is arbitrarily picked here), and the *RSTCFL* view is materialized. If a restart conflict exists, the requesting transaction is restarted. This corresponds to granting requests if no locks interfere, blocking a transaction if one or more locks are unobtainable, and restarting a transaction if it becomes the cause of a deadlock condition.

2.5.2. Basic Timestamp Ordering

In the basic timestamp ordering (BTO) algorithm [Bern81b], each transaction T has a timestamp, $TS(T)$, which is issued at the time that T begins executing. Associated with each data item x in the database is a read timestamp, $R\text{-}TS(x)$, and a write timestamp, $W\text{-}TS(x)$. These timestamps record the timestamps of the latest reader and writer (respectively) for x, and are

```
procedure readReq($T$,$x$);
begin
  if $TS(T) < W\text{-}TS(x)$ then
    restart($T$);
  else
    grant readReq;
    $R\text{-}TS(x) := \max(TS(T), R\text{-}TS(x))$;
  fi;
end;

procedure writeReq($T$,$x$);
begin
  if $TS(T) < R\text{-}TS(x)$ or $TS(T) < W\text{-}TS(x)$ then
    restart($T$);
  else
    grant writeReq;
    $W\text{-}TS(x) := TS(T)$;
  fi;
end;
```

Figure 2.10: Informal description of BTO algorithm.

maintained in a timestamp table. (Entries with timestamps older than the oldest active transaction may be deleted from the table since they will never cause an active transaction to be restarted.) A read request from T for x is rejected if $TS(T) < W\text{-}TS(x)$, and a write request from T for x is rejected if $TS(T) < W\text{-}TS(x)$ or $TS(T) < R\text{-}TS(x)$. Transactions whose requests are rejected are restarted, causing serialization to occur in timestamp order. Deadlock is impossible, although cyclic restarts are a possibility [Date82, Lin82, Ullm83]. The BTO algorithm is described informally in Figure 2.10.

For the purpose of this example, read requests will be processed as they arrive, and all write requests will be processed together just prior to transaction commit time. This simplifies the considerations involved in making BTO

```
XACT(xact-id,state,ts)
ACC(ts,mode,obj-id)
HIST(xact-id,obj-id)
```

Figure 2.11: Concurrency control database for BTO.

```
define view RSTCFL(obj-id = ACC.obj-id)
  where (REQ.obj-id = ACC.obj-id
    and ACC.ts > req-ts
    and (REQ.mode = "read"
      and ACC.mode = "write")
    and req-type = ACCESS)
  or (HIST.obj-id = ACC.obj-id
    and HIST.xact-id = req-xact-id
    and ACC.ts > req-ts
    and req-type = END)
```

Figure 2.12: Restart conflict view for BTO.

work with deferred updates, as otherwise some scheduling would be required to prevent transactions from reading objects for which a write request has been processed but the associated deferred update has not yet taken place [Bern81b, Agra83b]. Figures 2.11 through 2.13b give a description of BTO using the model. The global variable *req-ts* is assumed to contain the timestamp of the transaction making the current request. The macro-operation *CURRENT-TS*() is assumed to return the current timestamp value, implicitly increasing its value by one and setting the global variable *current-ts* to the value of the current timestamp. The global variable *oldest-ts* is assumed to contain the timestamp of the oldest active transaction. The global variable *req-type* is assumed to indicate the type of the current request.

```
on BEGIN:

   append to XACT
      (req-xact-id,"ready",CURRENT-TS())

on ACCESS:

   replace ACC(ts = max(ACC.ts,req-ts)
      where not any(RSTCFL)
      and REQ.mode = "read"
      and ACC.mode = "read"
      and ACC.obj-id = REQ.obj-id

   append to ACC
      (req-ts,REQ.mode,REQ.obj-id)
      where not any(RSTCFL)
      and REQ.mode = "read"
      and not any(ACC.obj-id
         where ACC.obj-id = REQ.obj-id
         and ACC.mode = "read")

   append to HIST(req-xact-d,REQ.obj-id)
      where REQ.mode = "write"

   RESTART(XACT.xact-id)
      where XACT.xact-id = REQ.xact-id
      and any(RSTCFL)
      and REQ.mode = "read"
```

Figure 2.13a: Request processing queries for BTO.

While this description appears a bit lengthy, its semantics are actually relatively simple. The *ACC* relation plays the role of the timestamp table for *BTO*. The "**append to** *ACC*..." portion of the query set for *ACCESS* requests in Figure 2.13a handles the case where there is no current timestamp for a requested object, recording a new one. The "**replace** *ACC*..." portion of the query set for *ACCESS* requests handles the case where there is a current timestamp for the object, updating it as called for by the BTO algorithm. The *HIST* relation is used to defer write timestamp checking until commit time, with similar timestamp checking and updating involving the *HIST* relation occurring in the *END* request portion of the description in Figure 2.13b.

```
on END:

   replace XACT(state = "committed")
      where XACT.xact-id = req-xact-id
      and not any(RSTCFL)

   replace ACC(ts = max(ACC.ts,req-ts)
      where not any(RSTCFL)
      and ACC.mode = "write"
      and ACC.obj-id = HIST.obj-id
      and HIST.xact-id = req-xact-id

   append to ACC(req-ts,HIST.obj-id,"write")
      where not any(RSTCFL)
      and HIST.xact-id = req-xact-id
      and not any(ACC.obj-id
         where ACC.obj-id = HIST.obj-id
         and ACC.mode = "write")

   RESTART(XACT.xact-id)
      where XACT.xact-id = req-xact-id
      and any(RSTCFL)

   delete HIST
      where HIST.xact-id = req-xact-id
   delete XACT
      where XACT.xact-id = req-xact-id
   delete ACC
      where ACC.ts < oldest-ts
```

Figure 2.13b: Request processing queries for BTO (cont.).

2.5.3. Serial Validation

The serial validation (SV) algorithm [Kung81] requires that the readsets and writesets of all transactions be recorded as they execute. These readsets and writesets are the sets of items which the transaction reads and writes,

```
procedure validate($T$);
begin
  valid := true;
  foreach $T_{rc}$ in $RC(T)$ do
    foreach $x_r$ in $readset(T)$ do
      foreach $x_w$ in $writeset(T_{rc})$ do
        if $x_r = x_w$ then
          valid := false;
        fi;
      od;
    od;
  od;
  if valid then
    commit $writeset(T)$ to database;
  else
    restart($T$);
  fi;
end;
```

Figure 2.14: Informal description of SV algorithm.

respectively. Transactions are allowed to execute freely until commit-time, writing their database changes into a list of deferred updates. Each transaction is subjected to a commit-time validation procedure in a critical section (a section of code which excludes other transactions from making concurrency control requests). This validation procedure is used to ensure that committing the transaction will not leave the database in an inconsistent state. Let $RC(T)$ be the set of *recently committed* transactions, i.e., those which commit between the time when T starts executing and the time at which T enters the critical section for validation. Transaction T is validated if $readset(T) \cap writeset(T_{rc}) = \emptyset$ for all transactions $T_{rc} \in RC(T)$. If T is validated, its updates are applied to the database; otherwise, it is restarted. An informal description of the serial validation algorithm is given in Figure

2.14.

Rather than write a description of serial validation as originally presented [Kung81], a new, potentially more efficient version with different but provably equivalent semantics will be described. In this version, each transaction is assigned a startup timestamp, $S\text{-}TS(T)$, at startup time, and each transaction receives a commit timestamp, $C\text{-}TS(T)$, when it enters its commit processing phase. A write timestamp, $TS(x)$, is maintained for each data item x; $TS(x)$ is the commit timestamp of the most recent (committed) writer of x. A transaction T will now be allowed to commit if and only if $S\text{-}TS(T) > TS(x_r)$ for each object x_r in its readset. Each transaction T which successfully commits will update $TS(x_w)$ to be $C\text{-}TS(T)$ for all data items x_w in its writeset.

```
procedure validate($T$);
begin
  valid := true;
  foreach $x_r$ in $readset(T)$ do
    if $S\text{-}TS(T) < TS(x_r)$ then
      valid := false;
    fi;
  od;
  if valid then
    foreach $x_w$ in $writeset(T)$ do
      $TS(x_w) := C\text{-}TS(T)$;
    od;
    commit $writeset(T)$ to database;
  else
    restart($T$);
  fi;
end;
```

Figure 2.15: Informal description of revised SV algorithm.

```
XACT(xact-id,state,ts)
ACC(ts,obj-id)
HIST(xact-id,mode,obj-id)
```

Figure 2.16: Concurrency control database for SV.

```
define view RSTCFL(obj-id = HIST.obj-id)
  where HIST.obj-id = ACC.obj-id
  where HIST.xact-id = req-xact-id
  where HIST.mode = "read"
  and ACC.ts > req-ts
```

Figure 2.17: Restart conflict view for SV.

It is fairly easy to show that this test is equivalent to the original readset/writeset test of [Kung81]. A formal equivalence proof is given in Appendix 1. An informal description of the revised SV algorithm is given in Figure 2.15, and Figures 2.16 through 2.18 give a description of SV using the model. For typical transaction mixes, it is expected that $RC(T)$ will tend to be larger than one and the writesets of transactions will not be overly large. The revised SV algorithm will entail less CPU cost than the original SV algorithm for such mixes. In the original version, the commit-time test involves checking $|RC(T)|$ writesets for each object x_r, whereas a single timestamp is checked for each x_r in the revised version. The revised version involves an additional cost for updating $TS(x_w)$ for each x_w, but this is unlikely to be significant compared to the cost reduction for testing the readset.

```
on BEGIN:
  append to XACT
    (req-xact-id,"ready",CURRENT-TS())

on ACCESS:
  append to HIST
    (req-xact-id,REQ.mode,REQ.obj-id)

on END:

  replace XACT(state = "committed")
    where XACT.xact-id = req-xact-id
    and not any(RSTCFL)

  RESTART(XACT.xact-id)
    where XACT.xact-id = req-xact-id
    and any(RSTCFL)

  replace ACC(ts = current-ts)
    where not any(RSTCFL)
    and HIST.mode = "write"
    and ACC.obj-id = HIST.obj-id
    and HIST.xact-id = req-xact-id

  append to ACC
    (obj-id = HIST.obj-id,ts = current-ts)
    where not any(RSTCFL)
    and HIST.mode = "write"
    and HIST.xact-id = req-xact-id
    and not any(ACC
      where ACC.obj-id = HIST.obj-id)

  delete HIST
    where HIST.xact-id = req-xact-id
  delete XACT
    where XACT.xact-id = req-xact-id
  delete ACC
    where ACC.ts < oldest-ts
```

Figure 2.18: Request processing queries for SV.

2.6. ALGORITHM COST COMPARISONS

In this section, techniques are presented for using the model to compare the relative cost characteristics of various concurrency control algorithms. The storage and CPU costs are compared via a simple complexity analysis, based on implementation-independent units of CPU and storage costs. These cost units are based on ideas presented in [Bern80], where table accesses and table entries were informally used to compare algorithm costs. The analysis techniques are illustrated by using them to compute and compare the costs of the three algorithms described in the previous section.

To facilitate cost analyses, a performance model based on a set of simple parameters is used. The parameters will be defined as though the transaction mix used to evaluate algorithm costs consists of transactions of the same fixed size. The performance parameters used here are not all independent, so it would be difficult to carry out an accurate expected-value analysis of algorithm costs. The techniques applied here can be thought of as a formal analysis of a simple transaction mix, or alternatively as a mean-value approximation [Ferr78] to an analysis of a mix where the parameters are interpreted as being averages. The problem of carrying out a more precise analysis of average costs is left for future work.

Let T_s be the number of transactions in the system (i.e., the multiprogramming level). Let R be the readset size for these transactions, and let F_w be the fraction of the readset also included in the writeset. Each transaction thus makes $R(1+F_w)$ data access requests. (It is assumed that the writeset is a subset of the readset for each transaction, and that transactions do not make the same request twice). Let F_b be the fraction of blocked transactions, so that $F_b T_s$ is the current number of blocked transactions. Let F_{rc} be the recent commit factor, so that $F_{rc}T_a$ is the number of recently committed transactions, where a recently committed transaction is one which committed since the oldest remaining active transaction began running. These parameters are summarized in Table 2.1.

The blocking and restart characteristics of algorithms will influence the parameters F_b and F_{rc}, so they will vary from algorithm to algorithm. The parameter F_w is determined solely by the transaction mix. To bound these parameters, note that $0 \leq F_b \leq 1$ and $0 \leq F_w \leq 1$. For the parameter F_{rc}, however, all that is certain is that $F_{rc} \geq 0$, as F_{rc} is determined by the variance in running times for transactions in the mix. One would expect transactions to commit roughly in their startup order if all are truly the same size, and this would produce a small value for F_{rc}. However, a very long transaction mixed with a collection of short transactions would result in a large value for F_{rc}, as many short transactions could complete during the lifetime of the long transaction.

Simple Cost Parameters	
T_a	number of transactions in system
R	readset size for transactions
F_w	fraction of readset that is written
F_b	fraction of blocked transactions
F_{rc}	recent commit factor

Table 2.1: Parameters for algorithm cost analyses.

2.6.1. Storage Cost

In order to compare the storage costs of various concurrency control algorithms, the sizes of the relations in the concurrency control database portion of their models may be analyzed. One field of one tuple of one relation is taken as the unit of storage cost for this analysis. Given an algorithm, the tuple widths of the concurrency control database relations are determined by the algorithm description. The cardinalities of the relations can be determined by considering the behavior of the query sets in the description using the simple performance model just described. The overall database size is simply the sum of the products of the cardinalities and tuple widths for each relation in the database. Both upper and lower bounds on the storage cost of algorithms may be determined by considering both possible extremes of the degree to which requests from different transactions have objects in common.

The 2PL algorithm will be analyzed first. The *XACT* relation represents a storage cost of $2T_a$, and the *BLKD* relation represents a cost of $2F_b T_a$. For the *ACC* relation, a storage cost of $3T_a(1-F_w)R$ is incurred for storing read locks (note that only one lock is set on objects that are to be written). For storing write locks, the cost can vary from as low as $3F_w R$, in the case where all T_a transactions write the same objects, to as high as $3T_a F_w R$, in the case where no two transactions write the same object. Thus, for 2PL:

$$STO_{2PL} \leq 2T_a(1+F_b)+3T_aR \qquad (1a)$$

$$STO_{2PL} \geq 2T_a(1+F_b)+3T_aR(1-F_w)+3F_wR \qquad \text{(1b)}$$

The BTO algorithm is considered next. The *XACT* relation represents a storage cost of $3T_a$. The *HIST* relation must store write request entries for the T_a active transactions, so it represents a storage cost of $2T_aF_wR$. In addition, the *ACC* relation must store the timestamps associated with recently-accessed objects. The amount of storage required for this information depends upon the degree of overlap between transactions. In the case where all transactions access totally different objects, the *ACC* relation must hold R read timestamp entries and RF_w write timestamp entries for each of the T_a active transactions plus the $F_{rc}T_a$ recently committed transactions. This yields a worst-case total storage cost for the *ACC* relation of $3T_a(1+F_{rc})R(1+F_w)$. At the other extreme, if all active and recently committed transactions access the exact same set of objects, the storage cost of the *ACC* relation is just $3R(1+F_w)$, since each object has at most one read timestamp entry and one write timestamp entry. Thus, for BTO:

$$STO_{BTO} \leq 3T_a(1+F_{rc})R(1+F_w)+T_a(3+2F_wR) \qquad \text{(2a)}$$

$$STO_{BTO} \geq 3R(1+F_w)+T_a(3+2F_wR) \qquad \text{(2b)}$$

The SV algorithm is considered last. The *XACT* relation again represents a storage cost of $3T_a$. The *HIST* relation must store read and write request entries for the T_a active transactions, so it represents a storage cost of $3T_aR(1+F_w)$. In addition, the *ACC* relation must store the timestamps associated with recently-accessed objects. As in the BTO algorithm, the amount of storage required for this information depends upon the degree of overlap between transactions. In the case where all transactions access totally different objects, the *ACC* relation must hold RF_w write timestamp entries for each of the T_a active transactions plus the $F_{rc}\,T_a$ recently committed transactions. This yields a worst-case total storage cost for the *ACC* relation of $2T_a(1+F_{rc})RF_w$. At the other extreme, if all active and recently committed transactions access the exact same set of objects, the storage cost of the *ACC* relation is just $2RF_w$, since each object has at most one timestamp entry. Thus, for SV:

$$STO_{SV} \leq 2T_a(1+F_{rc})RF_w + 3T_a(1+R(1+F_w)) \tag{3a}$$

$$STO_{SV} \geq 2RF_w + 3T_a(1+R(1+F_w)) \tag{3b}$$

Given the bounds on F_b and F_w, some conclusions can be drawn about the relative storage costs of the algorithms. From equations (1a), (2a), and (3a), it can be concluded that 2PL has the smallest worst-case storage cost of the three algorithms, which is $(4+3R)T_a$. The worst-case storage costs of the other two algorithms are dependent on the parameter F_{rc}, which is unbounded. A more detailed analysis of these equations reveals that the worst-case storage cost of SV is strictly smaller than that of BTO (assuming the same F_{rc} values for the two algorithms). Moreover, if $F_b \leq 1/2$, 2PL is certain to have a smaller worst-case storage cost than both SV and BTO.

The worst-case storage cost occurs when transactions do not compete for the same data items, which is likely to be the case for real mixes of transactions according to the analysis of the probability of conflicts in [Gray81a]. Thus, F_b is likely to be small for 2PL, leading to the conclusion that 2PL dominates SV, and SV in turn dominates BTO, with respect to worst-case storage cost.

A comparison of equations (2b) and (3b) reveals that, with respect to best-case storage cost, BTO dominates SV for $T_s \geq 3$. Comparing equations (1b) and (3b), it is seen that, if $F_b \leq 1/2$, 2PL is certain to dominate SV as well. Finally, a comparison of equations (1b) and (2b) indicates that BTO dominates 2PL unless $F_w \geq 3/5$ and $F_b \leq 1/2$. Since the best-case costs apply when transactions tend to conflict (access the same objects), this combination of F_w and F_b is impossible. If F_w is large, transactions will be competing for write locks on these shared objects. Lots of blocking will result, leading to the conclusion that the (non-independent) parameters F_b and F_w cannot realistically take on these values at the same time if transactions tend to request common data items. Hence, BTO dominates 2PL with respect to best-case storage cost.

To summarize the overall storage cost results, then, SV is the worst of the three algorithms. 2PL is best in terms of worst-case storage cost, indicating that it is superior under low-conflict transaction mixes. BTO is best in terms of best-case storage cost, meaning that it is best under high-conflict transaction mixes. These results are shown in Table 2.2 at the end of the following section.

2.6.2. CPU Cost

The number of operations involved in executing the query sets for various algorithms is analyzed in order to compare their CPU costs. The unit of CPU cost for this analysis is taken to be one tuple access, insertion, or replacement in one relation, so the assumption is that the CPU time required is proportional to the number of table lookups (as proposed in different terms by Bernstein and Goodman [Bern80]). CPU cost is not charged for accesses to the REQ relation, as this relation is simply used to facilitate modeling the way transactions pass requests to the scheduler in a QUEL query language setting.

Unfortunately, analyzing the CPU cost of a given concurrency control algorithm is, in general, considerably more complex than analyzing the storage cost of the algorithm. In this paper only the no-conflict CPU cost [Bada81], the CPU cost experienced by a transaction which does not conflict in any way with other concurrent transactions, is considered. Since actual conflicts are reported to be rare [Gray81a], the no-conflict CPU cost should be a reasonable "first-order" estimate. The problem of generalizing the analysis to include the additional sources of CPU cost associated with transactions which must restart or repeat requests due to blocking is left for future work.

2PL is again considered first. The cost of processing a *BEGIN* request is 1. The cost of materializing the *BLKCFL* view is 1, so the cost of processing $R(1+F_w)$ data access (*ACCESS*) requests is $2R(1+F_w)$ if no blocking occurs. The cost of processing an *END* request is $3+R$, 1 to change the state of the committing transaction and $2+R$ to delete all the information about the transaction (assuming one *BLKD* access to determine the lack of blocked transactions). Hence, for 2PL:

$$CPU_{2PL} = 4+R(3+2F_w) \tag{4}$$

BTO is considered next. The cost of processing a *BEGIN* request is again 1. The cost of materializing the *RSTCFL* view is 1 for an *ACCESS* request restart conflict check and 2 for an *END* request check for each write entry in the *HIST* relation. Thus, the cost of processing R read requests, each of which checks for a restart conflict and then conditionally inserts or updates a timestamp in the no-conflict case, is $2R$. The cost of processing RF_w write requests, each of which simply records the pending request in the *HIST* relation, is RF_w.

The cost of processing an *END* request for BTO depends on the number of timestamps deleted at that time. In the no-conflict case, it is assumed that all transactions access different data items, meaning that all timestamps associated with a given transaction must eventually be explicitly deleted. This timestamp deletion cost is charged to the transaction creating the timestamp, even though deletion may occur at some later point in time. Thus, the cost of processing an *END* request is $2RF_w$ to check the *HIST* relation contents for restart conflicts, 1 to change the state of the committing transaction, $2RF_w$ to update the write timestamp of each write request in the *HIST* relation once the transaction has indeed committed, and $1+R(1+2F_w)$ to delete the information associated with the transaction. Hence, for BTO:

$$CPU_{BTO} = 3+R(3+7F_w) \tag{5}$$

The analysis for SV is much like that for BTO. The cost of processing a *BEGIN* request is 1. The cost of processing $R(1+F_w)$ read and write requests, each of which simply records the pending request in the *HIST* relation for later consideration, is $R(1+F_w)$. The timestamp deletion cost for SV is charged to the transaction creating the timestamp, as in the analysis for BTO. The cost of materializing the *RSTCFL* view is 2 for an *END* request check for each read entry in the *HIST* relation. Thus, the cost of processing an *END* request for SV is $2R$ to check the *HIST* relation contents for restart conflicts, 1 to change the state of the committing transaction, $2RF_w$ to update the write timestamp for each write request in the *HIST* relation once the transaction has indeed committed, and $1+R(1+2F_w)$ to delete the information associated with the transaction. Hence, for SV:

$$CPU_{SV} = 3+R(4+5F_w) \tag{6}$$

Results of Cost Comparisons		
CC Algorithm	Storage Cost	CPU Cost
2PL	best under low conflicts	best no-conflict cost
BTO	best under high conflicts	second best under infrequent writing
SV	worst of the three	second best under frequent writing

Table 2.2: Summary of algorithm cost results.

Comparing equation (4) with equation (5), 2PL is seen to have a smaller no-conflict CPU cost than BTO unless F_w is extremely small, in which case 2PL and BTO are almost the same. Comparing equation (4) with equation (6), 2PL is also seen to have a smaller no-conflict CPU cost than SV. Comparing equations (5) and (6), BTO is found to have a smaller no-conflict CPU cost than SV if $F_w < 1/2$, and that SV has a smaller no-conflict CPU cost if $F_w > 1/2$. Thus, with respect to this CPU cost metric, 2PL is dominant, BTO is second-best if writing is infrequent, and SV is second-best if writing is frequent. These results are shown in Table 2.2.

2.6.3. Cost Comparison Summary

In the previous sections, the storage and CPU costs of 2PL, BTO, and SV were compared. 2PL was found to be the algorithm involving the least storage cost under low-conflict transaction mixes, with BTO being the best under high-conflict mixes. SV was the worst algorithm with respect to storage cost. 2PL was also found to be the algorithm with the smallest no-conflict CPU cost. BTO turned out to be second-best with respect to no-conflict CPU cost if writing is infrequent, with SV being second-best if writing is frequent. These results are summarized in Table 2.2.

The cost results for 2PL, BTO, and SV seem to be supported by intuitive reasoning. The worst-case (low-conflict) storage cost results will be considered first. All three algorithms store the same information about transactions. 2PL stores no information about blocked transactions, as there are none in this case. Thus, the difference lies in the amount of information stored by each algorithm about data accesses. 2PL simply stores one lock for each item in the readset of an active transaction. (Recall that the writeset is assumed to be a subset of the readset.) BTO stores one read timestamp for each item in the readset of an active or recently committed transaction, plus a write timestamp for each item which is also in the writeset of an active or recently committed transaction. SV stores both the readset and the writeset of each active transaction, plus it stores the write timestamp associated with each data item in the writeset of a recently committed transaction. Thus, intuition supports the conclusion that 2PL should have the smallest worst-case storage cost.

The best-case (high-conflict) storage cost results will be considered next. As before, all three algorithms store the same information about transactions. 2PL stores information about blocked transactions, and there will be many in this situation. 2PL also stores one read lock for each of the $R(1-F_w)$ items in the readset but not in the writeset of each of the T_a active transactions, and it stores one write lock for each of the RF_w objects written by all transactions. (It is assumed that transactions conflict totally, and write locks are held by just one transaction at a time). BTO only stores a total of R read timestamps and RF_w write timestamps, as transactions conflict totally and each object read has one read timestamp and each object written has one write timestamp. SV also stores only one write timestamp for each item written by recently committed transactions, but it must still separately store all R readset entries and RF_w writeset entries for each of the active transactions. Thus, BTO intuitively has the smallest best-case storage cost, again supporting the conclusions obtained from the model.

The no-conflict CPU cost is considered last. 2PL simply checks and obtains a lock for each of the R read requests and RF_w write requests in a transaction in the absence of conflicts. BTO checks and updates R timestamps for read requests and RF_w timestamps for write requests, but it also pays an additional cost for keeping a list of write requests during transaction execution for use in processing them together at commit time. SV keeps a list of read requests and a list of write requests for commit-time use, plus it must check R timestamps and update RF_w timestamps at commit time. Hence, intuition supports the conclusion that 2PL has the lowest no-conflict CPU cost of the three algorithms considered.

2.7. FUTURE MODEL EXTENSIONS

In this section, extensions for the abstract model which will facilitate studies of the costs of multiple version, hierarchical, and distributed concurrency control algorithms are briefly described.

2.7.1. Multiple Versions

Several recent concurrency control algorithm proposals involve maintaining multiple versions of data objects [Reed78, Baye80, Stea81, Chan82, Bern82b]. In order to describe such algorithms using the abstract model, a new concurrency control database relation, the *OBJ* relation, is introduced. This relation has *obj-id*, *version-id*, and *obj-value* fields, and each version of each object in the database has a corresponding tuple in this relation. In places where an *obj-id* was called for in single-site algorithms, an (*obj-id*, *version-id*) pair will be used in in the multiple version abstract model. The analysis techniques can be applied to this extended model in the same manner as for the single-site model, except that C_s units of storage cost will be assessed for *obj-value* fields of *OBJ* tuples to reflect the fact that objects require much more storage than typical concurrency control information.

2.7.2. Granularity Hierarchies

Several locking algorithms which operate using multiple levels of granules, organized as a hierarchy, have been proposed [Gray75, Gray79, Kort82]. Chapter 4 of this thesis presents hierarchical variants of other types of concurrency control algorithms [Care83b] and examines their performance characteristics. This section briefly sketches extensions to allow the abstract model to support descriptions of algorithms which use a two-level hierarchy of granules.

In order to describe hierarchical algorithms using the abstract model, the *obj-id* fields of all concurrency control database relations will become *gran-id* fields. These new granule identifiers can either identify objects (lower level granules) or parent granules of objects (higher level granules). In addition, a new concurrency control database relation, the *HIER* relation, is introduced. This relation has *parent-id* and *obj-id* fields which associate objects with their parent granules. This explicit mapping of parent granules to objects will make it possible for algorithm descriptions to easily extract the parent granule for a given object granule. Since the *HIER* relation is simply used to represent a mapping which would be implicit in an actual implementation, no additional costs will be assessed for storing or accessing this relation. The analysis techniques are otherwise directly applicable to this extended model.

2.7.3. Distributed Databases

Many recent concurrency control algorithm proposals are intended for use in distributed database systems [Rose78, Mena78, Ston79, Lind79, Bern80, Bern81b, Bern82a, Thom79, Ceri82]. In order to describe distributed concurrency control algorithms within the abstract model, each site will have a concurrency control scheduler with an associated concurrency control database, and the schedulers will interact via messages. To model this interaction, some new notation will be introduced for use in writing algorithm descriptions for distributed systems. Queries of the form <*command*> where <*predicate*> *AT-SITES-OF*(*obj-id*) will be used to indicate that the predicate must be true at all sites where the specified object resides, indicating the need for a round-trip message exchange to evaluate the predicate. In cases where the *AT-SITES-OF* clause is left out, just the local site will be involved in evaluating the predicate.

With this extension, algorithm descriptions will be formulated as before, except that the *AT-SITES-OF*(X) set must be described for all objects X. It is this set description which will serve to differentiate primary site, primary copy, and decentralized concurrency control schemes [Bern81b, Bern82a] from one another, for example. The cost analysis techniques again carry through, though it is necessary to account for the additional cost when the *AT-SITES-OF* set contains more than a single site. Also, a new type of cost, ***message cost***, arises in distributed systems. This cost may be characterized by analyzing the number of messages required to evaluate non-local predicates when executing the new query sets on behalf of transactions.

2.8. SUMMARY

This chapter described a new model of concurrency control algorithms that provides a unified framework for describing and comparing different algorithm proposals. Several sample descriptions were given, and it was shown how the model facilitates analyses of the relative storage and CPU costs of algorithms. It was found that the costs associated with two-phase locking are at least as low as those for basic timestamp ordering and serial validation. The model of this chapter differs from previous work [Bern80, Bern81b, Gall82], as other frameworks for describing concurrency control algorithms have not supported both algorithm descriptions and quantitative algorithm comparisons. Finally, extensions which should enable the model to be used for descriptions and cost analyses of multiple version, hierarchical, and distributed concurrency control algorithms were briefly described.

CHAPTER 3

CONCURRENCY CONTROL PERFORMANCE

This chapter describes a simulation model of a database system which is sufficiently general to allow the performance of many different single-site concurrency control algorithms to be evaluated under various transaction workloads and system costs. Performance results obtained using this simulation model are given for seven concurrency control algorithms based on locking, timestamp, and validation approaches.

3.1. BACKGROUND CONSIDERATIONS

Before describing the simulation model, it will be helpful to consider the nature of the problem which it addresses. The purpose of a concurrency control algorithm is to permit the simultaneous execution of a number of transactions in order to enhance system performance. The degree to which an algorithm allows transactions to execute concurrently and make progress towards completion is called its level of *useful concurrency*. In order to compare alternative algorithms, a measure of their relative levels of useful concurrency must be obtained. The number of active transactions at first seems like a reasonable metric for the level of useful concurrency achievable, as the active transactions are exactly those which are usefully executing in a concurrent manner. However, restarts are a possibility in many variants of locking, and they are the major conflict-resolving tactic in basic timestamp ordering and serial validation. The possibility of transaction restarts makes the number of active transactions a useless concurrency metric, as is illustrated by the following example based on serial validation.

Consider a mix of N transactions whose readsets and writesets all include some object X, and suppose serial validation is the concurrency control algorithm being used. All N transactions will be allowed to execute concurrently. When they are subjected to the commit-time validation test after executing all their reads, doing their respective computation, and caching their writes locally, $N-1$ will be forced to restart. Thus, knowing that N transactions are executing concurrently is not informative.

A better measure of the concurrency benefits offered by alternative algorithms would be a measure of the number of successful commits for some mix of transactions. Hence, in this thesis, the metric chosen is the number of commits per unit time, or *throughput*. Concurrency control semantics are actually implemented and simulated in a closed queuing model of a database system to obtain this throughput information. The queuing model, a workload model, and the techniques used to implement concurrency control algorithms will be described in the next section. First, however, the reasons for using a detailed simulation approach will be discussed.

There are several reasons why detailed simulation seemed like the best way to obtain performance information about alternative concurrency control schemes. Analytic queuing models of concurrency control algorithms are difficult to develop because the sharing of a large number of distinct data objects is a key factor in determining algorithm performance. Recent papers on the subject have dealt primarily with models of very simple concurrency control schemes, such as exclusive-only locking [Gall82, Good83]. Thus, it would be prohibitively hard to develop tractable analytical models of a significant number of concurrency control algorithms for comparative purposes. Second, by selecting a detailed simulation approach, a more realistic collection of transaction mixes and workloads can be studied. Finally, there are certain facts about the behavior of transactions in real systems which are difficult to represent in an analytical model. For example, restarted transactions re-request the same data objects that they requested the last time. Detailed simulation provides a way to model such facts and evaluate algorithm performance without requiring the implementation of many alternative algorithms in an actual database system.

3.2. MODEL DESCRIPTION

3.2.1. The Workload Model

An important component of the performance model is a transaction workload model. When a transaction is initiated from a terminal in the simulator, it is assigned a workload, consisting of a readset and a writeset, which determines the objects that the transaction will read and write during its execution. Two transaction classes, *large* and *small*, are recognized in order to aid in the modeling of realistic transaction workloads. The class of a transaction is determined at transaction initiation time and is used to determine the manner in which the readset and writeset for the transaction are to be assigned. Transaction classes, readsets, and writesets are generated using the workload parameters shown in Table 3.1.

The parameter *num_terms* determines the number of terminals, or level of multiprogramming, for the workload. The parameter *restart_delay* determines the mean length of time required for a terminal to resubmit a transac-

Workload Parameters	
db_size	size of database
gran_size	size of granules in database
num_terms	level of multiprogramming
delay_mean	mean xact restart delay
small_prob	Pr(xact is small)
small_mean	mean size for small xacts
large_mean	mean size for large xacts
small_xact_type	type for small xacts
large_xact_type	type for large xacts
small_size_dist	size distribution for small xacts
large_size_dist	size distribution for large xacts
small_write_prob	Pr(write X | read X) for small xacts
large_write_prob	Pr(write X | read X) for large xacts

Table 3.1: Workload parameters for simulation.

tion when it finds that its current transaction has been restarted, with the delay associated with each particular restart determined by sampling from an exponential distribution with this mean.

The parameter *db_size* determines the number of objects in the database. The parameter *gran_size* determines the number of objects in each granule of the database. Concurrency control requests are made on the basis of granules. Thus, when a transaction reads or writes an object, an associated concurrency control request is made for the granule which contains the object. In modeling read and write requests, objects and granules are given integer names ranging from 1 to *db_size* and 1 to $\lceil db\_size/gran\_size \rceil$, respectively. Object i is contained in granule $\lceil (i-1)/gran\_size \rceil + 1$.

The readset and writeset for a transaction are lists of the numbers of the objects to be read and written, respectively, by the transaction. These lists are assigned at transaction startup time. When a terminal initiates a transaction, *small_prob* is used to randomly determine the class of the transaction. If the class of the transaction is small, the parameters *small_mean*, *small_xact_type*, *small_size_dist*, and *small_write_prob* are used to choose the readset and writeset for the transaction. The readset size for a new small transaction is determined by the *small_dist* and *small_mean* parameters. The readset size distribution, given by *small_dist*, is either constant, uniform, or exponential. If it is constant, the readset size is simply *small_mean*. If the distribution is uniform, the readset size is selected from a uniform distribution on the range [1, 2*small_mean*]. If it is exponential, the readset size is selected from an exponential distribution with mean *small_mean* and truncated to an integer value. All transactions read at least one object, so the readset size is set to 1 if the exponentially determined value is less than one. The size of the readset is truncated to the size of the database if it exceeds *db_size*, as transactions cannot possibly access more data than the database holds.

The particular database objects accessed by a small transaction are determined by the parameter *small_xact_type*. This parameter determines the type, either random or sequential, for small transactions. If small transactions are random, the readset is assigned by randomly selecting objects without replacement from the set of all objects in the database. In the sequential case, all objects in the readset are adjacent, so the collection of objects in the readset is selected randomly from the set of all possible collections of adjacent objects of the appropriate size. The random transaction type is intended to model transactions which access objects using either a primary hashed index or a secondary index, whereas the sequential transaction type is intended to model transactions which access objects using either an ordered primary index or a sequential scan of an entire relation or file. Finally, given the objects in the readset for a small transaction, the objects in its writeset are determined as follows using the *small_write_prob* parameter: It is assumed that all objects written by a transaction are first read by the transaction ("no blind writes"). When an object is placed in the readset, it is also placed in the writeset with probability *small_write_prob*.

The readsets and writesets for the class of large transactions are determined in the obvious analogous manner using the *large_mean*, *large_zact_type*, *large_size_dist*, and *large_write_prob* parameters. Small and large transactions differ only in the choice of parameter values. The purpose of having two transaction classes, large and small, it to enable the workload model to better represent realistic database workloads. For instance, it is possible to use the workload model to represent a mix of small transactions which randomly read and update a single object with large transactions which read a large number of objects sequentially. Such mixes will be used in the performance experiments reported later in this chapter and in Chapter 4.

3.2.2. The Queuing Model

Central to the detailed simulation approach used in this thesis is the closed queuing model of a single-site database system shown in Figure 3.1. This model is an extended version of the model of Ries [Ries77, Ries79a, Ries79b]. There is a fixed number of terminals from which transactions originate. When a new transaction begins running, it enters the *startup queue*, where processing tasks such as query analysis, authentication, and other preliminary processing steps are performed. Once this phase of transaction processing is complete, the transaction enters the *concurrency control queue* (or *cc queue*) and makes the first of its concurrency control requests. If this

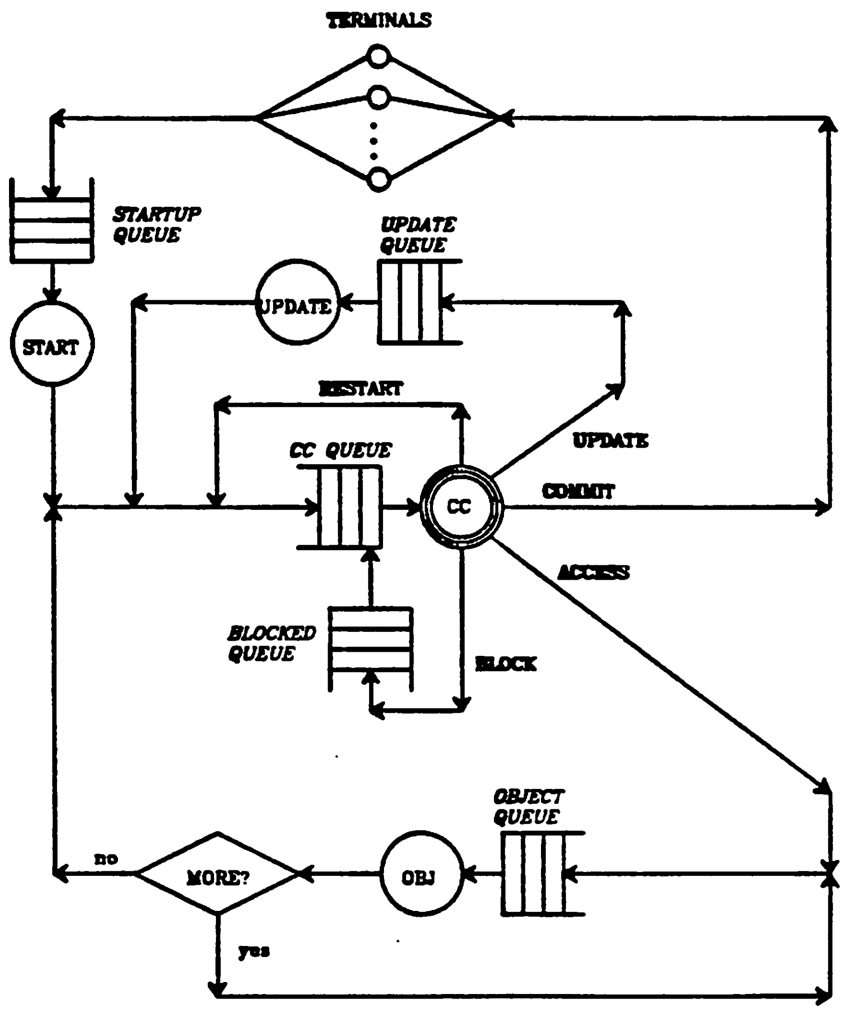

Figure 3.1: Logical database queuing model.

request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object may be accessed prior to the next concurrency control request, the transaction may cycle through this queue several times. (An example of this will be given shortly.) When the next concurrency control request is required, the transaction re-enters the concurrency control queue and makes the next desired request. It is assumed for convenience that transactions which read and write objects perform all of their reads before performing any writes. It would otherwise be necessary to introduce one or more additional parameters to determine how reads and writes are to be interleaved, and it is felt that this would be unlikely to affect the results significantly.

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to restart the transaction, it goes to the back of the concurrency control queue after a randomly determined restart delay period of mean *delay_mean*; it then begins making all of its concurrency control requests and object accesses over again. Eventually, the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, it must first enter the *update queue* and write its deferred updates into the database.

To further illustrate the operation of the logical model, suppose that an implementation of the two-phase locking algorithm studied in Chapter 2 is used in the model. Consider transaction T of Figure 3.2. Suppose that objects X_1 and X_2 are contained in granule G_1, but object X_5 is contained in granule G_2. T will begin by entering the startup queue in order to perform initialization tasks. When it is ready to begin processing objects, T enters

```
transaction T:
begin
  read x1_value from $X_1$;
  read x5_value from $X_5$;
  read x2_value from $X_2$;
  compute;
  write x1_value into $X_1$;
  write x2_value into $X_2$;
end;
```

Figure 3.2: Example transaction.

the concurrency control queue and tries to set a read lock on granule G_1. If it succeeds, it enters the object queue and reads X_1. T then returns to the concurrency control queue and requests a read lock on granule G_2. Assuming it succeeds in setting the required lock, it proceeds to the object queue and reads object X_5. Now, since T already holds a read lock on granule G_1, it proceeds directly to the back of the object queue and reads X_2. If T is ever unable to set a lock, it must wait in the blocked queue until the lock is available. If waiting introduces a deadlock, T may be restarted instead of being permitted to wait.

After completing its reads, T will proceed to the concurrency control queue and request a write lock on granule G_1. If the lock is granted, T will enter the object queue and write object X_1. It will then return to the back of the object queue and write X_2. After finishing its writes, T returns to the concurrency control queue in order to request permission to commit. (This request is never denied using locking, but the validation test would take place here in an implementation of serial validation.) T then proceeds to the update queue to perform its deferred updates. Once it has finished its updates, T enters the concurrency control queue one last time to release its locks. Finally, having completed all necessary activities, T finishes and a new transaction is generated in its place.

To illustrate another case where a transaction may access a number of objects following a single visit to the concurrency control queue, suppose that an algorithm which requires all locks to be preclaimed were used for the previous example. T will enter the startup queue and then proceed to the concurrency control queue, as before. At this point, however, T will request a write lock on granule G_1 and a read lock on granule G_2, and it will have to wait in the blocked queue if it is unable to set both locks. Once T succeeds in setting the required locks, it proceeds to the object queue. T reads object X_1, goes to the back of the object queue, reads X_5, goes to the back of the object queue, and reads X_2. T then goes to the back of the object queue, writes X_1, goes to the back of the object queue, and writes X_2. T finishes as before, entering the concurrency control queue, moving to the update queue, and finally returning to the concurrency control queue to release its locks.

Underlying the logical model of Figure 3.1 are two physical resources, the CPU and I/O (disk) resources. Associated with each logical service depicted in the figure (startup, concurrency control, object accesses, etc.) is some use of each of these two global resources. When a transaction enters the startup queue, it first performs its startup-related I/O processing and then performs its startup-related CPU processing. The same is true of each of the other services in the logical model. Each involves I/O processing followed by CPU processing, with the amounts of CPU and I/O per logical service being specified as simulation parameters. All services compete for portions of the global I/O and CPU resources for their I/O and CPU cycles. The underlying physical system model is depicted in Figure 3.3. As shown, the physical model is simply a collection of terminals, a CPU server, and an I/O server. Each of the two servers has one queue for concurrency control service and another queue for all other service.

The scheduling policy used to allocate resources to transactions in the concurrency control I/O and CPU queues of the underlying physical model is FCFS (first-come, first-served). Concurrency control requests are thus processed one at a time, as they would be in an actual implementation. The resource allocation policies used for the normal I/O and CPU service queues of the physical model are FCFS and round-robin scheduling, respectively. These policies are again chosen to approximately model the characteristics which a real database system implementation would have. When requests for both concurrency control service and normal service are present at either resource, such as when one or more concurrency control requests are pending while other transactions are processing objects, concurrency control service requests are given priority.

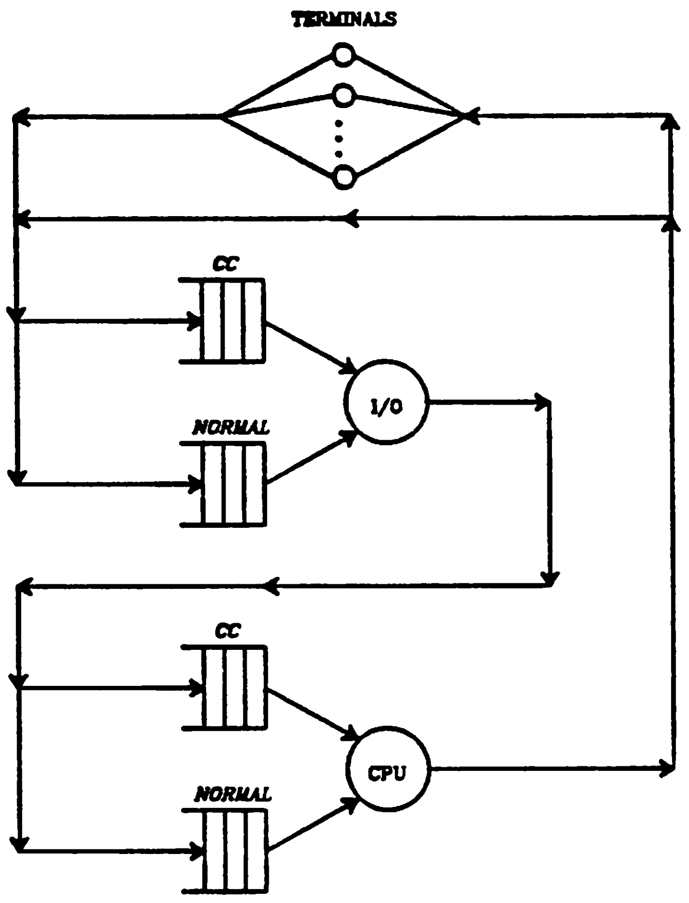

Figure 3.3: Physical database queuing model.

System Parameters	
startup_io	I/O time for transaction startup
startup_cpu	CPU time for transaction startup
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
cc_io	basic unit of concurrency control I/O time
cc_cpu	basic unit of concurrency control CPU time
stagger_mean	mean of exponential randomizing delay

Table 3.2: System parameters for simulation.

The parameters determining the service times (I/O and CPU) for the various logical resources in the model are given in Table 3.2. The parameters *startup_io* and *startup_cpu* are the amounts of I/O and CPU associated with transaction startup. Similarly, the parameters *obj_io* and *obj_cpu* are the amounts of I/O and CPU associated with reading and writing an object in the database. Reading an object takes resources equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and *obj_io* at deferred update time, as it is assumed that the deferred update list is maintained in buffers in main memory. The parameters *cc_io* and *cc_cpu* are the amounts of I/O and CPU associated with a concurrency control request. All these time parameters represent constant service time requirements rather than stochastic ones for simplicity. Finally, the *stagger_mean* parameter is the mean of an exponential time distribution which is used to randomly stagger transaction initiation times from terminals (not to model user thinking) each time a new transaction is started up. All parameters are specified in internal simulation units, the unit of CPU time allocated to a transaction in one sweep of the round-robin allocation code for the simulator.

3.2.3. Algorithm Descriptions

As mentioned previously, a major objective in the design of the simulator was to facilitate implementing a variety of concurrency control algorithms within a common framework. Concurrency control algorithms are described for simulation purposes as a collection of four routines, *Init_CC_Algorithm*, *Request_Semantics*, *Commit_Semantics*, and *Update_Semantics*. Each these routines is written in SIMPAS, a simulation language based on extending PASCAL with simulation-oriented constructs [Brya80a, Brya80b]. SIMPAS is the language in which the rest of the simulator is implemented as well. *Init_CC_Algorithm* is called when the simulation starts up, and it is responsible for initializing all algorithm-dependent data structures and variables. The other three routines are responsible for implementing the semantics of the concurrency control algorithm being modeled.

When a transaction reaches the front of the concurrency control queue, one of the three concurrency control semantics routines is invoked. If the transaction has any remaining read or write requests to make, the routine invoked is *Request_Semantics*. This routine processes the next request(s), returning information to the simulator informing it how many units of simulation time to charge for CPU and I/O associated with processing the concurrency control request. This cost is computed based on the *cc_cpu* and *cc_io* parameters. The *Request_Semantics* routine also returns a concurrency control decision of *access*, *block*, *restart*, *update*, or *commit*. This result informs the simulator which queue the transaction should go to next. It is the *Request_Semantics* routine which is responsible for checking concurrency control data structures such as a lock table.

When the transaction arrives in the concurrency control queue after finishing its last request, the *Commit_Semantics* routine is called. This routine is responsible for doing whatever algorithm-dependent commit time processing is called for. As an example, this routine is responsible for performing the validation test in simulations of concurrency control algorithms based on commit-time validation. Again, the routine returns cost and concurrency control result information to the simulator.

Finally, if the transaction executes to completion, the *Update_Semantics* routine is called after the transaction has committed and written its deferred updates to disk. This routine is responsible for any algorithm-dependent cleanup processing that is called for, such as releasing locks in simulations of locking algorithms. It returns cost information to the simulator which indicates the number of CPU and I/O units to charge for its post-update concurrency control processing.

3.3. PERFORMANCE EXPERIMENTS

This section reports on the results of a number of simulation experiments for seven concurrency control algorithm variants. Before presenting these experiments and results, however, the algorithms examined in the study will be reviewed briefly, the concurrency control cost modeling details for the algorithms will be described, and the statistical approach used in the experiments will be outlined.

3.3.1. Algorithms Studied

The algorithms investigated in the performance experiments of this chapter are all variants of the two-phase locking, basic timestamp ordering, and serial validation algorithms discussed earlier in this thesis.

Dynamic Two-Phase Locking (2PL). This algorithm is similar to the locking algorithm described and analyzed using the cost model of Chapter 2, except that its blocking behavior is a bit more complex. In this algorithm, transactions request read locks for granules which they read. Transactions later upgrade these read locks to write locks for granules which they also write (when they write the first object in the granule). When a lock request must be denied, the requesting transaction is blocked and placed at the end of a queue of transactions waiting to obtain the lock. When a lock is released, transactions are unblocked by removing transactions from the front of the lock wait queue until either the queue is empty or a transaction with an incompatible lock request reaches the front of the queue. A waits-for graph of transactions is maintained [Gray79], and deadlock cycle detection is performed each time a transaction blocks. If a deadlock is discovered, the transaction which just blocked and caused the deadlock is chosen as the victim and restarted. (This may not be the optimal victim choice, but it was selected initially for ease of implementation.)

Wait-Die Two-Phase Locking (WD). This algorithm is closely related to dynamic two-phase locking, except deadlock prevention is used instead of deadlock detection. Wait-die deadlock prevention [Rose78] is used, so deadlocks are prevented by ordering transactions by the times when they first start running. When a lock request from a transaction T_i conflicts with a lock held by another transaction T_j, T_i is permitted to wait only if it started running before T_j. Otherwise, T_i is restarted. This restart rule prevents the formation of deadlock cycles and does away with the need for deadlock detection. However, since a transaction T_i which waits for an older transaction T_j will not always result in a real deadlock, WD will result in more restarts than 2PL. WD was selected over the related wound-wait algorithm [Rose78] for two reasons. The first reason is that it was slightly simpler to implement. The second reason is that WD is the deadlock-preventing counterpart of 2PL in the sense that both algorithms are non-preemptive (they only restart the transaction making the current request).

Dynamic Two–Phase Locking, No Upgrades (2PLW). This algorithm is nearly identical to dynamic two-phase locking. The difference is that, if a transaction reads and writes an granule x, it does not request a read lock and then later upgrade its read lock to a write lock. Rather, for granules which are eventually read, write locks are requested the first time the granule is accessed, eliminating upgrades. Since transaction readsets and writesets are determined at startup time in the simulator, the necessary knowledge is available. Full deadlock detection is employed as before.

Exclusive, Preclaimed Two-Phase Locking (PRE). This algorithm is a simple form of two-phase locking, in which all granules read or written by a transaction are locked in exclusive access mode (write locked) at transaction startup time. If a transaction is unable to obtain the set of all required locks, it blocks without obtaining any of these locks and proceeds when all required locks are indeed available. This is basically the version of locking which Ries investigated in his experiments [Ries77, Ries79a, Ries79b]. Neither deadlocks nor upgrades are possible with this algorithm. The use of only exclusive locks will result in more blocking than would occur using both read and write locks. This will aid in establishing later performance results about the effects of blocking.

Basic Timestamp Ordering (BTO). This is the version of basic timestamp ordering described in Chapter 2, with read timestamps checked dynamically and all write timestamps checked at commit time.

Basic Timestamp Ordering, Thomas Write Rule (TWW). This is a version of basic timestamp ordering which differs from the previous version in the following manner: When a transaction T_i makes a write request for an object x and $TS(T_i) > R\text{-}TS(x)$ but $TS(T_i) < W\text{-}TS(x)$, the previous version of basic timestamp ordering would call for T_i to be restarted. In this version, T_i's request is granted, so T_i is not restarted, but the actual (outdated) write is ignored. In all of the experiments that follow, TWW never outperformed BTO; in fact, the performance of the two algorithms was always identical.

This is due to the fact that, under the "no blind writes" assumption which underlies the mechanism by which writesets are assigned in the model, TWW is identical to BTO (see Appendix 2). Thus, results for TWW are not presented separately.

Serial Validation (SV). This is the modified version of serial validation that was presented in Chapter 2, with the semantics of the original serial validation algorithm [Kung81] implemented via startup and commit timestamps.

3.3.2. Concurrency Control Costs

In order to simulate the concurrency control algorithms of interest, it is necessary to make some assumptions about their costs. To evaluate them fairly and determine how their blocking and restart decisions affect performance, the assumptions made for each of the algorithm simulations are consistent. This section will briefly describe how the *cc_cpu* and *cc_io* parameters are used in modeling the costs for each of the algorithms in the study. The concurrency control costs incurred by a transaction which makes N_r granule read requests and N_w granule write requests will be given for each algorithm.

For all of the locking algorithms except preclaimed locking (2PL, WD, and 2PLW), a CPU cost of *cc_cpu* and an I/O cost of *cc_io* are assessed each time the algorithm makes a read or write lock request for a granule. For 2PL and WD, the total concurrency control CPU and I/O costs for a transaction in the absence of restarts are $(N_r + N_w)cc\_cpu$ and $(N_r + N_w)cc\_io$, respectively. For 2PLW, which sets write locks initially on granules that are both read and written, the total concurrency control costs in the absence of restarts are $N_r cc\_cpu$ and $N_r cc\_io$. Thus, since 2PLW sets fewer locks overall, its concurrency control cost is lower. The PRE algorithm incurs the same total concurrency control costs as 2PLW, $N_r cc\_cpu$ and $N_r cc\_io$, but the entire lock-setting cost for PRE is charged at transaction startup time to model preclaiming.

For the BTO and TWW algorithms, a CPU cost of *cc_cpu* and an I/O cost of *cc_io* are assessed each time the algorithm checks a read or write access privilege for a granule. Thus, the total concurrency control cost for BTO and TWW in the absence of restarts is the same as for 2PL and WD, with one minor difference. For BTO and TWW, the read-related costs of $N_r cc\_cpu$ and $N_r cc\_io$ are charged dynamically, as read requests are received and processed, but the write-related costs of $N_w cc\_cpu$ and $N_w cc\_io$ are all charged together at transaction commit time to model the fact that BTO and TWW defer write request checking until then. The costs for BTO and TWW serve to model the timestamp-checking costs for these algorithms.

For the SV algorithm, a CPU cost of *cc_cpu* and an I/O cost of *cc_io* are charged for each of the granules read and written, and all charges are assessed at transaction commit time. The read-related charges model the testing of readset granules to make sure that none have timestamps which indicate that some recently committed transaction wrote the granule, and the write-related charges model the timestamp updating process once a transaction has committed. Thus, the total concurrency control costs in the absence of restarts for SV are the same as those for 2PL, WD, BTO, and TWW, $(N_r + N_w)cc\_cpu$ and $(N_r + N_w)cc\_io$, but the points where the costs are assessed differ somewhat.

In all cases, an attempt has been made to fairly but simply account for concurrency control costs. It is assumed that the unit costs for concurrency control operations in locking, timestamps, and validation are all the same, *cc_cpu* and *cc_io*, as a first-order approximation. This is reasonable since the basic steps in each algorithm involve doing one or two table lookups per request. Thus, the costs of processing requests in the various algorithms are not likely to differ by more than small constant factors. This is borne out by the CPU cost results of Chapter 2.

For transactions which must wait for locks, the concurrency control costs are assessed once the transaction has unblocked and succeeded in obtaining the desired lock(s). For transactions which are restarted during commit-time concurrency control checking, the costs for the commit-time checking are assessed in full for simplicity. It would otherwise be necessary to keep careful track of exactly how far along transactions are in the tests when they fail. Examples of transactions which might have to restart at commit time are those failing BTO or TWW write tests or the SV validation test. No other concurrency control costs are assessed for any of the algorithms for any reason.

Of course, if a transaction is restarted, it will pay the additional costs involved in executing again from the beginning. These include all the costs associated with reading and writing the objects that the transaction accesses, plus the costs for making all of its concurrency control requests over again.

3.3.3. The Statistical Approach

In the simulation experiments performed in this thesis, the metric chosen is the transaction throughput rate. This section sketches the methods used in this thesis to discriminate between throughput differences due simply to statistical variations and those actually due to algorithm performance characteristics.

Throughput results and confidence intervals for these results are obtained from the simulations using a slight variant of a standard simulation analysis technique. Surveys of alternative techniques may be found in [Sarg76, Ferr78, Saue81]. For several reasons, the method of *batch means* was chosen from the options of batch means, independent replications, and the regenerative method. First, due to a lack of exponential service times and the fact the transactions compete for a large number of shared logical resources (granules), the only true regeneration state for the simulations in this thesis is the state in which all terminals are in their "stagger delay" waiting periods prior to submitting new transactions. It was found experimentally that, following initial startup, this state does not occur with sufficient frequency to permit use of the regenerative method. Second, as described in [Sarg76], batch means has the advantage over independent replications that initial transients do not bias each of the throughput observations. Finally, implementation considerations made the use of batch means simpler than the method of independent replications, as the simulator would have to garbage collect and re-initialize simulation and algorithm-dependent data structures between observation periods if the method of independent replications were chosen.

Using the method of batch means, simulation runs are divided up into a set of *num_batches* individual batches or sub-runs, each of which is *batch_time* simulation time units long. Each batch within a simulation run provides one throughput observation, and these observations are averaged to estimate the overall throughput. Confidence intervals are usually computed using standard techniques assuming that the throughput observations from the batches are independent and identically distributed [Saue81]. Two questionable assumptions underly the use of batch means. The first assumption is that batches are long enough so that the results are not biased by startup transients. The second assumption is that batches are not correlated.

Appendix 3 addresses the assumptions underlying batch means, reviewing the mathematics associated with the method, describing how startup transients were excluded from the results, and detailing a method which was used to account for correlation between batches in computing confidence intervals. The appendix gives sample confidence interval results obtained for one of the experiments of this chapter to illustrate the usefulness of the statistical methods used in this thesis. Appendix 3 also describes the choice of the settings selected for the *batch_time* and *num_batches* simulation control parameters for the experiments and gives confidence interval results for each of the experiments in this thesis. Little will be said about confidence intervals beyond this section, but only differences which are statistically significant shall receive attention.

In order to make definitive statements and draw conclusions about concurrency control performance issues, it is necessary that confidence intervals for the experimental results be sufficiently small so that they do not overlap from algorithm to algorithm, at least where important differences are to be demonstrated. This consideration affected the choice of transaction sizes in the experiments. Small confidence intervals are achievable only when a "reasonably large" number of successful transaction commits is contained in the overall simulation run, as otherwise the variance in throughput results from the batches will be too large. (The results of experiments performed in this thesis seem to indicate that 1000 or more commits are desirable.) The "large" transactions studied in this thesis are therefore not all that large in terms objects accessed, as prohibitively long simulations would be necessary to produce statistically meaningful results for simulations involving very large transactions.

3.3.4. Experiments and Results

In this section, the results of six different performance experiments are reported. Each of these experiments was performed on the seven concurrency algorithms described earlier in this chapter. These experiments were designed to investigate the *relative* performance of the various algorithms, in hopes of identifying algorithms whose performance is either uniformly superior to that of the other algorithms or whose performance is superior under some set of reasonable conditions. The first three experiments investigate the relative performance of the algorithms under various transaction workloads; the fourth experiment investigates the effect which the level of multiprogramming has on the performance results; the fifth experiment investigates the effect of system balance on the performance results; and the sixth experiment examines the effect of concurrency control costs on the performance results. All of the experiments reported in this chapter were run with *batch_time* = 50,000 and *num_batches* = 20, or a total of 1,000,000 simulation time units, as described in Appendix 3.

3.3.4.1. Experiment 1: Transaction Size

This experiment investigates the performance characteristics of the seven concurrency control algorithms under workloads consisting of fixed-size transactions. The parameters varied in this experiment are the granularity of the database and the size of transactions. Since concurrency control requests are made for granules, rather than objects, varying the granularity of the database varies the probability that transactions will conflict with one another. When the finest granularity is chosen, where each granule contains a single object, conflicts will be rare for small transactions in a large database. When the granularity is coarse, the entire database will consist of only a few granules. Frequent conflicts between transactions will be inevitable in this case. The purpose of this experiment is to observe the behavior of the algorithms of interest under varying probabilities of conflicts, and also to see how the choice of transaction size affects this behavior.

The system parameter settings for this experiment are given in simulated time in Table 3.3. All simulations were run with one simulation unit interpreted as one millisecond of simulated time. With these system parameter

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	35
startup_cpu	10
obj_io	35
obj_cpu	10
cc_io	0
cc_cpu	1
stagger_mean	20

Table 3.3: System parameters for experiment 1.

settings, a transaction incurs a startup cost of one 35 millisecond disk access and 10 milliseconds of CPU time. In addition, this same cost is incurred for each read or write of an object. Charges for reading and writing objects are assessed in the manner described in the section which presented the details of the queuing model. The cost associated with processing each concurrency control request is 1 millisecond of CPU time and no I/O time. A 20 millisecond random delay time is used to stagger transaction startups.

The relevant workload parameter settings for this experiment are given in Table 3.4. The database consists of 10,000 objects, and its granularity is varied from 1 to 10,000 granules (or 10,000 down to 1 object per granule). This could correspond, for example, to a 10 megabyte database where objects are 1 kilobyte pages and granules are groups of one or more pages. The number of terminals submitting transactions against the database is 10, and

Workload Parameters	
db_size	10000 objects
gran_size	vary from 1 to 10000 objects/granule
num_terms	10
delay_mean	1 second
small_prob	1.0
small_mean	vary from 1 to 30 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5

Table 3.4: Workload parameters for experiment 1.

all transactions are the same size. Transactions each read a fixed number of objects selected at random from among all objects in the database. This number is varied for different simulation runs within this experiment. Transactions update each object that they read with fifty percent probability.

Table 3.5 shows the throughput results for experiment 1.1, where a transaction size of *small_mean* = 1 was used. Table 3.6 shows the throughput results for experiment 1.2, where *small_mean* = 2 was used. The throughputs are given in units of transactions per second of simulated time. The *Grans* column in each table contains the total number of granules in the

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	8.252	8.063	11.215	11.127	7.790	7.655
10	10.971	11.004	11.420	11.421	10.648	10.314
100	11.373	11.373	11.419	11.420	11.328	11.262
1000	11.413	11.413	11.420	11.420	11.405	11.402
10000	11.419	11.419	11.420	11.420	11.418	11.416

Table 3.5: Throughput, experiment 1.1 (*small_mean* = 1).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	3.400	3.638	6.479	6.241	2.505	3.634
10	5.974	5.790	7.096	7.161	5.119	5.231
100	7.039	6.966	7.161	7.163	6.906	6.714
1000	7.152	7.149	7.161	7.161	7.138	7.113
10000	7.159	7.159	7.160	7.161	7.158	7.158

Table 3.6: Throughput, experiment 1.2 (*small_mean* = 2).

10,000 object database. All of the algorithms studied have strikingly similar performance with more than 100 granules in the database for small transactions. For coarser granularities, the algorithms do display significant differences. In particular, the two best algorithms are 2PLW and PRE. 2PL and WD do somewhat worse than the two best algorithms, followed by SV and BTO.

Based on their virtually identical performance under fine granularities, the concurrency control cost of 1 millisecond of CPU time per granule request has little effect the performance of the algorithms. For example, although PRE and 2PLW make fewer concurrency control requests than 2PL and WD, they do not exhibit significantly greater throughputs. All differences may be attributed to the blocking and restart behavior of the algorithms. It was hypothesized that the differences under coarser granularities were due to differences in the number of restarts incurred using the algorithms, and an examination of the restart counts supports this hypothesis.

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	5960	7858	0	0	7185	7206
10	1096	3060	0	0	1909	2753
100	109	363	0	0	224	378
1000	14	40	0	0	31	40
10000	2	4	0	0	4	6

Table 3.7: Restarts, experiment 1.1 (*small_mean* = 1).

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	6311	7638	0	0	7594	6529
10	2366	4633	242	0	4071	3844
100	247	706	3	0	508	879
1000	20	73	0	0	51	102
10000	2	11	0	0	4	9

Table 3.8: Restarts, experiment 1.2 (*small_mean* = 2).

Tables 3.7 and 3.8 contain the restart counts for the two smallest transaction sizes tested. No restarts occurred with PRE, as PRE is deadlock free. Few restarts occurred with 2PLW. 2PLW is deadlock free when only one granule is accessed per transaction, and few deadlocks occurred at the next smallest transaction size setting. Restarts were more frequent with 2PL, as additional deadlocks occur when two or more transactions hold a read lock on a common granule and then each attempts to upgrade this read lock to a write lock. The WD algorithm also led to a number of restarts at upgrade time. More restarts occurred with WD than with 2PL because restarts were called for when deadlock was a possibility as opposed to only when it actually occurred. Finally, both SV and BTO use restarts as their single conflict-resolving tactic, and both led to many restarts under coarse granularity as a result.

The marginally superior performance of WD as compared to SV and BTO for coarse granularities is probably due to the fact that SV and (to a slightly lesser extent) BTO tend to allow transactions to get further along

Resources Consumed		
Resource	2PLW	PRE
I/O	905685.5	872204.0
CPU	265253.0	255452.0

Table 3.9: Utilization, experiment 1.2, coarsest granularity.

before restarting them, so less resource waste results from the average WD restart. The marginal performance advantage of 2PLW over PRE when the entire database is a single granule can be explained by the fact that PRE uses only exclusive locking and holds locks for entire transaction lifetimes, thereby totally serializing transaction execution in this case. 2PLW will permit the simultaneous execution of multiple read-only transactions, whereas PRE will not. This latter hypothesis is borne out by an examination of the relative resource utilization results in this case. Table 3.9 gives the total I/O and CPU resources consumed by transactions during 1,000,000 units of simulation time for experiment 1.2 (*small_mean* = 2) when the entire database was one granule.

As the transaction size parameter *small_mean* is increased, performance differences between the algorithms become more pronounced. Table 3.10 shows the throughput results for experiment 1.3, where a transaction size of *small_mean* = 5 was used. Table 3.11 shows the throughput results for experiment 1.4, where *small_mean* = 10 was used. The performance of the algorithms is beginning to be noticeably different even with 1000 granules in

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.748	0.788	2.917	2.741	0.169	0.889
10	0.946	1.065	2.097	3.028	0.406	1.200
100	2.823	2.633	3.335	3.360	2.231	2.408
1000	3.320	3.293	3.359	3.361	3.248	3.205
10000	3.357	3.351	3.362	3.361	3.355	3.347

Table 3.10: Throughput, experiment 1.3 (*small_mean* = 5).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.281	0.240	1.518	1.425	0.001	0.336
10	0.074	0.234	0.432	1.415	0.004	0.355
100	0.827	0.701	1.414	1.759	0.235	0.784
1000	1.676	1.599	1.784	1.790	1.473	1.480
10000	1.776	1.770	1.788	1.788	1.763	1.749

Table 3.11: Throughput, experiment 1.4 (*small_mean* = 10).

the database, a trend which will continue as the transaction size is increased further. This is due to an increased probability of conflicts between transactions when each accesses more objects in the database. The best algorithm at these transaction sizes under finer granularities is PRE, followed closely by 2PLW, then followed by 2PL, WD, SV, and lastly BTO. As the granularity becomes coarser, however, this ordering changes somewhat. In particular, WD actually does better than 2PL at the second coarsest granularity. Also, SV dominates 2PL and WD in addition to BTO at coarser granularities.

Tables 3.12 and 3.13 present the restart counts associated with experiments 1.3 and 1.4. As before, the differences between the performance of the

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	4242	6119	0	0	5317	4181
10	4061	5217	3123	0	4940	3651
100	907	1763	66	0	1910	1617
1000	71	164	0	0	188	258
10000	6	20	0	0	10	22

Table 3.12: Restarts, experiment 1.3 (*small_mean* $= 5$).

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	2351	4142	0	0	2855	2323
10	2816	3837	4223	0	2848	2293
100	1543	2360	769	0	2482	1612
1000	180	388	7	0	501	485
10000	16	31	0	0	38	56

Table 3.13: Restarts, experiment 1.4 (*small_mean* $= 10$).

concurrency control algorithms studied can be seen to correlate with differences in restarts. Here, though, where the number of objects accessed by a transaction is larger than it was previously, the penalty for restarting a transaction is much greater. Much more work is lost and must be repeated by a restarted transaction in this case. This increase in the restart penalty is the reason why throughput degradation occurs sooner as the granularity is varied from fine to coarse here. It is also the reason for the increase in the magnitude of this performance degradation effect.

The reason that PRE dominates 2PLW except when the entire database is one granule is that PRE is the only algorithm studied which is entirely free of restarts. Although 2PLW avoids deadlocks due to lock upgrades, deadlocks can arise when two transactions attempt to lock a pair of granules in the opposite order from each other. 2PLW ends up being the second best performer here because it has the fewest restarts of all the algorithms except PRE. An exception occurs with 10 granules in the database and *small_mean* = 10, where 2PLW actually has a larger number of restarts than some of the alternatives which it outperforms. An explanation in this case is that 2PLW restarts transactions much earlier in their lifetimes. In 2PLW, a transaction that is backed out due to a deadlock is restarted when it attempts to set a write lock prior to reading the granule it is attempting to lock. Less resources are wasted when a transaction has completed less work prior to being restarted.

There is a related explanation for the case where WD outperformed 2PL. In 2PL, the transaction picked as the victim when a deadlock arises is the one that induced the deadlock. When two transactions conflict over a granule in the WD algorithm, the transaction causing the conflict is allowed to wait if it is older and is forced to restart otherwise; this rule leads to the selection of younger transactions as victims for the purpose of deadlock avoidance in the WD algorithm. Since transactions which have accessed more objects are likely to have required more simulated time to execute so far, they are likely to be the older transactions when conflicts occur. The result is that, because the version of 2PL simulated here does does not take age into account in deciding which transaction to restart, the transactions selected are mixed with respect to the amount of work completed at restart time. Since WD makes an effort to avoid restarting old transactions, it restarts mostly transactions which have made less progress, thereby wasting fewer resources. This effect will become even more pronounced with larger transaction sizes.

Unfortunately, the simulation program does not keep track of restarts by transaction size, making verification of this hypothesis regarding the relative performance of 2PL and WD difficult. However, the restart counts for these experiments support this hypothesis. It is otherwise very difficult to explain why more restarts occurred using WD and yet its performance is superior to 2PL. This suggests that a better victim selection criteria could also improve the performance of 2PL, most probably to the point of surpassing WD. For instance, 2PL could choose the youngest transaction in a deadlock cycle as the victim to be restarted. 2PL would then restart the youngest transaction in a true deadlock cycle, whereas WD restarts the youngest transaction in a potential cycle, so 2PL would once again outperform WD.

The dominance of SV over 2PL and WD at fine granularities in experiments 1.3 and 1.4 is also related to their victim selection policies. The SV algorithm restarts a transaction only when it conflicts with a recently committed transaction. This ensures that at least one transaction in each group of conflicting transactions will succeed in committing, as otherwise there will be no recently committed transaction to cause others to restart. In 2PL, it is possible for a group of conflicting transactions to exhibit a thrashing-type restart behavior. Consider a transaction T_1 which is restarted due to a conflict with another transaction T_2 over a granule x. T_1 may begin again, lock a granule y needed by a third transaction T_3, and cause T_3 to subsequently restart due to a deadlock. T_3 may then do the same thing to T_2 over still another granule z. The "pick the conflict-causing transaction" rule for victim selection is the source of the problem, and the problem worsens as transaction size increases. In WD, a transaction T_1 which restarts due to a conflict with an older transaction T_2 over a granule x can restart and subsequently become involved in the very same conflict if T_2 has not finished with x. T_1 may thus be restarted a number of times due to the one conflict with T_2 using WD [Rose78], leading to a greater number of restarts than would otherwise be expected. As a result of the stability of SV and the victim selection problems of 2PL and WD, SV caused fewer restarts at coarse granularities in these experiments.

An exception to the statement that algorithm performance is correlated with the number of restarts is the performance of the BTO algorithm. BTO does worse than would be warranted by its restart counts. The explanation for this exceptionally poor performance is that, under a high probability of conflicts, BTO is unstable due an to anomaly in the algorithm. This anomaly, *cyclic restarts*, is mentioned briefly in [Date82, Lin82, Ullm83], and it is illustrated in Figures 3.4 and 3.5. Figure 3.4 depicts a pair of transactions with

```
transaction T1:
begin
   read x_value from X;
   compute;
   write x_value into X;
end;

transaction T2:
begin
   read x_value from X;
   compute;
   write x_value into X;
end;
```

Figure 3.4: Cyclic restart transactions for BTO.

workloads which make them prone to the cyclic restart anomaly, and Figure 3.5 demonstrates how these transactions can become involved in an infinite cycle, restarting each other repeatedly. Only the first ten steps in the infinite interleaving cycle are shown in the figure.

Step	*Action*	*Result*
1	T1: begin xact	TS(T1) = 1
2	T2: begin xact	TS(T2) = 2
3	T1: read *x_value* from *X*;	R-TS(*X*) = 1
4	T2: read *x_value* from *X*;	R-TS(*X*) = 2
5	T1: write *x_value* into *X*;	Restart(T1) with TS(T1) = 3
6	T1: read *x_value* from *X*;	R-TS(*X*) = 3
7	T2: write *x_value* into *X*;	Restart(T2) with TS(T2) = 4
8	T2: read *x_value* from *X*;	R-TS(*X*) = 4
9	T1: write *x_value* into *X*;	Restart(T1) with TS(T1) = 5
10	T1: read *x_value* from *X*;	R-TS(*X*) = 5
11	etc.	etc.

Figure 3.5: Example of cyclic restart anomaly.

The problem with the two transactions of Figure 3.4 is that they both wish to read and write the same granule. If they attempt to interleave execution in the manner shown in Figure 3.5, performing their reads in timestamp order and then attempting to do the same for their writes, they are liable to follow the pattern shown in the figure forever. The problem begins when T_1 is restarted at step 5 because X has been read by a younger transaction, T_2. At this point, T_1 actually becomes younger than T_2 and re-reads X. This dooms T_2's subsequent write to end in a restart, etc. If the computation delay between the read and write of X exceeds the delay from the time of a restart to the time of re-reading X for both transactions T_1 and T_2, this pattern can indeed persist forever. This is the case for the coarse granularity throughput results in experiments 1.3 and 1.4. Although the restart counts for BTO are comparable to those of some locking algorithms which have higher throughputs, its throughput is lower because the restarts caused by the anomaly always occur when write timestamp checking takes place. This does not take place until commit time, meaning that the amounts of CPU and I/O resources wasted due to the anomaly are very high. This anomaly was not observed in experiments 1.1 and 1.2 because the restart delay of one second of simulated time was sufficient to minimize its chances of occurring with such small transaction sizes.

Tables 3.14 and 3.15 give the throughput results for the largest transaction sizes tested, where *small_mean* = 15 and *small_mean* = 30 were used.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.179	0.139	1.031	0.964	0.000	0.205
10	0.009	0.121	0.143	0.959	0.000	0.207
100	0.203	0.237	0.583	1.111	0.009	0.371
1000	1.023	0.933	1.207	1.216	0.642	0.861
10000	1.193	1.186	1.217	1.218	1.148	1.159

Table 3.14: Throughput, experiment 1.5 (*small_mean* = 15).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.089	0.071	0.518	0.487	0.000	0.096
10	0.002	0.062	0.020	0.485	0.000	0.096
100	0.006	0.054	0.029	0.481	0.000	0.102
1000	0.266	0.201	0.483	0.592	0.025	0.267
10000	0.571	0.558	0.617	0.614	0.431	0.524

Table 3.15: Throughput, experiment 1.6 (*small_mean* = 30).

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	1592	3115	0	0	1908	1585
10	1928	3129	4217	0	1906	1581
100	1619	2363	1522	0	1891	1321
1000	302	582	25	0	900	556
10000	36	67	0	0	106	89

Table 3.16: Restarts, experiment 1.5 (*small_mean* = 15).

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	801	2178	0	0	953	805
10	977	2088	4131	0	953	805
100	1010	1781	2114	0	953	797
1000	538	956	285	0	917	545
10000	74	119	4	0	289	148

Table 3.17: Restarts, experiment 1.6 (*small_mean* = 30).

The restart counts for these cases, experiments 1.5 and 1.6, are given in Tables 3.16 and 3.17. For the most part, these results simply accentuate the trends observed in the previous pair of experiments. PRE is dominant due to its lack of restarts. WD again outperforms 2PL at coarser granularities because of its preference for restarting transactions which have completed less work. SV again outperforms 2PL and WD at coarse granularities, even outperforming 2PLW with 10 granules comprising the database, by causing fewer restarts. This is due to the thrashing behavior of the victim selection criteria used by 2PL and 2PLW, and to the possibility of repeating conflicts for WD. Both effects were discussed in conjunction with experiments 1.3 and 1.4. Finally, the anomalous behavior of BTO is extremely pronounced for these transaction sizes.

3.3.4.2. Experiment 2: Access Patterns

This experiment investigates the performance characteristics of the seven concurrency control algorithms under two workloads consisting solely of large transactions. The granularity of the database is varied, as before, in order to vary the probability of conflicts. The system parameter settings for this experiment are the same as those for experiment 1 (see Table 3.3). The objective of this experiment is to observe the effects of random versus sequential object access patterns on algorithm performance.

The relevant workload parameter settings for this experiment are given in Table 3.18. The database again consists of 10,000 objects, and its granularity is varied from 1 to 10,000 granules. The number of terminals submitting transactions against the database is 10, and all transactions are large. Transaction sizes are determined by sampling from a uniform distribution over the range [1,60], so transactions each read an average of about 30 objects. Both random and sequential access patterns are tested. In the random case, the objects are selected randomly from among all possible sets of the appropriate number of objects in the database. In the sequential case, the

Workload Parameters	
db_size	10000 objects
gran_size	vary from 1 to 10000 objects/granule
num_terms	10
delay_mean	1 second
small_prob	0.0
large_mean	30 objects
large_xact_type	random or sequential
large_size_dist	uniform
large_write_prob	0.1

Table 3.18: Workload parameters for experiment 2.

objects are selected at random from among all possible sequences of the appropriate size in the database. Transactions update each of the objects which they read with ten percent probability.

Table 3.19 gives the throughput results for experiment 2.1, where transactions with random access patterns were tested. These results are similar to those obtained for the larger transaction sizes in experiment 1, though some of the performance differences are less pronounced due to the reduced probability of writing. Table 3.20 gives the throughput results for experiment 2.2, where sequential transactions were used. Significant differences are apparent

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.062	0.068	0.759	0.609	0.000	0.098
10	0.001	0.060	0.035	0.608	0.000	0.099
100	0.024	0.103	0.049	0.612	0.015	0.142
1000	0.537	0.456	0.709	0.738	0.138	0.393
10000	0.780	0.759	0.783	0.787	0.677	0.675

Table 3.19: Throughput, experiment 2.1 (random access).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.062	0.071	0.770	0.646	0.000	0.097
10	0.418	0.414	0.797	0.801	0.052	0.408
100	0.712	0.718	0.799	0.799	0.443	0.685
1000	0.770	0.770	0.798	0.799	0.700	0.746
10000	0.775	0.775	0.798	0.799	0.728	0.754

Table 3.20: Throughput, experiment 2.2 (sequential access).

in the results of the second experiment. 2PLW and PRE are dominant, followed by WD and 2PL, with SV being the second worst algorithm and BTO being the worst of the algorithms investigated.

Tables 3.21 and 3.22 show the restart counts associated with experiments 2.1 and 2.2. The differences between the two experiments are clearly a function of restart behavior, and there is a simple explanation for the significant reduction in restarts from experiment 2.1 to experiment 2.2 for the locking algorithms. Since transactions request resources in sequential order in the latter experiment, all deadlocks related to random locking order disappear.

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	478	2296	0	0	711	455
10	697	1967	2590	0	711	461
100	535	1225	1157	0	602	468
1000	182	453	74	0	450	304
10000	5	30	2	0	76	84

Table 3.21: Restarts, experiment 2.1 (random access).

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	459	2258	0	0	642	445
10	284	1324	0	0	678	301
100	79	191	0	0	267	99
1000	29	58	0	0	78	49
10000	23	45	0	0	57	42

Table 3.22: Restarts, experiment 2.2 (sequential access).

2PLW is deadlock-free for the mix of experiment 2.2, and the only deadlocks (or potential deadlocks) for 2PL and WD occur due to lock upgrades. As a result of the reduction in the number of restarts for all of the locking algorithms, the restart-based SV and BTO algorithms are unable to compete with the performance of the locking algorithms for the mix of large sequential transactions.

3.3.4.3. Experiment 3: Mixed Workload

This experiment investigates the performance characteristics of the seven concurrency control algorithms under a workload consisting of a mix of large and small transactions. The granularity of the database is again varied in order to vary the probability of conflicts. The system parameters are the same as for experiment 1 (see Table 3.3), and the workload parameters for this experiment are listed in Table 3.23. The workload investigated here is a mix of the workloads of the two previous experiments, with some of the transactions being the small fixed-size transactions from experiment 1 and the remainder being the larger transactions from experiment 2. This workload is intended to represent a fairly realistic combination of small and large transactions. The fraction of small transactions in the mix is varied to investigate algorithm performance under different combinations of small and large transactions.

Workload Parameters	
db_size	10000 objects
gran_size	vary from 1 to 10000 objects/granule
num_terms	10
delay_mean	1 second
small_prob	vary from 0.2 to 0.8
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	30 objects
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.1

Table 3.23: Workload parameters for experiment 3.

The throughput results for experiment 3 are given in Tables 3.24 through 3.27. The associated restart counts are given in Tables 3.28 through 3.31. There are few surprises, as the results are mostly a combination of those observed previously in experiments 1.2 and 2.2. The locking algorithms are dominant, with 2PLW being the best and PRE tying except at the coarsest granularity setting. WD is next, outperforming 2PL for reasons related to the discussion of WD versus 2PL in experiments 1.3 and 1.4. WD picks the younger transaction to restart in order to avoid deadlocks when conflicts arise, and in this case the younger transaction is likely to be one of the small transactions in the mix. As a result, WD wastes fewer overall resources due to restarts than are wasted by 2PL. The advantage of WD over 2PL is more pronounced for mixes of mostly small transactions, as the likelihood of its selecting a large transaction to restart is the smallest for such mixes. SV and BTO perform as expected based on the observations from previous experiments. SV is generally the second worst algorithm here, outperforming 2PL only under very coarse granularities where the thrashing behavior of 2PL is the most serious. BTO is uniformly inferior to the other algorithms tested.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.066	0.091	0.919	0.771	0.000	0.111
10	0.464	0.517	0.967	0.963	0.124	0.450
100	0.883	0.894	0.966	0.964	0.691	0.858
1000	0.930	0.945	0.966	0.969	0.775	0.905
10000	0.942	0.944	0.966	0.967	0.874	0.913

Table 3.24: Throughput, experiment 3.1 (*small_prob* = 0.2).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.069	0.109	1.126	0.992	0.000	0.128
10	0.535	0.633	1.227	1.226	0.073	0.522
100	1.055	1.112	1.228	1.226	0.862	1.022
1000	1.183	1.183	1.229	1.228	1.081	1.112
10000	1.200	1.184	1.229	1.228	1.113	1.130

Table 3.25: Throughput, experiment 3.2 (*small_prob* = 0.4).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.089	0.203	1.535	1.344	0.002	0.202
10	0.639	0.857	1.709	1.709	0.159	0.710
100	1.462	1.556	1.713	1.717	0.410	1.352
1000	1.646	1.669	1.714	1.715	1.463	1.571
10000	1.688	1.679	1.713	1.715	1.576	1.578

Table 3.26: Throughput, experiment 3.3 (*small_prob* = 0.6).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.101	0.450	2.521	2.371	0.022	0.333
10	0.853	1.551	2.830	2.860	0.338	0.963
100	2.352	2.580	2.865	2.861	1.246	2.185
1000	2.675	2.752	2.859	2.864	2.415	2.504
10000	2.803	2.777	2.860	2.864	2.634	2.554

Table 3.27: Throughput, experiment 3.4 (*small_prob* = 0.8).

Restarts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	489	2892	0	0	631	469
10	317	1621	3	0	554	336
100	56	214	0	0	180	85
1000	24	72	0	0	135	46
10000	17	48	0	0	60	39

Table 3.28: Restarts, experiment 3.1 (*small_prob* = 0.2).

Restarts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	487	3494	0	0	669	460
10	386	2191	6	0	536	373
100	101	330	0	0	198	122
1000	21	71	0	0	71	58
10000	15	58	0	0	56	49

Table 3.29: Restarts, experiment 3.2 (*small_prob* = 0.4).

Restarts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	584	4621	0	0	861	508
10	484	2971	11	0	673	416
100	113	438	1	0	462	151
1000	22	98	0	0	84	52
10000	7	58	0	0	42	43

Table 3.30: Restarts, experiment 3.3 (*small_prob* = 0.6).

Restarts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	589	6003	0	0	762	481
10	562	3781	58	0	574	496
100	158	904	0	0	344	235
1000	43	211	0	0	115	102
10000	15	160	0	0	63	71

Table 3.31: Restarts, experiment 3.4 (*small_prob* = 0.8).

3.3.4.4. Experiment 4: Multiprogramming Level

This experiment investigates the effects of the multiprogramming level on the results of the previous experiments. Experiment 3.1 is repeated with two different levels of multiprogramming. Also, the effect of the level of multiprogramming on throughput when no concurrency control algorithm is used is investigated for the transaction mix of experiment 3.1.

The first part of this experiment consisted of running experiment 3.1 with all of the same system and workload parameters except for the level of multiprogramming. (See Tables 3.3 and 3.23 for these parameter settings.) The level of multiprogramming was set at 5 and then 20 terminals instead of its previous setting of 10. The throughput results from this experiment are presented in Tables 3.32 and 3.33. These results are basically the same as

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.134	0.183	0.915	0.771	0.002	0.219
10	0.681	0.749	0.965	0.967	0.326	0.660
100	0.921	0.921	0.964	0.964	0.869	0.908
1000	0.953	0.953	0.964	0.965	0.914	0.939
10000	0.956	0.955	0.964	0.964	0.930	0.942

Table 3.32: Throughput, experiment 4.1 (*num_terms* = 5).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.034	0.043	0.919	0.771	0.005	0.059
10	0.174	0.303	0.964	0.965	0.064	0.296
100	0.798	0.812	0.963	0.963	0.434	0.735
1000	0.899	0.900	0.961	0.965	0.738	0.847
10000	0.919	0.912	0.961	0.964	0.800	0.863

Table 3.33: Throughput, experiment 4.2 (*num_terms* = 20).

those of experiment 3.1, the only difference being that the throughputs of all algorithms are slightly higher in experiment 4.1 and slightly lower in experiment 4.2. The difference in throughputs observed here is due to the fact that the level of multiprogramming, like granularity, affects the probability of conflicts. Halving the multiprogramming level results in a lower rate of conflicts, and doubling it results in a higher rate of conflicts.

The second part of this experiment consisted of varying the multiprogramming level with concurrency control turned off. This was accomplished by using the serial validation simulation, with the validation code modified to always validate transactions. The result is a simulation in which transactions incur the usual concurrency control costs but are never blocked or restarted due to conflicts. The system and workload parameters, with the exception of the multiprogramming level, were again those those of experiment 3.1. The

Effects of Multiprogramming Level			
MP Level	Throughput Rate	CPU Used	I/O Used
1	2.121± 3.09% (90%*CI*)	227639.4	731175.7
2	2.711± 4.88% (90%*CI*)	293877.5	943777.7
3	2.838± 3.77% (90%*CI*)	308886.5	992059.0
4	2.860± 3.44% (90%*CI*)	311163.0	999280.0
5	2.861± 3.64% (90%*CI*)	311360.0	999935.0
6	2.865± 3.74% (90%*CI*)	311397.0	999966.0
7	2.866± 3.63% (90%*CI*)	311473.0	1000000.0
25	2.855± 4.38% (90%*CI*)	311601.0	1000000.0

Table 3.34: Throughput and utilization, experiment 4.3 (no CC).

results of this experiment, including confidence intervals, are given in Table 3.34. As is apparent in the table, the throughput rate reaches its maximum (within the limits of statistical variation) at a multiprogramming level of 4. Beyond this point, adding more transactions to the system has virtually no effect on throughput because the bottleneck resource, I/O in this case, is close to full utilization from this point on. (This is illustrated in the table by the amount of I/O time consumed out of a possible 1,000,000 total simulation units.) Thus, aside from affecting the probability of conflicts, the multiprogramming level has little or no effect once the number of transactions in the system is sufficient to allow the bottleneck resource to achieve full utilization.

3.3.4.5. Experiment 5: System Balance

This experiment investigates the effects of the system balance on the results of the previous experiments. Experiment 3.1 is repeated with the I/O cost parameters decreased so that the system is CPU bound and then balanced instead of being I/O bound (as it was in the original experiment). The system parameter settings used in this experiment are presented in Table 3.35. The I/O costs associated with startup processing and object processing have been decreased from from 35 to 5 and then 10 milliseconds. These correspond intuitively to systems which have multiple disks. The CPU cost associated with processing a concurrency control request is still set at 1 millisecond. The workload parameter settings used in this experiment are those of experiment 3.1 (see Table 3.23).

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	5 and 10
startup_cpu	10
obj_io	5 and 10
obj_cpu	10
cc_io	0
cc_cpu	1
stagger_mean	20

Table 3.35: System parameters for experiment 5.

The throughput results for this experiment are shown in Tables 3.36 and 3.37. These results are analogous to those of experiment 3.1, with higher throughputs overall because of the decrease in the I/O cost for processing each transaction. No other significant differences appear to be present in the data, the conclusion being that system balance is not a significant factor with respect to selecting from among a set of alternative concurrency control algorithms.

3.3.4.6. Experiment 6: Concurrency Control Cost

This experiment investigates the effects of concurrency control costs on the results of the previous experiments. Experiment 3.1 is repeated with the concurrency control cost parameters modified to investigate the effects of alternative concurrency control costs. The system parameter settings used in this experiment are those of experiments 1 through 3 (see Table 3.3), except

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.240	0.393	3.094	2.294	0.004	0.406
10	1.676	1.909	3.385	3.369	0.373	1.587
100	3.024	3.098	3.391	3.389	2.073	2.906
1000	3.211	3.215	3.370	3.370	2.928	3.084
10000	3.028	3.028	3.140	3.141	2.823	2.913

Table 3.36: Throughput results, experiment 5.1 (CPU bound).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.237	0.398	2.799	1.742	0.000	0.398
10	1.594	1.851	3.261	3.215	0.254	1.557
100	2.951	3.015	3.306	3.284	2.196	2.856
1000	3.142	3.144	3.272	3.270	2.958	3.049
10000	2.962	2.965	3.073	3.071	2.866	2.896

Table 3.37: Throughput results, experiment 5.2 (balanced).

for the *cc_cpu* and *cc_io* parameters. Experiments are performed with concurrency control being free (*cc_cpu* = 0, *cc_io* = 0), expensive in terms of CPU cost (*cc_cpu* = 5 milliseconds, *cc_io* = 0), and expensive in terms of I/O cost (*cc_cpu* = 1 millisecond, *cc_io* = 35 milliseconds). The last case is as though concurrency control information were kept on disk, with one disk access being required per concurrency control request processed. This is similar to the situation examined most thoroughly by Ries [Ries77, Ries79a, Ries79b]. The workload parameter settings are the same as for experiment 3.1 (see Table 3.23).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.066	0.088	0.919	0.771	0.000	0.111
10	0.473	0.515	0.967	0.963	0.121	0.460
100	0.883	0.893	0.967	0.964	0.698	0.863
1000	0.931	0.945	0.966	0.969	0.834	0.905
10000	0.942	0.944	0.966	0.967	0.874	0.912

Table 3.38: Throughput, experiment 6.1 ($cc\_cpu = 0$, $cc\_io = 0$).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.066	0.088	0.917	0.769	0.000	0.111
10	0.433	0.521	0.967	0.963	0.115	0.455
100	0.883	0.892	0.967	0.964	0.603	0.862
1000	0.932	0.943	0.966	0.969	0.834	0.905
10000	0.942	0.944	0.966	0.959	0.877	0.905

Table 3.39: Throughput, experiment 6.2 ($cc\_cpu = 5$, $cc\_io = 0$).

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.063	0.076	0.894	0.754	0.000	0.108
10	0.416	0.447	0.930	0.928	0.038	0.432
100	0.810	0.813	0.924	0.924	0.631	0.786
1000	0.787	0.794	0.870	0.872	0.662	0.766
10000	0.475	0.475	0.511	0.510	0.460	0.460

Table 3.40: Throughput, experiment 6.3 ($cc\_cpu = 1$, $cc\_io = 35$).

The throughput results from this experiment are shown in Tables 3.38 through 3.40. Again, the results are similar to those of experiment 3.1. The fact that Tables 3.38 and 3.39 are nearly identical to each other and to the results of experiment 3.1 (see Table 3.24) indicates that the effects of the cost of concurrency control are negligible as long as concurrency control overhead is small compared to the costs associated with object accesses. In Table 3.40, the effect of the vastly increased costs associated with processing concurrency control requests is apparent from the fact that a medium granularity is optimal rather than the finest granularity. This illustrates that the tradeoff observed by Ries for locking in [Ries77, Ries79a, Ries79b] exists for many concurrency control algorithms. This tradeoff arises in all algorithms because coarse granularities lead to high conflict probabilities and fine granularities lead to high concurrency control overhead. Other than this, however, nothing new is evident in the results of the experiment, as 2PLW and PRE display the best throughputs, with WD, 2PL, SV, and finally BTO following behind them, as in the previous experiments.

3.3.4.7. Algorithm Anomalies

Several preliminary experiments, not reported in detail here, were performed before a restart delay was added to the model. In these experiments, BTO did even worse than it did in the experiments reported here. Without a restart delay, the behavior of BTO was strongly dominated by the cyclic restart problem described earlier in this chapter. Cyclic restarts were identified as the cause of this performance problem after a log of transaction requests and responses was kept by the simulator and examined in detail. Another algorithm, multi-version timestamp ordering (MVTO), was also found to share the cyclic restart problem with BTO. (The MVTO algorithm is investigated in the next chapter.)

A related algorithm anomaly was uncovered in experiments with an earlier version of 2PL, a version where a transaction making a read request for a granule was not required to block unless the current lock on the granule was a write lock. This version of 2PL, which is basically the 2PL variant analyzed in Chapter 2, suffers from a fairness problem: If a transaction blocks waiting to write a granule, subsequent read requests may be granted for the granule even though the writer "asked first". Starvation (infinite waiting) of writers is one possible result of this policy. The anomaly shown in Figures 3.6 and 3.7 is another. The anomaly, repeated deadlocks, occurs when the three

```
transaction T1:
begin
  read x_value from X;
  compute;
  write x_value into X;
end;

transaction T2:
begin
  read x_value from X;
  compute;
  write x_value into X;
end;

transaction T3:
begin
  read x_value from X;
  compute;
  write x_value into X;
end;
```

Figure 3.6: Repeated deadlock transactions for anomalous 2PL.

update transactions of Figure 3.6 wish to access a common object. As shown in Figure 3.7, it is possible for one of the transactions to starve indefinitely while the other transactions repeatedly restart each other due to deadlocks.

In Figure 3.7, all three transactions start running, and then all three make read requests for X. After step 6, read locks are held on X by all three transactions. At step 7, T_1 makes a write request for X and is forced to block until T_2 and T_3 release their read locks on X. At step 8, T_2 also makes a write request for X and is restarted to resolve the deadlock that would arise between T_1 and T_2 if T_2 were allowed to wait. At step 9, T_2 resubmits its read request, and is again granted a read lock on X. Then, at step 10, T_3 submits a write request for X, at which point its fate is the same

Step	*Action*	*Result*
1	T1: begin xact	
2	T2: begin xact	
3	T3: begin xact	
4	T1: read *x_value* from *X*;	Read(T1,X)
5	T2: read *x_value* from *X*;	Read(T2,X)
6	T3: read *x_value* from *X*;	Read(T3,X)
7	T1: write *x_value* into *X*;	Block(T1)
8	T2: write *x_value* into *X*;	Restart(T2)
9	T2: read *x_value* from *X*;	Read(T2,X)
10	T3: write *x_value* into *X*;	Restart(T3)
11	T3: read *x_value* from *X*;	Read(T3,X)
12	T2: write *x_value* into *X*;	Restart(T2)
13	T2: read *x_value* from *X*;	Read(T2,X)
14	etc.	etc.

Figure 3.7: Example of cyclic restart anomaly.

as the previous fate of T_2. The result is that T_1 starves while T_2 and T_3 alternately get into deadlocks with T_1 and are restarted. Note that arbitrarily many transactions could actually become involved in such repeated deadlock situations. The 2PL algorithm examined in the experiments of this chapter was free of this anomaly because readers were made to wait for X when one or more conflicting transactions were already waiting to write X.

3.4. CONCLUSIONS

In this chapter, a simulation model of a single-site database system was described, and the model was used to compare the performance offered by seven alternative concurrency control algorithms under several different transaction workloads. Workloads consisting of all small transactions, all larger transactions, and a mix of transaction sizes were examined. Also, experiments were performed to learn about the effects of factors such as the level of multiprogramming, system balance, and concurrency control costs on the relative performance of the alternatives. A number of interesting conclusions are implied by the results of this study.

3.4.1. Performance Under Low Conflicts

A recent paper by Jim Gray et al [Gray81a] argued that, in most real database system environments, conflicting transactions are a rarity. An analysis based on a simple model of "real-world" transactions showed that blocking is rare even when all locks are exclusive, and that deadlocks are much more rare than blocking. The results of the experiments reported in this chapter indicate that, in such situations, the choice of a concurrency control algorithm can be made more or less arbitrarily. In situations where the granularity of the database led to few conflicts in the simulation studies, all algorithms performed almost identically.

3.4.2. Blocking versus Restarts

Some concurrency control algorithms, such as locking, use blocking as the primary mechanism for handling conflicting requests, restarting transactions only when absolutely necessary. Other algorithms, such as those based on timestamps or validation, choose to restart transactions in order to avoid or resolve conflicts. The results of the experiments of this chapter indicate that, in cases where concurrency control algorithms display significantly different performance characteristics, the algorithms which perform the best are the ones which choose blocking when it is a viable option.

Several aspects of the experiments support the conclusion that blocking is the conflict-resolving mechanism of choice. First of all, 2PLW and PRE outperformed all other algorithms when conflicts were not rare (under coarse granularities) in most of the experiments of this chapter. These algorithms were also the ones which called for the fewest number of restarts. The only case where performance suffered due to blocking was in the situation where PRE was used and the entire database was one granule. In this case, PRE totally serialized accesses to the database, and was outperformed by 2PLW, which allowed multiple concurrent readers. This was an extreme case, however, and would almost certainly disappear if PRE allowed both shared and exclusive locks to be preclaimed. Also, 2PL and WD, both of which caused some restarts, outperformed BTO and SV by requiring fewer restarts overall. In a few cases, where large, random transactions were involved, SV actually outperformed 2PL, WD, and even 2PLW under coarse granularities. This was due to the use of an overly simplistic victim selection policy for 2PL and 2PLW, and to a problem with repeated conflicts for WD. In these cases, SV won by actually requiring fewer restarts.

Another factor supporting this conclusion is the outcome of experiment 4, where the level of multiprogramming was varied with concurrency control turned off. It was found that the maximum possible throughput was achieved in all cases where four or more transactions were available to be run, as the bottleneck resource (I/O in the experiment) was fully utilized beyond this point. This indicates that, as long as blocking a transaction leaves the system with a sufficient number of runnable transactions, blocking will not degrade the performance of the system. (This also explains why PRE did so well in the other experiments even though PRE is quite conservative in terms of allowing transactions to execute concurrently.)

Given that blocking is preferable to restarts, the choice of a concurrency control algorithm is fairly simple. If sufficient knowledge about what a transaction will access is available at startup time, preclaimed locking with read and write locks is a good choice. Preclaiming avoids the possibility of deadlock, and the use of read and write locks will enhance its concurrency. The necessary knowledge may be available in relational database systems where query analysis can help identify the data to be accessed. If transaction readsets and writesets are not known a priori, but it is possible to know at read time whether or not a write will take place for the granule to be read, 2PLW with an improved deadlock victim selection criteria (to be discussed shortly) is a good choice. 2PLW avoids upgrade-induced deadlocks, and improving its victim selection criteria will avoid possible thrashing under mixes including large, random transactions. If no prior information is available, 2PL with an improved victim selection criteria would be the best choice, as restarts are avoided except when they are absolutely necessary to break dependency cycles. Algorithms such as BTO or SV, which resolve conflicts via restarts, are never recommended based on these results.

3.4.3. Transaction Size

Experiments 1 through 3 investigated the performance of the alternative algorithms under different transaction sizes, access patterns, and mixes. In all cases, the conclusions were the much the same. One difference observed between the small and large transaction results was that, with larger transactions, the performance differences were decidedly more noticeable. This was partly due to the increase in the probability of conflicts, but also due to the fact that more work was wasted by the average restart when larger transactions were involved. Another observation was that WD outperformed 2PL in the mixed size workload because WD chooses younger transactions to restart, and younger transactions are more likely to be the smaller ones. Choosing smaller transactions to restart lowered the wasted resources due to restarts, suggesting that 2PL should have incorporated an age or size based victim selection criteria for deadlock resolution. This was also indicated by the instances where thrashing by 2PL led to its being outperformed by SV.

3.4.4. Concurrency Control Overhead

The performance of the algorithms for a mix of small and large transactions was investigated with each concurrency control request requiring from 0 to 5 milliseconds of simulated CPU time in experiment 6. Concurrency control costs were seen to be insignificant compared to other factors such as the costs involved with the subsequent object reads and writes. This indicates that concurrency control overhead is not a problem as long as concurrency control decisions can be made in an amount of time which is fairly small compared with the overall amount of processing time required by transactions. This will be the case for most proposed algorithms as long as concurrency control tables can be maintained in primary memory.

3.4.5. Granularity and Performance

A second conclusion related to concurrency control overhead may be drawn from the results of experiment 6: All concurrency control algorithms have the same characteristics as those reported for locking by Ries [Ries77, Ries79a, Ries79b]. If the costs associated with concurrency control are indeed large enough to be significant, a medium granularity on the order of 100 or so granules is likely to be optimal due to the concurrency/overhead tradeoff. This indicates that granularity hierarchies may improve performance for other types of concurrency control algorithms, a hypothesis which will be tested in the next chapter. Also, if concurrency control overhead is not significant, granularities on the order of 1000 or more granules are needed to produce optimal performance, especially in the presence of large, random transactions.

3.4.6. System Balance

The effect of system balance on the choice of a concurrency control algorithm is minimal. This was illustrated by the results of experiment 5. Thus, while system balance is certainly of interest for determining the overall throughput which is achievable with a given transaction mix, it is not a factor which needs to be considered in selecting a concurrency control algorithm.

3.4.7. Algorithm Anomalies

Some concurrency control algorithms have anomalies which can hurt their performance in situations where conflicts are not rare. One example is the cyclic restart anomaly of BTO (and MVTO), and another is the repeated deadlock anomaly associated with some variants of locking. The degree to which these anomalies affect performance depends on the length of typical restart delays in the system of interest, as the anomalies arise due to the fact that transactions which are restarted request the same data items each time they run. However, algorithms with such anomalous behavior are probably worth avoiding, as there are a number of reasonable concurrency control alternatives which do not have such problems.

CHAPTER 4

PERFORMANCE ENHANCEMENT SCHEMES

In this chapter, two ideas which have been suggested for enhancing the performance of concurrency control algorithms are addressed. The first part of this chapter is concerned with *multiple version* concurrency control algorithms. Rather than keeping just one value for each object in the database, a set of values (the current one and several previous values) are maintained. These values are used to enhance concurrency for long-running transactions. Several multiple version algorithms are reviewed in the first half of this chapter, and one new multiple version algorithm is introduced. The simulation model of the previous chapter is then used to perform several experiments which investigate situations in which multiple versions offer improved performance.

The second proposal examined is the notion of structuring the database as a *granularity hierarchy* rather than a flat collection of granules (as in the algorithms of previous chapters). Granularity hierarchies can be used to enhance performance under mixes of large and small transactions. The second half of this chapter reviews the idea of a lock hierarchy and extends this notion to other forms of concurrency control. Three new hierarchical concurrency control algorithms are introduced in this part of the thesis. The performance model of Chapter 3 is extended to model a two-level granularity hierarchy, and experiments are performed to examine the potential benefits of granularity hierarchies.

4.1. MULTIPLE VERSIONS

There have been a number of recent papers proposing the use of multiple versions of data to increase potential concurrency [Reed78, Baye80, Stea81, Chan82, Bern82b]. For most of these algorithms, the idea is to allow long, read-only transactions to read old versions of data objects while allowing update transactions to create newer versions. This section of the thesis reviews several such algorithms, presents a new one based on modifying the serial validation algorithm of Kung and Robinson [Kung81], and then examines the performance of these algorithms using the simulation model developed in the previous chapter.

4.1.1. Multiversion Timestamp Ordering

The multiversion timestamp ordering (MVTO) algorithm [Reed78] is similar to the BTO algorithm in many ways. Associated with each transaction T is a timestamp, $TS(T)$, issued at the time at which T begins executing. Associated with each data item x in the database is a set of versions. Each version is a (*time*,*value*) pair indicating a value of x and the (timestamp) time at which the value was assigned to x. If T is a timestamp, let $x[T]$ be the value of the most recent version of x written at time less than or equal to T. A read request from T for x will be granted using the value $x[TS(T)]$, and a write request from T for x, if granted, will result in the creation of a new version of x. A write request from T for x will be denied if it attempts to create a new version of x when the previous version has been read by a transaction with a timestamp larger than $TS(T)$.

For concurrency control purposes, a read/write history, $H_{rw}(x)$, is maintained for each data item x. This history is a set of time intervals which correspond to versions of x. The starting time of each interval is the creation timestamp of the version, and the ending time of each interval is the largest timestamp of any reader of the version. For example, $H_{rw}(x) = \{(3,6), (10,13)\}$ means that x has two versions, one created at time 3 and last read at time 6, and the other created at time 10 and last read at time 13. Histories in MVTO play the role which timestamps played in BTO, allowing the concurrency control algorithm to know when potential conflicts arise.

The MVTO algorithm grants all read requests using the appropriate version, extending the interval in $H_{rw}(x)$ associated with the version read as necessary. A write request from T for x is rejected if any interval in $H_{rw}(x)$ contains the time $TS(T)$. Otherwise, transactions which have already read x between $TS(T)$ and the end of the interval containing $TS(T)$ would have their reads invalidated by T's write. If the write request is granted, a new version of x is created, and a new interval with starting and ending times of

$TS(T)$ is added to $H_{rw}(x)$. Continuing with our previous example, a read request for x from a transaction T with $TS(T) = 7$ would be granted using the version of x created at time 3, and the history for x would be changed to $H_{rw}(x) = \{(3,7), (10,13)\}$. A write request from T for x would now be rejected if $3 < TS(T) < 7$ or $10 < TS(T) < 13$. If $TS(T) = 8$, however, the request would be granted, a new version of x would be created, and the history for x would be changed to $H_{rw}(x) = \{(3,7), (8,8), (10,13)\}$.

An informal description of the MVTO algorithm is given in Figure 4.1. In the figure, the *extVers* operation is assumed to extend the version interval corresponding to $x[TS(T)]$ in $H_{rw}(x)$ if $TS(T)$ is larger than the ending time of the interval. The *newVers* operation is assumed to create a new interval in $H_{rw}(x)$, starting at $TS(T)$ and having length zero, recording the creation of a

```
procedure readReq($T$,$x$);
begin
  grant readReq using $x[TS(T)]$;
  extVers($H_{rw}(x)$, $TS(T)$);
end;

procedure writeReq($T$,$x$);
begin
  if $TS(T)$ in $H_{rw}(x)$ then
    restart($T$);
  else
    grant writeReq;
    newVers($H_{rw}(x)$, $TS(T)$);
  fi;
end;
```

Figure 4.1: MVTO algorithm.

new version of x. It is assumed that the *grant writeReq* operation creates this new version of x and stamps it as having been created at time $TS(T)$.

4.1.2. Multiversion Locking

There have been several proposals for multiversion locking algorithms [Baye80, Stea81, Chan82]. This section reviews one of the most recent ones, an algorithm proposed for use in the Ada-compatible database management system under development at CCA [Chan82]. This algorithm, which will be referred to as the *CCA version pool* algorithm, uses two-phase locking to synchronize update transactions and allows read-only transactions to run using older versions of data items. The CCA proposal includes schemes for implementing version selection efficiently and for dealing with maintenance and garbage collection of old versions in a bounded buffer pool, but this study will only be concerned with the concurrency control aspects of the proposal.

The semantics of the CCA version pool algorithm are actually quite simple, and can be explained as follows. As in the timestamp-based version of serial validation, transactions are assigned startup timestamps when they begin running and commit timestamps when they reach their commit point. Also, transactions are classified at startup time as being either *read-only* or *update* transactions. When an update transaction reads or writes a data item, it locks the item, just as it would in two-phase locking, and it reads or writes the most recent version of the item. When an item is written, a new version of the item is created, and versions of items are stamped with the commit timestamp of their creator.

When a read-only transaction wishes to access an item, no locking is needed. Instead, the transaction simply reads the latest version of the item with a timestamp less than the startup timestamp of the read-only transaction. Since the timestamp associated with a version is the commit timestamp of its writer, a read-only transaction T is made to read only versions which

```
procedure readReq($T$,$x$);
begin
  if readOnly($T$) then
    grant readReq using $x[TS(T)]$;
  else
    if writeLocked($x$) then
      block($T$);
      if cycle($T$) then
        restart($T$);
      fi;
    else
      grant readReq using $x[current]$;
      readLock($T$,$x$);
    fi;
  fi;
end;

procedure writeReq($T$,$x$);
begin
  if readLocked($x$) or writeLocked($x$) then
    block($T$);
    if cycle($T$) then
      restart($T$);
    fi;
  else
    grant writeReq;
    writeLock($T$,$x$);
  fi;
end;
```

Figure 4.2: CCA version pool algorithm.

were written by transactions which committed before T even began running. Thus, T is serialized after all transactions which committed prior to its startup, but before all transactions which are active but uncommitted during any portion of its lifetime. An informal description of the CCA version pool algorithm is given in Figure 4.2. The startup timestamp of transaction T is denoted as $TS(T)$, and the most recent committed version of x is denoted as $x[current]$ in the description. It is assumed that the *grant writeReq* operation creates a new version of x, and also that this version of x is stamped with the commit timestamp of T when T commits.

4.1.3. Multiversion Serial Validation

The previous section described a multiversion locking algorithm which enhanced a known concurrency control algorithm by permitting read-only transactions to read older version of objects. In this way, serializability was guaranteed for update transactions in the usual way, and it was guaranteed for read-only transactions by having them read a consistent set of older versions of data determined by their startup time. Conflicts between read-only transactions and update transactions were eliminated, increasing the level of concurrency which can be achieved using the algorithm. This idea can be applied outside the domain of locking. This section demonstrates its generality by developing a multiversion variant of the serial validation algorithm of Kung and Robinson [Kung81].

```
procedure validate($T$);
begin
  valid := true;
  if not readOnly($T$) then
    foreach $x_r$ in $readset(T)$ do
      if $S\text{-}TS(T) < TS(x_r)$ then
        valid := false;
      fi;
    od;
    if valid then
      foreach $x_w$ in $writeset(T)$ do
        $TS(x_w) := C\text{-}TS(T)$;
      od;
      commit $writeset(T)$ to database;
    else
      restart($T$);
    fi;
  fi;
end;
```

Figure 4.3: Informal description of multiversion SV algorithm.

The multiversion SV algorithm can be developed in a manner which follows naturally from the CCA version pool algorithm. Transactions are again classified as read-only or update transactions at startup time. Update transactions record their readsets and writesets, performing either the validation test of Kung and Robinson [Kung81] or the timestamp-based validation test developed in Chapter 2. As in the CCA version pool algorithm, versions are stamped with the commit timestamp of their creators, and read-only transactions read the latest versions of items with timestamps less than their startup timestamps. As a result, the serializability of update transactions is guaranteed by SV semantics and the serializability of read-only transactions is guaranteed by making sure they read consistent, committed versions of data.

An informal description of the multiversion SV algorithm is given in Figure 4.3. The timestamp-based validation test of Chapter 2 is used for update transactions. It is assumed that an appropriate version selection mechanism provides each transaction T with either $x[TS(T)]$ or $x[current]$ when it reads x, depending on whether T is a read-only transaction or an update transaction, respectively. It is also assumed that new versions are created and stamped with $C\text{-}TS(T)$ when $writeset(T)$ is committed to the database.

4.1.4. Multiple Versions and Performance

In this section, the performance characteristics of the three previous multiple version concurrency control algorithms are investigated using the simulation model of Chapter 3. Each multiple version algorithm is studied and compared to its single version counterpart. Before reporting on the experiments, however, some details of the concurrency control cost models of the algorithms and the simulation approach taken will be described.

4.1.4.1. Concurrency Control Costs

As in Chapter 3, in order to simulate the concurrency control algorithms of interest, it is necessary to make some assumptions about their costs. This section will briefly describe how the *cc_cpu* and *cc_io* parameters are used in modeling the costs for each of the multiversion algorithms in order to evaluate them using the simulation model. The concurrency control costs for a transaction which makes N_r granule read requests and N_w granule write requests will be given for each algorithm.

The cost model used for the MVTO algorithm is identical to that described in the previous chapter for the BTO algorithm. Each read access request for a granule is assessed a CPU cost of *cc_cpu* and an I/O cost of *cc_io* at the time of the request, and each write request is assessed the same CPU and I/O costs at transaction commit time. Thus, total read-related concurrency control costs of $N_r\,cc\_cpu$ and $N_r\,cc\_io$ are charged dynamically, and total write-related concurrency control costs of $N_w\,cc\_cpu$ and $N_w\,cc\_io$ are charged at commit time. Again, these costs are intended to model the costs of the required timestamp operations. In particular, the read request costs are intended to model the cost of extending the read/write history for the granules read, and the write request costs are intended to model version creation and timestamping costs.

The cost model used for the CCA version pool algorithm is nearly identical to the model described in the previous chapter for the 2PL algorithm. The one difference between this cost model and the 2PL cost model is that read-only transactions run nearly for free. At the time of startup for a read-only transaction, a CPU cost of *cc_cpu* and an I/O cost of *cc_io* are assessed to model the cost of recognizing and marking the transaction as read-only. After this point, no concurrency control costs are assessed for read-only transactions.

The cost model used for the multiversion SV algorithm is analogous to the model used for the CCA version pool algorithm. The original SV cost model is used here for update transactions, and read-only transactions pay a CPU cost of *cc_cpu* and an I/O cost of *cc_io* at transaction startup time. The costs assessed for read-only transactions model the cost of recognizing them as read-only and marking them as such.

4.1.4.2. The Simulation Approach

In order to simulate the multiple version algorithms, it was assumed that old versions of objects are accessible in as little time as the most recent version of each object. This assumption is reasonable if access paths for locating versions of active data items can be kept in primary memory. Such caching of version location information can be probably be achieved using algorithms such as those described in [Chan82]. Otherwise, the results reported here will be optimistic about the degree of performance improvements which are obtainable using the multiple version algorithms. Thus, the only salient aspect of simulating multiple version algorithms is representing their different concurrency control semantics.

The implementation of these multiple version algorithms was simplified by the observation that none of them ever rejects a read request. It can also be shown that, under the "no blind writes" assumption, MVTO accepts and rejects writes in exactly the same way that BTO does (see Appendix 2).

Thus, no transaction can ever write a version of an object other than the latest one in any of the multiple version algorithms. As a result, each multiple version algorithm was implemented by simply changing the simulation description for its single version counterpart to always accept read requests.

4.1.4.3. Experiments and Results

In this section, the results of three multiple version performance experiments are reported. Each of these experiments was performed on the three multiple version algorithms just described and their single version counterparts. The experiments were designed to investigate the performance of the multiple version algorithms under both low conflict transaction mixes and mixes which are more likely to benefit from the availability of multiple versions. The first experiment investigates the performance of the algorithms under mixes of transactions where the level of conflicts is fairly low. The second experiment investigates the performance of the algorithms as a function of read-only transaction size for a mix of small update transactions and larger read-only transactions. The third experiment investigates the performance of the algorithms as a function of the fraction of read-only transactions in the mix.

4.1.4.3.1. Experiment 1: Low Conflict Performance

This experiment investigates the performance of the multiple version algorithms under a transaction mix similar to that of experiment 3.4 of the previous chapter. The only change in the mix is that large transactions will be read-only in this case, as multiple version algorithms are specifically aimed at improving performance for such mixes. The objective of this experiment is to investigate the performance of the multiple version algorithms for situations where conflicts are fairly rare. The system parameters used for the experiment are given in Table 4.1, and the workload parameters are given in Table 4.2. The *batch_time* and *num_batches* parameter settings used in this experiment are the same as those used for the previous experiments. The database consists of 10,000 objects, and its granularity is one object per granule. The number of terminals used is 10. Small update transactions, which are eighty percent of the mix, each read two objects and then update them each with fifty percent probability. Large transactions, the other twenty percent of the mix, read a uniformly distributed number of objects

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	35
startup_cpu	10
obj_io	35
obj_cpu	10
cc_io	0
cc_cpu	1
stagger_mean	20

Table 4.1: System parameters for experiment 1.

sequentially. The mean size of the large transactions in the mix is 30, and they are read-only.

The results of this experiment are shown in Table 4.3, where throughput rates for the three multiple version algorithms and their single version counterparts are given for various granularities. The associated restart counts are given in Table 4.4. All of the multiple version algorithms yield similar performance (within limits of statistical variation) for all but the coarsest granularity. This occurs because the probability of conflicts between pairs of update transactions is quite small, given their size, and all of the multiple version algorithms allow read-only transactions to proceed without interference of any kind.

Workload Parameters	
db_size	10000 objects
gran_size	1 object/granule
num_terms	10
delay_mean	1 second
small_prob	0.8
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	30
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.0

Table 4.2: Workload parameters for experiment 1.

Throughput versus Granularity						
Grans	BTO	MVTO	2PL	VP	SV	MVSV
1	1.707	1.707	1.837	2.228	0.407	2.364
10	2.632	2.844	2.831	2.939	1.183	2.863
100	2.930	2.998	3.009	3.011	2.397	2.999
1000	2.918	3.012	3.012	3.013	2.691	3.012
10000	2.926	3.013	3.013	3.013	2.755	3.013

Table 4.3: Throughput, experiment 1.

Restarts versus Granularity						
Grans	BTO	MVTO	2PL	VP	SV	MVSV
1	6005	6005	3951	3865	494	3340
10	761	933	809	396	545	811
100	94	80	34	31	214	77
1000	42	7	0	0	82	7
10000	32	0	0	0	56	0

Table 4.4: Restarts, experiment 1.

The CCA version pool (VP) algorithm does not offer any significant performance improvement over 2PL except at the coarsest granularity examined. This is because 2PL is capable of handling the workload of the experiment quite well without multiple versions. Both of the other multiple version algorithms do succeed in outperforming their single version counterparts, although the advantage of MVTO over BTO is only slight and occurs only for coarse granularity (10 granules in the database). The advantages of multiple versions are much more pronounced for SV versus MVSV, as SV is the worst of the three single version algorithms for the transaction mix of this experiment. The poor performance of SV here occurs because transactions are not checked for conflicts until transaction commit time, a practice that strongly biases single version SV against large read-only transactions: They perform all their reads and then test to see if any of the granules have been updated, an occurrence which is likely with many small update transactions in the mix.

4.1.4.3.2. Experiment 2: Read-Only Transaction Size

This experiment investigates the performance characteristics of the multiple version algorithms under a workload consisting of a mix of small update transactions and larger read-only transactions. The parameter varied here is the size of the read-only transactions in the mix. The purpose of the experiment is to observe the behavior of the algorithms while varying the degree to which old versions may beneficial. The workload parameters used in this experiment were selected in order to emphasize situations in which multiple version algorithms are indeed beneficial.

The system parameter settings for this experiment are the same as those used for experiment 1 (see Table 4.1). The relevant workload parameter settings for this experiment are given in Table 4.5. The database consists of 100 objects, and its granularity is one object per granule. The number of terminals used is 10. Small update transactions, which are forty percent of the mix, read two objects and update each with fifty percent probability. Large read-only transactions, the other sixty percent of the mix, sequentially read a uniformly distributed number of objects. The mean size for large transactions is varied from 1 to 30 objects.

The motivation for selecting such a small database size was two-fold. First, it was desired that read-only transactions read a significant fraction of the database so that the probability of conflicts with update transactions would be significant. Otherwise, the multiple version algorithms would not be of use in improving performance. Second, it was necessary to keep the size of read-only transactions small in terms of the number of objects accessed so that reasonably tight confidence intervals could be obtained without using unreasonable amounts of simulation time (i.e., using the same *batch_time* and *num_batches* parameters that were used in Chapter 3). This tradeoff dictated the selection of a relatively small database size. One might also view these

Workload Parameters	
db_size	100 objects
gran_size	1 object/granule
num_terms	10
delay_mean	1 second
small_prob	0.4
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	vary from 1 to 30 objects
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.0

Table 4.5: Workload parameters for experiment 2.

parameter settings as an approximation to a large database with fairly coarse granularity.

The results of this experiment are shown in Table 4.6, where throughput rates for the three multiple version algorithms and their single version counterparts are given for six different read-only transaction sizes. Table 4.7 gives the associated restart counts. All of the multiple version algorithms again yield approximately the same performance, as the probability of conflicts between small transactions is still small.

Throughput versus Read-Only Transaction Size						
Size	BTO	MVTO	2PL	VP	SV	MVSV
1	7.613	7.613	7.716	7.717	7.386	7.669
2	6.545	6.573	6.641	6.641	6.110	6.610
5	4.435	4.649	4.668	4.675	3.722	4.660
10	2.725	3.174	3.157	3.183	1.957	3.177
15	1.903	2.462	2.452	2.468	1.271	2.464
30	0.812	1.336	1.282	1.336	0.483	1.336

Table 4.6: Throughput, experiment 2.

Restarts versus Read-Only Transaction Size						
Size	BTO	MVTO	2PL	VP	SV	MVSV
1	240	240	54	53	536	133
2	214	186	39	39	618	103
5	281	83	22	17	706	58
10	362	38	31	6	654	27
15	431	27	24	7	562	17
30	372	4	50	2	373	2

Table 4.7: Restarts, experiment 2.

The CCA version pool algorithm only outperforms the 2PL algorithm upon which it is based for the largest size setting examined, and not by much. This is because 2PL itself is again quite successful in handling the mix in the experiment. Neither 2PL nor VP are deadlock-free or particularly close to deadlock-free for this mix, as both permit lock upgrades. Any slight advantage of VP over 2PL may thus be explained by the fact that large read-only transactions can cause update transactions to queue up waiting for a common granule in 2PL. When this granule subsequently becomes available, the waiting update transactions will each obtain read locks and then deadlock when they attempt to upgrade these locks to write locks. It is expected that a version pool variant of 2PLW, in which upgrades are not permitted, would have no performance advantages whatsoever over 2PLW.

Both of the other multiple version algorithms do succeed in outperforming their single version counterparts. The advantages of multiple versions are again most pronounced for SV versus MVSV. The poor performance of SV is due to its bias against large read-only transactions for this workload. BTO is also somewhat biased against large read-only transactions, as objects which are read late in their execution are likely to have been updated by younger transactions. With BTO, though, read-only transactions have some chance of causing conflicting update transactions to be restarted. Also, read-only transactions are likely to be restarted less far into their execution with BTO, thus wasting fewer resources. As a result, MVTO does not provide the same relative improvement over BTO that MVSV provides over SV.

4.1.4.3.3. Experiment 3: Read-Only Transaction Fraction

This experiment again investigates the performance characteristics of the multiple version algorithms under a workload consisting of a mix of small update transactions and large read-only transactions. The parameter varied in this case is the fraction of read-only transactions in the mix. The workload parameters of the previous experiment were used again in order to emphasize situations in which multiple version algorithms are beneficial.

The system parameter settings for this experiment are the same as those used in previous experiments (see Table 4.1). The workload parameter settings for this experiment are the same as those of the previous experiment with *large_mean* = 30 (see Table 4.5). The fraction of read-only transactions in the mix is varied by varying the the *small_prob* parameter (the fraction of update transactions in the mix) from 0.0 to 1.0.

Throughput versus Update Transaction Fraction						
Pr(Sm)	BTO	MVTO	2PL	VP	SV	MVSV
0.0	0.878	0.878	0.878	0.878	0.878	0.878
0.2	0.815	1.043	1.021	1.043	0.540	1.043
0.4	0.812	1.336	1.282	1.336	0.483	1.336
0.6	0.981	1.868	1.739	1.872	0.526	1.867
0.8	1.130	2.947	2.606	2.956	0.546	2.943
1.0	6.842	6.842	7.013	7.010	6.691	6.790

Table 4.8: Throughput, experiment 3.

Restarts versus Update Transaction Fraction						
Pr(Sm)	BTO	MVTO	2PL	VP	SV	MVSV
0.0	0	0	0	0	0	0
0.2	224	1	35	0	286	2
0.4	372	4	50	2	373	2
0.6	506	24	73	12	418	23
0.8	730	72	163	30	453	83
1.0	605	605	263	267	908	702

Table 4.9: Throughput, experiment 3.

The throughput results of this experiment are shown in Table 4.8. The restart counts are shown in Table 4.9. Again, all three of the multiple version algorithms yield virtually identical performance. The main new finding of this experiment is that multiple version algorithms outperform their single version counterparts most significantly when the mix contains mostly small update transactions. For MVTO and MVSV, an explanation is that when many small update transactions can run to completion during the execution of a few large read-only transactions, the update transactions are likely to do something which will force the read-only transactions to be restarted. All it takes, roughly speaking, is for one update transaction to write something that a read-only transaction reads. An explanation for VP is that a mix of many small update transactions and a few large read-only transactions is the most likely mix to suffer from the increased lock upgrade deadlock problem described previously.

4.2. GRANULARITY HIERARCHIES

In addition to studying alternative locking protocols and their correctness, several researchers have examined issues associated with selecting the appropriate level of granularity for partitioning a database into lockable units [Gray75, Gray79, Ries79a, Ries79b]. It was found that a database can be organized as a hierarchy of lockable units, called a *lock hierarchy*, and locking protocols for such a hierarchy have been developed [Gray75, Gray79, Kort82]. It was also found that, under some typical transaction mixes, a lock hierarchy offers improved system performance [Ries79a, Ries79b].

Other types of concurrency control algorithms have been proposed, some of which have been studied in previous chapters of this thesis. Most proposals ignore the granularity issue, modeling a database simply as a homogeneous, unstructured collection of fixed-size objects. Timestamp-based algorithms have been criticized for this very reason [Gray81b]. In this section of the thesis, it is shown that granularity hierarchies can be used outside the domain of locking. Hierarchical versions of a validation algorithm, a timestamp algorithm, and a multiversion algorithm are presented. These hierarchical algorithms were first introduced in [Care83b]. The simulation model of the previous chapter is extended to handle hierarchical algorithms, and the performance characteristics of several hierarchical concurrency control algorithms are studied.

4.2.1. Hierarchical Algorithms

This section reviews of the concept of hierarchical locking and presents a hierarchical version of the PRE algorithm whose performance was studied in the previous chapter. Following a review of of lock hierarchies, three new hierarchical concurrency control algorithms are developed. First, however, some useful notation must be introduced.

For a granule g, the notation *parent*(g) will refer to the granule immediately above g in the hierarchy. The notation *children*(g) will refer to the set of granules right below g in the hierarchy. The notation *descendents*(g) will refer to the set of all descendents of g in the hierarchy. Finally, the notation *ancestors*(g) will refer to the set of all ancestors of g in the hierarchy. Granules at the bottom level of the hierarchy will be referred to as *leaf* granules.

4.2.1.1. Hierarchical Locking

In locking algorithms, a transaction wishing to access some item in the database must first lock the item. A key performance question is: How big should the lockable items (or *granules*) be? To maximize potential concurrency for small transactions, many small granules are best, and to minimize locking overhead for large transactions, a few large granules are best. The notion of *hierarchical locking* was introduced to allow more than one level of granularity to be used.

In hierarchical locking [Gray75, Gray79, Kort82], the database is viewed as a hierarchy of granules. When a transaction sets a lock on an item at a given level of the hierarchy, it is implicitly locking all of its descendents as well. Small transactions obey a locking protocol whereby they set intention locks at higher levels of the hierarchy before setting access locks at a lower level. An intention lock on a granule indicates that some lower-level granule is (or will be) locked. Large transactions can then avoid setting many lower-level locks. The result is a small increase in locking overhead for small transactions, a penalty which is hopefully offset by a large decrease in locking overhead for large transactions.

A hierarchical version of preclaimed exclusive locking (H-PRE) is described informally in Figure 4.4. The functions *ILocked*(g) and *XLocked*(g) return true if is g is currently locked in intention or exclusive mode, respectively. Procedures *ILock*(g) and *XLock*(g) are used to set these two types of locks. The function *ableToRun*(T) returns true if all locks required by transaction T are available. All locks are determined to be available if no exclusive lock is set on any ancestor of any granule which T wishes to access, and no intention or exclusive lock is set on any of the granules themselves. The procedure *setLocks*(T) is the called by each transaction T at startup time in order to preclaim its locks. This routine sets all locks for T if they are available, blocking T if not. Intention locks are set on each ancestor of each granule to be accessed by T, and exclusive locks are set on each of the

```
function ableToRun($T$);
begin
  runnable := true;
  foreach $g$ in $readset(T) \bigcup writeset(T)$ do
    foreach $G$ in $ancestors(g)$ do
      if XLock($G$) then
        runnable := false;
      fi;
    od;
    if ILock($g$) or XLock($g$) then
      runnable := false;
    fi;
  od;
  return runnable;
end;


procedure setLocks($T$);
begin
  if ableToRun($T$) then
    foreach $g$ in $readset(T) \bigcup writeset(T)$ do
      foreach $G$ in $ancestors(g)$ do
        ILock($G$);
      od;
      XLock($g$);
    od;
  else
    block($T$);
  fi;
end;
```

Figure 4.4: Informal description of H-PRE algorithm.

granules themselves.

4.2.1.1.1. Hierarchical Validation

In this section, a hierarchical version of the serial validation algorithm (SV) of [Kung81] will be presented. In describing the hierarchical version of the algorithm, the notation and assumptions of Chapter 2 are employed.

For hierarchical serial validation (H-SV), the read and write sets of a transaction will be sets of granules. Short transactions may specify these sets in terms of small granules, and large transactions may specify them in terms of granules higher up in the granularity hierarchy. As in SV, these sets are used for commit-time conflict testing. A transaction T is validated if $readset(T) \cap writeset(T_{rc}) = \emptyset$ for all transactions $T_{rc} \in RC(T)$, where $RC(T)$ is the set of recently committed transactions for T. In testing for possible conflicts under H-SV, the algorithm must recognize that a granule g_1 has some data in common with another granule g_2 if $g_1 = g_2$, $g_1 \in ancestors(g_2)$, or $g_2 \in ancestors(g_1)$. The H-SV algorithm is given in Figure 4.5. The validation test used here is the original test of Kung and Robinson [Kung81]. A hierarchical version of the timestamp-based SV algorithm can be developed in a manner analogous to the hierarchical version of BTO which will be presented in the next section.

Theorem: The hierarchical version of SV is correct in the sense that serializability is guaranteed.

Proof: The SV algorithm is known to be correct [Kung81]. Thus, it suffices to show that H-SV only commits transactions which would be committed by SV. This may be shown as follows:

When a transaction T requests access to a granule g, it is requesting permission to access some or all of the granules in $descendents(g)$. The H-SV

```
procedure validate($T$);
begin
  valid := true;
  foreach $T_{rc}$ in $RC(T)$ do
    foreach $g_r$ in $readset(T)$ do
      foreach $g_w$ in $writeset(T_{rc})$ do
        if $g_r = g_w$ or $g_r \in ancestors(g_w)$
        or $g_w \in ancestors(g_r)$ then
          valid := false;
        fi;
      od;
    od;
  od;
  if valid then
    commit $writeset(T)$ to database;
  else
    restart($T$);
  fi;
end;
```

Figure 4.5: H-SV algorithm.

algorithm will commit T as long as no granule g_r in its readset either contains, equals, or is a sub-granule of another granule g_w in the writeset of any recently committed transaction. This ensures that H-SV commits T only if there is no overlap between the leaf granules associated with granules in the readset of T and those associated with granules in the writeset of any recently committed transaction. These associated leaf granules are a superset of those which would comprise the readset of T and the writesets of the recently committed transactions for SV. Thus, SV would commit T as well.

●

4.2.1.1.2. Hierarchical Timestamps

This section presents a hierarchical version of the basic timestamp ordering algorithm (BTO) of [Bern81b]. The assumptions and notation used in describing the hierarchical version of the algorithm are the same as those used in the initial description of the algorithm in Chapter 2.

To extend the BTO algorithm for hierarchical use, each granule g will have read and write *summary* timestamps, $R_S\text{-}TS(g)$ and $W_S\text{-}TS(g)$, in addition to its actual read and write timestamps. Its read and write summary timestamp values will be:

$$R_S\text{-}TS(g) = \max\{R\text{-}TS(G) \mid G \in g \cup descendents(g)\}$$

$$W_S\text{-}TS(g) = \max\{W\text{-}TS(G) \mid G \in g \cup descendents(g)\}$$

The actual read (write) timestamp for a granule g is the largest timestamp of any transaction for which a read (write) request for g has been granted. The summary read (write) timestamp for g is the largest timestamp of any transaction for which a read (write) request for g or any sub-granule of g has been granted. With these timestamps at each level of the hierarchy, the BTO algorithm requires two extensions. First, when a transaction T wishes to access a granule g, it must make sure that no granule in $ancestors(g)$ has an actual timestamp that violates the BTO ordering rules when compared with $TS(T)$. This would mean that some transaction younger than T has already made a request that potentially conflicts with

```
procedure readReq($T$,$g$);
begin
   $okay$ := true;
   foreach $G$ in $ancestors(g)$ do
      if $TS(T) < W\text{-}TS(G)$ then
         $okay$ := false;
      fi;
   od;
   if $TS(T) < W_S\text{-}TS(g)$ then
      $okay$ := false;
   fi;
   if not $okay$ then
      restart($T$);
   else
      grant readReq;
      $R\text{-}TS(g) := \max(TS(T), R\text{-}TS(g))$;
      $R_S\text{-}TS(g) := \max(TS(T), R_S\text{-}TS(g))$;
      while $parent(g)$ exists do
         $g := parent(g)$;
         $R_S\text{-}TS(g) := \max(TS(T), R_S\text{-}TS(g))$;
      od;
   fi;
end;


procedure writeReq($T$,$g$);
begin
   $okay$ := true;
   foreach $G$ in $ancestors(g)$ do
      if $TS(T) < R\text{-}TS(G)$ or $TS(T) < W\text{-}TS(G)$ then
         $okay$ := false;
      fi;
   od;
   if $TS(T) < R_S\text{-}TS(g)$ or $TS(T) < W_S\text{-}TS(g)$ then
      $okay$ := false;
   fi;
   if not $okay$ then
      restart($T$);
   else
      grant writeReq;
      $W\text{-}TS(g) := TS(T)$;
      $W_S\text{-}TS(g) := TS(T)$;
      while $parent(g)$ exists do
         $g := parent(g)$;
         $W_S\text{-}TS(g) := \max(TS(T), W_S\text{-}TS(g))$;
      od;
   fi;
end;
```

Figure 4.6: H-BTO algorithm.

T's request. Second, the algorithm must propagate timestamp changes upwards in the hierarchy to keep the summary timestamp values accurate. The hierarchical version of BTO (H-BTO) is given in Figure 4.6.

To illustrate the roles played by the actual and summary timestamps in H-BTO, consider the simple hierarchy of Figure 4.7, where there are two lower-level granules, X and Y, and one upper-level granule, XY. Suppose that $R\text{-}TS(X) = 8$, $W\text{-}TS(X) = 8$, $R\text{-}TS(Y) = 15$, $W\text{-}TS(Y) = 13$, $R\text{-}TS(XY) = 5$, and $W\text{-}TS(XY) = 5$. This implies that X and Y have been accessed since time 5, but that their parent granule XY has not been accessed as a whole since that time. The summary timestamp values will be $R_S\text{-}TS(X) = 8$, $W_S\text{-}TS(X) = 8$, $R_S\text{-}TS(Y) = 15$, $W_S\text{-}TS(Y) = 13$, $R_S\text{-}TS(XY) = 15$, and $W_S\text{-}TS(XY) = 13$. (Note that the actual and summary timestamp values are always the same for leaf granules, so it is not actually necessary to maintain them separately at the bottom level of the hierarchy.)

Now, suppose that a transaction T with timestamp $TS(T) = 10$ wishes to read X. H-BTO checks $W\text{-}TS(XY)$, finds that the request is acceptable thus far, then checks $W_S\text{-}TS(X)$ and finds that the request is indeed acceptable. H-BTO grants the request, setting $R\text{-}TS(X)$ and $R_S\text{-}TS(X)$ equal to 10. Suppose instead that T had wished to read XY. H-BTO would have checked $W_S\text{-}TS(XY)$, found that the request violated the BTO ordering rules for reading because some sub-granule of XY had been written since time 10,

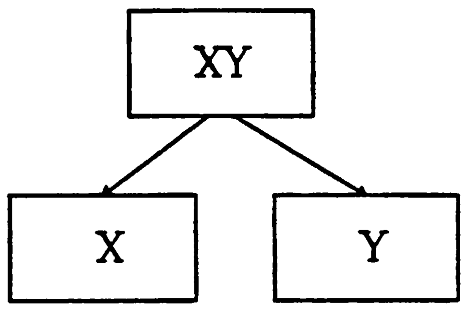

Figure 4.7: Simple example hierarchy.

and rejected the request.

Theorem: The hierarchical version of BTO is correct in the sense that serializability is guaranteed.

Proof: The BTO algorithm is known to be correct [Bern82a]. Thus, it suffices to show that H-BTO only grants requests which would be granted by BTO. This may be shown as follows:

When a transaction T requests access to a granule g, it is requesting permission to access some or all of the granules in $descendents(g)$. The H-BTO algorithm grants a read request as long as two conditions hold:

(1) $TS(T) \geq W\text{-}TS(G)$ for all $G \in ancestors(g)$

(2) $TS(T) \geq W_S\text{-}TS(g)$

The first condition guarantees that no transaction younger than T has written any granule which contains g, and thus possibly g itself. The second condition guarantees that no transaction younger than T has written any portion of g. If both conditions (1) and (2) hold, H-BTO grants the request. This occurs only when no transaction younger than T has written g or any portion thereof, so no write timestamps of leaf granules associated with g in the hierarchy would exceed $TS(T)$ in the BTO algorithm. BTO would therefore grant the request as well.

The H-BTO algorithm grants write requests as long as two conditions hold:

(1) $TS(T) \geq R\text{-}TS(G)$ and $TS(T) \geq W\text{-}TS(G)$ for all $G \in ancestors(g)$

(2) $TS(T) \geq R_S\text{-}TS(g)$ and $TS(T) \geq W_S\text{-}TS(g)$

The first condition guarantees that no transaction younger than T has read or written any granule which contains g, and thus possibly g itself. The second condition guarantees that no transaction younger than T has read or written any portion of g. If both conditions (1) and (2) hold, H-BTO grants the request. This occurs only when no transaction younger than T has read or written g or any portion thereof, so no read or write timestamps of leaf granules associated with g in the hierarchy would exceed $TS(T)$ in the BTO algorithm. BTO would therefore grant the request as well. •

4.2.1.1.3. Hierarchical Multiple Version Algorithms

In this section, a hierarchical variant of the multi-version timestamp ordering algorithm (MVTO) of Reed [Reed78] is presented. Version management and concurrency control are treated separately in the description, making the hierarchical version of MVTO a natural extension of its non-hierarchical counterpart. The notation and definitions used here are the same as those used in the description of the non-hierarchical version of the algorithm.

To extend the MVTO algorithm for hierarchical use, each granule g will have a read/write *summary* history, $H_S(g)$, in addition to its actual history, $H_{rw}(g)$. This summary history will be:

$$H_S(g) = \bigcup \{H_{rw}(G) \mid G \in g \cup descendents(g)\}$$

Thus, $H_S(g)$ is the union of $H_{rw}(G)$ for all granules G having any data in common with g. This union operation may be interpreted graphically. The read/write history for a granule can be thought of as a timeline, with the intervals in the history being line segments drawn on this timeline. The union of two or more histories is computed by overlaying their timelines, with the intervals in the resulting history being those intervals included in one or more of the histories being unioned. For example, the union of {(3,7), (10,13)} and {(1,2), (5,11) (15,17)} would be {(1,2), (3,13), (15,17)}. This example is also depicted graphically in Figure 4.8.

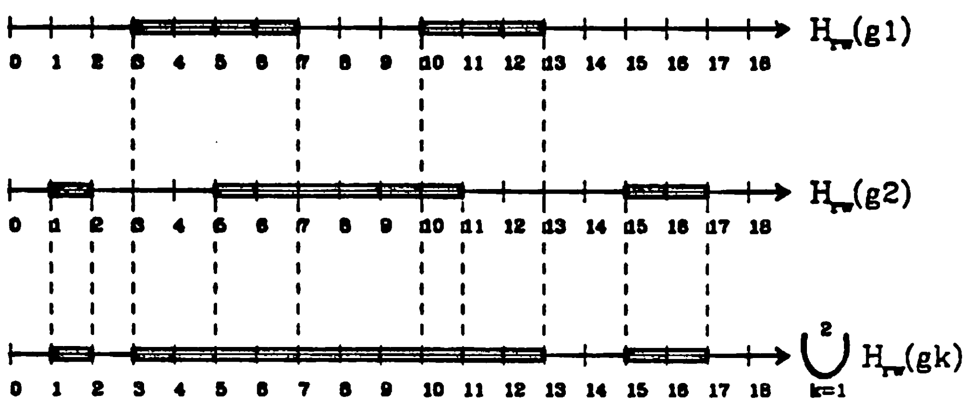

Figure 4.8: Union operation for read/write histories.

These actual and summary histories are analogous to the actual and summary timestamps used in creating the H-BTO algorithm from the BTO algorithm. With these histories at each level of the hierarchy, the MVTO algorithm requires two extensions. First, when a transaction T wishes to write a granule g, it must make sure that no higher-level granules have actual histories with an interval containing $TS(T)$. Second, when a transaction T causes some history to be updated, the algorithm must propagate the change upwards in the hierarchy to keep the summary histories accurate. For the lowest level granules in the hierarchy, the actual and summary histories will always be equal (just as for timestamps in the H-BTO algorithm), so they need not be separately maintained for leaf granules.

```
procedure readReq($T$,$g$);
begin
   grant readReq;
   extVers($H_{rv}(g)$, $TS(T)$);
   extVers($H_S(g)$, $TS(T)$);
   while $parent(g)$ exists do
      $g$ := $parent(g)$;
      extVers($H_S(g)$, $TS(T)$);
   od;
end;


procedure writeReq($T$,$g$);
begin
   $okay$ := true;
   foreach $G$ in $ancestors(g)$ do
      if $TS(T)$ in $H_{rv}(G)$ then
         $okay$ := false;
      fi;
   od;
   if $TS(T)$ in $H_S(g)$ then
      $okay$ := false;
   fi;
   if not $okay$ then
      restart($T$);
   else
      grant writeReq;
      newVers($H_{rv}(g)$, $TS(T)$);
      newVers($H_S(g)$, $TS(T)$);
      while $parent(g)$ exists do
         $g$ := $parent(g)$;
         newVers($H_S(g)$, $TS(T)$);
      od;
   fi;
end;
```

Figure 4.9: H-MVTO algorithm.

The hierarchical version of MVTO (H-MVTO) is given in Figure 4.9. It is assumed in the figure that the *newVers* operation creates a new interval in a history by taking the union of the history and the new interval, and that the *extVers* operation merges intervals when extending one causes it to overlap with another.

Theorem: The hierarchical version of MVTO is correct in the sense that serializability is guaranteed.

Proof: The MVTO algorithm is known to be correct [Bern82b]. Thus, it suffices to show that H-MVTO only grants requests which would be granted by MVTO. This may be shown as follows:

When a transaction T requests access to a granule g, it is requesting permission to access some or all of the granules in *descendents*(g). The H-MVTO algorithm always grants read requests, just as the MVTO algorithm does. The H-MVTO algorithm grants write requests as long as two conditions hold:

(1) $TS(T)$ is in no interval in $H_{rw}(G)$ for any $G \in$ *ancestors*(g)

(2) $TS(T)$ is in no interval in $H_S(g)$

The first condition guarantees that no granule containing g has an interval which contains $TS(T)$, so the version of g to be written cannot have been read by a younger transaction. The second case guarantees that no granule contained within g has an interval which contains $TS(T)$, so no portion of the version of g to be written can have been read by a younger transaction. If both conditions (1) and (2) hold, H-MVTO grants the request. This occurs only when neither g nor any portion thereof has a version which was written before $TS(T)$ and read after $TS(T)$, so no read/write histories of leaf granules associated with g in the hierarchy would have intervals containing $TS(T)$ under MVTO. MVTO would therefore grant the request as well. •

4.2.2. Hierarchies and Performance

In this section, the performance characteristics of several hierarchical concurrency control algorithms are investigated using the simulation model of Chapter 3. The algorithms studied are a hierarchical version of PRE and the three new hierarchical concurrency control algorithms presented in the previous section. PRE was chosen for hierarchical study because it was felt to be representative of the class of hierarchical locking algorithms and was the easiest of the locking algorithms to implement hierarchically. The other three algorithms were chosen as being representative of hierarchical versions of the validation, timestamp, and multiple version approaches. The purpose of this performance study is to investigate the hypothesis that any hierarchical concurrency control algorithm should display much the same performance advantages over its non-hierarchical counterpart as locking does in some situations [Ries79a, Ries79b]. Before presenting the details of the performance experiments of this section, however, the manner in which the performance model of Chapter 3 was extended to accommodate the study of a two-level granularity hierarchy is described.

4.2.2.1. Modeling a Hierarchy

In order to allow hierarchical concurrency control algorithms to be simulated, a new parameter, *size_threshold*, was added to the simulator. This parameter defines the threshold, in objects, used to classify transactions as being small or large for concurrency control purposes. Small transactions are those with readset sizes which are less than or equal to *size_threshold*, and large transactions are those whose readsets exceed *size_threshold*. The other modification made to support hierarchical algorithms involved changing the interpretation of the *gran_size* parameter somewhat. A two-level hierarchy is simulated, and it is assumed that each lower level granule in the hierarchy contains just one database object. The *gran_size* parameter is thus used to determine the size of the higher-level database granules in the hierarchy.

4.2.2.2. Concurrency Control Costs

As with previous simulations, it is necessary to make some assumptions about the concurrency costs of the various algorithms to be simulated. The cost models used for the H-PRE, H-SV, H-BTO, and H-MVTO algorithms are simple variations on the cost models previously described for PRE, SV, BTO, and MVTO. For transactions which access higher-level granules, the costs are assessed as previously described for PRE, SV, BTO, and MVTO using the number of unique higher-level granules read and written in place of the number of database granules read and written. For transactions which access lower-level granules, concurrency control costs are computed using the number of lower-level granules (objects) read and written, and then the resulting costs are doubled. This cost doubling models the fact that transactions which access lower-level granules incur extra overhead to descend the two-level hierarchy of concurrency control information each time a request is processed.

4.2.2.3. Experiments and Results

In this section, the results of two hierarchical performance experiments are reported. The experiments were performed on H-PRE, H-SV, H-BTO, H-MVTO, and the non-hierarchical counterparts of these four hierarchical algorithms. The first experiment investigates the performance of the hierarchical algorithms under low concurrency control costs. The second experiment was designed to investigate the potential performance advantages offered by hierarchical algorithms in certain cases.

Before discussing the experiments and results, one other point should be made to motivate the choice of large values for the concurrency control cost parameters: Hierarchical algorithms are useful for enhancing performance only when the costs associated with each concurrency control request are large. In Chapter 3, where a 1 millisecond (simulated time) CPU cost was charged for processing concurrency control requests, the results of the experiments with this CPU cost showed no performance degradation at fine granularities. Only in the last experiment in the chapter, where the cost of concurrency control request processing was extremely high, was the need for a hierarchy indicated. Thus, hierarchies may not be useful in practice today. However, as large primary memories allow more and more data to be buffered for long periods of time in main memory, the ratio of the cost of object processing to the cost of concurrency control should decrease. This may make hierarchies more attractive, making a study of their behavior worthwhile.

4.2.2.3.1. Experiment 1: Low Concurrency Control Cost

This experiment investigates the performance characteristics of hierarchical concurrency control algorithms under a workload consisting of a mix of small and large transactions. The parameter varied in this experiment is the size of the large transactions in the mix, the intention being to observe the behavior of the algorithms as the degree to which hierarchies are potentially helpful is varied. The system parameter settings used are those of Table 4.1, where concurrency control costs are low. The *batch_time* and *num_batches* parameter settings used in this experiment are the same as those used for the previous experiments. The purpose of this experiment is to investigate the performance of the hierarchical algorithms when concurrency control costs are not particularly high.

The workload parameter settings for this experiment are given in Table 4.10. The database consists of 10,000 objects, and its granularity is 10 objects per higher-level granule in the two-level hierarchy. The number of terminals used is 10. Small transactions, which are forty percent of the mix, each read two objects, updating each object with fifty percent probability. Large transactions, the other sixty percent of the mix, sequentially read a uniformly distributed number of objects. The mean size for large transactions is varied

Workload Parameters	
db_size	10000 objects
gran_size	10 object/granule
num_terms	10
delay_mean	1 second
small_prob	0.4
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	vary from 1 to 30 objects
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.1
size_threshold	4 objects

Table 4.10: Workload parameters for experiment 1.

from 1 to 30 objects. Each object is updated with ten percent probability. The size threshold for distinguishing between small and large transactions is 4 objects, so transactions which access 4 or fewer objects are considered small and make their concurrency control requests based on objects (lower-level granules). Transactions which access 5 or more objects are considered large and make their requests based on higher-level granules.

The throughput results for experiment 1 are given in Tables 4.11a and 4.11b, with the associated restart counts given in Tables 4.12a and 4.12b. The results are exactly as one would expect with low concurrency control costs: No significant performance improvements are offered by the hierarchical algorithms in this case. The tiny performance differences that do show up

Throughput versus Large Transaction Size				
Size	PRE	H-PRE	SV	H-SV
1	7.444	7.445	7.435	7.434
2	6.346	6.346	6.337	6.327
5	4.414	4.414	4.386	4.368
10	2.930	2.930	2.888	2.873
15	2.281	2.281	2.215	2.189
30	1.228	1.228	1.130	1.112

Table 4.11a: Throughput, experiment 1.

Throughput versus Large Transaction Size				
Size	BTO	H-BTO	MVTO	H-MVTO
1	7.444	7.442	7.444	7.442
2	6.342	6.331	6.342	6.331
5	4.404	4.374	4.405	4.385
10	2.895	2.867	2.905	2.861
15	2.234	2.181	2.235	2.178
30	1.113	1.081	1.111	1.079

Table 4.11b: Throughput, experiment 1 (cont.).

Restarts versus Large Transaction Size				
Size	PRE	H-PRE	SV	H-SV
1	0	0	13	14
2	0	0	11	24
5	0	0	18	37
10	0	0	25	40
15	0	0	37	53
30	0	0	49	58

Table 4.12a: Restarts, experiment 1.

Restarts versus Large Transaction Size				
Size	BTO	H-BTO	MVTO	H-MVTO
1	5	7	5	7
2	4	18	4	18
5	9	34	6	24
10	19	46	12	48
15	24	55	19	51
30	56	71	51	70

Table 4.12b: Restarts, experiment 1 (cont.).

are in the favor of the non-hierarchical algorithms. These arise because of a slightly elevated restart count due to the fact that large transactions claim somewhat more data than needed with the hierarchical algorithms, thus increasing the probability of conflicts. In this case, no significant gains in terms of concurrency control overhead are available to offset this effect. Of course, since the size of large transactions is never chosen to be very large due to statistical considerations, this does not imply that hierarchies would not be useful for mixes including much larger transactions.

4.2.2.3.2. Experiment 2: Large Transaction Size

This experiment repeats the study of the previous experiment in a situation where concurrency control costs are large. The parameter varied in this experiment is again the size of the large transactions in the mix, the intention being to observe the behavior of the algorithms as the degree to which hierarchies are potentially helpful is varied. The system and workload parameters used in the experiment were selected in order to emphasize situations where hierarchical algorithms are indeed beneficial. The system parameter settings for this experiment, are summarized in Table 4.13. The concurrency control costs for this experiment are a combination of the largest *cc_cpu* and *cc_io* parameters used in experiment 6 of Chapter 3. The workload parameters are the same as those used for the previous experiment (see Table 4.10).

The results of this experiment are shown in Tables 4.14a and 4.14b, where throughput rates for the four hierarchical algorithms and their non-

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	35
startup_cpu	10
obj_io	35
obj_cpu	10
cc_io	35
cc_cpu	5
stagger_mean	20

Table 4.13: System parameters for experiment 2.

Throughput versus Large Transaction Size				
Size	PRE	H-PRE	SV	H-SV
1	4.662	3.395	4.292	3.010
2	3.869	3.314	3.577	2.942
5	2.534	2.928	2.353	2.601
10	1.608	2.204	1.505	1.956
15	1.245	1.781	1.134	1.578
30	0.662	1.049	0.571	0.884

Table 4.14a: Throughput, experiment 2.

Throughput versus Large Transaction Size				
Size	BTO	H-BTO	MVTO	H-MVTO
1	4.293	3.013	4.293	3.013
2	3.580	2.944	3.580	2.944
5	2.363	2.602	2.365	2.607
10	1.514	1.945	1.514	1.942
15	1.153	1.575	1.153	1.558
30	0.584	0.897	0.590	0.891

Table 4.14b: Throughput, experiment 2 (cont.).

Restarts versus Large Transaction Size				
Size	PRE	H-PRE	SV	H-SV
1	0	0	3	3
2	0	0	8	12
5	0	0	9	21
10	0	0	13	27
15	0	0	24	38
30	0	0	23	47

Table 4.15a: Restarts, experiment 2.

Restarts versus Large Transaction Size				
Size	BTO	H-BTO	MVTO	H-MVTO
1	1	2	1	2
2	3	9	3	9
5	5	18	2	13
10	8	36	8	37
15	15	37	12	47
30	16	40	11	39

Table 4.15b: Restarts, experiment 2 (cont.).

hierarchical counterparts are given for six different large transaction sizes. The non-hierarchical algorithms made all their concurrency control requests based on objects for this study. The results are consistent with what one would expect based on the results of the previous chapter and by Ries for a two-level lock hierarchy [Ries79a, Ries79b]. That is, all of the concurrency control algorithms studied do benefit from the use of a granularity hierarchy.

The first result worth noting is that the performance of the hierarchical algorithms follow the same trend as was observed in Chapter 3 for their non-hierarchical counterparts. The hierarchical version of PRE, which uses blocking rather than restarts, exhibits the best performance of the three algorithms. The H-SV algorithm performs second best, and the H-BTO and H-MVTO algorithms perform the least well of the three algorithms studied. The restart counts for this experiment are given in Tables 4.15a and 4.15b. Thus, the use of a granularity hierarchy does not alter the result that blocking is the mechanism of choice for dealing with transaction conflicts.

Second, the use of a granularity hierarchy helps improve performance as expected for all of the algorithms. Initially, when large transactions are of mean size 1, the hierarchical algorithms exhibit worse performance than their non-hierarchical counterparts, as the overhead for small transactions is greater using the hierarchical algorithms. However, as the size of the large transactions in the mix is increased, the hierarchical algorithms perform better due to reduced concurrency control overhead for large transactions. With a mean large transaction size of 5 objects, large transactions make concurrency control requests in terms of higher-level granules, and a slight performance improvement is observed. As large transaction size is increased further, the degree of the benefit increases. As much as fifty percent additional throughput is available using the hierarchical algorithms instead of the non-hierarchical algorithms by the time *large_mean* = 30 objects is reached.

4.3. SUMMARY

This chapter has examined two performance enhancement schemes for concurrency control algorithms, multiple versions and granularity hierarchies. In terms of algorithmic concepts, several multiple version algorithms were reviewed, and a new, multiple version serial validation algorithm was presented. The notion of hierarchical locking was reviewed, and the notion was extended so as to be applicable to other types of algorithms. New hierarchical versions of serial validation, basic timestamp ordering, and multiversion timestamp ordering were presented.

Three experiments were presented which addressed the performance issues associated with multiple versions, and a number of conclusions were drawn from the results of the experiments. First, multiple versions did not improve performance much for 2PL, except in extreme cases, as 2PL was quite capable of handling the transaction mixes tested without multiple versions. The improvement of MVTO over BTO was also fairly limited. Multiple versions helped improve performance the most for SV, as SV was the worst of the single-version algorithms for the transaction mixes examined.

Second, all of the multiple version algorithms performed virtually alike in the experiments, as they all enabled large read-only transactions to execute without interference. The sizes of small update transactions and the database were such that conflicts between update transactions were very unlikely, so differences in the performance of the algorithms for update transactions did not play a role in the results. Finally, it was found that multiple versions improved performance the most for workloads consisting of many small update transactions and a few large read-only transactions.

Two experiments were presented which addressed the performance issues associated with granularity hierarchies. The experiments investigated using a two-level hierarchy in scenerios where the cost of concurrency control was normal and extremely large, respectively. It was found that hierarchies are not helpful if concurrency control costs are not significant, as is the case normally. The results for hierarchical versions of PRE, SV, BTO, and MVTO indicate that hierarchies are indeed beneficial for mixes of small and large transactions when concurrency control is extremely expensive, however. The hierarchical PRE algorithm performed the best out of all of the hierarchical algorithms investigated, reinforcing the conclusions of Chapter 3 about blocking versus restarts.

CHAPTER 5

CONCLUSIONS

In this chapter, the results of the previous chapters are reviewed. The overall conclusions of this thesis are compared and contrasted with those of other concurrency control researchers. Implications of the results of this work for other types of concurrency control algorithms, such as distributed concurrency control algorithms and algorithms which use information about transaction semantics, are presented. Finally, directions are suggested for future research in the area of concurrency control algorithms and performance.

5.1. SUMMARY OF RESULTS

Chapter 2 of this thesis presented an abstract model of concurrency control algorithms which was useful for describing algorithms and analyzing their relative storage and CPU costs. Results obtained from a comparison of the costs of a two-phase locking variant, basic timestamp ordering, and serial validation indicated that the two-phase locking variant examined has the best overall cost characteristics. The storage costs of basic timestamp ordering and serial validation each contained factors dependent upon the number of recently committed transactions as well as the currently active transactions, making them potentially more expensive (especially when transactions tend to access disjoint sets of data items). The no-conflict CPU costs of the algorithms were less distinct, with differences involving constant factors on the order of a factor of two or less, although locking was again the lowest-cost algorithm. In light of the results of later chapters, which indicate that locking is superior in terms of performance, it suffices to summarize the main cost result obtained using the model of Chapter 2 as follows:

(2.1) The costs associated with two-phase locking are at least as low as those for basic timestamp ordering and serial validation.

Chapter 3 presented a performance model of concurrency control algorithms which was implemented in the form of a fairly flexible, detailed simulation program. Seven concurrency control algorithms, including two-phase locking with full deadlock detection (with and without read/write upgrades), two-phase locking with wait-die deadlock prevention, preclaimed exclusive two-phase locking, basic timestamp ordering (with and without the Thomas write rule), and serial validation. The main conclusions of Chapter 3 can be summarized as follows:

(3.1) When conflicts between transactions were rare, all of the concurrency control algorithms examined performed equally well. If Gray is right about conflicts being rare in most real database systems [Gray81a], the choice of a concurrency control algorithm will not affect performance.

(3.2) For workloads in which conflicts were not rare, the concurrency control algorithms that performed the best were those which minimized the number of transaction restarts. Blocking is thus the mechanism of choice for dealing with transaction conflicts.

(3.3) The main difference between workloads of small transactions and those with larger transactions is that larger transactions make the penalty associated with restarting a transaction even greater. For mixes of small and large transactions, performance can be improved by attempting to select small transactions to restart when restarts are required.

(3.4) If concurrency control costs are low compared to the cost of object accesses, which is likely for current database systems, concurrency control overhead does not affect performance. Fine granularities are recommended in this case, although optimal performance can be obtained with as few as 1000 or more granules unless large, random transactions are anticipated. If concurrency control costs are high, the conclusions regarding blocking versus restarts still hold, but concurrency control overhead will be a significant factor. Medium granularities or a hierarchy will be necessary for optimal performance.

(3.5) Some algorithms have anomalies in their behavior which can degrade their performance in the absence of a sufficient period of delay following transaction restarts. An example is the cyclic restart anomaly shared by basic timestamp ordering and multiversion timestamp ordering.

Chapter 4 investigated two concurrency control performance enhancement schemes, multiple versions and granularity hierarchies. Previously proposed algorithms using each of these schemes were reviewed. A new multiple version algorithm was proposed, and several new hierarchical algorithms were also proposed. Performance experiments were performed to investigate situations in which multiple versions and granularity hierarchies offer performance benefits. The main conclusions of Chapter 4 were:

(4.1) Multiple versions may be used in conjunction with serial validation in order to improve performance when the workload includes update transactions and large read-only transactions.

(4.2) For workloads consisting of read-only transactions and update transactions with the property that conflicts between update transactions were of low probability, all of the multiple version algorithms performed alike, enabling transactions to execute with little interference.

(4.3) Multiple versions did little to improve the performance of 2PL, as 2PL did well for the workloads examined without multiple versions. They did help somewhat for basic timestamp ordering and significantly for serial validation, however.

(4.4) Multiple versions were the most effective for workloads consisting of many small update transactions and a few large read-only transactions.

(4.5) Granularity hierarchies may be used in conjunction with serial validation, basic timestamp ordering, and multiversion timestamp ordering to attempt to improve performance when the cost of concurrency control is high and the workload includes a mix of small and large transactions.

(4.6) Hierarchical versions of preclaimed locking, serial validation, basic timestamp ordering, and multiversion timestamp ordering all succeeded in improving performance for a mix of small and large transactions. Hierarchical preclaimed locking performed the best of the hierarchical algorithms investigated. However, these performance improvements were only obtained when the cost of concurrency control was extremely high. No improvements were obtained for the transaction mixes examined with normal concurrency control costs.

5.2. COMPARISON WITH OTHER WORK

In this section, the results of this thesis are compared with those of other related studies. The most relevant related work to date has been performed by Ries [Ries77, Ries79a, Ries79b], Bernstein and Goodman [Bern80], Lin and Nolte [Lin82, Lin83], Peinl and Reuter [Pein83], Galler [Gall82], Robinson

[Robi82a, Robi82b], and Agrawal and DeWitt [Agra83b, Agra83c]. Each of these studies will be reviewed in turn, and the results of each will be examined in light of the conclusions of this thesis.

In his thesis studies, Ries [Ries77, Ries79a, Ries79b] examined the effects of granularity on performance. Most of his studies assumed that an I/O cost was associated with concurrency control, as in experiment 6 of Chapter 3. Ries found that relatively coarse granularities, on the order of 100 granules, were sufficient to obtain optimal performance for locking. The findings of this thesis agree with those of Ries, as granularities of 100 to 1000 granules were sufficient to achieve the best performance for each algorithm studied in Chapter 3 under most conditions. Also, Ries compared a pair of locking algorithms similar to PRE and 2PL of Chapter 3, and he found that PRE tended to outperform 2PL. PRE won over 2PL in the studies of this thesis as well, the reason being that no restarts occurred with PRE. Ries also attributed the dominance of PRE to its lack of restarts. Ries observed a tradeoff between maximizing concurrency and minimizing concurrency control overhead. The results of experiment 6 of Chapter 3 demonstrated the existence of this tradeoff for the seven algorithms studied in this chapter. Ries found that a lock hierarchy was useful in the presence of this tradeoff, and experiment 2 of the granularity hierarchy section of this thesis yielded similar results for all algorithms tested.

Bernstein and Goodman performed a comprehensive study of concurrency control algorithms for distributed database systems [Bern80]. They identified four metrics — blocking, restarts, messages, and local processing cost — for rating alternative algorithms, and they qualitatively examined a large number of algorithms using the metrics. Their results were inconclusive, leading to the identification of eleven algorithms based on locking and timestamps as "dominant" over all other alternatives. To the extent that several locking variants were included in the dominant set, the results of this thesis agree with those of Bernstein and Goodman. However, a main result of this thesis is that restarts are a driving performance factor, whereas blocking is acceptable as long as a sufficient number of transactions remain unblocked to keep critical system resources well utilized. Thus, the use of blocking and restarts as separate metrics seems inappropriate in light of the results of the experiments reported here. Also, the results of the first five experiments of Chapter 3 suggest that the local processing cost metric may not be of interest for most concurrency control algorithms.

Lin and Nolte have performed a number of simulation studies of locking versus timestamps for distributed database systems [Lin82, Lin83]. The conclusions of the most recent of their papers are that multiversion timestamp ordering is only marginally better than basic timestamp ordering, that basic timestamp ordering performs better than two-phase locking if the average transaction size is small, and that two-phase locking performs better than basic timestamp ordering if the average transaction size is large. The conclusions that basic timestamp ordering outperforms locking in some situations contradicts the findings of this thesis. This contradiction occurs for several reasons, the primary one being that the system model used by Lin and Nolte does not account for the effects of CPU and I/O resource sharing, as discussed below. Also, differences between centralized and distributed systems may contribute to differences in the results of the studies.

The model of Lin and Nolte assumes that the message delays, CPU processing times, and I/O processing times associated with concurrency control requests and subsequent object processing can be legitimately combined into a single, exponentially determined "communications delay" time. This totally eliminates the fact that CPU and I/O resources are shared by all transactions, so service times for transactions in their model are not dependent on the number of other transactions in service. It is precisely the sharing of CPU and I/O resources which allows blocking to have little or no negative performance impact. The sharing of resources also makes the penalty associated with restarts greater, as restarted transactions waste CPU and I/O resources which could have been used by other transactions instead. The absence of shared resource modeling in Lin and Nolte's work would clearly cause their results to differ from those reported here. They state in their most recent paper [Lin83] that they intend to investigate more detailed models, and this thesis would suggest that their future results will differ from previous ones.

A recent paper by Peinl and Reuter reports using a metric based on combining the average level of multiprogramming and the number of restarts to evaluate several alternative concurrency control algorithms driven by reference strings from an actual database system [Pein83]. Algorithms studied in the paper include two-phase locking, a two-version variant of locking, and serial validation. Peinl and Reuter obtain results which indicate that the serial validation algorithm leads to the largest number of restarts, yet they end up concluding that, in terms of their performance parameter, it performs well. The results of this thesis indicate that their parameter, which is the ratio of the level of multiprogramming and the factor by which the number of requests is increased by restarts, is not the best choice for a performance metric. A metric based on number of restarts alone seems more appropriate as long as the level of multiprogramming is five or more. Peinl and Reuter also stated some conclusions about the relative performance of one-version versus two-version algorithms, but this thesis did not examine multiple version algorithms which restricted the number of versions available.

In his thesis, Galler [Gall82] presented a performance model for exclusive two-phase locking in a single-site database system, a qualitative framework for selecting among alternative distributed concurrency control algorithms, and some simulation results for basic timestamp ordering versus locking in a distributed database system. Of these, the simulation results are most relevant to the conclusions drawn here. In particular, Galler reported finding that basic timestamp ordering outperformed two-phase locking in a variety of environments for mixes of small transactions. Galler attributed this to longer waits until necessary restarts are performed in locking and greater parallelism between sites with basic timestamp ordering.

Differences between distributed and single-site systems are thought to be a factor in the contradiction between the results of this thesis and the work of Galler. Modeling differences are another factor. The model used by Galler is similar to that of Lin and Nolte, although Galler claims to account for load-dependence of transaction service times. Galler concentrates solely on mixes of transactions which read and write a single object, and his locking algorithm uses timeouts in place of true deadlock detection. In one study, Galler shows the throughput for basic timestamp ordering increasing quite steadily up to a multiprogramming level of at least ten, suggesting that his model of CPU and I/O sharing may be unrealistic. Results from the multiprogramming level experiments of this thesis suggest that throughput is not likely to increase once a multiprogramming level of four or five has been reached. If anything, throughput would be expected to *decrease* beyond this point due to an increase in the probability of conflicts and the use of restarts.

Robinson performed research on the design of general transaction processing systems [Robi82a, Robi82b]. Robinson designed and implemented such a system on the Cm* multiprocessor system at CMU [Full78]. As a test of the generality of his design, Robinson performed experiments in which several different concurrency control algorithms were executed by the system. His results for locking versus serial validation indicated that the throughput produced using locking was higher than that produced using serial validation, and that locking led to fewer restarts. Although Robinson did not intend for his work to be interpreted as a conclusive study of concurrency control performance, the results of his experiments concur with those of this thesis.

Finally, Agrawal and DeWitt have recently completed a performance study of several combinations of concurrency control and recovery algorithms [Agra83b, Agra83c]. The result of their study, which was based on an analytical model of the "burden" experienced by transactions operating under the alternative concurrency control and recovery algorithm combinations, was that locking (combined with several different recovery mechanisms) was the best choice examined. They also concluded that serial validation was only reasonable under workloads consisting of small transactions for which conflicts are rare. Both of their conclusions are consistent with the results reported here.

5.3. IMPLICATIONS OF THE RESULTS

The results of this thesis have several implications. For single-site database systems, it is clear that algorithms which prefer blocking to restarts should be chosen. If conflicts are not rare, such algorithms will outperform their competitors, and if conflicts are truly rare, they will perform at least as well. To summarize the discussion at the end of Chapter 3, if sufficient knowledge of transaction reads and writes is available at transaction startup time, preclaiming with read and write lock modes is probably best. Alternatively, if writes can be anticipated when objects are read, an upgrade-free two-phase locking variant with a victim selection criteria based on the amount of work completed by transactions is also an excellent alternative. If reads and writes cannot be predicted before they occur, two-phase locking with deadlock detection and an improved victim selection criteria is the algorithm of choice. Also, in special cases where structural knowledge makes using deadlock-free locking protocols an option, such as in hierarchical databases or operations on a tree-structured index [Silb80, Moha82], such protocols are to be recommended.

For algorithms which utilize transaction semantics or data-type-specific operation information in an attempt to improve concurrency [Allc82, Bern78, Bern81a, Kort83, Garc83, Hsu83, Schw82, Spec83], the implication of this thesis is that specialized locking protocols are probably the most promising approach. The use of semantic information will not change the fact that restarts degrade performance in situations where conflicts occur with other than low probability.

Even for distributed concurrency control algorithms, this thesis would seem to indicate that locking protocols are likely to dominate those which resolve conflicts using restarts. The main result of the performance experiments of Chapter 3 was that blocking did not significantly degrade performance. With the multiprogramming level of the single-site database system modeled kept at five or more, the bottleneck resource was fully utilized and the maximum throughput was obtained. Suppose that the sites in a distributed database system each have a number of active transactions, and further suppose that locking can succeed at keeping this number at levels comparable to those which produced optimal throughputs in the single-site system. Each site in the system should then operate at its maximum throughput, and optimal performance should be achieved for the entire distributed system. This argument should hold as long as the number of messages required for locking algorithms is comparable to those for other algorithms, so the costs for locking are not greater than the costs for its competitors in distributed database environments. Similar arguments also apply to the selection of a concurrency control algorithm for a database server in a local computer network or for a distributed database system in which a central site is to be used for concurrency control purposes.

5.4. LIMITATIONS OF RESULTS

The results reported in this thesis are subject to the limitations of the models used to obtain them. There are a number of assumptions which were made in the performance model of Chapters 3 and 4, and each of these assumptions has influenced the results of the thesis to some extent. This section reviews the major assumptions which underly the model and discusses their expected influence. The major underlying assumptions are:

(1) Transactions do not pause during their execution.

(2) The cost of processing a collection of concurrency control requests is proportional to the size of the collection.

(3) Buffer contents are flushed for restarted transactions, so the cost of executing a transaction from beginning to end is the same independent of the restart history of the transaction.

(4) The overhead associated with switching contexts from one transaction to another is not large.

(5) Each object read by a transaction is read only once, and all objects written must previously have been read.

The first assumption may be interpreted as a decision to model a transaction processing system rather than an interactive query processing system, a decision which may be justified by noting that the performance impact of concurrency control is probably most important for transaction processing environments in which high throughputs are required. New applications such as engineering design systems or database browsers weaken the validity of this assumption. Ries briefly examined the impact of such transaction idle times on the performance of locking, and he found that coarse granularities were still sufficient to produce near-optimal performance [Ries79a]. This result suggests that additional blocking due to idle transactions would not change the thesis result that locking is the algorithm of choice. Otherwise, this additional blocking should have led to a need for finer lock granularity in his study.

The second assumption may be restated as the assumption that the overhead of performing a concurrency control system call is not the dominant factor in the cost of concurrency control request processing. Otherwise, the simple cost modeling approach taken in the simulations, based on the *cc_io* and *cc_cpu* parameters, poorly reflects reality. Because one of the conclusions of Chapter 3 was that concurrency control overhead is insignificant as long as it is small compared to the other costs (like object processing) incurred by transactions, it is not expected that altering this assumption would significantly alter the results.

The third and fourth assumptions have to do with the costs of restarts and blocking. The third assumption basically says that restarted transactions were modeled by starting them all over again, having them re-read all of the objects in their readsets and re-write all of the objects in their writeset. The fourth assumption says that blocking was modeled by setting blocked transactions aside, and that the cost of blocking was assumed to be some fraction of the average locking cost modeled by *cc_io* and *cc_cpu*. If these assumptions were drastically modified, so that restarts were nearly free and the cost of blocking (and context switching) was very high in comparison, it is expected that the results for blocking versus restarts would come out differently.

(Robinson made comments along these lines in a discussion of the outcome of his experiments, though his results indicated that blocking was cheap and restarts were expensive in his Cm* transaction processing system.) It is felt that such a cost reversal is unlikely for real database systems, as environments where restarts are cheap due to vast amounts of buffer space are also likely to have sufficient memory to make context switches inexpensive by keeping many active transactions in primary memory.

The final assumption can be interpreted as a combination of assuming that transactions have sufficient buffer space to maintain all items which may be re-read in primary memory, and that transactions do not make "blind writes". Both assumptions were made for convenience in generating and manipulating transaction read and write sets in the simulator. It is not expected that changing either of these assumptions would to lead to major changes in the results of this thesis. If the "no blind writes" assumption were relaxed, however, minor differences would be expected to arise among certain of the timestamp algorithms. (Appendix 2 discusses the algorithmic aspects of the effects of the "no blind writes" assumption on basic timestamp ordering with the Thomas write rule and on multiversion timestamp ordering.)

5.5. FUTURE RESEARCH DIRECTIONS

One area of further research which may be appropriate is to extend the simulation model of Chapter 3, relaxing some of the assumptions that were discussed in the previous section. It was argued that this should not alter the conclusions significantly, but only by trying can the validity of these arguments be demonstrated.

Another obvious area for future work is to extend the performance model of Chapter 3 to the distributed case, introducing a network model to connect a collection of the single-site queuing models in order to investigate the hypothesis that locking algorithms will be dominant for distributed concurrency control as well. Much of the work done in this area suffers from modeling deficiencies, discussed earlier in this chapter, and might benefit from the use of the more detailed single-site model of this thesis. An alternative to modeling would be to be to measure the performance of alternative algorithms in a real system or a representative testbed system of some sort. This undertaking would be worthwhile if a reasonably modular database system were used as a starting point so that the implementation difficulties would not be prohibitive.

Finally, with main memory sizes increasing dramatically each year, it is anticipated that more and more data will be maintained in primary rather than secondary memory as database systems progress towards meeting the transaction throughput demands of the late 1980's [Gray83]. This may happen by mapping databases into the virtual address space of transactions [Trai82, Ston83], or by managing a large buffer pool in such a way that most data of interest is kept in primary memory. In either case, new concurrency control and recovery techniqués may need to be investigated to provide synchronization and recovery for transactions in such extremely high-throughput environments, as the overhead associated with current techniques may become prohibitive there.

REFERENCES

[Agra83a] Agrawal, R., Carey, M., and DeWitt, D., "Deadlock Detection is Cheap", ACM SIGMOD Record, January 1983.

[Agra83b] Agrawal, R., and DeWitt, D., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", Technical Report No. 497, Computer Sciences Department, University of Wisconsin-Madison, February 1983.

[Agra83c] Agrawal, R., "Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation", Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1983.

[Allc82] Allchin, J., and McKendry, M., "Object-Based Synchronization and Recovery", School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings of the COMPSAC '79 Conference, Chicago, Illinois, November 1979.

[Bada81] Badal, D., "Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms", Proceedings of the Fifth Berkeley Workshop on Distributed

Data Management and Computer Networks, February 1981.

[Baye80] Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems 5(2), June 1980.

[Bern78] Bernstein, P., Rothnie, J., Goodman, N., and Papadimitriou, C., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Transactions on Software Engineering 4(3), May 1978.

[Bern79] Bernstein, P., Shipman, D., and Wong, W., "Formal Aspects of Serializability in Database Concurrency Control", IEEE Transactions on Software Engineering 5(3), May 1979.

[Bern80] Bernstein, P., and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Technical Report, Computer Corporation of America, 1980.

[Bern81a] Bernstein, P., Goodman, N., and Lai, M., "Two Part Proof Schema for Database Concurrency Control", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, February 1981.

[Bern81b] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys 13(2), June 1981.

[Bern82a] Bernstein, P., and Goodman, N., "A Sophisticate's Introduction to Distributed Database Concurrency Control", Proceedings of the Eighth International Conference on Very Large Data Bases, September 1982.

[Bern82b] Bernstein, P., and Goodman, N., "Multiversion Concurrency Control Theory and Algorithms", Technical Report No. TR-20-82, Aiken Computation Laboratory, Harvard University, June 1982.

[Brya80a] Bryant, R., "SIMPAS – A Simulation Language Based on PASCAL", Technical Report No. 390, Computer Sciences Department, University of Wisconsin-Madison, June 1980.

[Brya80b] Bryant, R., SIMPAS User Manual, Computer Sciences Department and Madison Academic Computing Center, University of Wisconsin-Madison, December 1980.

[Care83a] Carey, M., "An Abstract Model of Database Concurrency Control Algorithms", Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1983.

[Care83b] Carey, M., "Granularity Hierarchies in Concurrency Control", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March

1983.

[Casa79] Casanova, M., "The Concurrency Control Problem for Database Systems", Ph.D. Thesis, Computer Science Department, Harvard University, 1979.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February 1982.

[Chan82] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., "The Implementation of An Integrated Concurrency Control and Recovery Scheme", Proceedings of the ACM SIGMOD International Conference on Management of Data, March 1982.

[Conw63] Conway, R., "Some Tactical Problems in Digital Simulation", Management Science 10(1), January 1963.

[Date82] Date, C., An Introduction to Database Systems (Volume II), Addison-Wesley Publishing Company, 1982.

[Elli77] Ellis, C., "A Robust Algorithm for Updating Duplicate Databases", Proceedings of the 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.

[Eswa76] Eswaren, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM 19(11), November 1976.

[Ferr78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Full78] Fuller, S., Ousterhout, J., Raskin, L., Rubinfeld, P., Sindhu, P., Swan, R., "Multi-Microprocessors: An Overview and Working Example", Proceedings of the IEEE 66(2), February 1978.

[Gall82] Galler, B., "Concurrency Control Performance Issues" Ph.D. Thesis, Computer Science Department, University of Toronto, September 1982.

[Garc79] Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database", Ph.D. Thesis, Computer Science Department, Stanford University, June 1979.

[Garc83] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database", ACM Transactions on Database Systems 8(2), June 1983.

[Good83] Goodman, N., Suri, R., and Tay, Y., "A Simple Analytic Model for Performance of Exclusive Locking in Database Systems", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March 1983.

[Gray75] Gray, J., Lorie, R., Putzulo, G., and Traiger, I., "Granularity of Locks and Degrees of Consistency in a Shared Database", Report No. RJ1654, IBM San Jose Research Laboratory, September 1975.

[Gray79] Gray, J., "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[Gray81a] Gray, J., Homan, P., Korth, H., and Obermarck, R., "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System", Report No. RJ3066, IBM San Jose Research Laboratory, February 1981.

[Gray81b] Gray, J., "The Transaction Concept: Virtues and Limitations", Proceedings of the Seventh International Conference on Very Large Databases, September 1981.

[Gray83] Gray, J., "Practical Problems in Data Management -- A Position Paper", Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1983.

[Grif83] Griffeth, N., and Morsi, M., "SORCERER: A Distributed Database Testbed and Simulation Tool", School of Information and Computer Science, Georgia Institute of Technology, 1983.

[Hsu83] Hsu, M., and Madnick, S., "Hierarchical Database Decomposition -- A Technique for Database Concurrency Control", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March 1983.

[Iran79] Irani, K., and Lin, H., "Queueing Network Models for Concurrent Transaction Processing in a Database System", Proceedings of the ACM SIGMOD International Symposium on Management of Data, 1979.

[Kort82] Korth, H., "Deadlock Freedom Using Edge Locks", ACM Transactions on Database Systems 7(4), December 1982.

[Kort83] Korth, H., "Locking Primitives in a Database System", Journal of the ACM 30(1), January 1983.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6(2), June 1981.

[Lin82] Lin, W., and Nolte, J., "Distributed Database Control and Allocation: Semi-Annual Report", Technical Report, Computer Corporation of America, Cambridge, Massachusetts, January 1982.

[Lin83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking", submitted to Symposium on Reliability in Distributed Software and Database Systems, Palo Alto, California, October 1983.

[Lind79] Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Price, T., Putzolu, F., Traiger, I., and Wade, B., "Notes on Distributed Databases", Report No. RJ2571, IBM San Jose Research Laboratory, 1979.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

[Moha82] Mohan, C., Fussel, D., and Silberschatz, A., "Compatibility and Commutativity in Non-Two-Phase Locking Protocols", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March 1983.

[Papa79] Papadimitriou, C., "Serializability of Concurrent Updates", Journal of the ACM 26(4), October 1979.

[Pein83] Peinl, P., and Reuter, A., "Empirical Comparison of Database Concurrency Control Schemes", Department of Computer Sciences, University of Kaiserslautern, West Germany, 1983.

[Poti80] Potier, D., and LeBlanc, P., "Analysis of Locking Policies in Database Management Systems", Proceedings of the Performance '80 Conference, 7th IFIP W.G.7.3 International Symposium on Computer Performance Modeling, Measurement, and Evaluation, Toronto, May 1980.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Ries77] Ries, D., and Stonebraker, M., "Effects of Locking Granularity on Database Management System Performance", ACM Transactions on Database Systems 2(3), September 1977.

[Ries79a] Ries, D., "The Effects of Concurrency Control on Database Management System Performance", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1979.

[Ries79b] Ries, D., and Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems 4(2), June 1979.

[Robi82a] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1982.

[Robi82b] Robinson, J., "Experiments with Transaction Processing on a Multi-Microprocessor", Report No. RC9725, IBM Thomas J.

Watson Research Center, December 1982.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems 3(2), June 1978.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data", Proceedings of the Fourth Annual Symposium on the Simulation of Computer Systems, August 1976.

[Saue81] Sauer, C., and Chandy, N., Computer Systems Performance Modeling, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Schw82] Schwarz, P., and Spector, A., "Synchronizing Shared Abstract Types", Technical Report CMU-CS-82-128, Carnegie-Mellon University, September 1982.

[Silb80] Silberschatz, A., and Kedem, Z., "Consistency in Hierarchical Database Systems", Journal of the ACM 27(1), January 1980.

[Spec83] Spector, A., and Schwarz, P., "Transactions: A Construct for Reliable Distributed Computing", Operating Systems Review 17(2), April 1983.

[Stea81] Stearns, R., and Rosenkrantz, D., "Distributed Database Concurrency Controls Using Before-Values", Technical Report, SUNY Albany, February 1981.

[Ston76] Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES", ACM Transactions on Database Systems 1(3), September 1976.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering 5(3), May 1979.

[Ston83] Stonebraker, M., "Virtual Memory Transaction Management", in preparation.

[Svob81] Svoboda, L., "A Reliable Object-Oriented Repository for a Distributed Computer System", Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 4(2), June 1979.

[Trai82] Traiger, I., "Virtual Memory Management for Database Systems", Operating Systems Review 16(4), October 1982.

[Ullm83] Ullman, J., Principles of Database Systems, Second Edition, Computer Science Press, Rockville, Maryland, 1983.

[Wolf83] Wolff, R., personal communication.

APPENDIX 1

TIMESTAMP-BASED SERIAL VALIDATION

In this appendix, it is shown that the timestamp-based version of serial validation (T-SV) from Chapter 2 of the thesis preserves the semantics of the original serial validation (SV) algorithm of Kung and Robinson [Kung81]. The equivalence proof is based on showing that the T-SV algorithm commits exactly those transactions which would be committed by the SV algorithm, restarting all transactions which SV would restart.

Lemma 1: All transactions which are committed by the SV algorithm are also committed by the T-SV algorithm.

Proof: Suppose some transaction T is committed by SV but restarted by T-SV. Let $RC(T)$ be the set of recently committed transactions, those which committed between the time when T started executing and the time at which it entered the validation critical section. Since T is committed by SV, it must be true that $\mathit{readset}(T) \cap \mathit{writeset}(T_{rc}) = \emptyset$ for all transactions $T_{rc} \in RC(T)$. Since T is restarted by T-SV, it must also be true that $TS(x) > S\text{-}TS(T)$ for some $x \in \mathit{readset}(T)$. However, $TS(x) > S\text{-}TS(T)$ implies that x was written by a transaction which committed subsequent to the startup of T, as $TS(x)$ is the commit timestamp of the most recent writer of x and $S\text{-}TS(T)$ is the startup timestamp of T. Thus, x must be in *writeset*(T_{rc}) for some transaction $T_{rc} \in RC(T)$. This contradicts the assumption that SV committed T, proving the lemma. •

Lemma 2: All transactions which are restarted by the SV algorithm are also restarted by the T-SV algorithm.

Proof: Suppose some transaction T is restarted by SV but committed by T-SV. Let $RC(T)$ be the set of recently committed transactions, those which committed between the time when T started executing and the time at which it entered the validation critical section. Since T is restarted by SV, it must be true that some $x \in$ *readset*(T) is also in *writeset*(T_{rc}) for some transaction $T_{rc} \in RC(T)$. Since T is committed by T-SV, it must also be true that $TS(x) < S\text{-}TS(T)$ for all $x \in$ *readset*(T). However, $TS(x) < S\text{-}TS(T)$ implies that x has not been written by any transaction which committed subsequent to the startup of T, as $TS(x)$ is the commit timestamp of the most recent writer of x. Thus, x cannot be in *writeset*(T_{rc}) for any transaction $T_{rc} \in RC(T)$. This contradicts the assumption that SV restarted T, proving the lemma. •

Theorem: The set of transactions committed by the T-SV algorithm is precisely that set of transactions which would be committed by the SV algorithm, so the T-SV algorithm preserves the semantics of the SV algorithm.

Proof: The theorem follows directly from a combination of Lemmas 1 and 2.

•

APPENDIX 2

THE "NO BLIND WRITES" ASSUMPTION

The "no blind writes" assumption says that transactions never write a data item without having first read the item. This appendix investigates the consequences of this assumption on two timestamp-based concurrency control algorithms, basic timestamp ordering (BTO) with the Thomas write rule [Bern81b] and Reed's multiversion timestamp ordering (MVTO). In particular, it is shown that certain conditions anticipated by each of these algorithms cannot arise if transactions always read items before writing them.

Thomas Write Rule

The only difference between the BTO algorithm and BTO with the Thomas write rule is that BTO will reject a write request from a transaction T for a data item x if either $R\text{-}TS(x) > TS(T)$ or $W\text{-}TS(x) > TS(T)$, whereas BTO with the Thomas write rule will only reject the write request if $R\text{-}TS(x) > TS(T)$. In the latter case, where $W\text{-}TS(x) > TS(T)$, BTO with the Thomas write rule accepts the request and ignores the actual write. The idea of the Thomas write rule, then, is to reduce the number of restarts by ignoring old writes.

Lemma: Under the "no blind writes" assumption, no write request from from a transaction T for an item x will ever find $R\text{-}TS(x) \leq TS(T)$ when $W\text{-}TS(x) > TS(T)$.

Proof: The "no blind writes" assumption implies that $R\text{-}TS(x) \geq W\text{-}TS(x)$ for all x, as $R\text{-}TS(x)$ is the timestamp of the youngest transaction which has read x, $W\text{-}TS(x)$ is the timestamp of the youngest transaction which has written x, and $R\text{-}TS(x) < W\text{-}TS(x)$ would mean that the youngest writer of x never read x before writing it. The lemma follows trivially. •

Theorem: Under the "no blind writes" assumption, the behavior of the BTO algorithm and the BTO algorithm with the Thomas write rule will be exactly the same.

Proof: Since the lemma shows that the only difference between the two algorithms disappears under the "no blind writes" assumption, BTO and BTO with the Thomas write rule must behave identically under this assumption. •

Multiversion Timestamp Ordering

In the variant of multiversion timestamp ordering (MVTO) presented in [Reed78], a write request from a transaction T for an item x is accepted as long as no interval in $H_{rw}(x)$ contains the time $TS(T)$. In other words, if $TS(T)$ lies in a hole in $H_{rw}(x)$, a space between the last read of one version of x and the creation (write) of the next newer version of x, the write request is accepted. When a write request is accepted, a new version of x is created, and the interval $(TS(T), TS(T'))$ is added to x to denote the new, unread version. Reads are always accepted, and are granted by returning to T the latest version of x which was created prior to timestamp time $TS(T)$.

The write request processing logic of MVTO in this original version is a bit complex, and has it been simplified in more recent related work [Svob81]. The simplification, made so that versions other than the most recent one cannot be written (for laser disk implementation purposes), involves granting a write request from T for x only if $TS(T)$ is greater than the timestamp of the latest read of the most recent version of x. More simply, writes are only accepted if they define the newest, most recent version of x, in which case the MVTO algorithm basically becomes BTO enhanced with a version pool for use by read-only transactions. It can be shown that versions of MVTO with and without this simplification treat write requests identically under the "no blind writes" assumption.

Lemma: Under the "no blind writes" assumption, the read/write history of each item x will have no holes. The rightmost point of each interval associated with each version of x, except for the most recent version, will coincide with the leftmost point of the interval associated with the next newer version of x.

Proof: When a transaction T submits a write request for x, it has already successfully read x under the "no blind writes" assumption. At the time of this read request, then, the interval in $H_{rw}(x)$ associated with the version of x

read by T will have been extended to $TS(T)$. If T's write request is now granted, the interval associated with the new version of x will begin at $TS(T)$, exactly where the previous interval stopped. Thus, all writes create versions which pick up where the previous version left off, making holes in $H_{rw}(x)$ impossible. •

Theorem: Let $R\text{-}TS(x)$ denote the maximum timestamp of any transaction which has read the most recent version of x using MVTO (i.e., the rightmost read in $H_{rw}(x)$). Under the "no blind writes" assumption, write requests will be accepted only if $TS(T)$ equals or exceeds $R\text{-}TS(T)$.

Proof: The lemma showed that holes do not exist in read/write histories under the "no blind writes" assumption. Hence, outdated writes can never be accepted under the rules of MVTO. Only a write which creates a new, most recent version of x will be accepted. Such a write can only be submitted by a transaction T with timestamp $TS(T)$ such that $TS(T) \geq R\text{-}TS(T)$. •

APPENDIX 3

SIMULATION OUTPUT ANALYSIS

This appendix presents the statistical analysis techniques used to interpret the simulation results of the experiments of Chapters 3 and 4. The methods employed here are based on a combination of the traditional batch means approach [Sarg76, Ferr78, Saue81] and a slightly more sophisticated technique for estimating variance for use in computing confidence intervals for the results. The improved variance estimation technique was proposed by Wolff [Wolf83]. Sample results, obtained from simulations presented in the thesis, are given to demonstrate the utility of this approach. Also, the confidence interval results from each of the experiments of Chapters 3 and 4 are given for reference at the end of this appendix.

A Statistical Model

This section describes the modified batch means approach used in this thesis. For brevity, the ***batch_time*** parameter will be referred to as T_b and the ***num_batches*** parameter will be referred to as n_b. Each simulation is run for T_b simulation time units, and the overall simulation is divided into n_b batches of T_b/n_b simulation time units apiece. The throughput estimate for the i^{th} batch, denoted as X_i for $1 \leq i \leq n_b$, is the ratio of the number of transactions which commit during the batch to the length of the batch in simulation time units. (This number is multiplied by a scaling factor of 1000 for display purposes.) These estimates are summed and divided by the total number of batches in order to compute the overall throughput estimate for a simulation:

$$\overline{X} = \frac{1}{n_b} \sum_{i=1}^{n_b} X_i \tag{1}$$

Variance Estimation

To model the statistical characteristics of the throughput observations, it is assumed that $\{X_i\}$ is a stationary sequence and that each X_i has mean μ and variance σ^2. It is further assumed that the correlation between adjacent batches is significant, but that the correlation between non-adjacent batches is negligible, an idea proposed in [Conw63]. More formally:

$$Cov(X_i, X_j) = \begin{cases} \varsigma & \text{if } |i-j| = 1 \\ 0 & \text{if } |i-j| > 1 \end{cases} \tag{2}$$

Given this model of the correlation between batches, the variance of the sum of the X_i's may be expressed as:

$$Var(\sum_{i=1}^{n_b} X_i) = \sum_{i=1}^{n_b} Var(X_i) + 2 \sum_{i=1}^{n_b-1} \sum_{j=i+1}^{n_b} Cov(X_i, X_j) \tag{3}$$

$$= n_b \sigma^2 + 2(n_b - 1)\varsigma \tag{4}$$

Thus, the variance of the mean throughput estimate $\overline{X}$ is obtained by dividing this result by $n_b{}^2$:

$$Var(\overline{X}) = \frac{\sigma^2}{n_b} + \frac{2(n_b-1)\varsigma}{n_b^{\,2}} \tag{5}$$

In order to compute a confidence interval for the throughput estimate $\overline{X}$, $Var(\overline{X})$ must be estimated. This, in turn, requires that estimates be obtained for σ^2 and ς. The estimate often used for σ^2 in computing confidence intervals is s^2, the sample variance of the throughput observations:

$$s^2 = \frac{1}{n_b-1}\sum_{i=1}^{n_b}(X_i-\overline{X})^2 \tag{6}$$

If the X_i's are correlated, the usual sample variance is not a good estimate for σ^2. Instead, σ^2 may be estimated by taking advantage of the fact that only adjacent batches are correlated. The sample variance of the throughputs from the even batches provides an unbiased estimate of σ^2, as does the sample variance of the throughputs from the odd batches, so these two sample variances are computed and averaged in order to obtain a better unbiased estimate of σ^2. (This improved estimate of σ^2 is denoted s_ς^2 to indicate that it has been introduced because of the covariance ς between adjacent batches, and it is assumed that n_b, the number of batches, is chosen to be even.) To estimate σ^2, then:

$$\overline{X}_{odd} = \frac{1}{(n_b/2)}\sum_{i\ odd} X_i \tag{7}$$

$$\overline{X}_{even} = \frac{1}{(n_b/2)}\sum_{i\ even} X_i \tag{8}$$

$$s_{odd}^2 = \frac{1}{(n_b/2)-1}\sum_{i\ odd}(X_i-\overline{X}_{odd})^2 \tag{9}$$

$$s_{even}^2 = \frac{1}{(n_b/2)-1} \sum_{i\ even} (X_i - \bar{X}_{even})^2 \tag{10}$$

$$s_\varsigma^2 = \frac{s_{even}^2 + s_{odd}^2}{2} \tag{11}$$

The covariance term in equation (5) may then be estimated once s_ς^2 has been computed. This estimate of the covariance ς, denoted c, may be computed as follows. First, an unbiased estimate κ of the quantity $2(\sigma^2 - \varsigma)$ is:

$$\kappa = \frac{1}{n_b-1} \sum_{i=1}^{n_b-1} (X_{i+1} - X_i)^2 \tag{12}$$

Given κ, an estimate of ς is:

$$c = s_\varsigma^2 - \frac{\kappa}{2} \tag{13}$$

Finally, then, the variance estimate $s_{\bar{x}}^2$ for the overall throughput $\bar{X}$ may itself be computed by substituting the variance and covariance estimates of equations (11) and (13) into equation (5):

$$s_{\bar{x}}^2 = \frac{s_\varsigma^2}{n_b} + \frac{2(n_b-1)c}{{n_b}^2} \tag{14}$$

Confidence Intervals

Given an overall throughput estimate $\bar{X}$ and an estimate $s_{\bar{x}}^2$ of its variance, confidence intervals can be computed in a fairly simple manner. The confidence interval computation is performed as though the X_i's were independent, as is typically assumed, with the improved variance estimate $s_{\bar{x}}^2$ used in place of the usual estimate of s^2/n_b. Thus, the 100(1–α)% confidence interval for the mean throughput μ is computed as:

$$\bar{X} \pm \delta \tag{15}$$

where:

$$\delta = s_{\bar{x}} t_{\alpha/2;n_b-1}, \quad s_{\bar{x}} = \sqrt{s_{\bar{x}}^2} \tag{16}$$

The $t_{\alpha/2;n_b-1}$ term is chosen from the Student-t distribution with n_b-1 degrees of freedom. That is, if Y has this distribution, then:

$$Prob(Y > t_{\alpha/2;n_b-1}) = \frac{\alpha}{2} \tag{17}$$

In order to obtain the most reasonable confidence interval estimates, the computational procedure used in this thesis differs slightly from what has been described thus far. First, the confidence interval estimates obtained using the preceding method will be slightly narrow because the correlation between adjacent X_i's reduces the "effective" number of degrees of freedom [Wolf83]. The actual confidence interval computations in this thesis are therefore performed assuming only $n_b/2$ degrees of freedom, a heuristic intended to make the confidence intervals obtained using the methods described here even more realistic.

Second, since the estimator c of the covariance ς is itself just a random variable, actual experimental data may occasionally yield a negative covariance estimate. Since correlations tend to be positive in this type of study, $c \leq 0$ is taken to indicate that the actual covariance ς is itself negligible. When such values are obtained, the associated confidence interval estimate is computed by reverting to standard methods, using s^2/n_b to estimate $Var(\bar{X})$

and using n_b-1 for the number of degrees of freedom.

Finally, simulations are actually run for a total of n_b+1 batches worth of simulation time. The purpose of the extra batch is to allow the results of the first batch (batch 0) to be discarded in order to eliminate any transient effects which might result from starting the simulation in an unrealistic state [Conw63]. (The startup state used in implementing the simulations is where all terminals are in the "stagger delay" state, a state which indeed proved to be visited rarely following the start of the simulations.) The mean throughputs and associated confidence intervals are actually computed using the results of batches 1 through n_b, and all other data (restart counts, etc.) is also obtained from these batches only.

Experimental Results

This section presents confidence interval results obtained by applying the methods described in this appendix to the simulation results obtained from the experiments of Chapters 3 and 4. As mentioned briefly in the chapters, all of the experiments of this thesis were run with control parameter settings of *num_batches* = 20 and *batch_time* = 50,000. These settings were selected based on confidence interval results obtained from preliminary experiments.

Table A3.1 contains actual throughput observations which were obtained from the simulation studies of 2PL in experiment 3.4 of Chapter 3. Table A3.2 gives the various estimator and confidence interval values which result

Sample Throughput Observations					
Batch Number	10000 Grans	1000 Grans	100 Grans	10 Grans	1 Gran
0	2.420	2.440	2.140	1.000	0.440
1	3.140	3.140	2.640	0.920	0.220
2	2.780	2.560	2.640	0.600	0.120
3	2.820	2.120	1.860	0.580	0.140
4	2.780	2.960	2.500	1.300	0.080
5	2.780	2.460	2.180	1.080	0.100
6	2.660	2.580	1.860	1.120	0.060
7	3.320	3.100	2.000	0.840	0.080
8	2.680	2.460	2.820	0.960	0.120
9	2.680	2.860	2.600	0.980	0.080
10	2.740	2.680	2.420	0.880	0.120
11	2.640	2.100	2.280	0.980	0.060
12	3.100	3.260	2.500	0.860	0.100
13	2.620	2.840	1.860	0.600	0.100
14	3.420	3.120	2.600	0.860	0.080
15	2.960	2.900	2.680	0.860	0.100
16	3.040	2.940	2.500	0.560	0.100
17	2.360	2.380	2.360	0.540	0.060
18	2.320	2.600	2.280	0.600	0.080
19	2.380	1.920	2.520	0.880	0.140
20	2.840	2.520	1.940	1.060	0.080

Table A3.1: Chapter 3, experiment 3.4, 2PL observations.

when the statistical methods detailed in this appendix are applied to this data. The statistics in Table A3.2 were produced with $n_b = 20$, as mentioned above. The observations used in the computation are those from batches 1 through 20 in Table A3.1. The confidence intervals given in Table A3.2 are for $\overline{X}$, the mean throughput, and are given as a percentage of $\overline{X}$. These data samples and results serve to illustrate the utility of the statistical approach used here.

It is evident from Table A3.1 that discarding batch 0 actually does help to eliminate transient throughput observations. Table A3.2 shows that the sample variance s^2 is generally smaller than the improved variance estimate s_ζ^2, an expected result since the sample variance tends to underestimate the actual variance when the data is positively correlated. In the cases where the number of granules (*Grans*) is 10 and 10,000 in the table, the value of the covariance estimate c is as large as 15-30% of the size of s_ζ^2. This indicates that it is indeed worthwhile using these methods to reduce the error that would occur if the correlation were ignored. In one instance c is seen to be negative, and in this case the standard variance estimate s^2 is used in the confidence interval computation.

The remaining tables in this appendix document the 90% confidence interval estimates associated with the throughput results given for the experiments of Chapters 3 and 4. Again, each confidence interval is expressed as a percentage of its corresponding throughput estimate. Confidence interval esti-

Sample Confidence Interval Computations						
Grans	$\bar{X}$	s^2	s_ζ^2	c	$s_{\bar{x}}^2$	C.I.
1	0.101	0.001	0.001	0.000	0.000	± 18.85%
10	0.853	0.046	0.047	0.017	0.004	± 13.45%
100	2.352	0.093	0.095	0.004	0.005	± 5.51%
1000	2.675	0.139	0.137	- 0.009	0.007	± 5.39%
10000	2.803	0.087	0.091	0.012	0.006	± 4.88%

Table A3.2: Chapter 3, experiment 3.4, 2PL results.

mates for experiment 4.3 of Chapter 3 are omitted, as they were reported in the chapter. These statistical results are included for the sake of completeness.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	8.252 ± 1.39%	8.063 ± 1.27%	11.215 ± 0.40%	11.127 ± 0.39%	7.790 ± 0.97%	7.655 ± 0.75%
10	10.971 ± 0.56%	11.004 ± 0.61%	11.420 ± 0.44%	11.421 ± 0.44%	10.648 ± 0.72%	10.314 ± 0.58%
100	11.373 ± 0.44%	11.373 ± 0.46%	11.419 ± 0.46%	11.420 ± 0.46%	11.328 ± 0.44%	11.262 ± 0.43%
1000	11.413 ± 0.43%	11.413 ± 0.43%	11.420 ± 0.45%	11.420 ± 0.45%	11.405 ± 0.42%	11.402 ± 0.43%
10000	11.419 ± 0.44%	11.419 ± 0.44%	11.420 ± 0.45%	11.420 ± 0.45%	11.418 ± 0.44%	11.416 ± 0.42%

Table A3.3: Chapter 3, experiment 1.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	3.400 ± 2.31%	3.638 ± 2.24%	6.479 ± 0.91%	6.241 ± 0.82%	2.595 ± 2.32%	3.634 ± 1.47%
10	5.974 ± 1.21%	5.790 ± 1.16%	7.096 ± 0.64%	7.161 ± 0.67%	5.119 ± 1.78%	5.231 ± 1.38%
100	7.039 ± 0.82%	6.966 ± 0.90%	7.161 ± 0.68%	7.163 ± 0.68%	6.906 ± 0.86%	6.714 ± 0.69%
1000	7.152 ± 0.64%	7.149 ± 0.62%	7.161 ± 0.67%	7.161 ± 0.68%	7.138 ± 0.66%	7.113 ± 0.66%
10000	7.159 ± 0.72%	7.159 ± 0.70%	7.160 ± 0.69%	7.161 ± 0.68%	7.158 ± 0.66%	7.158 ± 0.66%

Table A3.4: Chapter 3, experiment 1.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.748 ± 2.56%	0.788 ± 3.51%	2.917 ± 1.40%	2.741 ± 0.73%	0.169 ± 11.06%	0.889 ± 1.75%
10	0.946 ± 8.10%	1.065 ± 3.45%	2.097 ± 1.92%	3.028 ± 0.75%	0.406 ± 11.19%	1.200 ± 3.09%
100	2.823 ± 1.53%	2.633 ± 1.65%	3.335 ± 0.79%	3.360 ± 0.74%	2.231 ± 3.57%	2.408 ± 1.14%
1000	3.320 ± 0.81%	3.293 ± 0.80%	3.359 ± 0.57%	3.361 ± 0.82%	3.248 ± 1.36%	3.205 ± 0.84%
10000	3.357 ± 0.64%	3.351 ± 0.67%	3.362 ± 0.68%	3.361 ± 0.88%	3.355 ± 0.68%	3.347 ± 0.77%

Table A3.5: Chapter 3, experiment 1.3.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.281 ± 1.08%	0.240 ± 2.86%	1.518 ± 1.60%	1.425 ± 0.68%	0.001 ± 172.90%	0.336 ± 1.20%
10	0.074 ± 34.59%	0.234 ± 6.05%	0.432 ± 6.61%	1.415 ± 0.68%	0.004 ± 79.33%	0.355 ± 2.22%
100	0.827 ± 3.89%	0.701 ± 5.73%	1.414 ± 1.57%	1.759 ± 0.51%	0.235 ± 24.09%	0.784 ± 2.71%
1000	1.676 ± 1.17%	1.599 ± 1.70%	1.784 ± 0.79%	1.790 ± 0.80%	1.473 ± 3.88%	1.480 ± 1.43%
10000	1.776 ± 0.79%	1.770 ± 0.81%	1.788 ± 0.53%	1.788 ± 1.05%	1.763 ± 0.66%	1.749 ± 0.65%

Table A3.6: Chapter 3, experiment 1.4.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.179 ± 0.97%	0.139 ± 4.59%	1.031 ± 1.47%	0.964 ± 0.49%	0.000 ± 0.00%	0.205 ± 1.68%
10	0.009 ± 72.12%	0.121 ± 5.12%	0.143 ± 18.41%	0.959 ± 0.72%	0.000 ± 0.00%	0.207 ± 1.83%
100	0.203 ± 17.41%	0.237 ± 6.55%	0.583 ± 6.54%	1.111 ± 1.39%	0.009 ± 169.26%	0.371 ± 4.84%
1000	1.023 ± 1.74%	0.933 ± 4.46%	1.207 ± 1.09%	1.216 ± 1.10%	0.642 ± 13.61%	0.861 ± 2.17%
10000	1.193 ± 0.92%	1.186 ± 1.00%	1.217 ± 0.73%	1.218 ± 1.77%	1.148 ± 2.60%	1.159 ± 1.60%

Table A3.7: Chapter 3, experiment 1.5.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.089 ± 4.43%	0.071 ± 6.59%	0.518 ± 1.60%	0.487 ± 0.78%	0.000 ± 0.00%	0.096 ± 3.31%
10	0.002 ± 172.90%	0.062 ± 7.99%	0.020 ± 56.18%	0.485 ± 0.94%	0.000 ± 0.00%	0.096 ± 3.31%
100	0.006 ± 84.82%	0.054 ± 9.41%	0.029 ± 47.64%	0.481 ± 0.69%	0.000 ± 0.00%	0.102 ± 3.39%
1000	0.266 ± 8.14%	0.201 ± 12.92%	0.483 ± 4.05%	0.592 ± 1.30%	0.025 ± 136.42%	0.267 ± 3.79%
10000	0.571 ± 2.62%	0.558 ± 3.34%	0.617 ± 1.42%	0.614 ± 2.71%	0.431 ± 11.21%	0.524 ± 3.79%

Table A3.8: Chapter 3, experiment 1.6.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.062 ± 12.07%	0.068 ± 8.57%	0.759 ± 3.85%	0.609 ± 5.35%	0.000 ± 0.00%	0.098 ± 10.66%
10	0.001 ± 172.90%	0.060 ± 12.54%	0.035 ± 98.00%	0.608 ± 5.11%	0.000 ± 0.00%	0.099 ± 14.88%
100	0.024 ± 73.46%	0.103 ± 28.07%	0.049 ± 91.92%	0.612 ± 4.75%	0.015 ± 144.47%	0.142 ± 15.27%
1000	0.537 ± 12.23%	0.456 ± 13.91%	0.709 ± 5.41%	0.738 ± 4.36%	0.138 ± 30.52%	0.393 ± 7.87%
10000	0.780 ± 3.54%	0.759 ± 4.07%	0.783 ± 3.70%	0.787 ± 4.13%	0.677 ± 6.39%	0.675 ± 6.70%

Table A3.9: Chapter 3, experiment 2.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.062 ± 11.37%	0.071 ± 13.90%	0.770 ± 5.35%	0.646 ± 4.80%	0.000 ± 0.00%	0.097 ± 10.15%
10	0.418 ± 9.48%	0.414 ± 11.71%	0.797 ± 5.44%	0.801 ± 4.48%	0.052 ± 88.93%	0.408 ± 7.76%
100	0.712 ± 5.98%	0.718 ± 5.58%	0.799 ± 5.06%	0.799 ± 5.23%	0.443 ± 18.13%	0.685 ± 6.17%
1000	0.770 ± 4.36%	0.770 ± 4.35%	0.798 ± 5.27%	0.799 ± 5.45%	0.700 ± 7.04%	0.746 ± 5.65%
10000	0.775 ± 4.44%	0.775 ± 4.95%	0.798 ± 5.24%	0.799 ± 5.36%	0.728 ± 5.32%	0.754 ± 4.13%

Table A3.10: Chapter 3, experiment 2.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.066 ± 8.58%	0.091 ± 16.82%	0.919 ± 4.09%	0.771 ± 5.21%	0.000 ± 0.00%	0.111 ± 12.48%
10	0.464 ± 11.78%	0.517 ± 10.35%	0.967 ± 3.15%	0.963 ± 3.78%	0.124 ± 24.66%	0.450 ± 8.81%
100	0.883 ± 5.65%	0.894 ± 6.44%	0.966 ± 3.35%	0.964 ± 3.29%	0.691 ± 9.93%	0.858 ± 6.89%
1000	0.930 ± 4.52%	0.945 ± 5.40%	0.966 ± 2.92%	0.969 ± 3.02%	0.775 ± 11.31%	0.905 ± 4.79%
10000	0.942 ± 5.46%	0.944 ± 5.49%	0.966 ± 3.13%	0.967 ± 3.01%	0.874 ± 6.98%	0.913 ± 4.37%

Table A3.11: Chapter 3, experiment 3.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.069 ± 11.91%	0.109 ± 12.89%	1.126 ± 4.93%	0.992 ± 6.00%	0.000 ± 0.00%	0.128 ± 14.85%
10	0.535 ± 5.43%	0.633 ± 9.82%	1.227 ± 6.48%	1.226 ± 5.71%	0.073 ± 80.88%	0.522 ± 6.66%
100	1.055 ± 7.18%	1.112 ± 6.02%	1.228 ± 5.75%	1.226 ± 5.58%	0.862 ± 8.14%	1.022 ± 4.75%
1000	1.183 ± 4.69%	1.183 ± 4.72%	1.229 ± 5.75%	1.228 ± 5.90%	1.081 ± 10.96%	1.112 ± 5.45%
10000	1.200 ± 4.96%	1.184 ± 5.15%	1.229 ± 5.67%	1.228 ± 5.26%	1.113 ± 9.61%	1.130 ± 5.60%

Table A3.12: Chapter 3, experiment 3.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.089 ± 16.92%	0.203 ± 20.25%	1.535 ± 4.25%	1.344 ± 5.65%	0.002 ± 249.77%	0.202 ± 13.31%
10	0.639 ± 12.30%	0.857 ± 7.33%	1.709 ± 4.85%	1.709 ± 5.04%	0.159 ± 59.52%	0.710 ± 8.56%
100	1.462 ± 5.52%	1.556 ± 4.64%	1.713 ± 5.18%	1.717 ± 4.82%	0.410 ± 62.60%	1.352 ± 5.37%
1000	1.646 ± 6.12%	1.669 ± 5.20%	1.714 ± 5.12%	1.715 ± 5.13%	1.463 ± 6.47%	1.571 ± 4.56%
10000	1.688 ± 5.12%	1.679 ± 5.21%	1.713 ± 5.09%	1.715 ± 5.10%	1.576 ± 5.55%	1.578 ± 4.56%

Table A3.13: Chapter 3, experiment 3.3.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.101 ± 18.85%	0.450 ± 22.02%	2.521 ± 4.25%	2.371 ± 5.01%	0.022 ± 221.16%	0.333 ± 11.90%
10	0.853 ± 13.45%	1.551 ± 9.98%	2.830 ± 4.20%	2.860 ± 3.30%	0.338 ± 27.01%	0.963 ± 13.36%
100	2.352 ± 5.51%	2.580 ± 5.81%	2.865 ± 3.78%	2.861 ± 3.20%	1.246 ± 25.21%	2.185 ± 7.44%
1000	2.675 ± 5.39%	2.752 ± 4.99%	2.859 ± 3.86%	2.864 ± 3.92%	2.415 ± 6.27%	2.504 ± 5.39%
10000	2.803 ± 4.88%	2.777 ± 4.25%	2.860 ± 3.84%	2.864 ± 4.02%	2.634 ± 5.33%	2.554 ± 5.22%

Table A3.14: Chapter 3, experiment 3.4.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.134 ± 7.96%	0.183 ± 13.45%	0.915 ± 4.31%	0.771 ± 5.19%	0.002 ± 176.61%	0.219 ± 9.34%
10	0.681 ± 6.78%	0.749 ± 7.00%	0.965 ± 3.62%	0.967 ± 3.97%	0.326 ± 25.71%	0.660 ± 5.88%
100	0.921 ± 4.98%	0.921 ± 4.41%	0.964 ± 3.88%	0.964 ± 3.89%	0.869 ± 7.67%	0.908 ± 6.50%
1000	0.953 ± 4.51%	0.953 ± 4.52%	0.964 ± 3.79%	0.965 ± 3.68%	0.914 ± 6.66%	0.939 ± 5.09%
10000	0.956 ± 4.42%	0.955 ± 4.57%	0.964 ± 3.76%	0.964 ± 3.82%	0.930 ± 6.25%	0.942 ± 5.18%

Table A3.15: Chapter 3, experiment 4.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.034 ± 14.94%	0.043 ± 19.59%	0.919 ± 4.20%	0.771 ± 5.16%	0.005 ± 227.19%	0.059 ± 18.76%
10	0.174 ± 45.20%	0.303 ± 9.70%	0.964 ± 3.39%	0.965 ± 4.37%	0.064 ± 62.01%	0.296 ± 6.86%
100	0.798 ± 6.44%	0.812 ± 5.48%	0.963 ± 4.37%	0.963 ± 4.14%	0.434 ± 10.89%	0.735 ± 6.99%
1000	0.899 ± 4.75%	0.900 ± 4.88%	0.961 ± 3.60%	0.965 ± 3.95%	0.738 ± 6.49%	0.847 ± 3.98%
10000	0.919 ± 3.79%	0.912 ± 4.57%	0.961 ± 3.84%	0.964 ± 3.98%	0.800 ± 6.87%	0.863 ± 3.33%

Table A3.16: Chapter 3, experiment 4.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.240 ± 4.90%	0.393 ± 9.79%	3.094 ± 2.67%	2.294 ± 2.95%	0.004 ± 172.90%	0.406 ± 4.41%
10	1.676 ± 7.34%	1.909 ± 6.28%	3.385 ± 2.45%	3.369 ± 2.42%	0.373 ± 33.37%	1.587 ± 4.65%
100	3.024 ± 3.09%	3.098 ± 2.89%	3.391 ± 2.40%	3.389 ± 2.29%	2.073 ± 10.67%	2.906 ± 2.71%
1000	3.211 ± 3.15%	3.215 ± 3.00%	3.370 ± 2.18%	3.370 ± 2.28%	2.928 ± 3.23%	3.084 ± 3.76%
10000	3.028 ± 2.83%	3.028 ± 2.94%	3.140 ± 2.77%	3.141 ± 2.79%	2.823 ± 3.82%	2.913 ± 3.16%

Table A3.17: Chapter 3, experiment 5.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.237 ± 5.22%	0.398 ± 8.17%	2.799 ± 3.19%	1.742 ± 3.55%	0.000 ± 0.00%	0.398 ± 5.27%
10	1.594 ± 9.08%	1.851 ± 5.28%	3.261 ± 2.41%	3.215 ± 2.92%	0.254 ± 60.83%	1.557 ± 5.67%
100	2.951 ± 2.60%	3.015 ± 3.31%	3.306 ± 2.38%	3.284 ± 2.48%	2.196 ± 10.43%	2.856 ± 3.33%
1000	3.142 ± 2.99%	3.144 ± 3.19%	3.272 ± 2.53%	3.270 ± 2.49%	2.958 ± 2.93%	3.049 ± 3.21%
10000	2.962 ± 2.41%	2.965 ± 2.59%	3.073 ± 2.82%	3.071 ± 2.78%	2.866 ± 3.40%	2.896 ± 3.20%

Table A3.18: Chapter 3, experiment 5.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.066	0.088	0.919	0.771	0.000	0.111
1	± 8.58%	± 16.26%	± 4.17%	± 5.15%	± 0.00%	± 15.07%
10	0.473	0.515	0.967	0.963	0.121	0.460
10	± 10.07%	± 10.30%	± 3.15%	± 3.78%	± 27.68%	± 9.43%
100	0.883	0.893	0.967	0.964	0.698	0.863
100	± 5.91%	± 6.96%	± 3.25%	± 3.28%	± 9.09%	± 5.87%
1000	0.931	0.945	0.966	0.969	0.834	0.905
1000	± 4.66%	± 5.58%	± 3.13%	± 2.88%	± 9.27%	± 4.74%
10000	0.942	0.944	0.966	0.967	0.874	0.912
10000	± 5.46%	± 5.60%	± 3.02%	± 3.03%	± 6.70%	± 4.78%

Table A3.19: Chapter 3, experiment 6.1.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.066	0.088	0.917	0.769	0.000	0.111
1	± 8.58%	± 16.80%	± 4.13%	± 4.70%	± 0.00%	± 14.67%
10	0.433	0.521	0.967	0.963	0.115	0.455
10	± 8.69%	± 9.26%	± 3.29%	± 3.77%	± 35.11%	± 9.79%
100	0.883	0.892	0.967	0.964	0.603	0.862
100	± 5.84%	± 6.54%	± 3.41%	± 3.21%	± 20.14%	± 6.13%
1000	0.932	0.943	0.966	0.969	0.834	0.905
1000	± 4.39%	± 5.36%	± 3.03%	± 3.00%	± 9.26%	± 4.86%
10000	0.942	0.944	0.966	0.959	0.877	0.905
10000	± 5.41%	± 5.37%	± 3.20%	± 4.26%	± 6.22%	± 5.04%

Table A3.20: Chapter 3, experiment 6.2.

Throughput Confidence Intervals						
Grans	TPL	WD	TPLW	PRE	BTO	SV
1	0.063	0.076	0.894	0.754	0.000	0.108
1	± 7.21%	± 19.19%	± 3.68%	± 4.66%	± 0.00%	± 14.10%
10	0.416	0.447	0.930	0.928	0.038	0.432
10	± 12.27%	± 12.16%	± 3.98%	± 4.44%	± 96.05%	± 10.51%
100	0.810	0.813	0.924	0.924	0.631	0.786
100	± 5.96%	± 5.27%	± 3.80%	± 4.19%	± 10.97%	± 6.46%
1000	0.787	0.794	0.870	0.872	0.662	0.766
1000	± 5.89%	± 5.01%	± 3.83%	± 3.99%	± 11.10%	± 4.68%
10000	0.475	0.475	0.511	0.510	0.460	0.460
10000	± 4.92%	± 4.95%	± 5.04%	± 5.46%	± 5.61%	± 6.75%

Table A3.21: Chapter 3, experiment 6.3.

Throughput Confidence Intervals						
Grans	BTO	MVTO	TPL	VP	SV	MVSV
1	1.707	1.707	1.837	2.228	0.407	2.364
	± 5.04%	± 5.04%	± 4.06%	± 2.69%	± 11.60%	± 2.61%
10	2.632	2.844	2.831	2.939	1.183	2.863
	± 4.53%	± 3.42%	± 3.92%	± 4.13%	± 8.36%	± 2.93%
100	2.930	2.998	3.009	3.011	2.397	2.999
	± 4.33%	± 4.39%	± 4.15%	± 3.92%	± 6.28%	± 4.46%
1000	2.918	3.012	3.012	3.013	2.691	3.012
	± 3.82%	± 4.05%	± 4.45%	± 4.41%	± 5.01%	± 4.31%
10000	2.926	3.013	3.013	3.013	2.755	3.013
	± 3.79%	± 4.33%	± 4.32%	± 4.44%	± 4.55%	± 4.36%

Table A3.22: Chapter 4, experiment 1, multiple versions.

Throughput Confidence Intervals						
Size	BTO	MVTO	TPL	VP	SV	MVSV
1	7.613 ± 0.37%	7.613 ± 0.39%	7.716 ± 0.34%	7.717 ± 0.37%	7.386 ± 0.60%	7.669 ± 0.42%
2	6.545 ± 0.55%	6.573 ± 0.59%	6.641 ± 0.55%	6.641 ± 0.60%	6.110 ± 1.41%	6.610 ± 0.60%
5	4.435 ± 1.52%	4.649 ± 1.30%	4.668 ± 1.30%	4.675 ± 1.16%	3.722 ± 1.85%	4.660 ± 1.20%
10	2.725 ± 3.65%	3.174 ± 2.49%	3.157 ± 2.52%	3.183 ± 2.45%	1.957 ± 4.30%	3.177 ± 2.57%
15	1.903 ± 3.94%	2.462 ± 2.37%	2.452 ± 2.50%	2.468 ± 2.50%	1.271 ± 6.06%	2.464 ± 2.35%
30	0.812 ± 7.04%	1.336 ± 4.11%	1.282 ± 5.35%	1.336 ± 4.24%	0.483 ± 10.71%	1.336 ± 4.06%

Table A3.23: Chapter 4, experiment 2, multiple versions.

Throughput Confidence Intervals						
Pr(Sm)	BTO	MVTO	TPL	VP	SV	MVSV
0.0	0.878 ± 4.73%	0.878 ± 4.73%	0.878 ± 4.73%	0.878 ± 4.67%	0.878 ± 4.67%	0.878 ± 4.67%
0.2	0.815 ± 5.02%	1.043 ± 4.36%	1.021 ± 4.09%	1.043 ± 4.33%	0.540 ± 9.88%	1.043 ± 4.33%
0.4	0.812 ± 7.04%	1.336 ± 4.11%	1.282 ± 5.35%	1.336 ± 4.24%	0.483 ± 10.71%	1.336 ± 4.06%
0.6	0.981 ± 8.22%	1.868 ± 4.62%	1.739 ± 5.03%	1.872 ± 4.77%	0.526 ± 11.15%	1.867 ± 4.81%
0.8	1.130 ± 11.38%	2.947 ± 4.92%	2.606 ± 4.30%	2.956 ± 4.75%	0.546 ± 13.05%	2.943 ± 4.90%
1.0	6.842 ± 0.43%	6.842 ± 0.43%	7.013 ± 0.42%	7.010 ± 0.36%	6.691 ± 0.56%	6.790 ± 0.59%

Table A3.24: Chapter 4, experiment 3, multiple versions.

Throughput Confidence Intervals				
Size	PRE	H-PRE	SV	H-SV
1	7.444 ± 0.53%	7.445 ± 0.54%	7.435 ± 0.63%	7.434 ± 0.64%
2	6.346 ± 0.59%	6.346 ± 0.63%	6.337 ± 0.74%	6.327 ± 0.72%
5	4.414 ± 1.45%	4.414 ± 1.46%	4.386 ± 1.31%	4.368 ± 1.24%
10	2.930 ± 1.85%	2.930 ± 1.79%	2.888 ± 2.14%	2.873 ± 2.75%
15	2.281 ± 2.10%	2.281 ± 2.12%	2.215 ± 3.09%	2.189 ± 3.35%
30	1.228 ± 5.26%	1.228 ± 5.91%	1.130 ± 5.60%	1.112 ± 5.80%

Table A3.25a: Chapter 4, experiment 1, hierarchies.

Throughput Confidence Intervals				
Size	BTO	H-BTO	MVTO	H-MVTO
1	7.444 ± 0.56%	7.442 ± 0.58%	7.444 ± 0.56%	7.442 ± 0.58%
2	6.342 ± 0.60%	6.331 ± 0.65%	6.342 ± 0.60%	6.331 ± 0.65%
5	4.404 ± 1.35%	4.374 ± 1.34%	4.405 ± 1.45%	4.385 ± 1.38%
10	2.895 ± 2.67%	2.867 ± 3.24%	2.905 ± 2.30%	2.861 ± 3.27%
15	2.234 ± 2.90%	2.181 ± 3.27%	2.235 ± 2.84%	2.178 ± 4.16%
30	1.113 ± 9.61%	1.081 ± 10.96%	1.111 ± 12.16%	1.079 ± 11.72%

Table A3.25b: Chapter 4, experiment 1, hierarchies (cont.).

Throughput Confidence Intervals				
Size	PRE	H-PRE	SV	H-SV
1	4.662 ± 0.55%	3.395 ± 0.58%	4.292 ± 0.65%	3.010 ± 0.89%
2	3.869 ± 1.07%	3.314 ± 0.97%	3.577 ± 0.97%	2.942 ± 0.86%
5	2.534 ± 1.30%	2.928 ± 0.69%	2.353 ± 2.31%	2.601 ± 0.82%
10	1.608 ± 2.91%	2.204 ± 1.24%	1.505 ± 3.16%	1.956 ± 1.92%
15	1.245 ± 5.68%	1.781 ± 2.42%	1.134 ± 6.36%	1.578 ± 2.95%
30	0.662 ± 7.65%	1.049 ± 5.07%	0.571 ± 5.79%	0.884 ± 5.38%

Table A3.26a: Chapter 4, experiment 2, hierarchies.

Throughput Confidence Intervals				
Size	BTO	H-BTO	MVTO	H-MVTO
1	4.293 ± 0.72%	3.013 ± 0.69%	4.293 ± 0.72%	3.013 ± 0.69%
2	3.580 ± 0.81%	2.944 ± 0.88%	3.580 ± 0.81%	2.944 ± 0.88%
5	2.363 ± 2.10%	2.602 ± 1.05%	2.365 ± 2.04%	2.607 ± 0.80%
10	1.514 ± 3.31%	1.945 ± 2.55%	1.514 ± 3.36%	1.942 ± 2.82%
15	1.153 ± 5.12%	1.575 ± 3.64%	1.153 ± 5.09%	1.558 ± 4.25%
30	0.584 ± 6.70%	0.897 ± 5.41%	0.590 ± 7.20%	0.891 ± 6.04%

Table A3.26b: Chapter 4, experiment 2, hierarchies (cont.).