CONCURRENCY CONTROL
USING TRANSACTION CHARACTERISTICS

by

ADEBOLA OLATUNJI AKINSANYA

A thesis presented to the University
of Lagos in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

December 1989

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF LAGOS
LAGOS

SCHOOL OF POSTGRADUATE STUDIES
UNIVERSITY OF LAGOS

CERTIFICATION

THIS IS TO CERTIFY THAT THE THESIS -

CONCURRENCY CONTROL

USING TRANSACTION CHARACTERISTICS

SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES

UNIVERSITY OF LAGOS FOR THE AWARD OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

IS A RECORD OF ORIGINAL RESEARCH CARRIED OUT BY

ADEBOLA OLATUNJI AKINSANYA

IN THE DEPARTMENT OF

COMPUTER SCIENCES

AKINSANYA, A.O.
AUTHOR'S NAME — SIGNATURE — 21/4/90
DATE

Prof O. Asass — 21/4/90
Dr. H.O.D. Longe
SUPERVISOR'S NAME — SIGNATURE — 21/4/90
DATE

Dr. H.O.D. Longe
INTERNAL EXAMINER'S NAME — SIGNATURE — 21/4/90
DATE

Dr. A. Akinbode
INTERNAL EXAMINER'S NAME — SIGNATURE — 21/4/90
DATE

PROF. S.B. JAIYESIMI
EXTERNAL EXAMINER'S NAME — SIGNATURE — 21-4-90
DATE

ABSTRACT

In this dissertation a new concurrency control algorithm is proposed and compared for efficiency with two of the existing algorithms. An abstract model is used to evaluate the costs associated with the algorithms and performances of the algorithms are determined using simulation technique.

Simulation results show that none of the algorithms is uniformly better than the others. However, the newly proposed algorithm, the Integrated Transaction Characteristics, is generally better when concurrency control information is kept in primary memory.

ACKNOWLEGEMENTS

I wish to express my sincere gratitude to all lovers of knowledge who have offered their time, direction and encouragement to this work. While many nice people contributed to this effort, I must specially mention the following.

I am most grateful to my Supervisors, Professor Olayide Abass and Dr. H.O.D. Longe who provided assistance above and beyond the call of duty. In particular I must mention Professor Abass' helpful and detailed criticisms of the design of my simulation experiments and Dr. Longe's criticisms of the content and style of this work. They were simply wonderful.

I am quite grateful to Dr. J.O.A. Ayeni from whom I initially collected a lot of papers on concurrency control. His formal and informal discussions with me on this work had been very encouraging and fruitful.

I would also like to thank Dr. A.B. Sofoluwe and Dr. Lani Akinbode, the former and the current departmental postgraduate studies coordinator respectively, for their understanding and willingness to help.

I must also thank all other academic and non-academic staff of the Department of Computer Sciences for their assistance.

Finally, I wish to express a most special thanks to my family and close friends who insisted, despite all odds, that this work must be successfully completed.

DEDICATION

This work is dedicated to the memory of my grandmother, **NUSIRAT JAIYEOLA OYEDELE**, who died on 11th September, 1989 while I was working on this thesis.

TABLE OF CONTENTS

Page

ABSTRACT I

ACKNOWLEDGEMENTS II

DEDICATION IV

Chapter 1 INTRODUCTION 1

1.1 Database and transaction abstraction 1

1.2 Database consistency problems and coordination of transactions 3

1.2.1 Lost update problem 5

1.2.2 Data inconsistency problem 5

1.3 Serializable executions 6

1.4 Scope of work 8

1.5 Organisation of the Thesis 9

Chapter 2 LITERATURE REVIEW 11

2.1 Fundamental Mechanisms 11

2.1.1 Locking 13

2.1.2 Timestamping 16

2.1.3 Serial Validation 17

2.2 Transaction Class algorithm 19

2.3 Semantically Consistent Schedules algorithms 23

2.4 Performance of concurrency control algorithms 25

2.4.1 Locking Performance 27

2.4.2 Performance of other algorithms 31

Chapter 3 INTEGRATED TRANSACTION CHARACTERISTICS ALGORITHM AND A COST MODEL 33

3.1 Integrated Transaction Characteristics algorithm 34

3.1.1 Syntactic attributes 34

3.1.2 Semantic attributes 35

3.1.3 The synchronization mechanism 38

3.2 Concurrency Control Cost Model 41

3.2.1 Transaction Requests 41

3.2.2 The Concurrency Control Database 44

3.2.3 Algorithm cost comparison 46

3.2.4 Storage Cost 49

3.2.4.1 The Storage Cost of Transaction Classes Algorithm 50

3.2.4.2 The Storage Cost of Semantically Consistent Schedules Algorithm 51

3.2.4.3 The Storage Cost of Integrated Transaction Characteristics Algorithm 52

3.2.4.4 Relative Storage Cost of the Algorithms 52

3.2.5 CPU Cost 54

3.2.5.1 The CPU cost of Transaction Classes algorithm 55

3.2.5.2 The CPU cost of Semantically Consistent Schedules Algorithm 57

3.2.5.3 The CPU cost of Integrated Transaction Characteristics Algorithm 58

3.5.4 Relative CPU Cost of the Algorithms 58

Chapter 4 CONCURRENCY CONTROL PERFORMANCE 60

4.1 Background 60

4.2 Justification for a simulation approach 62

4.3 Model description 63

4.3.1 The Workload Model 63

4.3.2 The queueing Model 65

4.4 Pre-analysis of Transaction and data generation 73

4.5 Design of Experiment 76

4.5.1 Identification of the factors 77

4.5.2 Selection of level for each Simulation run 77

4.5.3 Concurrency Control and Searching costs 78

4.5.4 Simulation Output data analysis 81

4.6 Assumptions 85

4.7 Program Development 89

4.8 Experiments and Results 92

4.8.1 Multiprogramming level 93

4.8.1.1 Summary of results of the multiprogramming level experiments 108

4.8.2 Concurrency Control Cost 112

4.8.2.1 Summary of result of Concurrency Control experiments 126

4.8.3 Searching Costs 126

4.8.3.1 Summary of results of the searching cost experiments 139

Chapter 5 CONCLUSION 140

5.1 Summary of Results 140

5.2 Comparison with other work 145

5.3 Future Research directions 146

BIBLIOGRAPHY 148

APPENDIX 1 164

APPENDIX 2 175

CHAPTER 1

INTRODUCTION

1.1 Database and transaction abstraction

A database is a model of some part of the real world. It consists of some named resources called entities. For the database to be a reliable real-world model, the value of the entities must be, at all times, related in a way that satisfies certain integrity constraints.

However, it may be impossible to carry out an operation, consisting of a sequence of indivisible (atomic) steps, without temporarily violating the integrity constraints at some intermediate stage(s). For example, it may be impossible to transfer funds from one bank account to another without violating at some intermediate point the consistency constraint stating that the sum of all balances equals the total liability of the bank.

It is for this and some other reasons that indivisible steps are grouped into an abstraction called a transaction. Thus if a set of atomic steps are applied on a consistent database and the database remains consistent at the end of the operation, then the set of atomic steps is a unit of consistency. Such unit of consistency is a transaction.

If a transaction is run by itself on a consistent database, then it is guaranteed to produce a consistent database at the end. As a corollary, the same holds for any serial (one-at-a-time) execution of many transactions. However, in practice, transactions are run concurrently as many users, unaware of each other's presence, initiate their own transactions. This may result in inconsistencies despite the guaranteed correctness of individual transactions.

Therefore, in order to avoid inconsistencies and thus ensure database correctness when several transactions access (read and update) the same database concurrently, there must be some kind of coordination of the various actions to ensure that the resulting sequence of actions is correct, that is every user must receive a consistent view of the data and the actions must in turn leave the database in a consistent state at the end.

1.2. Database consistency problems and coordination of transactions

An important component of a shared database system is a transaction coordinator or scheduler. This portion of the system is responsible for dealing with the recovery and synchronization aspects of database management.

It coordinates transaction actions such that the integrity of the database is assured. An example will illustrate database consistency problems and the need for proper transaction coordination.

Consider a database of three bank accounts A,B,C, and the three sample banking transactions shown in Figure 1.1. The read and write operations in Figure 1.1 respectively represent reading values from the database into local variables and writing values from local variables into the database.

Transaction T1 transfers forty Naira from account A to account B, transaction T2 computes interest for account A, and transaction T3 deposits fifty naira into account C. If the three transactions are all allowed to run concurrently without proper coordination or synchronization, database consistency problems such as those highlighted below may arise.

```
Transaction T1
        Begin;
              read a-value from A;
              read b-value from B;
              a-value: = a value - 40;
              b-value: = b-value + 40;
              write a-value into A;
              write b-value into B;
        end;
Transaction T2
        Begin;
              read a-value from A;
              a-value: = a-value * 1.10;
              write a-value into A;
        end;
Transaction T3
        Begin;
```

```
            c-value: = c-value + 50;
            write c-value into c;
        end;
```

Figure 1.1: Transactions T1, T2, T3 Typical interdependent

1.2.1. Lost Update Problem

Consider the actions of transactions T1 and T2 below. The subtraction of forty naira by T1 from account A is lost when T2 writes the results of its interest computation. This is referred to as lost update.

```
T2: read a-value from A;
T2: a-value: = a-value * 1.10;
T1: read a-value from A;
T1: read b-value from B;
T1: a-value: = a-value-40;
T1: b-value: = b-value + 40;
T1: write a-value into A;
T1: write b-value into B;
T2: write a-value into A;
```

1.2.2 Data inconsistency Problem

Another problem which could occur when

transactions execute concurrently without controls is the reading of inconsistent data by some transaction(s). Take for instance the interleaving of transaction T1 and T4 below. Transaction T4 is reading while transaction T1 is writing resulting in accessing of inconsistent data by T4.

The inconsistency arises because T4 reads balances from both account A and B after T1 has written its new value for A but before writing its new value for B. T4 therefore sees forty naira missing from A but not yet added to B. T4 could not see such partial results in a correctly synchronized execution of T1 and T4.

```
T1: read a-value from A;
T1: read b-value from B;
T1: a-value: = a-value - 40;
T1: b-value: = b-value + 40;
T1: write a-value into A;
T4: read a-value from A;
T4: read b-value from B;
T1: write b-value into B;
```

1.3 Serializable executions

If the transactions T1, T2, T3 and T4 are executed serially, one after the other, the problems highlighted in section 1.2.1 will not occur. The results of executing these transactions serially, one after the other, are known as serial outcomes.

Obviously, different serial outcomes would result depending on the order of executing the transactions. But each of the outcomes preserves the integrity of the database.

But since in practice transactions are not run serially there is a need for an organised interleaving of the actions of the transactions such that the integrity of the database is still preserved. Such an organised interleaved actions of transaction which produce the same results as some serial outcome is called serializable execution [Eswa76, Yann84].

The problem of guaranteeing serializability for non-serial or concurrent transactions is known as the concurrency control problem. A number of solutions to this problem are known [Bern81].

Some of these solutions, called concurrency control algorithms, are discussed in the next chapter. Some algorithms which are not forcy about serializable executions but ensures preservation of database integrity constraints are also discussed.

1.4 Scope of Work

This study examines a special class of concurrency control algorithms. These algorithms are those that use characteristics of transactions to further improve the performance of basic concurrency control mechanisms. The algorithms are Transaction Classes [Bern80c], Semantically Consistent Schedules [Garc83] and Integrated Transaction Characteristics. Transaction Classes and Semantically Consistent Schedules use syntactic properties and semantic properties of transactions respectively.

We are proposing the Integrated Transaction Characteristics algorithm. It is based on both the syntactic information in Transaction Classes algorithm and semantic information in

Semantically Consistent schedule algorithm.

Using an abstract concurrency control model [Care83a] the storage and CPU costs of the algorithms are determined. Also given some set of conditions, the relative performance of the algorithms is evaluated. This is with the hope of identifying algorithm which performs uniformly better than others and the algorithms that are best under certain given conditions.

1.5 Organisation of the Thesis

The remaining part of the thesis is organised as follows.

Chapter 2 reviews concurrency control algorithms and their performance.

Chapter 3 describes Integrated Transaction Characteristics algorithm and uses an abstract concurrency control model to determine the storage and CPU costs of the transaction knowledge based algorithms. Chapter 4 evaluates relative performance of the transaction knowledge based algorithms using a workload model and a logical database queuing model.

Chapter 5 contains conclusions and recommendations for further research.

CHAPTER 2

LITERATURE REVIEW

2.1. Fundamental Mechanisms

Much research on algorithm construction has been done in the area of concurrency control for both single-site and distributed database systems. Some of these researches have focused on the theory involved in proving the correctness of concurrency control algorithms [Eswa76, Bern80d, Papa82]. Early work has led to the development of new concurrency control algorithms, most of which are either hybrids of or based on one of three mechanisms: locking [Mena78, Gray79], timestamps [Thom78, Bern80a] and serial validation [Kung81, Mena82].

Most of the algorithms deal mainly with two operations READ and WRITE. Some of the algorithms make use of more information about transactions or recognize additional operations [Bern80c] while some allow non-serializable but integrity preserving executions [Garc83]. The order in which data items are accessed, the particular set of items which the transactions might access, and the manner in which WRITES are computed from READS are examples of additional information that could be supplied to a transaction processing mechanism. Examples of additional operations which might be recognised are INSERT operation to create a new data item and a DELETE operation to eliminate a data item.

When only READ and WRITE operations are allowed, serializability theory is concerned with two types of dependencies which can arise between transactions. They are the READ/WRITE and WRITE/WRITE dependencies. Consider the execution of two transactions Ti and Tj. A READ/WRITE dependency holds from transaction Ti to transaction Tj if either Ti reads some data item which is later written by Tj or Ti writes some data item which is later read by Tj. A WRITE/WRITE dependency holds from Ti to Tj if Ti writes some data item which is later written by Tj. The existence of a READ/WRITE or WRITE/WRITE dependency from Ti to Tj implies that Ti must precede Tj in any serial execution of transactions which produces the same result as their concurrent execution. One can construct a graph, called a DEPENDENCY or CONFLICT 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 (or conflict) graph is acyclic, that is if the graph contains no cycles [Papa77,Papa79, Stea76]. For concurrency control algorithms based on other types of operations, other types of dependencies are defined between operations to provide the basis for serializability [Bern81].

2.1.1. Locking

Most solutions to concurrency control problem are based on some explicit or implicit locking scheme [Eswa76]. A transaction may lock objects to ensure their inaccessibility while in a temporarily inconsistent state. Locks can be set either statically (at transaction startup time) or dynamically as READS and WRITES are performed.

In the simplest case each object has a unique lock which is held by at most one transaction at a time. If a transaction attempts to lock an object that is already locked it must either wait, abort itself, or preempt the other transaction.

Typical of locking algorithms is two-phase locking (2PL). It requires all transactions to be well-formed and two-phased to guarantee consistency [Eswa76].

A transaction is well-formed if it

(i) locks an object before accessing it

(ii) does not lock an object which is already locked

and (iii) unlocks each object it has locked before it completes.

It is two-phased if no object is unlocked before all objects are locked.

Formally, suppose a transaction $T =((t, a_i, e_i) \; i = 1, n)$

where t is the transaction name

a_i is the operation at step i.

e_i is the object accessed at step i.

Then

(1) T is <u>well-formed</u> if for each step

$i=1, \ldots\ldots\ldots\ldots, n$

if a_i = lock then e_i is not locked by

T through step $i - 1$

if a_i # lock then e_i is locked by T

through step i

and at step n, only e_n is still

locked by T and a_n = unlock;

and

(2) T is <u>two-phased</u> if for some $j < n$

$i < j$ implies a_i # unlock

$i = j$ implies a_i = unlock

$i > j$ implies a_i # lock;

steps $1, \ldots\ldots, j-1$ are

called the <u>growing phase</u>

and steps j-1, ..., n the shrinking phase.

2.1.2 Timestamping

A timestamp is a unique number assigned to a transaction or object and is chosen from a monotonically increasing sequence which is often a function of the time of day. Some solutions to concurrency control problem are based on timestamping [Rose78, Thom79, Reed78, Bern80a].

Typical of timestamp algorithms is basic timestamp ordering [Bern80a]. In addition to transaction timestamps, each object (the smallest logical unit of data) has read timestamp and a write timestamp in basic timestamp ordering. These are the timestamps of the latest reader and writer (respectively) for the object of interest, and are maintained in a timestamp table.

(A transaction Ti is later than another transaction Tj if Ti 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 initiation time order. Transactions attempting to violate timestamp ordering are restarted (aborted and started over); causing serialization to occur in timestamp order.

2.1.3 Serial Validation

In Serial Validation algorithms [Kung81, Bada79, Schl81, Mena82], transactions are permitted to run freely until they reach their commit point (the point at which the effects of the transactions are about to be registered in the database). Upon reaching this point, each transaction is subjected to a test which ensures that committing it will not lead to violation of the database integrity constraints or non-serializable results. Transactions which fail this test are restarted.

Typical of Serial Validation algorithms is Kung and Robinson's Optimistic Serial Validation. It is optimistic since database resources are not locked hoping that they will not be modified by other transactions.

Kung and Robinson [Kung81] divided update transaction into three phases: the read phase, the validation phase and the write phase. In the read *phase* the intended work of the transaction is done, all updates are made on local copies of the database objects. In the *write phase* the local database objects are made global. In the *validation phase* a test for serializability of the developing parallel transaction system is performed in the following way:

each transaction is assigned a unique transaction number after positive validation, before the write phase

Let t_s be the highest transaction number at the start of transaction T,

Let t_f be the highest transaction number at the begining of the validation phase of T,

Then in the validation phase the following check is performed:

```
Valid : = true;
for t from $t_s$ + 1 to $t_f$ do
if (write set of transaction t
      intersects readset of T)
      then valid: = false
```

Validation and subsequent write are one critical section.

In the validation phase, all transactions t which had their write phase after the begining of T and before the validation of T are considered. For these transactions we check whether the set of objects written by t interesects with the set of objects read by T. In this case we possibly have a conflict which may destroy serializability, such that T has to be backed-up and restarted.

Read transactions, of course, do not have a write phase, but they also have to be validated. Again, the write sets of all transactions with numbers from $t_s + 1$ to t_f have to be examined to detect intersections with the read set of T. The difference being that the validation need not be done in a critical section.

2.2 **Transaction Classes algorithm.**

In most cases, a particular transaction

depends only on a small part of the system state and by implication it depends on a small part of the database and other resources.

Therefore one technique for avoiding conflict is to partition database entities into disjoint classes of entities. Transaction using common parts of the database must still be scheduled serially. If such a policy is adopted then each transaction will see a consistent version of the state [Eswa76]. Bernstein et al used this idea in the design of Transaction Classes algorithm of the system for the Distributed Databases (SDD-1) [Bern80c]

In the Transaction Classes algorithm, avoidance of undesirable interleavings is accomplished using two methods:

(1) Examination of each transaction to determine if it is conceivable that it could participate in a non-serializable execution. Most of these efforts is done statically during database design stage. The database administrator establishes a static set of transaction classes during database

design stage. Each transaction class is defined by a logical readset and writeset. A transaction fits in a class if the readset and writeset of the transaction are contained (respectively) in the readset and writeset of the class. An example of class definition is shown in figure 2.1 below.

2. If the examination in (1) indicates that some transactions are dangerous because they can participate in non-serializable executions then READs and WRITEs of such transaction are synchronized using timestamping mechanism.

Relation Schema	INVENTORY (ITEM #, DESCRIPTION, PRICE, QUANTITY)
Class 1	
Readset:	INVENTORY (ITEM #, PRICE)
Writeset:	INVENTORY (PRICE)
Comments:	Transactions that update prices
Class 2	
Readset:	INVENTORY (ITEM #,

QUANTITY WHERE PRICE > 100)

Writeset: INVENTORY (QUANTITY)

Comments: Transactions that update quantities of high priced items

Class 3

Readset: INVENTORY (ITEM #, DESCRIPTION, PRICE WHERE QUANTITY > 0)

Writeset: Users Terminal

Comments: Transactions that display information about items currently in stock.

Fig 2.1 Class definitions using simple predicates

A simple illustration of how a transaction is synchronized at run time and how the algorithm works is shown in fig 2.2.

```
Do forever;
Wait for a transaction T to arrive;
Find a class C, in which T fits;
```

```
Look up the transaction classes table for the synchronization rules of class C;
Effect READ operations on behalf of T, sychronizing where necessary;
Effect WRITE operations on behalf of T, synchronizing where necessary;
End;
```

Fig. 2.2 How a transaction is processed for single site database.

2.3. Semantically Consistent Schedules Algorithm

It has been shown by Yannakakis [Yann82b] that serializability is much more than what is required to guarantee consistency in database. It has also been reported that in some applications users may be satisfied with a schedule that preserves consistency even though it is not serializable. Allowing the system to run these non-serializable but consistency preserving schedules may result in higher parallelism and better performance [Eswa76]. These facts are used by Garcia-Molina [Garc83] in proposing Semantically

Consistent Schedules (SCS) algorithm.

Like the Transaction Classes (TC) algorithm, SCS algorithm uses transaction information. But Garcia-Molina disagrees with the restricted way transaction information is used in TC algorithm. In TC algorithm, transactions T1 and T2 of the same type have similar access patterns, conflict heavily, and should be synchronized. In Garcia-Molina's proposal, the fact that T1 and T2 are of the same type does not convey exhaustive information about conflicts. Hence, the SCS algorithm requires information about the compatibility or otherwise of T1 and T2. So while the TC algorithm uses transaction conflict (syntactic) information, the SCS algorithm uses compatibility (Semantic) information with the hope of allowing more concurrency.

The basic idea of the proposal is that transactions fall into a collection of semantic types. It is assumed that through the user's semantic knowledge of the transactions and the actions they perform, users may be able to group the actions of the transactions into steps. Then, the users may be able to indicate that the steps of a transaction of a given semantic type can be interleaved, without violating consistency constraints, with the steps of another type of transaction.

The semantic knowledge is supplied in form of rules that describe the most common (if not all) of the Semantically Consistent Schedules to a locking mechanism.

2.4. Performance of Concurrency Control Algorithms

All concurrency control algorithms have a cost with the controls which they provide. Since it would be easy to simply require transaction 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 achievement of 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 long transactions which precede them regardless of whether or not they actually conflict. This leads to poor average response time. Allowing controlled concurrent database accesses by transactions solves these potential problems. Given that a concurrency control algorithm is needed, and that some algorithms are available, the database system designer is faced with a difficult decision: which concurrency control algorithm should be chosen?

Below is a review of performance of the basic concurrency control algorithms.

2.4.1 Locking Performance

Several of the studies on locking performance were based on simulations while some were based on qualitative and analytical techniques. The various studies have produced a wealth of data and some empirical results.

Some of the earliest work on locking performance was done by Munz and Krenz [Munz77] in a simulation study of two problems concerning deadlocks. The following questions were asked in the study.

(i) when deadlock occurs which transaction should be restarted?

(ii) if the state of a transaction is saved periodically (checkpointing), then a transaction needs not restart from the begining;
it can be rolled back to the nearest check-point necessary for breaking the deadlock, then continued from there; does check-pointing help?

The study involved about six hundred simulation runs and the conclusions of the study are:

(i) if the cheapest (by some measure) transaction in a deadlock is restarted, this would be considerably better than simply restarting the transaction that caused the deadlock.

(ii) checkpointing does not pay; i.e. transactions should release all locks when they restart.

Balter, Berard and Decitre [Balt82] investigated the relative effects of blocking and deadlock in locking algorithm. They simulated several concurrency control algorithms and found that deadlocks are secondary to blocking in performance degradation; for when the system thrashes (when there are many transactions doing useless work) for locking, the waiting time for locks is much greater than the time cost in restart delays.

An extensive simulation study has been done by Ries and Stonebraker [Ries77, Ries79]. This is a study of the effect of granularity on locking, in which they considered exclusive locking, both static and dynamic, in centralized and distributed systems.

They have found that dynamic locking is better than static locking when transactions are short, and worse when they are long. They also considered sequential data access, locking hierarchies and predicate locks and concluded that for sequential access; a medium granularity is usually optimum; locking hierarchies should be used if transactions are long and granularity is fine; and it is doubtful that predicate locking [Eswa76] can improve performance.

In considering the benefit of coarse granularity, they argued that with coarse granularity, a transaction needs to set a few locks only, and so cuts down on the cost of setting locks. Another benefit of coarse granularity was shown by Tay, Goodman and Suri [Tay84] to be reduced workload.

Peinl et al [Pein83, Hard85, Pein87] studied locking as well as certification. Peinl disagreed with the highly idealized simulation approach taken by previous authors and used a bench mark-like technique instead. He took page reference strings from some applications on a CODASYL-LIKE database, and fed them to a string converter which generated from them reference strings for a given number of concurrent transactions. Essentially to get the multiprogramming level up to the desired number, a transaction is picked from further down the reference string. These transactions are then run on a fictitious database with different concurrency controls, so as to compare the performance of the algorithms. Unfortunately, the experimental results show a high restart rate. Two main reasons could be attributed for this.

First, in the applications that were monitored, transactions may release read locks once they were done with them (and acquire more locks later), whereas the experiments used two-phase locking. The probability of conflict is therefore higher in the experiments. Second, the technique of picking transactions from further down the reference string to match the desired multiprogramming level makes it possible that two very similar transactions are run at the same time, thus raising the probability of conflict above that in the original applications themselves.

2.4.2. Performance of other algorithms

To date, literature on performance of concurrency control algorithms based on other mechanisms than locking has been very scanty or indeed lacking. It has, however, been established that some of the concurrency control algorithms have anomalies that could hurt their performance particularly in situations where conflicts are not rare.

For example, timestamping cyclic restart anomaly could affect timestamping performance [Bern80a]. Transaction starvation, a peculiarity of Optimistic Serial Validation [Kung81] can also hurt serial validation performance particularly in non-query dominant applications. The degree to which this anomaly affects 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 during reincarnation (each time they run).

Race condition anomaly of the SCS algorithm [Garc83] could also hurt its performance. For example, the anomaly may make it impossible to obtain the results of a semantically consistent schedule with any serializable schedule. This may have a limiting effect on the anticipated performance of SCS.

In general, pre-analysis of transactions in TC and SCS algorithms promises a reduction of transactions requiring synchronization and therefore may allow more concurrency. But this advantage does not come for free since users and/or database administrators are burdened with the task of identifying and indicating transactions that could be involved in dangerous interleavings.

As special purpose algorithms, performance of TC and SCS depends on application. The algorithm may not improve performance in some applications while performance will definitely improve in others.

CHAPTER 3

INTEGRATED TRANSACTION CHARACTERISTICS ALGORITHM AND A COST MODEL

This chapter proposes Integrated Characteristics (ITC) algorithm. The ITC algorithm makes use of syntactic properties of transactions as in [Bern80c] as well as semantic properties of transactions as in [Garc83].

The Integrated Transaction Characteristics (ITC) algorithm is expected to further improve on the performance gains of TC and SCS algorithms over the basic timestamping and locking mechanisms.

The ITC algorithm is designed for a single site database environment and does not have to contend with and provide for node failures, communication overhead and other problems peculiar to distributed database environment.

Also described in this chapter is a model for evaluating the costs associated with alternative concurrency control algorithms. The model, a variant of the abstract cost model reported by Michael Carey [Care83a], is subsequently used to analyse storage and CPU costs associated with Transaction Classes (TC), Semantically Consistent Schedules (SCS) and Integrated Transaction Characteristics (ITC) algorithms.

3.1 **Integrated Transaction Characteristics algorithm**

The basic idea about ITC algorithm is that each transaction has a set of attributes in form of its readsets, writesets and the transactions it is potentially in conflict with. So, given a set of transactions that can run on a database, the attributes indicate which of the transaction(s) can run simultaneously without being involved in non-serializable execution.

The attributes are divided into syntactic and semantic attributes. Syntactic attributes are identified during database design stage while semantic attributes are determined and supplied by the users of the database as explained below.

3.1.1. **Syntactic attributes**

Logical readsets and writesets of transactions are used by the database administrator to establish a set of syntactic groups or transaction classes. Formally, each transaction class is defined by a logical readset and writeset. A transaction fits into a class if the readset and writeset of the transaction are contained (respectively) in the readset and writeset of the class. Furthermore, a transaction fits into one and only one class and there may be some classes with only one transaction.

Two classes conflict if the readset or writeset of one class intersects with the writeset of the other class. So, by examining the readsets and writesets of each class vis a vis the readsets and writesets of other classes conflicting classes are determined.

3.1.2. Semantic attributes

Transactions within a class have intersecting readsets and writesets and therefore traditionally require synchronization at run time. But through the semantic knowledge of transactions and the actions they perform the database administrator and/or users may be able to tell which of the transactions with overlapping readsets/writesets are compatible [Garc83]. This semantic knowledge is given to the ITC algorithm in order to allow the non-interfering transactions to run faster.

For this purpose the transactions are grouped into semantic types which indicate the transactions operation (e.g. delete, write, read, re-write etc). A compatibility set which describes the interleavings that do not violate consistency is associated with each of these semantic types. Each compatibility set consists of interleaving descriptors which depict a specific type of allowable interleaving.

As an example, if Y1,Y2, Y3 are semantic types of three transactions. We may have a compatibility set CS (Y1) = {{Y1, Y2}, {Y1, Y3}}, meaning that transactions of type Y1 can be interleaved with transactions of type Y2 and with those of Y3. However, Y2 and Y3 transactions cannot be interleaved. The interleaving descriptor sets are {Y1, Y2} and {Y1, Y3}.

Semantic type, compatibility set and interleaving descriptor are defined more formally in [Garc83] as follows.

Semantic Type: Transactions are classified (by the users) into a set of semantic types. Let TYPES be the set of all transaction types. (for a given application). Then each transaction T submitted to the system will have a type ty (T) $\in$ TYPES, associated with it.

Compatibility Set: With each semantic type Y $\in$ TYPES, associate a compatibility set CS(Y). Each element of CS(Y) is an interleaving descriptor set.

Interleaving desriptor set: An interleaving descriptor set of a compatibility set CS(Y), h (h $\in$ CS(Y), Y $\in$ TYPES), must have the following properties:

(1) h $\subseteq$ TYPES

(2) any schedule S of a set of transactions T must be semantically consistent for all initial database

states if the following characteristics exist:

(a) There is a transaction $T_1 \in T$ such that ty $(T_1) = Y$;

(b) All other transactions $T_2 \in T$ $(T_2 \# T_1)$ are such that ty $(T_2) \in h$ and

(c) S is stepwise serial, that is a schedule which represents an execution in which a set of actions (which make up a step of a transaction) are performed as indivisible units.

3.1.3 <u>The Synchronization Mechanism</u>

The basic mechanism used for synchronization in ITC algorithm is locking. It is obviously not a new mechanism since all the ingredients (e.g. exclusive/shared locks deadlock detection) are well known. However, the ingredients are used in a fashion that guarantees that only compatible transactions are interleaved.

Associated with each object in the

database (at run time) is an interleaving descriptor ID(O). The descriptor indicates that one or more transactions with the interleaving descriptor associated with the object have accessed the locked object, are being interleaved, and have not finished yet. In addition to the basic locking requirements of the locking mechanism, the interleaving descriptor indicates that a requesting transaction could only access the locked object if the interleaving descriptor is an element of the requesting transaction’s compatibility set otherwise the transaction must wait.

For example, suppose a transaction T with semantic type ty (T) = Y1 and interleaving descriptor id(Y1) = {Y1, Y2, Y3}. If T accesses an unlocked object O, it sets a lock on it with ID(O) = {Y1, Y2, Y3} indicating that T could be interleaved with transactions of semantic types Y1, Y2, Y3. If T requires another object O’ which is already locked, then T must check if it can proceed: If ID(O’) # {Y1, Y2, Y3} then object O’ is participating in a different interleaving and T must wait. Otherwise T can proceed.

Also, when a transaction has multiple interleaving descriptors, it can choose what interleaving it wishes to participate in. For example, consider a transaction T such that semantic type ty (T) = Y1 and compatibility set CS (Y1) = {{Y1, Y2},{Y1,Y3}}. Transaction T must decide which interleaving it is to participate in after requesting for an object O which has been locked by other transactions. If ID(O) = {Y1, Y2}, then T selects the {Y1, Y2} descriptor and from then on acts as if it had no other ones. (If, later on, T wishes to access an object O with ID (O) = {Y1, Y3}, it will have to wait). Similarly, if ID (O) = {Y1, Y3}, then T selects the {Y1, Y3} descriptor as its only one. If ID(O) is neither {Y1, Y2} nor {Y1, Y3}, then T must wait until O is unlocked. Finally the object locked by T before a descriptor decision is reached, are locked with ID(O)=0 (null). When a decision is reached all these descriptors are set to the decided set.

3.2. Concurrency Control Cost Model

The model contains a single concurrency scheduler which makes scheduling decisions based on information that it maintains about the history of requests received to date and predefined transaction characteristics. This information is referred to as the concurrency control database, and is treated conceptually as a simple relational database, ignoring the many data structures which might be used in an actual implementation. For a particular concurrency control algorithm, the scheduler obeys a well-defined set of rules as indicated in their description which describes how it should respond to incoming requests, based both on the requests themselves and on the contents of the concurrency control database. The model is as shown in figure 3.1.

3.2.1. Transaction Requests

The model recognises three types of requests from transaction viz: BEGIN, END, and ACCESS. The first two mark the begining 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. It is assumed in the model that transactions abide by the responses received from the scheduler, accessing

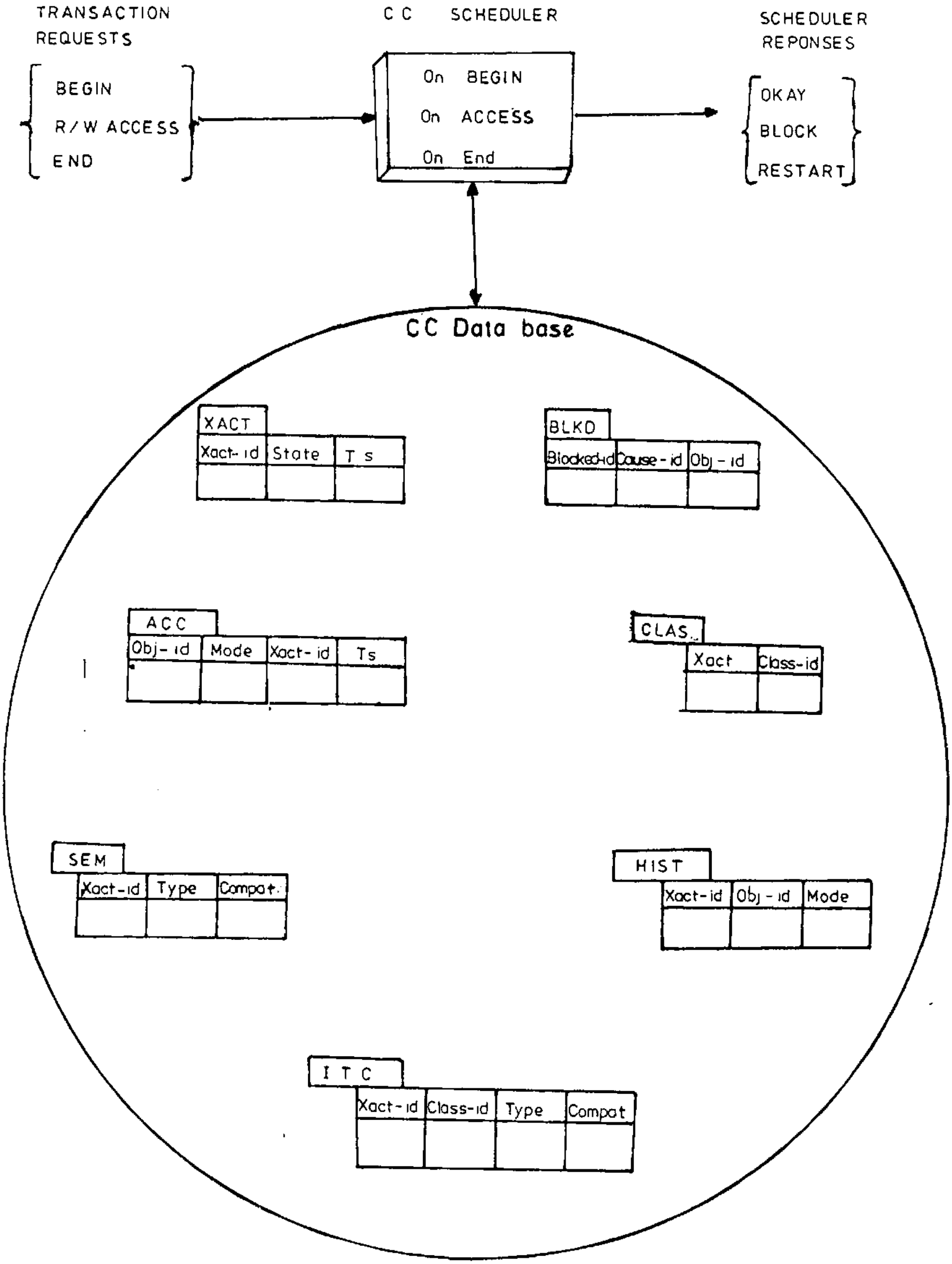

Figure 3.1 : Concurrency Control Model

data objects accordingly. It is also assumed that writes go to a list of deferred updates [Gray79] to be installed at transaction commit time.

3.2.2. **The Concurrency Control database**

The concurrency control database, shown in figure 3.1 consists of seven relations.

The XACT relation contains transaction state information, specifying the transaction identifier, state (ready, blocked, committed, aborted) and time-stamp of each current transaction.

The ACC relation contains 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. For Semantically Consistent Schedules algorithm and Integrated Transaction Characteristics algorithm, 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 Transaction Classes algorithm.

The BLKD relation contains information about any blocked transactions, the transaction identifiers 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 implicitly unblocks the corresponding transaction, allowing it to continue processing from where it left off.

The CLAS relation contains information about the class a transaction belongs and timestamp of the transaction. The SEM relation describes transaction identification, transaction semantic type, and its compatibility set.

The ITC relation stores transaction information such as transaction identification, the class a transaction belongs, the semantic type of the transaction and its compatibility set.

The HIST relation stores histories of ACCESS requests which are conditionally granted until a concurrency control decision is made. 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 relation 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 relation. Thus, the portion of the concurrency control database used by an algorithm is used for the costing discussed in the next section.

3.2.3 Algorithm Cost Comparison

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 used to compare algorithm cost in [Bern80f]. The analysis techniques are illustrated by using them to compute and compare the cost of the three transaction characteristic based algorithms.

To facilitate cost analysis, a performance model on a set of simple parameters is used. The parameters are defined as though the transaction mix used to evaluate algorithm cost consists of transaction of the same fixed size. The technique 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.

Let Tc be the number of transactions in the system (that is, the multiprogramming level)

Let R be the readset size for these transactions.

If Fw is the fraction of the writeset included in the readset then each transaction makes R + RFw or R (1+Fw) data access requests (assuming there are no blind writes)

Let Fb be the fraction of blocked transactions so that FbTc is the current number of blocked transactions

Let Frc be the recent commit factor, so that Frc Tc is the number of recently committed transactions, where a recently committed transaction is one which committed since the oldest remaining active transaction began running.

Let Ct, Cs and Cc be the number of comparisons (search length) for Transaction Classes, Semanticaly Consistent Schedules and Integrated Transaction Characteristics algorithms respectively.

The blocking and restarting characteristics of the algorithms will influence the parameter Fb and Frc, so they will vary depending on the underlining concurrency control algorithm. The parameter Fw is determined by the transaction mix. To bound these parameter, note that $0 \leq Fb \leq 1$, $0 \leq Fw \leq 1$. It is certain that the recent commit factor $Frc \geq 0$. Also the lower bound of Ct, Cs and Cc are one each. That is $Ct \geq 1$; $Cs \geq 1$; $Cc \geq Ct$

It is expected that transactions commit roughly in their startup order if transactions are of equal size thus producing a small value for Frc. However a very long transaction mixed with a collection of short transactions would result in a large value for Frc since many short transactions could complete during the life-time of the long transaction.

3.2.4 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 analysed. One field of one tuple of one relation is taken as the unit of storage cost for this analysis. The overall database size is 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.

3.2.4.1. The storage Cost of Transaction Classes Algorithm

The XACT relation represents a storage cost of 3Tc.

The HIST relation must store write request entries for the Tc active transactions, so it represents a storage cost of 2TcFwR. 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 RFw write timestamp entries for each of Tc active transactions plus the FrcTc recently committed transactions. This yields a worst case total storage cost for the ACC relation of 3Tc(1+Frc) R(1+Fw).

At the other extreme, if all active and recently committed transactions access the same set of objects, the storage cost of the ACC relation is just 3R(1+Fw), since each object has at most one read timestamp entry and one write timestamp entry. The CLAS relation represents a storage cost of 2Tc. Thus for Transaction Classes (TC) algorithm the storage cost (STORtc) is:

$3R(1+Fw)+ Tc(5+2FwR) \leq STORtc \leq 3Tc(1+Frc)R(1+Fw)+Tc(5+2FwR)$ ....(1)

3.2.4.2 The Storage Cost of Semantically Consistent Schedules algorithm

The XACT relation represents a cost of 2Tc while BLKD relation costs 2FbTc; for the ACC relation, a storage cost of 3Tc(1-Fw)R is incurred for storing read locks. For storing write locks, the cost can vary from as low as 3FwR in the case where all Tc transactions write the same object.

The SEM relation represents a storage cost of 3Tc.

Thus for Semantically Consistent Schedules(SCS) algorithm the storage cost (STORscs) is: $2Tc(1+Fb)+ 3TcR(1-Fw)+3FwR+3Tc \leq STORscs \leq 2Tc(1+Fb) + Tc(3R+3)$ which is equivalent to $2Tc(1+Fb)+3TcR(1-Fw)+3FwR+3Tc \leq STORscs \leq Tc(5+2Fb+3R)$ ...(2)

3.2.4.3 The Storage Cost of Integrated Transaction Characteriatics Algorithm

The XACT, BLKD and ACC relation are used as in 3.2.4.2. above.

The ITC relation represents 4Tc. Thus for Integrated Transaction Characteristics (ITC) algorithm the storage cost (STORitc) is:
$2Tc\ (1+Fb)+3TcR\ (1-Fw) + 3FwR+4Tc \le STORitc \le 2Tc\ (1+Fb)\ +3Tc\ (R+2)$ which is equivalent to $2Tc\ (1+Fb) + 3TcR\ (1-Fw)\ +3FwR+4Tc \le STORitc \le 2Tc+2TcFb + 3TcR+4Tc$ which is equivalent to $2Tc\ (1+Fb)+3TcR(1-Fw)+3FwR+4Tc \le STORitc \le Tc(6+2Fb+3R)$ .....(3)

3.2.4.4. Relative Storage Costs Of The Algorithms

Given the bound of Fb and Fw (which is between 0 and 1) some conclusion can be drawn about the relative storage costs of the concurrency control algorithms. From equations (1), (2) and (3) it can be concluded that Semantically Consistent Schedules algorithm has the smallest worst-case storage of the three algorithms which is (7+3R) Tc. The Integrated Transaction Charateristics algorithm has the next worst-case storage of (8+3R)Tc; while the Transaction Classes algorithm has the highest worst case storage since parameter Frc is unbounded.

The worst-case storage 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 probability of conflicts in [Gray81a].

For the best-case storage, we compare the three equations again.The best-case storage for Transactions Classes algorithm (7Tc+6R) is smaller and therefore better than the best-case storage for either Semantically Consistent Schedules (9Tc+3R) algorithm or Integrated Transactions Characteristics algorithm (10Tc+3R). The Semantically Consistent Schedules algorithm dominates the Integrated Transaction Characteristics algorithm.

The Semantically Consistent Schedules algorithm is best in terms of worst case storage cost, indicating that it is superior under low-conflict transaction mixes. Transaction Classes algorithm is best in terms of best-case storage cost. This implies that it is better under high-conflict transaction mixes. The Integrated Transaction Characteristics algorithm is worst in terms of best-case storage cost, meaning that it performs worst under high-conflict transaction mixes but better than the Transaction Classes algorithm when transaction conflicts are low.

3.2.5. CPU COST

The number of operation involved in executing the query sets for various algorithm is analysed, in the following, in order to compare their CPU costs. The unit of CPU cost for this analysis is taken to be one tuple access, insertion, replacement or comparison in one relation. So the assumption implies that the CPU time required is proportional to the number of table lookups and/or class or semantic type comparisons [Bern80d].

But, analysing the CPU cost of a given concurrency control algorithm is, in general, considerably more complex than analysing the storage cost of the algorithm. So for simplicity and clarity only the no-conflict CPU cost [Bada81] (the CPU cost experienced by a transaction which does not conflict in any way with other concurrent transaction) is considered. Since in practice actual conflicts are rare [Gray81a], the no-conflict CPU cost should be a reasonable "first-order" estimate.

3.2.5.1. The CPU Cost Of Transaction Classes algorithm.

For this computation, we assume that read requests are processed as they arrive, and all write requests are processed together just prior to transaction commit time. This simplifies the considerations involved in making the underlining timestamping mechanism work with deferred updates; otherwise some scheduling would be required to prevent transactions from reading objects for which a write request has been processed but the associated deferred updates has not yet taken place [Bern82, Agra83]. The HIST relation is used to defer write timestamp checking until commit time, with similar timestamp checking and updating involving HIST relation occurring with END request. So, the cost of processing a BEGIN request is 1.

For identification of the class of the transaction a Ct class comparisons are made so we charge a CPU cost of Ct for the operation. The cost of checking restart conflict is 1; while a CPU cost of 2 is charged for each write entry in the HIST relation for an END request check. 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 non-conflict case, is 2R. The cost of processing RFw write requests, each of which simply records the pending request in the HIST relation, is RFw.

The cost of processing an END request depends on the number of timestamps deleted. 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 2RFw to check the HIST relation contents for restart conflicts, 1 to change the state of the committing transaction, 2RFw to update the write timestamp of each write request in the HIST relation once the transaction has indeed committed, and 1+R(1+2Fw) to delete the information associated with the transaction. Hence the CPU cost for the transaction classes algorithm is CPUtc = Ct+3+R(3+7Fw)........(4)

3.2.5.2. The CPU cost of Semantically Consistent Schedules Algorithm

The cost of processing a BEGIN request is 1. The cost for checking blocking conflict is 1, so the cost of processing R(1+Fw) data access request is 2R (1+Fw) if no blocking occurs. The CPU cost for comparing and identifying transactions of the same semantic types is Cs. 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 blocking transactions). Hence the CPU cost of Semantically

Consistent schedules algorithm is CPUscs = Cs + 4 +R (3+2Fw) - (5)

3.2.5.3. The CPU cost of Integrated Transaction Characteristics Algorithm

The analysis for Integrated Transaction Characteristics algorithm is much like that for Semantically Consistent Schedules algorithm except that it incurs the cost of identifying transaction class as in Transaction Classes algorithm and additionally identifies semantic type within a transaction class. Hence the CPU cost of Integrated Transaction Characteristics algorithm is CPUitc = Cc+4 + R (3+2Fw). ....(6)

3.5.4 Relative CPU Costs of the Algorithms

Garcia-Molina [Garc83] has reported that given an application and assuming a relational database the SEM relation cardinality is less or equal to CLAS relation cardinality. This implies that Cs is at most equal to Ct. Therefore using the bounds of Fw,Ct,Cs, Cc and considering equations (4) and (5) it is clear that the CPU cost for the Semantically Consistent Schedules algorithm is less than the Transaction Classes algorithm CPU cost except when Fw is very small and Ct equals Cs when CPUscs could be the same as CPUtc. Now we consider equations (5) and (6). Cc is greater than Ct; therefore if Fw is very small CPUitc is greater than CPUtc otherwise if Fw is not small and Cc is sufficiently close to Ct then CPUtc is greater than CPUitc. This of course implies that CPUscs is less than CPUitc.

CHAPTER 4

CONCURRENCY CONTROL PERFORMANCE

This chapter describes a simulation model of a database system. It is used to compare performance of the three single-site concurrency control algorithms evaluated under various transaction workloads and system costs. Performance results using this simulation model are given for Transaction Classes, Semantically Consistent Schedules, and Integrated Transaction Characteristics algorithms.

4.1. Background

Before describing the simulation model, it will be helpful to consider the nature of the problem it addresses. The purpose of a concurrency algorithm is to facilitate the simultaneous execution of a number of transactions in order to enhance performance. The degree to which an algorithm allows transactions to execute concurrently and make progress towards completion is its level of useful concurrency. In order to compare alternative algorithms, a measure of their relative levels of useful concurrency must be obtained.

Usually, the number of active transactions in a system is more than the system's level of useful concurrency because of the possibility of restarts and idling. This implies that there could be some active transactions doing useless work. The number of active transactions as a measure of performance is therefore obviously undesirable as 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 catching their writes locally, N-1 will be forced to restart. Thus, knowing that N transactions are executing concurrently is not sufficiently informative. A better measure of concurrency benefits offered by alternative algorithms is a measure of successful commits per unit time or throughput. We have therefore considered throughput as a fair measure of performance and it will be used for the purpose of this work.

Concurrency control semantics are actually implemented and simulated in a closed queuing model of a database system to obtain throughput information.

4.2. JUSTIFICATION FOR A SIMULATION APPROACH

There are several reasons why simulation is most appropriate to obtain relative performance information about alternative concurrency control algorithms. First, 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. It would, thus, be prohibitively hard to develop tractable analytical models of three concurrency control algorithms for comparative purposes. Second, by selecting a simulation approach, a more realistic collection of transaction mixes and workloads can be studied.

Finally, there are certain facts about the behaviour of transaction in real systems which are difficult to represent in an analytical model. For example, restarted transactions could re-request the same data objects that they requested the last time. Simulation provides a way to model such facts and evaluate algorithm performance.

4.3. Model Description

4.3.1. The Workload Model

An important component of the performance model is a transaction workload model. The workload model is a variant of Tay's workload model [Tay84].

Initiation of a transaction from a terminal essentially entails picking a transaction randomly from a transaction table (see Appendix 1). Each transaction thus picked has a transaction type, a transaction class, a compatibility set and a workload consisting of a readset and a writeset (determined during pre-analysis stage). Transaction type, transaction class and compatibility set are the attributes of the transaction while the readset and writeset are the objects the transaction will read and write during its execution. Other workload parameters apart from the predetermined readsets and writesets are shown in Table 4.1.

Workload Parameters	
Readset	} determined during pre-analysis
Writeset	}
Nterms	No of terminals (level of multi-programming)
Dbsize	number of objects in database
Gransize	number of object in a granule
Conflict-delay	Conflict avoidance delay

Table 4.1. Workload Parameters for simulation

The parameter Nterms determines the number of terminals or level of multiprogramming.

The parameter Conflict-delay determines the mean length of time required for a terminal to resubmit a transaction after finding that its current transaction has been restarted with the delay associated with each particular restart determined from an exponential distribution with this mean.

The parameter Dbsize determines the number of objects in the database, that is the database size.

The parameter Gransize determines the number of objects in each granule of the database, that is the granule size.

Concurrency Control requests are made on granule basis. Thus when a transaction reads or writes an object, an associated concurrency control request is made for the granule which contains the object.

In order to model read and write request, objects are given integer names ranging from 1 to Dbsize while granules are also given integer names ranging from 1 to Dbsize/Gransize. Since concurrency control request is made for a granule which contains the objects, it follows that a request should be made for granules ((i - Dbsize/Gransize)+1) in order to access object i.

4.3.2. The Queueing Model

The second component of the performance

model is a close queueing model (figure 4.1) of a single-site database system. The model is an extended version of Ries model [Reis77, Ries79].

In the model there is a fixed number of terminals <u>Nterm</u> from which transactions originate. Each transaction enters concurrency control queue (CCqueue) and makes the first of its concurrency control requests. Here, several tables are looked up depending on the requirements of the algorithm under consideration. The tables include active transaction table, lock table, transaction classes table, compatibility sets table and timestamp table. The transaction table contains active transactions that

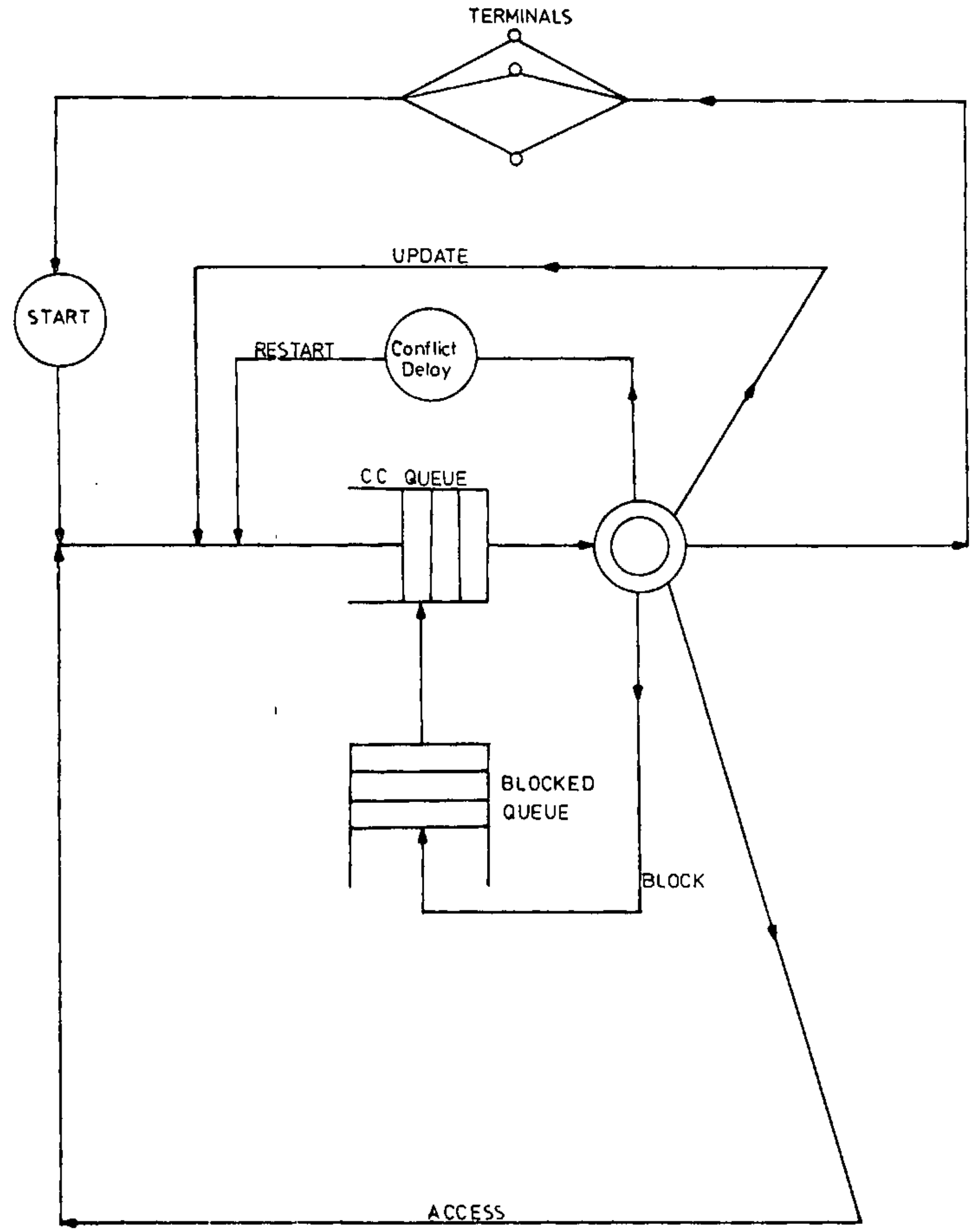

Fig. 4.1 Logical DB queuing model.

are not yet committed. Objects that are being held by transactions are contained in the lock-table. Transaction classes and semantic compatibility sets tables contain information about the characteristics of transactions that are expected in the system.

A history of transaction accesses is also maintained mainly to determine whether the granule that has been accessed contain the object that is to be accessed next. It is also used to ensure that transaction doesnot request for objects, and hence granules, that it has accessed or held.

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 ease of implementation that transactions which read and write objects perform all of their reads before performing any write.

If access is denied the transaction either blocks or restarts. For example if the outcome of 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 abort and restart, the transaction goes to the back of the concurrency control after a randomly determined conflict avoidance delay period of mean conflict-delay; it then begins to make all of its concurrency control request over again. The conflict-delay ensures that, as much as possible, the transaction with which the restarted transaction is in conflict with has committed in order to avoid cyclic restarts. Eventually, the transaction may complete and the concurrency control algorithm may decide to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, it must first write its deferred updates into the database. In either case the transaction returns to its terminal for re-cycling, after committing.

Associated with each logical service (concurrency control queue and blocked queue) and table look ups is some use of CPU and I/O resources. When a transaction enters a queue it first performs the queue-related I/O processing and then performs the queue-related CPU processsing, with the amounts

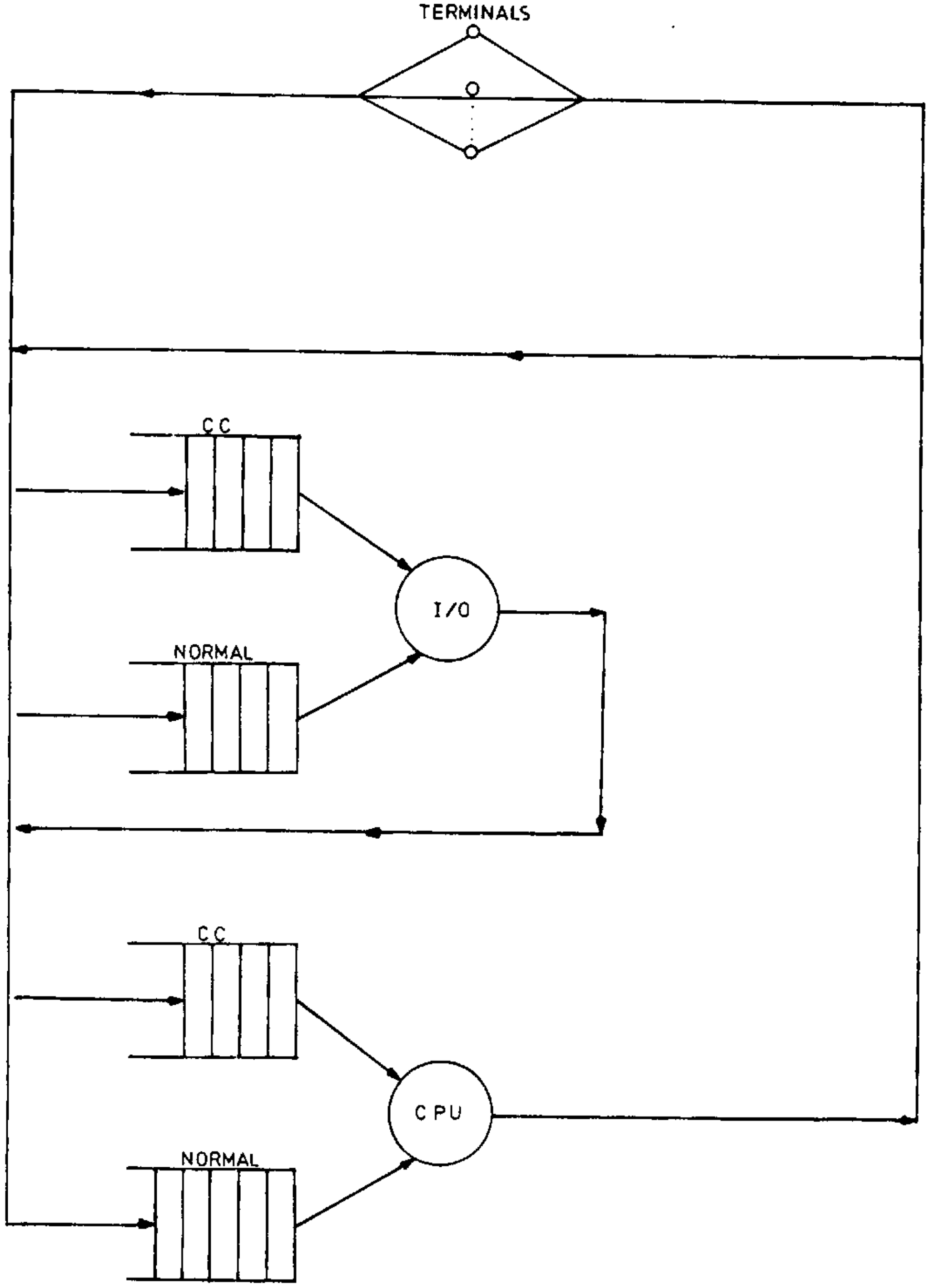

Fig 4.2 Physicol DB queuing model

of CPU and I/O per logical service being specified as simulation parameters. The underlying physical systems model is depicted in figure 4.2. 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 used to allocate resources to transactions in the concurrency control I/O and CPU queues of the underlying physical model is first come, first served (FCFS). Concurrency control requests are thus processed one at a time, as they would in an actual implementation. The resource allocation policy for the normal I/O and CPU service queue of the physical model are FCFS and round-robin respectively. These policies are again chosen to approximately model the characteristic which a real database implementation would have. When request 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.

The parameters determining the I/O and CPU service times for the logical resources, that is the system parameters are given in Table 4.2.

SYSTEM PARAMETERS	
IO-time	I/O time for accessing an object
CPU-time	CPU time for accessing an object
CC-IO-time	Concurrency Control I/O time
CC-CPU-time	Concurrency Control CPU time
TAB-IO-time	Searching I/O time
TAB-CPU-time	Searching CPU time
Stagger-mean	Mean of exponential randomizing delay

Table 4.2: System Parameters for Simulation

The parameters IO-time and CPU-time 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 CPU-time at the time of the write request and IO-time at deferred update time, as it is assumed that deferred update list is maintained in buffers in main memory.

The parameters CC-IO-time and CC-CPU-time are the times associated with a concurrency control request.

The parameters TAB-IO-time and TAB-CPU-time are the times associated with a table look up.

The Stagger-mean parameter is the mean of an exponential time distribution which is used to randomly stagger transaction initiation time from terminals.

For ease of implementation all the time parameters represent constant service times rather than stochastic ones.

4.4 Pre-analysis of transactions and data generation

For a user to supply helpful indicators

of the characteristics of transactions at job submission time, an analysis of transactions ought to have been done at database design stage and relevant guide made available to the user. So in preparation for the simulation experiments, a pre-analysis involving the determination of transactions and their readsets is done. Assuming that there are no blind writes, writesets which are subsets of readsets are statically determined.

By examining the readsets and writesets, transaction classes and conflicting classes are determined. The transaction classes are later submitted along with randomly initiated transactions in the simulation so as to determine which class a transaction should run in parallel with or whether synchronization within a class is necessary.

Also the semantic types are determined by examining the generated transactions. For ease of implementation the semantic types we consider are READ-ONLY and READ-WRITE. So, all transactions fall into READ-ONLY and READ-WRITE semantic types. In addition only READ actions and WRITE actions are considered; counteractions are ignored so as to avoid unnecessary complexity. To determine compatible sets, actions of transactions are checked against actions of other transactions thus ascertaining whether the actions, when interleaved, would ensure serializablity.

Unlike trace driven simulations of Peter Peinl [Pein83, Pein87] where empirical trace input called object reference strings (ORS) were taken from real DB-application of different types and sizes, this simulation is random-number driven. The choice is informed by the unavailability of appropriate local database environment and the fact that result based on traces do not yield general results [Hard85].

As in [Munz77], readset of a transaction is determined using a mean readsize and exponential distribution. That is the readset size is selected from an exponential distribution with mean readsize 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 1. The readset size is truncated to the size of the database if it exceeds the database size as transactions cannot possibly access more objects than the database holds.

Only sequential transaction types are considered in this implementation while random transaction types are left for future work.

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 sequential transaction type is intended to model transaction which access objects using either an ordered primary index or a sequential scan of an entire relation or file.

4.5. Design of Experiment

In order to obtain the relative performance of each algorithm, the parameters (factors) which influence appreciably the value of the chosen performance index (throughput) are identified. The values (levels) of each factor are then suitably selected.

Multiple simulation runs are also performed in order to determine the influences of a factor on the performance of each algorithm.

4.5.1 Identification of the factors

In the factor identification task we rely on our knowledge of the algorithms [Ferr78]. For example all the algorithms incur some concurrency control costs while synchronizing transactions. Also,a lot of probes (table lookups) have to be done in order to synchronize transactions or permit access to database objects. We also know that the number of active transactions (multiprogramming level) has some effect on throughput.

Concurrency control costs, searching costs and multiprogramming level are thus identified as factors that could contribute significantly to throughput performance.

4.5.2 Selection of level for each simulation run

The values (or levels) of each factor are selected such that the levels adequately cover the range of plausible variability of that factor.

Since our goal is to compare different algorithms we use non-factorial approach [Ferr78], that is one-factor-at-a-time approach.

This approach requires variation of one factor at a time while other two factors are kept constant. The experiment is thus broken down into three distinct experiments, each having as its objective the study of the effects of one factor. Each of the three experiment,in turn has as many runs as there are values (or levels) of the factor under consideration.

4.5.3. Concurrency Control and Searching Costs

To evaluate the concurrency control algorithms fairly and determine how their searching actions and blocking/restart decisions affect performance some assumptions are made about their concurrency control and searching costs. This section will briefly describe how the CC-CPU-time, CC-IO-TIME, TAB-CPU-time and TAB-IO-time are used in modeling the costs for each of the algorithms in this thesis.

For Semantically Consistent Schedules (SCS) algorithm a CPU cost of CC-CPU-time and an I/O cost of CC-IO-time are assessed each time the algorithm makes a read or write concurrency control request. So in the absence of restarts the CPU and I/O concurrency control cost for a transaction are (Nsr + Nsw) x CC-CPU-time and (Nsr+Nsw)x CC-IO-time where Nsr and Nsw are the number of read objects and write objects requiring synchronized access using SCS.

In addition,the algorithm will also incur a compatibility set table lookup cost of TAB-CPU-time and TAB-IO-time for each probe. For simplicity and since the number of compatibility set entries is small a linear search is used.

So, for a table of ntab entries a transaction requires an average of ntab/2 probes and therefore an average cost of (ntab/2)x TAB-IO-time and (ntab/2)x TAB CPU-time.

For Transaction Classes (TC) algorithm a CPU cost of CC-CPU-time and an I/O cost of CC-IO-time are also assessed. In the absence of restarts (Ntr+Ntw)xCC-CPU-time and (Ntr+Ntw)xCC-10-time concurrency control costs are incurred respectively with write related costs NtwxCC-CPU-time and NtwxCC-

IO-time being charged together at transaction commit time thus modeling the fact that it is a timestamp based algorithm;
where Ntr and Ntw are the number of read and write objects requiring synchronized access using TC.

The Transaction Classes algorithm also incurs a transaction class table look up cost of TAB-CPU-TIME and TAB-IO-TIME for each probe. Assuming for simplicity that the number of classes in TC algorithm is the same as the number of compatibility sets in SCS algorithm, SCS also incurs an average table probe cost of (ntab/2)x TAB-IO-time-and (ntab/2)x TAB-CPU-time.

For Integrated Transaction Characteristic (ITC) algorithm a CPU cost of CC-CPU-time and an I/O cost of CC-IO -time are again assessed each time the algorithm makes a read or write concurrency control request. Hence for objects requiring synchronized access, ITC algorithm incurs (Ncr+ Ncw)x CC-CPU-time and (Ncr ± Ncw)x CC-IO-time for CPU and I/O operations.

/2)x TAB-IO-time and (ntab/2)x TAB-CPU-time table look up costs as the TC algorithm plus additional table look up costs of (ntabl/2)xTAB-IO-time and (ntabl/2)xTAB-CPU-time for compatibilty set within a class where ntabl is the number of entries for a compatibilty set within a class.

It is assumed that the unit costs for concurrency control operations, CC-IO-time and CC-CPU-time for the three algorithms are all the same. It is also assumed that the TAB-IO-time and TAB-CPU-time are of the same magnitude. This is reasonable since the basic steps in each algorithm only involve doing simple table look-ups or probes per request.

4.5.4. Simulation Output data analysis

Due to the variability of workload, each individual performance index is very often unreliable. Usually, our observations of a simulator's behaviour cannot be considered independent and identically distributed. We can therefore not apply central limit theorems to estimate the variance of the estimators based on our observations.

These variances about the unknown true means are needed to calculate the accuracies of the estimates of our performance indices. When all the estimates reach their desired accuracies, the run can be stopped. Variance reduction methods (e.g. autocovariances, indepedent replications, regenerative, batch means) could be applied. We use the option of batch means in this thesis for the following reasons. First, due to a lack of exponential service times and the fact that 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 from preliminary experiments that this state does not occur with sufficient frequency to permit use of the regenerative method. Second, if the run is long enough and the sub-runs (batches) are about 40, batch means has the advantage over independent replications that initial transients do not bias each of the throughput observations [Law82]. Finally, a practical programming difficulty made the use of batch mean simpler than the method of independent replications as independent replications method requires starting the simulator 'all over'. This implies that 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 into a set of n_b individual batches or sub-runs each of which is t_b 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 [Law82, Saue81, Ferr78]. Two 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 the batches are not correlated.

Appendix 2 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.

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. Small confidence intervals are achievable only when "reasonably large" number of sucessful transaction commits is contained in the overall simulation run, as otherwise the variance in throughput results from the batches will be too large. Preliminary experiments in this study seem to indicate that 2000 or more commits are desirable.

The results in this chapter are obtained using $t_b = 50,000$ and $n_b = 40$ or a total of 2,000,000 simulation time unts as described in Appendix 2. These settings were selected based on confidence interval results obtained from preliminary experiments and also from suggestions in [Law82].

4.6. Assumptions

The results of this study are subject to the limitations of the models used to obtain them. There are a number of assumptions inherent in the performance model and simplifications purposely for ease of implementation. Obviously each of these assumptions and simplifications must have had, to some extent, an influence on the results.

Some of the underlying assumptions and simplifications include:

(a) transactions do not pause during their executions

(b) the cost of processing a collection of concurrency control or search requests is proportional to the size of the collection.

(c) READ-ONLY and READ/WRITE semantic types are representative of most transaction types.

(d) due to only READ-ONLY and READ/WRITE semantic types, compensating actions are unnecessary and therefore unimplemented.

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

(f) the overhead associated with switching contexts from one transaction to another is not large

(g) each object read by a transaction is read only once and all objects written must have been read previously.

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 is probably most important for transaction processing environments in which high throughputs are required.

The second assumption may be restated as the assumption that the overhead of concurrency control routine or search routine call is not the dominant factor in the cost of concurrency control or search request processing. Otherwise the simple cost modeling approach taken in the simulations, based on the CC-IO-time, CC-CPU time, TAB-IO-time and TAB-CPU-time may have to be modified.

The third and fourth simplifications imply that only READ and WRITE operations were recognised in the implementation. This suggests the possibility of other operations like DELETE and INSERT in some real systems. The introduction of additional operations would necessitate implementation of compensating actions in Semantically Consistent Schedules and Integrated Transaction Characteristics algorithms instead of restarting transactions. This could affect throughput performance of the two algorithms and thus the results obtained may not be applicable in an environment with DELETE and/or INSERT operations.

The fifth and sixth assumptions have to do with the costs of restarting and blocking. The fifth assumption 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 write set. The sixth 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-time and CC-CPU-time.

If these assumptions are modified, so that restarts are nearly free and the cost of blocking (and context switching) is very high in comparison, it is expected that the results would come out differently. 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 "blindwrites". Both assumptions were made for convinience in generating and manipulating transaction read and write sets in the simulator. It is not expected that changing either of these assumptions would lead to major changes in the results.

4.7. Program Development

In the absence of simulation languages locally, the simulator is written using Fortran77. The lack of appropriate simulation language inevitably results in the development of a lot of routines some of which are used to handle special tasks such as queue management which otherwise could have perhaps been more efficiently handled with special purpose simulation languages.

The simulator consists of six main modules each of which consists of several subroutines. The six main modules are all centrally controlled by a control program CNTPRG.

The modules include

Initialisation Routine

Start routine

Request routine

Search routine

Commit routine and

Update routine

The initialisation routine initializes all algorithm-dependent data structures and variables. It also loads transaction table, transaction classes table and semantic type/compatibility sets table.

Start routine is called whenever a transaction is to be started. It randomly determines which transaction to start and calls search routine to pick the transaction from transaction table. The newly started routine is then made to join the concurrency control queue. It also assigns timestamp to transactions as may be required.

The request routine is invoked when there are reads or writes to be done. It returns cost information about the resources utilised e.g. units of simulation to charge for CPU and I/O associated with processing the concurrency control request. It also returns concurrency control decisions e.g. access block, restart, update, commit which determine which subroutine to invoke next. The request routine is responsible for checking concurrency control data structures such as transaction classes table, semantic type/compatibility sets table and lock table.

The search routine is called by start and request routines. It picks required information about transactions on relevant tables and return the information to the caller. It also returns cost information associated with table lookups.

When the transaction arrives in the concurrency control queue after finishing its last request (with 'commit' decision) the commit routine is called. The routine is responsible for validating the correctness or otherwise of the objects read or updated using the timestamps of the transactions initiated since the committing transaction started running. Again, the routine returns cost and concurrency control decision to CNTPRG. If a transaction completes and it has written some object during execution, update routine is called after the transaction has committed and has written its deferred updates. The routine releases transaction locks if any in lock based methods. It also returns cost information to CNTPRG.

The number of read-only transactions that sucessfully commit and the number of read-write transactions that sucessfully commit and update is accumulated by CNTPRG. The number of commits and simulation time are recorded for various parameter settings and throughput is computed.

4.8. Experiments and Results

Results of three different performance experiments are reported below. Each of these experiments is performed on the three concurrency control algorithms. The experiments are designed to investigate the relative performance of the various algorithms, in hopes of identifying algorithm whose performance is either uniformly superior to that of other two or whose performance is superior under some set of reasonable conditions.

The first experiment investigates the effect which the level of multiprogramming has on performance of the algorithms.

The second experiment investigates the effect of concurrency control cost on the performance of the algorithms while the third experiment examines the effect of searching overhead on performance.

4.8.1. Multiprogramming level

This experiment investigates the effect of multiprogramming level on performance of the algorithms.

The parameters varied in this experiment are the granularity of database and number of transactions (multiprogramming level). Since concurrency control requests are made for granules rather than objects, varying the granularity of the database varies the probability that transactions requiring synchronization will conflict with one another. When the finest granularity is chosen, where each granule contains a single object, conflict should be rare for transactions with small readsets and writesets and many transactions are expected to run sucessfully.

But when the granularity is coarse, and in particular when the entire database is a granule frequent conflicts are inevitable for transaction that either belong to the same class or do not belong to the same compatibility set. The purpose of this experiment is to observe the behaviour of the algorithms under varying probabilities of conflicts and also to see how multiprogramming level affects this behaviour.

The system parameter settings for this experiment are given below. All simulations are run with one simulation unit interpreted as one millisecond of simulated time. With these system parameter settings, a transaction incurs a cost of 40 milliseconds disk access and 15 milliseconds of CPU time for each read or write of an object. In addition, a cost of o and 1 milliseconds would be incurred respectively for transaction classes and compatibility sets table lookup since the tables are loaded into memory at the onset. The cost associated with processing each concurrency control request is 1 millisecond of CPU time and no I/O time. A 25 millisecond random delay time is used to stagger transaction initiations.

System parameter Settings

System Parameter	Time (milliseconds)
OBJ-IO-time	40
OBJ-CPU-time	15
CC-IO-time	0
CC-CPU-time	1
TAB-IO-time	0
TAB-CPU-time	1
stagger-mean	25

Table 4.3. System parameter Setting for experiment 1

The relevant workload parameter setting for the experiment are given below. The database consists of 1,000 objects and its granularity is varied from 1 to 1,000 granules (or 1,000 down to 1 object per granule). The number of transactions is varied from 5 to 20 and all transactions are of the sizes generated during pre-analysis. Transaction numbers are randomly generated and picked from the transaction table for submission. Conflict advoidance delay mean (conflict-delay) meant to avoid cyclic restart anomaly as much as possible is 1 second.

Workload Parameters	
Nterms	Vary from 5 to 20
Dbsize	1000 objects
Gransize	Vary from 1 to 1000 objects/granule
Conflict-delay	1 Second

Table 4.4. Workload Parameters for Experiment 1

Figure 4.3. through 4.6. show throughput results in experiments 1.1. through 1.4. for multiprogramming level of 5,10, 15 and 20 respectively. The throughputs are given in units of committed transactions per second of simulated time.

The Gran column for each associated table indicates the number of granules, in the database, used for each experiment.

Gran	TC	SCS	ITC
1	7.890	8.164	9.063
5	9.506	10.312	11.489
10	10.748	11.105	11.982
50	10.811	11.412	12.023
100	11.428	11.474	12.383
500	11.451	11.483	12.395
1000	11.505	11.504	12.424

Table 4.5. Throughput experiment 1.1.

(multiprogramming level = 5).

Experiment 1.1 investigates the performance behaviour of the three algorithms when multiprogramming level is 5. The experiment shows that throughputs in the three cases are very close, with Integrated Transaction Characteristics algorithm outperforming the other two.

The Semantically Consistent Schedules algorithm performs slightly better than the Transaction Classes algorithm when granularity is coarse. But as granularity is made finer the Transaction Classes algorithm exhibits a performance edge over the Semantically Consistent Schedules algorithm.

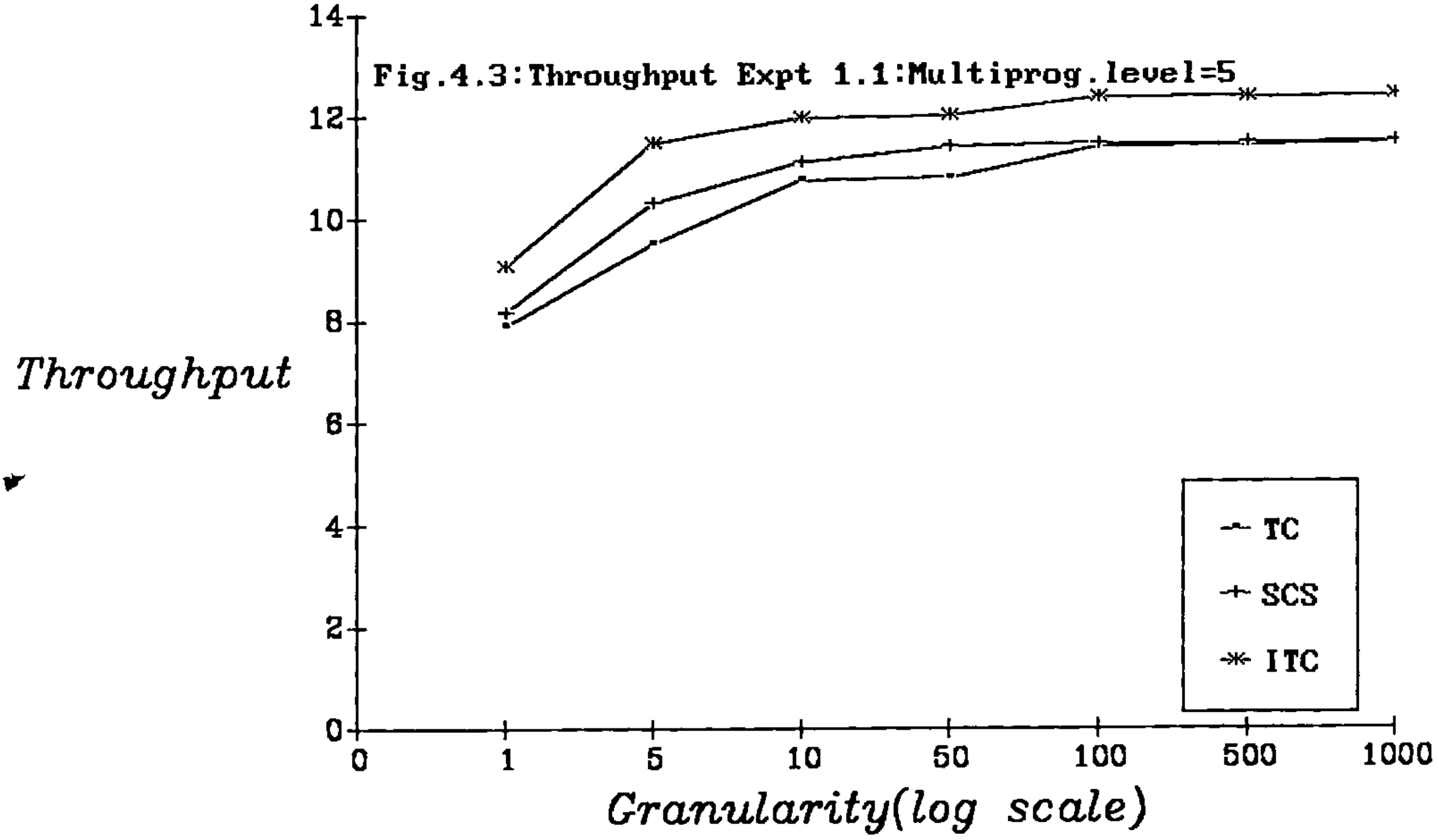
Fig.4.3:Throughput Expt 1.1:Multiprog.level=5

In this experiment, the closeness in the performance of the algorithms could be attributed to low probability of conflicts since transactions are just 5. Also the additional costs incurred as a result of restarting has to be borne by the Transaction Classes algorithm. This cost is more pronounced when granularity is coarse.

Gran	TC	SCS	ITC
1	2.596	3.709	4.411
5	3.988	4.898	6.204
10	5.120	5.890	6.985
50	6.108	6.723	7.891
100	6.907	7.067	8.050
500	7.018	7.112	8.126
1000	7.139	7.250	8.163

Table 4.6: Thoughput experiment 1.2

(multiprogramming level = 10

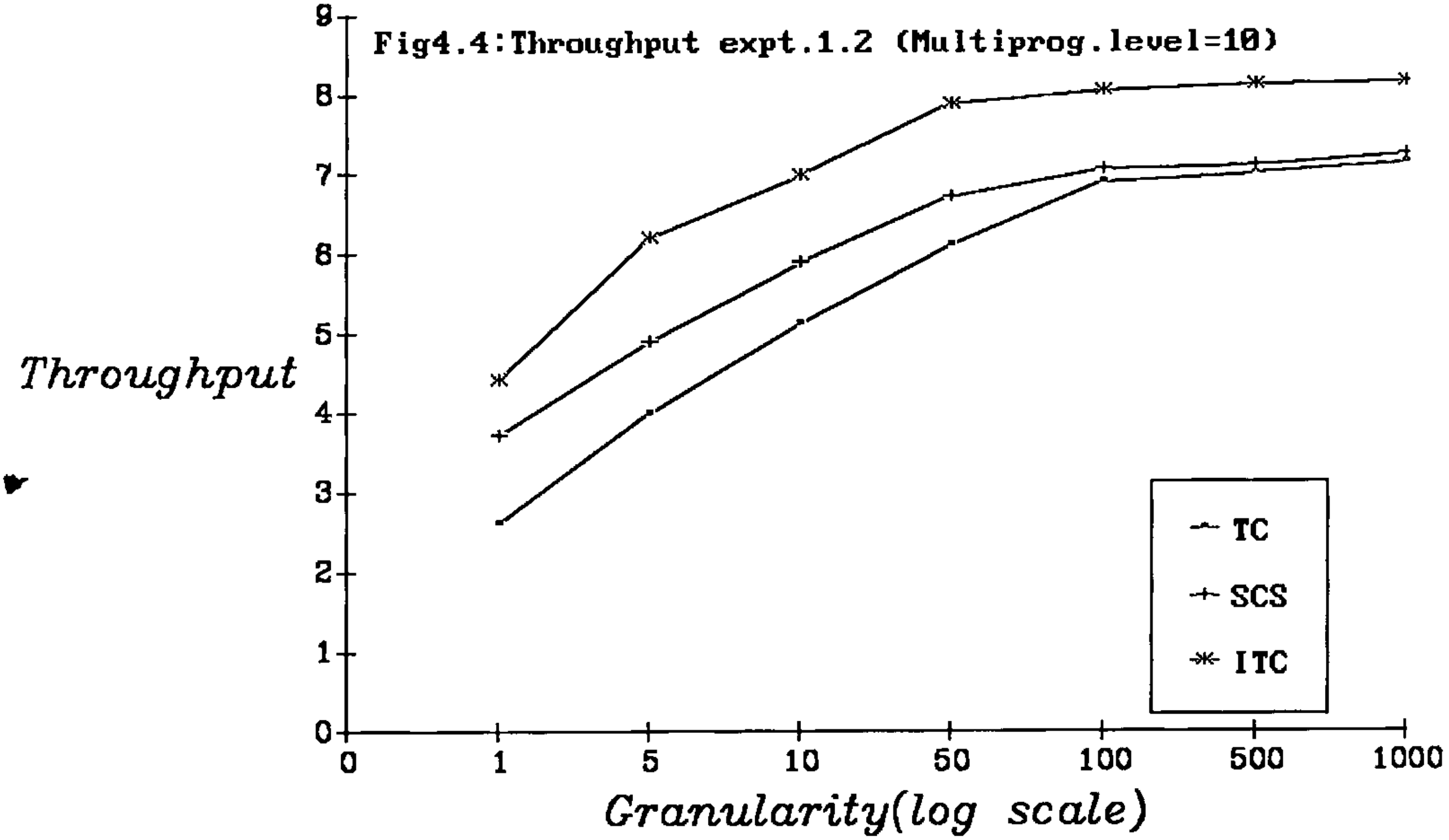

Fig4.4:Throughput expt.1.2 (Multiprog.level=10)

Experiment 1.2 checks throughput of the algorithm while the multiprogramming level is 10. Again, there is a noticeable improvement in the throughput as granularity becomes finer. But here, relative to results of experiment 1.1, the throughputs generally decrease. Also, the Transaction Classes algorithm is consistently worst while the Integrated Transaction Characteristics algorithm maintains its lead. The difference in behaviour could again be attributed to the conflict resolution peculiarities of the algorithms.

Gran	TC	SCS	ITC
1	.280	.809	1.859
5	.314	.912	1.934
10	.517	1.166	2.057
50	1.200	1.304	2.311
100	2.342	2.704	3.934
500	2.581	2.813	4.056
1000	3.359	3.394	4.431

Table 4.7: Throughput experiment 1.3

(multiprogramming level = 15)

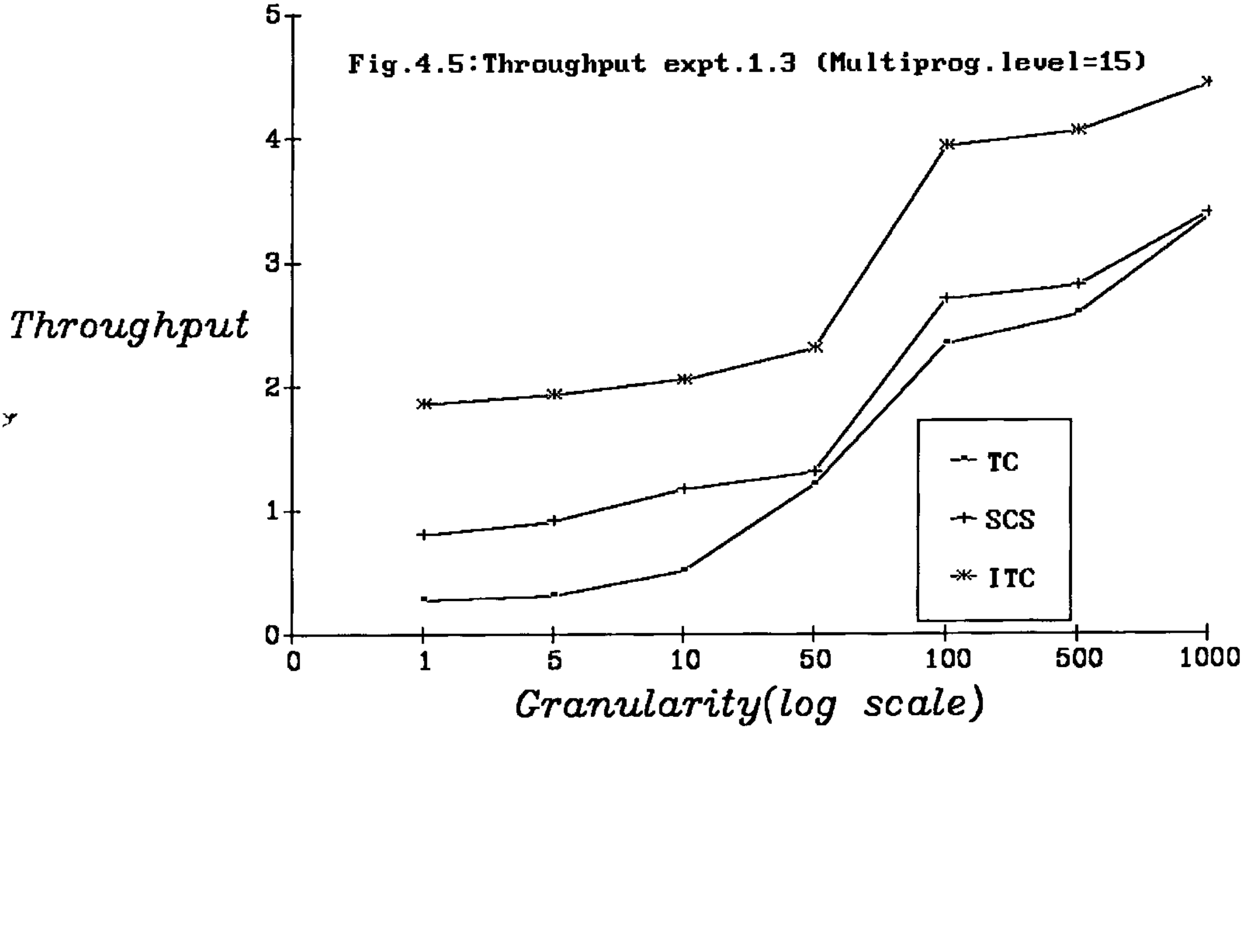

Fig.4.5:Throughput expt.1.3 (Multiprog.level=15)

Experiment 1.3 sets multiprogramming level to 15. A sharp drop in the throughputs of the methods is noticed in this experiment. Relatively the Integrated Transaction Characteristics algorithm still leads while the semantically consistent schedules algorithm follows. This experiment perhaps shows that quite apart from conflict probability, the resources of the system is beginning to be stretched close to its limit.

Gran	TC	SCS	ITC
1	.104	.351	.393
5	.104	.312	.327
10	.106	.104	.184
50	.241	.511	.433
100	.302	.802	.939
500	.696	1.019	1.312
1000	1.574	1.700	1.782

Table 4.8: Throughput Experiment 1.4

(Multiprogramming level = 20)

For once, while the multiprogramming level is 20, the Semantically Consistent Schedules algorithm performs better than Integrated

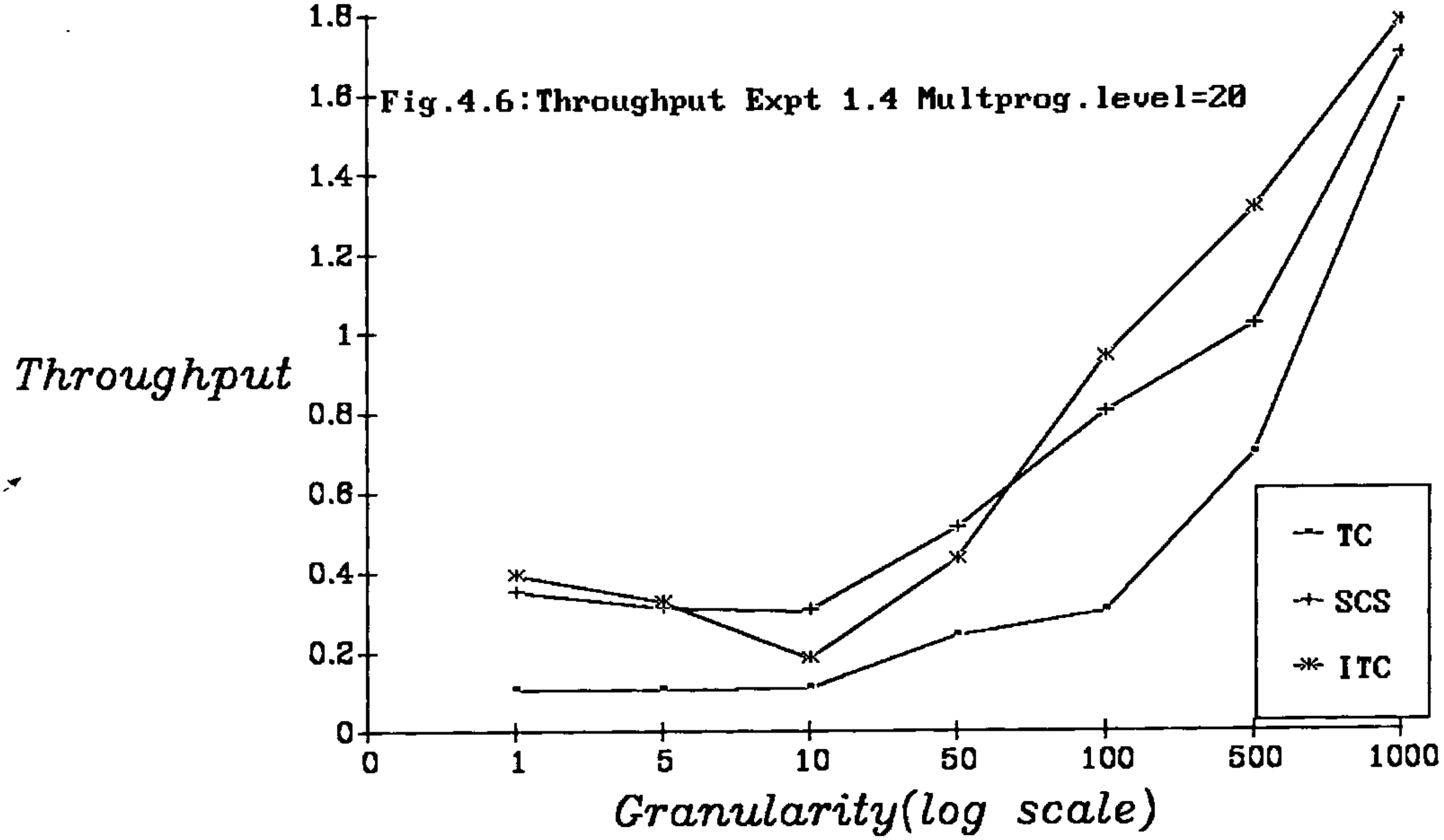

Fig.4.6:Throughput Expt 1.4 Multprog.level=20

Transaction Characteristics algorithm when graularity is 10 or in the neighbourhood of 10.

The curves get most depressed in this experiment, thus showing that as the multiprogramming level is increased the probability of conflict gets higher and resource contention gets worse.

4.8.1.1 Summary of results of the multiprogramming level Experiments.

Perfomance of each algorithm improves with increase in the number of granules whereas their perfomances steadily decrease with increase in multiprogramming level. This result is corroborated by intuition and various results in performance literature [Ries77, Garc78, Tay84a]. Also, the graphs are similar to a segment of the general granularity curve reported by Tay [Tay84a].

The Integrated Transaction Characteristics algorithm performs uniformly better than the other two algorithms in all the experiments except when multiprogramming level becomes large and the throughputs of the three algorithms become very close. In particular, when multiprograming level is 20 and number of granules is about 10 the Semantically Consistent Schedules algorithm edges out the Integrated Transaction Characteristics algorithm. The performance of Transaction classes algorithm is in general worse than that of Semantically Consistent Schedules algorithm.

The poor performance of Transaction Classes algorithm is attributable to its conflict resolution technique. Its conflict resolution technique is time stamp based, it therefore resolves conflicts by restarting transactions that have already incurred some costs. Another explanation of its worst performance is that, under a high probability of conflict (when granularity is coarse and/or level of parallelism is high), cyclic retarts [Date82] may occur.

This anomaly is illustrated in figures 4.6 and 4.7

```
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 4.6 Cyclic restart transactions

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 4.7: Example of cyclic restart anomaly

Figure 4.6 shows a pair of transactions both reading from object X and writing into object X, which make the two transactions prone to cyclic restart anomaly.

Figure 4.7 demonstrates how these transactions can be involved in an infinite cycle, restarting each other repeatedly and incurring costs. Only the first ten steps in the infinite interleaving cycle are shown in the figure.

The problem with the two transactions is that they are reading and attempting to write the same granule. If they attempt to interleave execution in the manner shown in Figure 4.7, performing their reads in time stamp 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 T1 is restarted at step 5 because X has been read by a younger transaction T2. At this point,T1 actually becomes younger than T2 and re-reads X. This dooms T2's subsequent write to end in a restart. If the computation delay between the read and write of X exceeds the conflict avoidance delay,that is the delay from the time of a restart to the time of re-reading X for both transactions T1 and T2, this pattern can indeed persist forever.

4.8.2 Concurrency Control Cost

This experiment investigates the effects of concurrency control costs on the result of experiment 1.2. The experiment is repeated with concurrency control parameters modified to investigate the effects of alternative concurrency control costs. The system parameter and workload parameter settings are as shown in Table 4.9 and 4.10 respectively.

SYSTEM PARAMETER SETTINGS	
System Parameter	Time (Milliseconds)
OBJ-10-time	40
OBJ-CPU-time	15
CC-10-time	Vary from 0 to 40
CC-CPU-time	Vary from 0 to 10
TAB-10-time	0
TAB-CPU-time	1
Stagger-mean	25

Table 4.9 : System Parameter Settings for Experiment 2

WORKLOAD	PARAMETERS
Dbsize	1000 objects
Granssize	Vary from 1 to 1000 objects/ granule
Nterms	10
Conflict-delay	1 second

Table 4.10: Workload Parameters for Experiment 2.

Figure 4.8 through 4.10 show throughput results in experiments 2.1 through 2.4 with respect to varying CC-IO-time and CC-CPU-time.

Gran	TC	SCS	ITC
1	2.609	3.818	4.488
5	3.988	5.212	6.193
10	5.252	6.416	7.469
50	5.807	6.736	8.012
100	6.918	7.069	8.051
500	6.932	7.117	8.153
1000	7.140	7.252	8.163

Table 4.11: Throughput Experiment 2.1

(CC-CPU-time=0,CC-10-time=0)

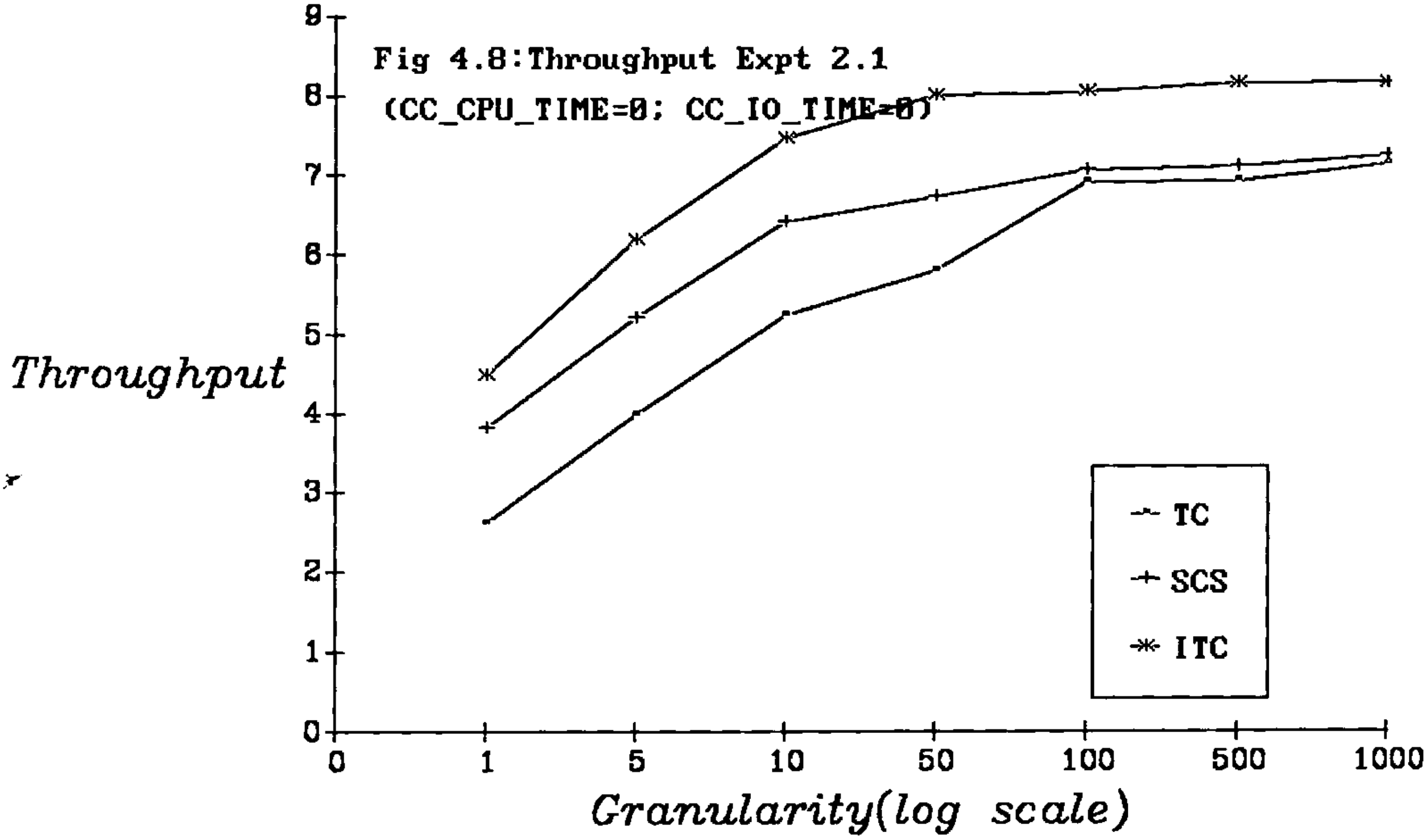

Fig 4.8:Throughput Expt 2.1
(CC_CPU_TIME=0; CC_IO_TIME=0)

Experiment 2.1 investigates a situation where both CC-IO-time and CC-CPU-time are very small and therefore could be regarded as being zeros.

Except for some minor variations when granularity is coarse (few objects per granule) the throughputs and hence the behaviour of the algorithms are similar to results of experiment 1.2.

This result suggests that the CC-CPU-time has little or no effect on the relative behaviour of the algorithms. This observation is further corroborated by the result of experiment 2.2 below.

In experiment 2.2 the CC-CPU-time is 5. The experiment is intended to check results in 2.1 and 1.2 vis a vis concurrency control CC-CPU-time cost. The results in 1.2, 2.1 and 2.2 are nearly identical. This indicates that the effects of the cost of concurrency control could be negligible as long as concurrency control overhead is small compared to the costs associated with object accesses. This observation is checked out in the next two experiments by increasing concurrency control costs. The two experiments model a situation where concurrency control information is kept on disk rather than in memory, with one disk access being required per conccurency control request processed.

GRAN	TC	SCS	ITC
1	2.611	3.818	4.489
5	4.001	5.193	6.200
10	5.250	6.411	7.456
50	6.203	6.334	7.837
100	6.709	7.070	8.051
500	6.812	7.122	8.094
1000	7.142	7.252	8.162

Table 4.12: Throughput Experiment 2.2

(CC-CPU-time=5, CC-10-time=0)

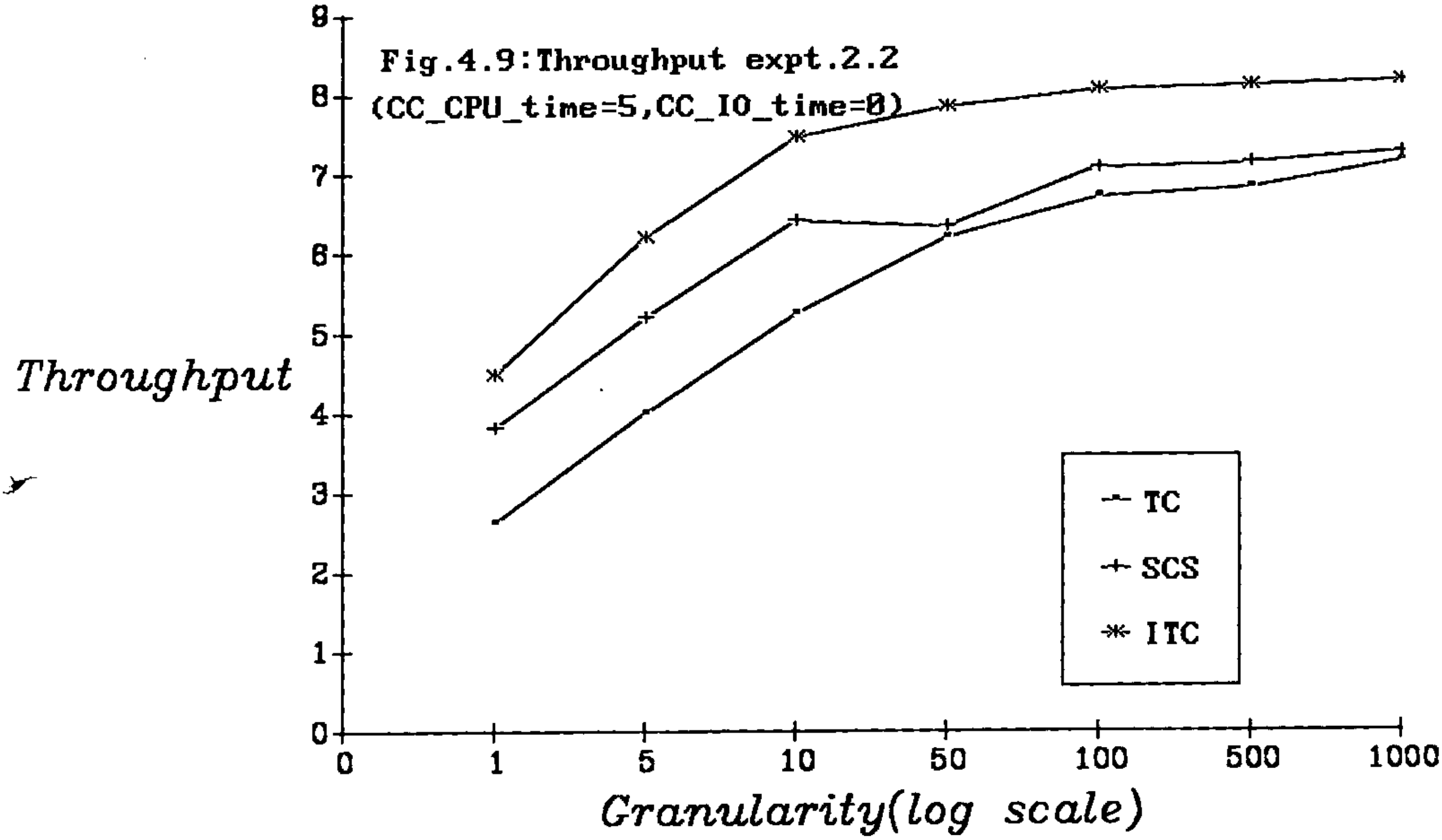

Fig.4.9:Throughput expt.2.2
(CC_CPU_time=5,CC_IO_time=0)

Table 4.13 and Figure 4.10 show the result of experiment 2.3.

This result is interesting in more than one way. First, it confirms the observation in experiments 1.2, 2.1 and 2.2 that the effect of Concurrency Control cost is significant with respect to throughput when its cost is very close to object access costs. Secondly, the throughputs take a dip when the granularity is more than 100. An implication of this is that a lot more Concurrency Control information is required at finer granularities and if access to this information is expensive throughput would inevitably drop.

GRAN	TC	SCS	ITC
1	2.599	3.019	4.415
5	4.000	4.234	5.924
10	5.223	6.013	7.112
50	5.998	6.036	7.530
100	6.107	6.904	7.635
500	4.982	6.112	6.497
1000	3.703	3.698	4.011

Table 4.13: Throughput Experiment 2.3

(CC-CPU-time=1, cc-Io-time=40)

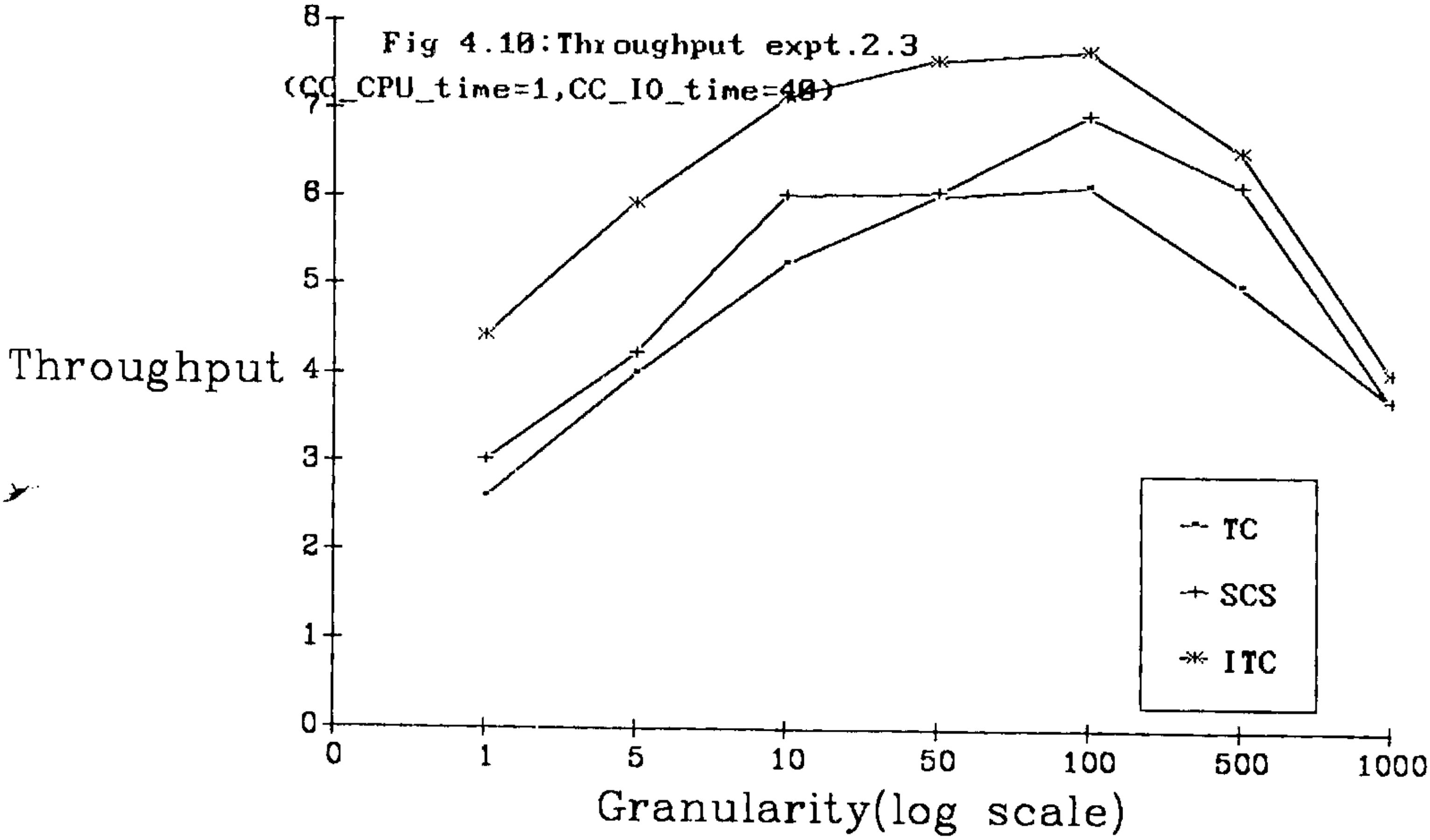

Fig 4.10:Throughput expt.2.3
(CC_CPU_time=1,CC_IO_time=40)

Gran	TC	SCS	ITC
1	2.587	3.017	4.201
5	3.665	4.672	5.723
10	5.101	5.989	6.910
50	5.213	6.181	7.006
100	5.997	6.781	7.513
500	3.813	5.529	6.102
1000	3.702	3.577	3.934

Table 4.14: Throughput Experiment 2.4

(CC-CPU-time = 10; CC-10-time=40)

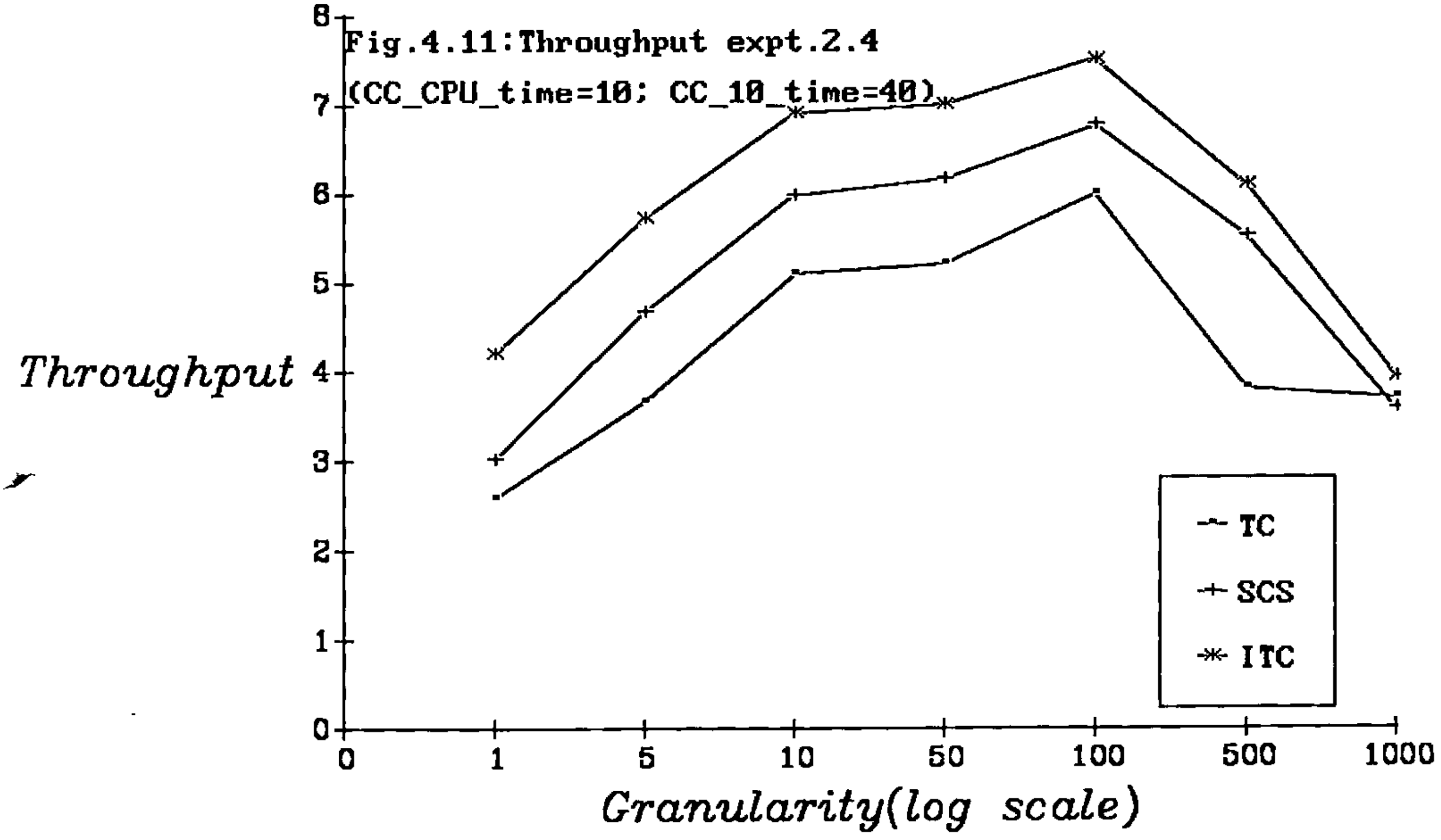

Fig.4.11:Throughput expt.2.4
(CC_CPU_time=10; CC_10_time=40)

Table 4.14 and Figure 4.11 illustrate the result of experiment 2.4. The relative performance is similar to that of experiment 2.3. A striking difference is that the throughput has generally decreased.

4.8.2.1 Summary of result of Concurrency Control experiments.

There is a general decrease in the performance of each algorithm with increase in concurrency control cost. The Integrated Transaction Characteristics algorithm is still the best.

The CC-CPU-time Concurrency Control cost has little or no effect on the relative performance of the algorithm while an increase in the CC-IO-time reduces the throughput of the algorithms.

It is also observed that the effect of Concurrency control cost is negligible in performance determination provided concurrency control overhead is small compared to the cost associated with object accesses.

4.8.3 Searching Costs

This experiment investigates the effects of searching costs on throughputs of the algorithm. Experiment 1.2 is repeated with TAB-IO-time and TAB-CPU-time parameters modified. The system parameter and workload parameter settings are as follow.

SYSTEM PARAMETER SETTINGS	
System Parameter	Time (Milliseconds)
OBJ-10-time	48
OBJ-CPU-time	15
CC-10-time	0
CC-CPU-time	1
TAB-10-time	Vary from 0 to 40
TAB-10-time	Vary from 0 to 10
Stagger-mean	25

Table 4.15: System parameter settings for experiment 3

WORKLOAD	PARAMETER
Dbsize	1000 objects
Gransize	Vary from 1 to 1000 object/granule
Nterms	10
Conflict-delay	1 second

Table 4.16: Workload parameters for experiment 3.

Figures 4.12 through 4.15 show throughput results in experiments 3.1 through 3.4.

Experiment 3.1 investigates the performance of the algorithms when no cost is charged for searching class or compatibility set matches. The result of the experiment is identical to those of experiments 1.2, 2.1 and 2.2. This implies that searching overhead, like concurrency control overhead is not significant whenever searching cost is small compared with the cost of object accesses.

Gran	TC	SCS	ITC
1	2.718	3.819	4.511
5	4.344	5.481	6.963
10	5.247	6.443	7.472
50	6.222	7.003	7.853
100	7.103	7.069	8.057
500	6.950	7.103	8.100
1000	7.141	7.257	8.163

Table 4.17: Throughput experiment 3.1

(TAB-IO-time=0, TAB-CPU-time = 0)

Experiment 3.2 is performed with cost associated with CPU set to 5. Except for some minor differences in throughput the result of this experiment is similar to those of experiments 1.2, 2.1, 2.2 and 3.1. Experiment 3.2 further confirms that searching overhead is not significant whenever searching cost is small compared with cost of object accesses.

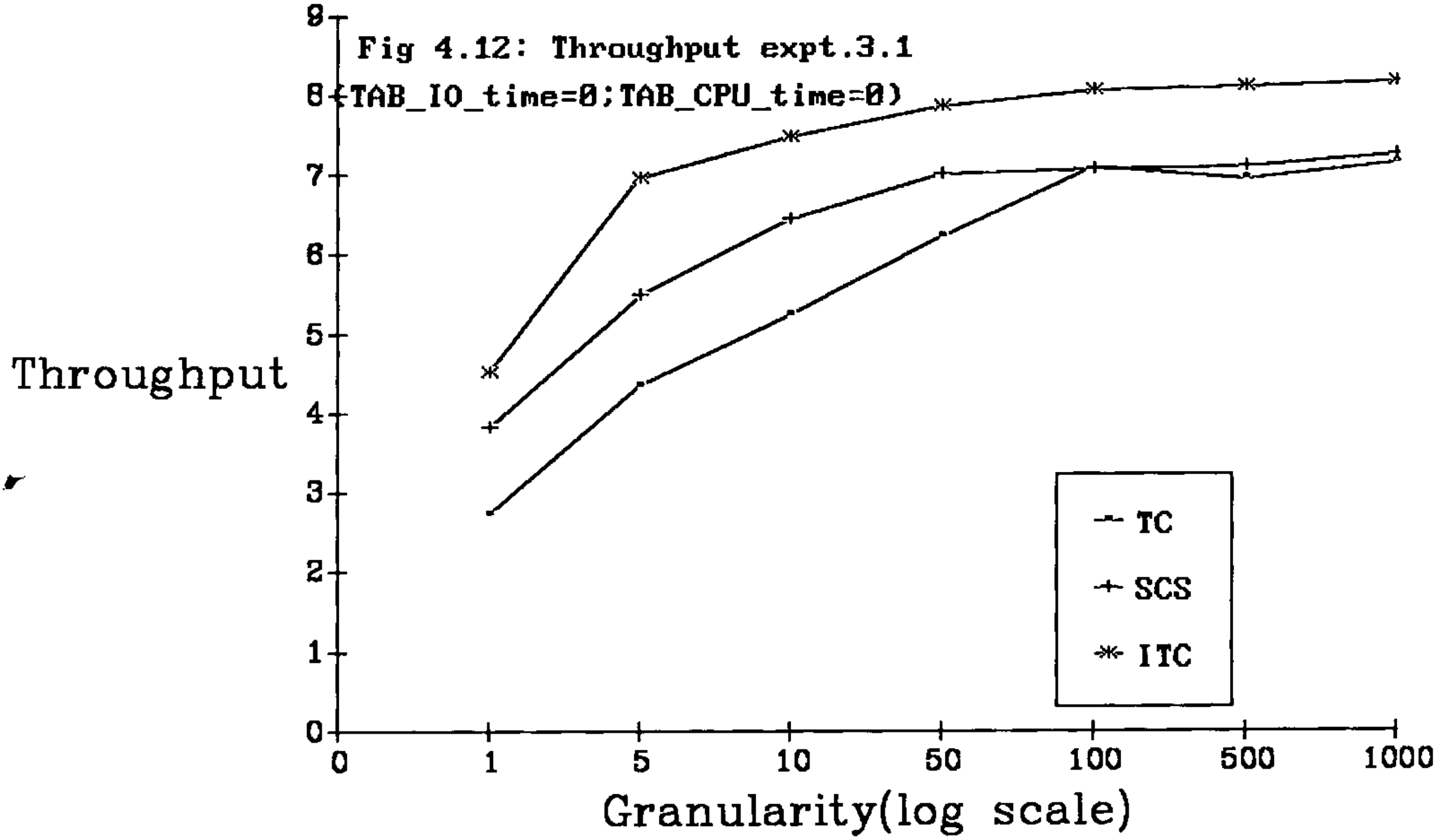

Fig 4.12: Throughput expt.3.1 (TAB_IO_time=0;TAB_CPU_time=0)

Gran	TC	SCS	ITC
1	2.716	3.820	4.314
2	4.212	5.739	7.202
10	5.239	6.441	7.461
50	5.744	7.001	7.935
100	6.933	7.069	8.003
500	7.107	7.116	8.092
1000	7.141	7.258	8.102

Table 4.18: Throughput Experiment 3.2

(TAB-IO-time = 0; TAB-CPU-time = 5)

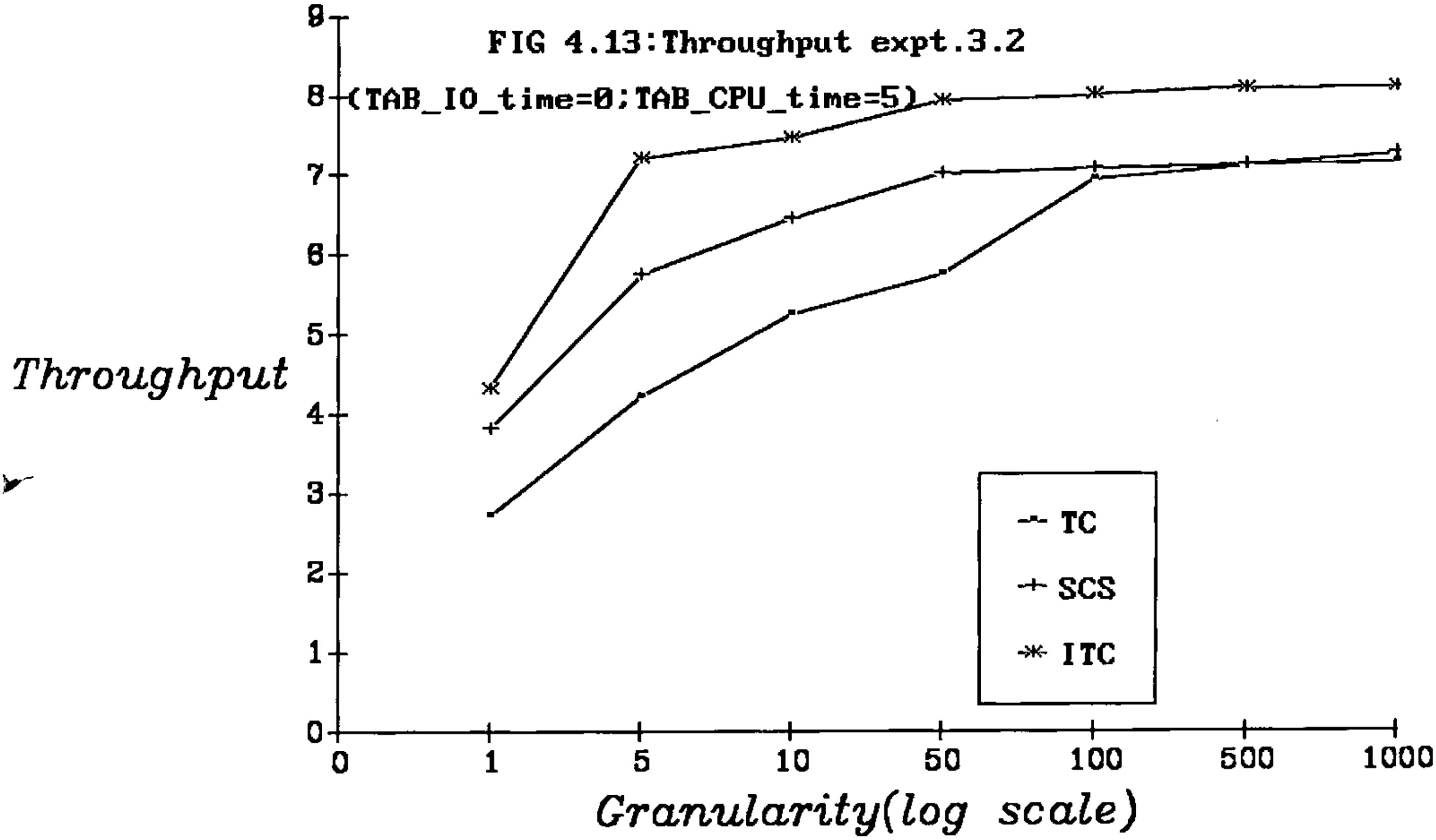
FIG 4.13:Throughput expt.3.2
(TAB_IO_time=0;TAB_CPU_time=5)
Throughput
9
8
7
6
5
4
3
2
1
0
0
1
5
10
50
100
500
1000
Granularity(log scale)
TC
SCS
ITC

Table 4.19 and Figure 4.14 show the results of experiment 3.3. Searching overhead has an effect on throughputs of the three algorithms. But the most significant result here is that the Integrated Transaction Characteristics algorithms for the first time is out-performed by the Semantically Consistent Schedules algorithm. This perhaps shows that the number of table probes could adversely affect throughputs especially when the table is large and resident on a secondary storage. Another observation here as in experiment 2.3 and 2.4, is that the throughputs peaked between 10 and 100 granules and dropped thereafter.

Gran	TC	SCS	ITC
1	1.913	3.107	2.112
5	2.426	3.410	2.561
10	3.235	3.946	3.127
50	3.364	4.372	3.489
100	3.721	4.801	3.943
500	3.213	4.013	3.217
1000	2.614	3.310	2.633

Table 4.19: Throughput Experiment 3.3

(TAB-IO-time=40; TAB-CPU-time=1)

Table 4.20 and Figure 4.15 show the results of experiment 3.4.

There is a general drop in the performance

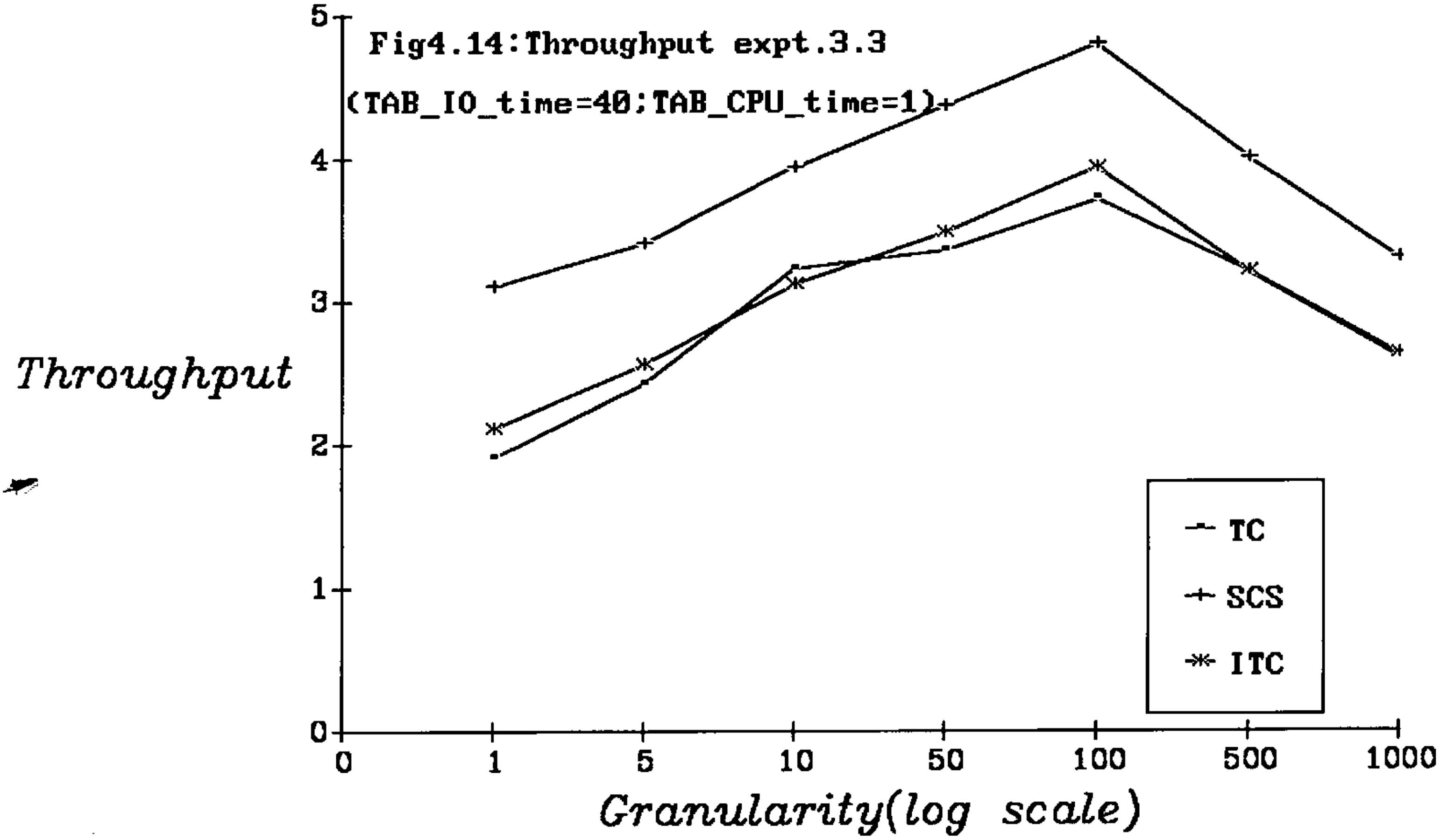

Fig4.14:Throughput expt.3.3

(TAB_IO_time=40;TAB_CPU_time=1)

of the algorithms with throughput at their maximum when granularity is about 100.

The Integrated Transaction Characteristics algorithm still lags behind (in terms of performance) when compared with the other two algorithm between granules 1-50. Between granules 100-1000, the order of performances is SCS, ITC, and TC.

Gran	TC	SCS	ITC
1	1.891	3.107	2.109
5	2.903	3.774	2.856
10	3.231	3.945	3.126
50	3.804	4.500	3.620
100	3.719	4.796	3.738
500	3.111	3.903	3.444
1000	2.513	3.295	2.620

Table 4.20: Throughput Expeperiment 3.4

(TAB-CPU-time=10; TAB-10-time=40)

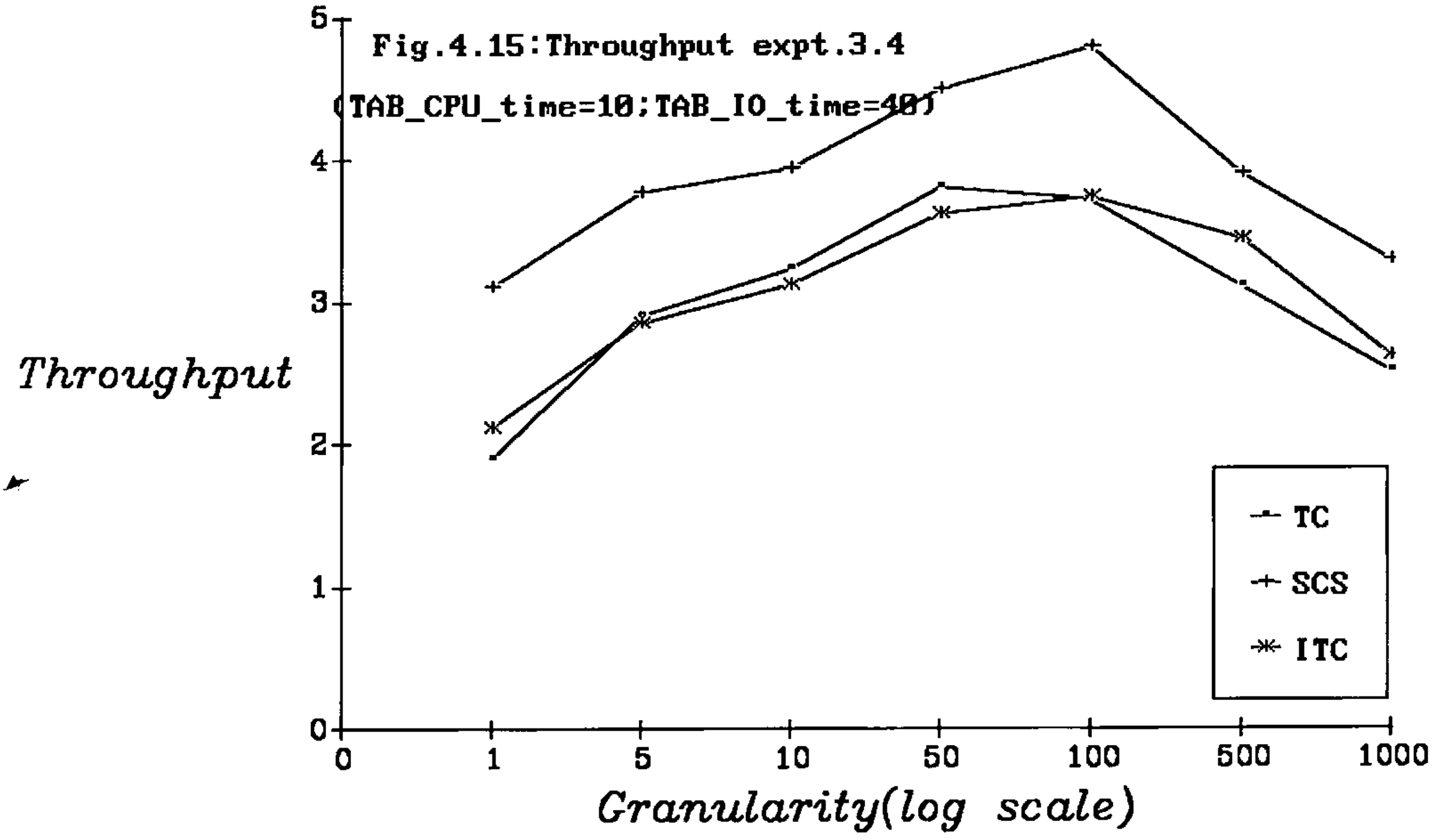

Fig.4.15:Throughput expt.3.4

(TAB_CPU_time=10;TAB_IO_time=40)

4.8.3.1. Summary of results of the Searching Cost Experiments

Results of searching cost experiments show that an increase in searching overhead has a degrading effect on the throughput performance of the three algorithms.

The TAB-CPU-time Searching cost has little or no effect on the relative performance of the algorithms while an increase in TAB-IO-time adversely affects throughputs of the algorithms in general and the Integrated Transaction Characteristics algorithm in particular. So when the table to be searched is so large that it has to be kept on an auxiliary (secondary) storage, the throughput of Integrated Transaction Characteristics algorithm and indeed the other two algorithms could be badly affected.

It is also observed that the effects of searching cost, on performance, is negligible if the cost is small compared to the costs associated with object accesses.

CHAPTER 5

CONCLUSION

5.1 Summary of Results

Results obtained from a comparison of the costs of Transaction Classes, Semantically Consistent Schedules and Integrated Transaction Characteristics algorithm show that,

(i) Semantically consistent schedules algorithm has the smallest worst-case storage of the three algorithms. The Integrated Transaction Characteristics algorithm has the next worst-case storage while the Transaction Classes algorithm has the highest worst-case storage.

(ii) The best-case storage for Transaction Classes algorithm is smaller and therefore better than the best-case storage for either Semantically Consistent Schedules algorithm or Integrated Transaction Characteristics algorithm.

(iii) The Semantically Consistent Schedules algorithm is best in terms of worst-case storage cost, indicating that its storage requirements is superior under low-conflict transaction mixes.

Transaction classes algorithm is best in terms of best-case storage implying that it is better under high conflict transaction mixes.

(iv) CPU cost for Semantically Consistent Schedules algorithm is less than the CPU cost of Transaction Classes algorithm except when writes/reads ratio is very small and transaction class entries are equal to compatibility sets entries in which case the cost could be the same.

(v) If writes/reads ratio is very small, CPU cost for Integrated Transaction characteristics algorithm is greater than CPU cost for Transaction Classes algorithm

(vi) If writes/reads ratio is not small, the total entries for transaction classes and compatibility sets for Integrated Transaction Characteristics algorithm are sufficiently close to transaction classes entries (for example when just a compatibility set is contained in a class) then CPU cost for Transaction Classes algorithm is greater than CPU cost for Integrated Transaction

Characteristics algorithm.

In general the cost associated with Semantically Consistent Schedules algorithm is at least as low as those for Transaction Classes algorithm and Integrated Transaction characteristics algorithm.

The three algorithms were also tested under varying multiprogramming levels, concurrency control costs and searching costs.

The results of the experiments show that

(vii) Performance of each algorithm improves with increase in number of granules while their performances steadily decrease with increase in multiprogramming level.

(viii) The Integrated Transaction Characteristics algorithm performs better than the other two algorithms in all the multiprogramming level experiments. But when the multiprogramming level becomes large, the throughputs of the algorithms become very close.

(ix) Given a multiprogramming level, the performance of Transaction Classes algorithm is in general worse than that of Semantically Consistent Schedules algorithm.

(x) In general, increase in concurrency control costs degrades performance.

(xi) Concurrency control cost has little or no effect on the relative performance of the algorithms

(xii) Effects of concurrency control cost is negligible in throughput performance whenever concurrency control overhead is small compared to the costs associated with object accesses.

(xiii) The algorithm of choice under varying concurrency control costs is Integrated Transaction Classes algorithm.

(xiv) Increase in searching overhead has a degrading effect on throughput performance of the three algorithms.

(xv) The input/output cost increase for table probes have a more severe performance degradation effect than CPU cost associated with such searches.

(xvi) Effects of searching cost are negligible if the cost is small compared to the costs associated with object accesses.

(xvii) For the maintenance of performance edge of transaction knowledge based algorithms (TC,SCS and ITC) over traditional algorithms (e.g. two-phase locking, Timestamp ordering and serial validation), information about the characteristics of transactions must be as small as possible and must be kept in primary memory.

(xviii) The algorithm of choice under varying searching costs is Semantically Consistent Schedules algorithm.

A major conclusion here is that no algorithm is uniformly better than the other two. the throughputs of the algorithms depend heavily on availability of relevant resources (e.g. fast input/output devices large memory that could accommodate both concurrency control and searching information), the structure of the database and the environment (e.g. the number of concurrent transactions)

5.2 Comparison with other work

It has been reported by Eswaran et al [Eswa76] that availability and usage of transaction characteristics by concurrency control algorithms could improve throughput. This fact was used by Bernstein et al [Bern80c] and Garcia-Molina [Garc83] in proposing Transaction classes and Semantically Consistent Schedules algorithm respectively.

A comparison of the relative throughput performance of the two algorithms is hitherto non-existent in concurrency control performance literature. A major contribution of this study is the evaluation of the throughput performance of the two algorithms and the determination of a better choice which is Semantically Consistent Schedules algorithm.

In effect there is no available results in the performance evaluation of this class of concurrency control algorithms with which our results could be compared. The result should however be a template for results of subsequent performance evaluation efforts.

5.3 Future research directions

The abstract model in chapter 2 considers only the non-conflict CPU cost [Bada81] - the CPU cost experienced by a transaction which does not conflict in any way with other concurrent transaction. One area of further work is therefore a generalisation of the analysis to include the additional resources of CPU cost associated with transaction restarts and blocking.

Another area of further research is to relax the assumptions discussed in 4.6. Although it is not expected that this should significantly alter the results, it is by trying that the expectation can be confirmed.

Only sequential transaction types and well-placed granules were considered in this study, future efforts could be extended to random transaction types and random-placed granules respectively.

The results in this thesis are based on simulation experiments. An alternative is an analytic approach. This option is also left for future work.

BIBLIOGRAPHY

[AGRA83] Agrawal, R., and Dewitt, D.:"Intergrated concurrency control and Recovery Mechanisms: Design and performance Evaluation", Technical Report No. 497, Computer Sciences Department, University of Wisconsin-Madison, February 1983

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

[Bada80] Badal, D.Z.:"On the degree of Concurrency Provided by Concurrency Control Mechanisms for distributed database", North Holland Publishing Company INRIA, 1980 pp35-48.

[Bada81] Badal, D.Z.: "Concurrency Control Overhead or Closer Look at Blocking vs non-blocking Concurrency Control Mechanisms", Proceedings of the fifth Berkeley Workshop on Distributed Data Management and Computer Networks, February 1981

[Balt82] Balter, R., Berard, P., and Decitre, P.: "why control of concurrency level in distributed Systems is more fundamental than deadlock managment. Proc. 1st ACM SPGACT-SPGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada (Aug. 1982), pages 183-193.

[Beer89] Beeri, C., Bernstein, P.A. and Goodman, N.: "A Model for Concurrency in Nested Transactions Systems" Journal of the ACM April 1989 pp 230 - 269.

[Bern79a] Berstein, P. A., and Goodman, N.: "Approaches to Concurrency Control in Distributed database Systems" National Computer Conference 1979.

[Bern79b] Berstein, P. A., Shipman, D. W., Wong W. S.: "Formal Aspects of Serializability in Database Concurrency Control", IEEE Transaction on software Engineering Vol. SE - 5, No.3, May 1979.

[Bern80a] Bernstein, P. A., and Goodman, N.: "Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems" Proc. 6th Int. Conf. on very Large Data Bases oct 1980 pp 285 - 300.

[Bern80b] Bernstein, P. A., and Shipman, D. W.: "The Correctness of Concurrency Control Mechanisms in a system for Distributed databases (SDD-1)", ACM TODS, Vol 5, No.1, March 1980 pp 52-68.

[Bern80c] Bernstein, P. A., Shipman, D. W., Rothnie Jr. J. B.: "Concurrency Control in a system for distributed databases (SDD-1)", ACM TODS Vol.5

No.1, March 1980 pp. 18-51.

[Bern80d] Bernstein, P. and Goodman, N.: "Fundamental Algorithms for Concurrency Control in distributed database systems", Technical Report, Computer Corporation of America, 1980.

[Bern81] Bernstein, P. A., Goodman, N.: "Concurrency Control in Distributed Database systems" ACM Computing Surveys Vol. 13 No.2 June 1981.

[Bern82] Bernstein, P., and Goodman, N.: "Multiversion Concurrency Control Theory and Algorithms", Technical Report No. TR-20-82,Aiken Computation Laboratory, Havard University, June 1982.

[Bern84] Bernstein, P.A., and Goodman, N.: "An Algorithm for concurrency control and recovery inreplicated Distributed Data base" ACM TODS vol 9 No. 4 Dec 1984 pp 596-615.

[Bern87] Bernstein, P.A, Hadzilacos, V., and Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, Mass, 1987.

[Care83] Carey, M. : An Abstract Model of Database Concurrency control Algorithms" Proceeding of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1983.

[Care83a] Carey, M. "Granularity Hierachies in Concurrency control" Proceedings of the second ACM SIGACT-SIGMOD Symposium on principles of Database Systems, Atlanta, Georgia, March 1983

[Care84] Carey, M. Stonebraker, M: "The performance of Concurrency control algorithms for DBMSs" Proc. Int.Conf. on very Large Data Bases Singapore (Aug 1984) pp 107-118

[Date82] Date, C., An Introduction to Database systems (vol II), Addison-Wesley publishing company, 1982

[Demi77] Demillo, R.A., Vairavan, K., and Sycara-cyranski,E.,: "A study of schedules as Models of synchronous parallel Computation "JACM Vol. 24 No. 4 Oct 1977

[Denn76] Denning, P J., Kahn K. C., Leroudier, J. Potier, D., and Suri, R., "Optimal multi-Porgramming" Informatica 7, (1976) PP 197-217

[Eswa76] Eswaran, K. P., Gray, J. N. Lorie, R. A. and Traiger, I. L.: "The Notions of Consistency and Predicate Locks in a database system" CACM 19, 11 (Nov 1976) pp 624-633.

[Ferr78] Ferrari, D.: "Computer Systems performance Exaluation" Prentice-Hall, Englewood Cliffs, New Jersey, 1978

[Fran85] Franaszek, P., and Robinson, J. T.: "Limitations of Concurrency in Transaction Processing" ACM

Transactions on Database systems vol. 10 No.1 March 1985

[Garc78] Garcia-Molina, H.: "Performance Comparison of two update Algorithms for Distributed Database" in proceedings of the third Berkeley Workshop on Distributed Data Management and computer Networks Calofornia University, Berkeley (USA) Lawrence Berkeley Lab Aug 19.

[Garc82] Garcia-Molina, H., and Wiederhold, Gio,: "Read only Transactions in a Distributed Databases" ACM TODS Vol 7 No. 2 Jun 1982 pp 209-234

[Garc83] Garcia-Molina, H., "Using Semantic knowledge for Transaction Processing in a Distributed Database", ACM TODS, Vol 8 No. 2, June 1983, pp 186-213.

[Gray75] Gray, J., Lorie, R., Putzulo, G., and Traiger, I.:"Granularity of Locks and Degree 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 Advance Course, Springer-Verlag, 1979

[Gray81] Gray, J., Homan, P., Korth, H., and Obermarck, R.: "A Strawman Analysis of the probability of Waiting and Deadlock in a Database system", Report No. RJ3066, IBM San Jose Research Laboratory, Feb 1981

[Hard85] Harder, T., Peinl, P., Reuter, A: "Performance Analysis of Synchronization and Recovery Schemes" IEEE Transactions in Database Engineering, June 1985 pp 50-57.

[Heun78] Heuner, A. R., Yao, S.B.: "Query Processing on a Distributed Database" Proceedings of the third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley (USA) Aug 1978.

[Hull87] Hull, R. and King, R.: "Semantic Database Modelling: Survey, Applications and Research Issues" ACM Computing Surveys Vol. 19, No. 3, Sept. 1987 pp 201-260.

[Ibar83] Ibaraki, T., Kameda T., and Minoura, T.: "Disjoint Interval topological sort: a useful concept in Serializability" in proceedings 9th International Conference on VLDB Oct/Nov 1983.

[Isao] Isao, Miyamoto: "Hierarchical Performance Analysis Models for Database systems" Nippon Electric Company Ltd. Tokyo Japan.

[Kato85] Katoh. N., Ibaraki, T., and Kameda, T.: Cautious Transactions schedulers with Admission Control" ACM TODS Vol 10 No. 2 June 1985 pp 205-209

[Kenn73] Kennedy, S. R. "The Use of Acess frequencies in database organisation

Ph.D. Thesis, Cornell University, 1973.

[Klug83] Klug, A.: "Locking Expressions for Increased Database Concurrency" JACM Vol 30 No. 1 Jan 1983 pp 36-54.

[Knap87] Knapp, E.,: "Deadlock Detection in Distributed Database" ACM Computing Surveys, Vol.19 No. 4 Dec. 1987

[Kohl81] Kohler, W. H.: "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems" ACM Computing Surveys Vol. 13 No. 2, June '81.

[Kort82] Korth, H. F.: "Deadlock Freedom Using Edge Locks" ACM TODS Vol. 7 No. 4 Dec 1982 pp 632-652

[Kort83] Korth, H. F.: "Locking Primitives in a Database System" JACM Vol 30 No. 1 Jan 1983 pp 55-79

[Kung81] Kung, H. T., and Robinson, J.: "On Optimistic Methods for Concurrency Control, "ACM TODS Vol. 6 No. 2, June

1981 pp 213-226

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

[Mena80] Menasce, D. A., Popek, G. J. and Muntz R. R.: "A locking protocol for Resource Coordination in Distributed Database ACM TODS, Vol 5 No.2, June 1980

[Mena82] Menasce, D. A. and Nakanishi, T.: "Optimistic versus Pessimistic Concurrency Control Mechanism in Database Management system" Inform. Systems Vol. 7 No. 1 pp 13-27, 1982.

[Munz77] Munz, R. and Krenz, G.: "Concurrency in database systems - a simulation study" Proc. ACM SIGMOD International Conference on Management of Data, Toronto, Canada (Aug 1977), 111-120

[Papa82] Papadimitriou, C. H., : "A theorem in database Concurrency Control", Journal of ACM, Vol. 29, No.4, October, 1982 pp 998-1006

[Pein83] Peinl, P.: "Empirical Comparison of database Concurrency Control schemes", VLDB International Conference Florence, Italy Oct/Nov 1983 pp 97-108

[Pein87] Peinl, P.: "Load Balancing Policies vs Concurrency Control.An Empirical Comparison of DBMS performance Criteria" University of Stuttgart, Department of Computer Science West Germany.

[Poti80] Potier, D. and Leblanc, ph.: "Analysis of locking policies in database Management system", CACM Oct, 1980 Vol. 23, No.10, pp 584-593.

[Reut84] Reuter, Andreas: "Performance Analysis of Recovery Techniques" ACM TODS Vol. 9 No. 4 Dec 1984 ,pp 526-559

[Ries77] Ries, D. R., Stonebraker, M: "Effects of locking granularity in a database Management system." ACM TODS Vol. 2, No. 3 Sept 1977, pp 233-246.

[Ries79] Ries, D. R. and Stonebraker, M.: "Locking granularity revisited", ACM TODS Vol. 4 No.2, June 1979 pp 210-227

[Rose78] Rosenkrantz, D. J., Stearns, R. E., Lewis II, P. M.: "System Level Concurrency Control for Distributed Database systems" ACM TODS Vol 3 No. 2 June 1978 pp 178-198.

[Rypk79] Rypka, D. J. and Lucido, A. P.: "Detection and Avoidance for shared Logical Resources" IEEE Transactions on software Engineering, Vol SE-5, No.5, Sept 1979. pp 465-471.

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

[Schl81] Schlageter, G.: "Optimistic Methods for Concurrency Control in Distributed Database systems" IEEE 1981, University of Hagen, Germany West.

[Seth81] Sethi, R.: "A model of Concurrent database transactions (summary)", Bell Laboratories Murray Hill, New Jersey; IEEE 1981

[Seth82] Sethi, R.: "Useless Actions Make a Difference: Strict Serializability of Database Updates" JACM Vol. 29 No.2 April 1982 pp 394-403

[Silb80] Silberschatz, A. and Kedem, Z.: "Consistency in Hierarchical Database systems" JACM Vol. 27 No. 1 Jan 1980 pp 72-80

[Tay84a] Tay, Y. C., Goodman, N., Suri, R.: "Performance evaluation of locking in database: A survey" Technical Report TR - 17-84, Harvard University, Cambridge, Massachusets, 1984

[Tay84b] Tay, Y. C. and Suri, R.: "Choice and performance in locking for database, In Proc of tenth Int. Conf. on VLDB Singapore Aug 1984

[Tay85a] Tay, Y. C., Suri, R., Goodman, N.: "A Mean Value performance Model for locking in database: The No-Waiting Case", JACM Vol 32, No.3, July 1985 pp 618-651

[Tay85b] Tay, Y. C., Suri, R., Goodman, N.: "Locking Performance in centralized Databases" ACM TODS Vol 10 No. 4 Dec 1985 pp 415-462

[Thom79] Thomas R. H.: "A majority Consensus Approach to Concurrency Control" ACM TODS Vol 4 No 2 June 1979

[Trai82] Traiger, P. L., Gray, J., Galiteri, C. A., Lindsay, B. g.: "Transactions and Consistency in Distributed Database systems" ACM TODS Vol 7 No. 3 September 1982 pp 323-342

[Yann81] Yannakakis, M.: "Issues of Correctness in Database Concurrency Control by locking" STOC (Milwacikee 1981) pp 363-367.

[Yann82a] Yannakakis, M.: "Freedom from deadlock of safe locking policies" Siam J. Comput. Vol. II, No. 2, May 1982.

[Yann82b] Yannakakis, M.: "A theory of safe locking policies in Database Systems" JACM Vol. 29 No.3 July 1982 pp 718-740

[Yann84] Yannakakis, M.: "Serializability by Locking" JACM Vol. 31, No. 2 April 1984 pp 227-244.

APPENDIX 1

Generation of transaction, transaction classes and Compatibility set tables

1. A random number generator is used to identify the first of the objects and the number of objects to be accessed by each transaction.

2. It is assumed that writeset of a transaction is a subset of its readset. It is further assumed that Pr(write/read) ranges from 0 to 0.5.

3. For implementation ease, each transaction constitutes a transaction class

4. By inspecting readset and writeset of each class, conflicting classes are determined.

5. Also, transaction types (e.g. READ-ONLY and READ/WRITE transactions) are determined while compatibility sets are randomly picked among the transactions.

6. Using compatibility sets and transaction classes, compatible transactions within

transaction classes are identified.

TRANSACTION TABLE

Transaction No.	Transaction Type	Transaction Class	Compatibility Set	Read Range	Write Range
1	W	1	1	001,015	001,008
2	R	2	1	002,002	NIL
3	W	3	1	003,008	003,005
4	W	4	2	006,011	009,011
5	R	5	2	008,019	NIL
6	W	6	2	013,059	036,059
7	W	7	3	014,014	014,014
8	W	8	3	017,017	017,017
9	R	9	3	025,032	NIL
10	W	10	3	029,029	029,029
11	W	11	4	034,042	041,047
12	R	12	4	043,065	NIL
13	R	13	13	075,102	NIL
14	W	14	5	054,062	054,059
15	R	15	12	062,071	NIL
16	R	16	12	086,062	NIL

17	W	17	6	090,089	090,094
18	W	18	6	094,100	097,100
19	R	19	8	100,107	100,102
20	W	20	8	106,110	108,110
21	W	21	8	119,122	119,120
22	W	22	7	120,141	120,131
23	W	23	9	126,127	127,127
24	R	24	7	130,161	NIL
25	W	25	9	132,132	132,132
26	W	26	9	133,133	133,133
27	W	27	9	144,168	156,168
28	W	28	9	168,186	168,177
29	W	29	10	170,197	183,197
30	R	30	10	172,174	NIL
31	W	31	10	174,231	203,231
32	W	32	10	181,225	181,203
33	W	33	10	185,185	185,185
34	W	34	10	191,191	191,191
35	W	35	11	201,229	201,224
36	R	36	11	220,238	NIL
37	W	37	12	257,257	256,257
38	R	38	12	263,281	NIL

39	W	39	12	274,278	274,276
40	R	40	14	331,341	NIL
41	W	41	14	335,346	335,346
42	W	42	13	358,362	358,360
43	W	43	12	397,403	400,403
44	W	44	13	435,456	NIL
45	W	45	15	452,478	465,478
46	W	46	15	457,487	472,487
47	W	47	15	503,515	503,508
48	W	48	18	510,510	510,510
49	R	49	16	537,578	NIL
50	W	50	16	545,559	552,559
51	W	51	16	557,561	557,559
52	W	52	17	573,573	573,573
53	W	53	18	607,618	612,618
54	W	54	18	621,662	621,642
55	W	55	18	636,636	636,636
56	R	56	17	677,706	NIL
57	W	57	17	689,703	689,697
58	W	58	17	697,702	697,699
59	W	59	17	698,727	712,727

60	W	60	17	731,745	731,738
61	W	61	18	741,755	741,748
62	W	62	19	777,780	779,780
63	W	63	19	785,818	785,801
64	W	64	19	797,797	797,797
65	W	65	19	804,804	804,804
66	W	66	19	852,864	852,857
67	W	67	19	859,892	875,892
68	W	68	21	892,908	899,908
69	W	69	21	905,917	905,910
70	W	70	20	918,927	918,922
71	R	71	22	919,966	Nil
72	W	72	22	935,959	935,942
73	W	73	22	951,953	953,953
74	W	74	20	994,1000	994,997

TRANSACTION CLASSES TABLE

Transaction Class	Conflicting Classes	Conmpatible Transactions within conflicting Classes
1.	1,2,3,4,5,7	(1,2,3) (4, 5)
2.	1	Nil
3.	4,3,	Nil
4.	1,4,5	(4,5)
5.	1,4,7,8	(7,8)
6.	1,5,7,8,9,10,6	(7,8,9,10)
7.	7	Nil
8.	8	Nil
9.	10	Nil
10.	10	Nil
11.	6,11,12,13	Nil
12.	6,11	Nil
13.	16,17,18,19	(17,18)
14.	14	Nil
15.	15	Nil
16.	13,16,17,18	(17,18)
17.	13,16,17,18	(17,18)

18.	13,17,18,19	(17,18)
19.	13,18,	Nil
20.	19,20	(19,20)
21.	21,22	Nil
22.	21,22,23,24,25,26	(22,24) (25,26)
23.	22,23	Nil
24.	25,26,27	(25,26,27)
25.	24,25	Nil
26.	24,26	Nil
27.	28,29	(27,28)
28.	27,28,29,30,32,33	(29,30,32,33)
29.	28,29,30,32,33	(29,30,32,33)
30.	28	Nil
31.	28,29,31,32,33	(29,31,32,33)
32.	29,31,32,33	(29,31,32,33)
33.	29,32,33	(29,32,33)
34.	29,33,34	(29,33,34)
35.	31,32,35	(31,32)
36.	31	Nil
37.	37	Nil
38.	38	Nil
39.	39	Nil

40.	41	Nil
41.	40,41	(40,41)
42.	42	Nil
43.	43	Nil
44.	44	Nil
45.	45,46	(45,46)
46.	45,46	(45,46)
47.	47,48	Nil
48.	48	NIL
49.	50,51,52	(50,51)
50.	49,50,51	(49,50,51)
51.	49,50,51	(49,50,51)
52.	52	NIL
53.	53	NIL
54.	54,55	(54,55)
55.	57,58	(57,58)
56.	57,58	(57,58)
57.	56,57,58	(56,57,58)
58.	58,59	(58,59)
59.	61,60	(58,59)
60.	61	NIL

61.	62	NIL
62.	63	NIL
63.	64	NIL
64.	65	NIL
65.	66	NIL
66.	67	NIL
67.	68,69	(68,69)
68.	68,69	(68,69)
69.	70	NIL
70.	70,73	NIL
71.	71,72,73	(71,72,73)
72.	71,73	(71,73)
73.	73	NIL
74.	74	NIL

COMPATIBILITY SETS TABLE

Compatibility Set No.	Compatible Transactions
1.	1,2,3
2.	4,5,6
3.	7,8,9,10
4.	11,12
5.	14,15
6.	17,18,19
7.	22,24
8.	19,20,21
9.	23,25,26,27,28
10.	29,30,31,32,33,34
11.	35,36
12.	43,15,37,38,39,16
13.	13,42,44
14.	41,40
15.	45,46,47
16.	49,50,51
17.	54,56,57,58,59,60
18.	48,52,53,55,61
19.	62,63,64,65,66,67

20.	70,74
21.	68,69
22.	71,72,73

APPENDIX 2

OUTPUT DATA ANALYSIS

This appendix represents the statistical analysis techniques used to interpret the simulation results of experiments of Chapter 4. The methods employed here are based on a combination of the traditional batch means approach [Ferr78, Law82, Saue81] and an improved technique for estimating variance for use in computing confidence intervals for the results. The improved variance estimation technique was proposed by Law et al [Law82].

A. Statistical Model

This section describes the improved batch means approach used in this thesis. Each simulation is run for t_b simulation time units, and the overall simulation is divided into n individual 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). This estimates are summed and divided by the total number of batches in order to compute the overall throughput estimate for a simulation:

$$\bar{X} = \frac{1}{n_b} \sum_{i=1}^{n_b} X_i \text{ ....(1)}$$

To model the statistical characteristics of the throughput observations, it is assumed that $\underline{\{Xi\}}$ is a stationary sequence and that each $\underline{Xi}$ has mean $\underline{U}$ and variance $\underline{\mathbf{9}}^{2}$. It is further assumed that the correlation between adjacent batches is significant, but the correlation between non-adjacent batches is negligible [Conw63]. More formally:

$$\mathrm{Cov}\ (Xi,Xj) = \begin{cases} d \text{ if } \mid i-j \mid = 1 \\ o \text{ if } \mid i-j \mid > 1 \text{ ....(2)} \end{cases}$$

Given this model of the correlation between batches, the variance of the sum of the $\underline{Xi's}$ may be expressed as:

$$\mathrm{Var}\left(\sum_{i=1}^{n_b} Xi\right) = \sum_{i=1}^{n_b} \mathrm{Var}\ (Xi) + 2\times$$

$$\sum_{i=1}^{n_b - 1} \ \sum_{j=i+1}^{n_b} \mathrm{Cov}\ (Xi,\ Xj) \ \ldots\ldots\ldots\ldots \quad (3)$$

$$= n_b\, g^2 + 2\ (n_b - 1)d$$

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

$$\mathrm{Var}\ (\overline{X}) = \frac{g^2}{n_b} + \frac{2(n_b - 1)d}{n_b^2} \ \ldots\ldots\ldots\ldots\ldots\ldots \quad (5)$$

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 $\underline{g}^2$ and $\underline{d}$. The estimate often used for g^2

in computing confidence intervals is S^2, the sample variance of the throughput observations:

$$S^2 = \frac{1}{n_b - 1} \sum_{i=1}^{n_b} (Xi-\bar{X})^2 \ldots\ldots\ldots\ldots\ldots \quad (6)$$

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

$$\bar{X}_{odd} = \frac{1}{(n_b/2)} \sum_{i\ odd} Xi \ldots\ldots\ldots\ldots\ldots\ldots (7)$$

$$\bar{X}_{even} = \frac{1}{(n_b/2)} \sum_{i\ even} Xi \ldots\ldots\ldots\ldots\ldots\ldots (8)$$

$$S^2_{odd} = \frac{1}{(n_b/2)-1} \sum_{i\ odd} (Xi-\bar{X}\ odd)^2 \ldots\ldots (9)$$

$$S^2_{even} = \frac{1}{(n_b/2)-1} \sum_{i\ even} (Xi-\bar{X}\ even) \ldots (10)$$

$$S^2_d = \frac{S^2\,even + S^2\,odd}{2} \ldots\ldots\ldots\ldots\ldots (11)$$

The covariance term in equation (5) may then be estimated once S^2_d has been computed. This estimate of the covariance d, denoted c, may be computed as follows. First, an unbiased estimate k of the quantity $2(g^2-d)$ is:

$$K = \frac{1}{\frac{n}{b} - 1} \sum_{i=1}^{\frac{n}{b} - 1} (X_{i+1} - X_i)^2 \quad \ldots\ldots\ldots\ldots \quad (12)$$

Given K, an estimate of d is:

$$C = S_d^2 - \frac{k}{2} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (13)$$

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):

$$S_{\bar{x}}^2 = \frac{S_d^2}{\frac{n}{b}} + \frac{2(\frac{n}{b}-1)c}{\left(\frac{n}{b}\right)^2} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (14)$$

Confidence Intervals

Given an overall throughput estimate $\bar{X}$ and an estimate $S^2_{\bar{x}}$ of its variance, confidence intervals can be computed in a fairly simple manner. The confidence interval computation is performed as though the Xi's were independent, as is typically assumed, with the improved variance estimate $S^2_{\bar{x}}$ used in place of the usual estimate of S^2 / n_b. Thus, the 100 (1-a)% confidence interval for the mean throughput t is computed as:

$$\bar{X} \pm \ell \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (15)$$

Where: $\ell = S_{\bar{x}} \; t_{\alpha} / 2; \; n_{b-1}, \qquad S_{\bar{x}} = \sqrt{S_{\bar{x}}} \quad \ldots \quad (16)$

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:

$$\text{Prob } (Y > ta/2; \; n_{b-1}) = \frac{a}{2} \quad \ldots\ldots\ldots\ldots\ldots \quad (17)$$

In order to obtain the most reasonable confidence interval estimates, the computational procedure in this thesis differs slightly from what has been discribed thus far. First, the confidence interval estimate obtained using the preceding method will be slightly narrow because correlation betwwen adjacent Xi's reduces the "effective" number of degrees of freedom [Law82]. 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 method describe here even more realistic.

Second, since the estimator $\underline{c}$ of the covariance $\underline{d}$ is itself just a random variable, actual experimental data may occasionally yield a negative covariance estimate. Since correlation tend to be positive in this type of study, $\underline{c \leq o}$ is taken to indicate that the actual covariance $\underline{d}$ 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(X) and using n_{b-1} for the number of degrees of freedom.