INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

University of Oklahoma

Graduate College

AN INTEGRATED CONCURRENCY CONTROL IN OBJECT-ORIENTED DATABASE SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

By

Woochun Jun

Norman, Oklahoma

1998

AN INTEGRATED CONCURRENCY CONTROL IN OBJECT-ORIENTED DATABASE SYSTEMS

A DISSERTATION

APPROVED FOR THE SCHOOL OF COMPUTER SCIENCE

By

Acknowledgments

First of all, I would like to express my appreciation to my advisor Dr. Le Gruenwald, who is a great researcher and teacher, for her encouragement and valuable advice. Especially, she has taught me how to do research during the past four years.

I also would like to thank Dr. S. Lakshmivarahan, Dr. S. Dhall, Dr. D. Trytten and Dr. S. Pulat for their guidance and serving as my committee members.

I am indebted to my officemate Jing Huang for her friendship and suggestions, especially helping in experiments.

I do not know how to express my deep gratitude to my father Youngjoon Jun, my mother Euisoon Kim, and my sister Yooja Jun, for their support and patience. Without their sacrifice, I would never have finished my research.

Finally, this work would not be possible without the help of God.

Table of Contents

List of Figures

List of Tables

Abstract

Object-oriented databases (OODBs) have been adopted for non-standard applications requiring advanced modeling power, in order to handle complex data and relationships among such data. One of the important characteristics in database system is manipulation of shared data. That is, database systems, including OODBs, allow shared data to be accessed by multiple users at the same time. Concurrency control is a mechanism used to coordinate access to the multi-user databases so that the consistency of the database is maintained. In order to provide good performance, it is very important that concurrency control schemes incur low overhead and increase concurrency among users. This dissertation presents a concurrency control scheme in OODBs that meets those requirements.

First, the dissertation discusses three important issues of concurrency control in OODBs. These include conflict among methods, class hierarchy locking, and nested method invocations. The previous works for each issue are presented, and their advantages and disadvantages are also discussed. Then, an integrated concurrency control which addresses all three issues is proposed. For conflict among methods, a finer locking granularity, such as an attribute and an individual class object, is adopted for instance access and class definition access so that higher concurrency is achieved. Especially, for instance access, higher concurrency is obtained using run-time information. Also, locks are required for instance method invocations instead of atomic operation invocations so that locking

overhead is reduced. For class hierarchy locking, locking overheads are reduced using special classes which are based on access frequency information on classes. Finally, for nested method invocations, semantic information is used in order to provide higher concurrency among methods. Also, parent/children parallelism is adopted for better performance.

Secondly, an analytical model is constructed to measure the performance of concurrency control in an OODB system. Using this model, the proposed technique is then compared with the two existing techniques, Orion and Malta. The analytical results show that the proposed scheme gives the best transaction response time, Malta the second best, and Orion the worst.

Finally, a performance study is conducted by means of simulation using the 007 benchmark. The simulation results show that, in terms of transaction response time and lock waiting time, the proposed scheme performs the best, Malta the second best, and Orion the worst.

CHAPTER 1

INTRODUCTION

1. Motivation

Recently, many new database applications such as computer-aided design (CAD), computer-aided software engineering (CASE), office information systems, and artificial intelligence have emerged. These new areas require advanced modeling capabilities to handle complex data and complex relationships among data. In those areas, complex modeling is impossible or very difficult, if relational data model is adopted. An object-oriented database (OODB) is suitable for such applications, since it provides modeling power as grouping similar objects into class, and organizing all classes into a hierarchy where a subclass inherits all definitions from its superclasses.

In [Kim,1990], an OODB is defined as "a collection of objects whose behavior and state, and the relationships are defined in accordance with an object-oriented data model". Also, an object-oriented database system (OODBS) is defined as "a database system which allows the definition and manipulation of an OODB". The followings are basic concepts in OODBs ([Kim,1990],[Olse,1995]).

- Object: any real world entity can be an object. Also, each object is associated with a unique identifier.
- Attribute: an object has one or more *attributes* whose values are also objects. The values of an attribute represent the state of an object.
- Method: an object has one or more *methods* which operates on the state of the object.

- Class: all objects sharing the same set of attributes and methods can be grouped into a class. An object belongs to only one class as an instance of the class.
- Encapsulation: it is the process of packaging the data elements and functionality together [Denc,1994]. That is, the state of an object can be manipulated and read only by invoking the object's methods.
- Class hierarchy: the classes form a hierarchy (which is directed-acyclic graph) called a *class hierarchy*. It is based on generalization and specialization concepts, which will be discussed later.

Usually, several operations on the database form a logical unit of work. For example, consider customer's fund transfer in which one account is debited and the other account is credited. It is important to maintain database consistency so that either both debit and credit occur or none of them occurs. That is, the fund transfer is done atomically. A transaction is a collection of operations that performs a single logical function in a database application [Kort,1991]. In general, a transaction has four properties: *atomicity*, *consistency*, *isolation*, and *durability*. Each transaction is a unit of atomicity (*atomicity*). Thus, a successful execution of a transaction maps one consistent database state into another (*consistency*). Also, an executing transaction's intermediate results cannot be revealed to other concurrently running transactions before commitment (*isolation*). Once a transaction commits, its results are recorded in the database permanently and cannot be erased (*durability*). In OODBs, a database is a collection of classes and instances where classes and instances are called *objects*. Users can access objects by invoking methods. To make sure atomicity of user interactions, the traditional

transaction model can be used in OODBs. That is, users can access an OODB by executing transactions, each of which is defined as a partially ordered set of method invocations on a class or an instance object [Agra,1992].

Transactions in OODBs have the following characteristics: first, unlike traditional applications, transactions in advanced applications such as CAD and CASE require long-duration running time (hours or even days). In particular, in CAD, design tasks generally make a team of designers cooperate for days to months [Jose,1991]. Second, in OODBs, since a method on an object can invoke other methods on other objects, this results in transaction executions with a nested form ([Agra,1992], [Hadz,1991]). Third, some advanced applications require user transactions to cooperate to perform a common task. This results in the concept of *cooperating* transactions.

One of the important characteristics in database systems is manipulation of shared data. That is, database systems, including OODBs, allow shared data to be accessed by multiple users at the same time. Concurrency control involves synchronization of access to the database so that the consistency of the database is maintained ([Ozsu,1991], [Bern,1987]). Like in conventional databases, concurrency control in OODBs also requires logical consistency of data and transactions. Concurrency control requires an application-dependent correctness criterion to maintain database consistency while transactions are running concurrently on the same object. *Serializability* is a widely used correctness criterion. Transactions are *serializable* if the interleaved execution of their operations produces the same output and has the same effects on the database as some serial execution of the same transactions ([Bern,1987], [Bern,1981]).

In general, three concurrency control schemes have been used: locking-based ([Eswa,1976], [Hung,1992], [Kort,1983]), optimistic [Kung,1981], and timestamp ordering ([Bern,1987], [Sing,1985], [Ulos,1992]). The locking-based (also called *pessimistic*) schemes assume that there will likely be conflicts among concurrent transactions and transactions must acquire locks before accessing the database. One locking_based scheme that ensures serializability is the two-phase locking scheme [Kort,1991]. The basic rules of this protocol require that two phases be observed by every transaction as follows. First, in *growing phase*, a transaction may only obtain locks (releasing any lock is prohibited). Second, in *shrinking phase*, a transaction may only release locks (obtaining further lock is prohibited likewise). Initially, a transaction belongs to growing phase. After the transaction obtains as many locks as needed, it enters to the shrinking phase and further lock requests are prohibited. The optimistic schemes assume that transactions will not conflict with each other, and the isolation of a transaction is checked only at its commit time. The timestamp ordering scheme is a technique in which the transaction execution is ordered on a priori serialization order. To obey this order, each transaction is assigned a timestamp when it is initiated. Conflicting operations of transactions are processed in the timestamp order. There are many variations of these three basic schemes ([Care,1987], [Hali,1989], [Serv,1990], [Hali,1991], [Levy,1994], [Naka,1994]). Locking-based scheme are the most widely used schemes in database systems.

Supporting concurrency control in an OODB is more complicated than in a relational database for the following reasons ([Jaga,1993], [Muth,1993]). First, the

semantics of methods on encapsulated objects can be exploited to provide better concurrency. That is, although two methods conflict with each other in terms of read and write conflict relationships, OODB systems can provide concurrency between two methods using semantics of the methods since the methods usually represent behaviors of objects [Kwon,1997]. Second, an object participates in various forms of hierarchies among objects such as class hierarchy or composite-object hierarchy [Garz,1988]. Access to one object may affect other objects in the hierarchies; thus access control is more complicated. For example, if a definition of a class is updated, this blocks any access to the class as well as its subclasses.

As we discussed earlier, typical transactions are long-lived nature. Thus, in order to meet database consistency, blocking or aborting transactions may delay transaction's response time. For better performance in transaction processing in OODBs, it is essential for a concurrency control scheme to incur low overhead whenever invoked and provide higher concurrency so that as many as transactions can run concurrently.

1.2. Problem Statement

A number of concurrency control techniques have been developed for OODBs ([Jaga,1993],[Malt,1993],[Muth,1993],[Olse,1995],[Rese,1994],[Lee,1996],[Shar,1996],[Kwon,1997]). These existing works deal with three features of access: *conflicts among methods*, *class hierarchy locking*, and *nested method invocations*. In order to illustrate each type of access, consider the following Figure 1-1. Assume that class *vehicle* has four

attributes *id*, *color*, *drivetrain* and *manufacturer* and class *company* has three attributes *name*, *location*, and *president*. Class *employee* has three attributes *ssn*, *name*, *age*.

1.2.1. Conflicts among methods

In general, there are two types of access to an object : *instance access* and *class definition access* [Cart,1990]. An instance access consists of consultations and modifications of attribute values in an instance or a set of instances. A class definition access includes consulting class definition, adding/deleting an attribute or a method, changing the implementation code of a method or changing the inheritance relationship between classes, etc. In Figure 1-1, for class *vehicle*, a possible instance access is a modification of the attribute *color* of an instance, and a possible class definition access is changing domain of the attribute *id* from integer to character.

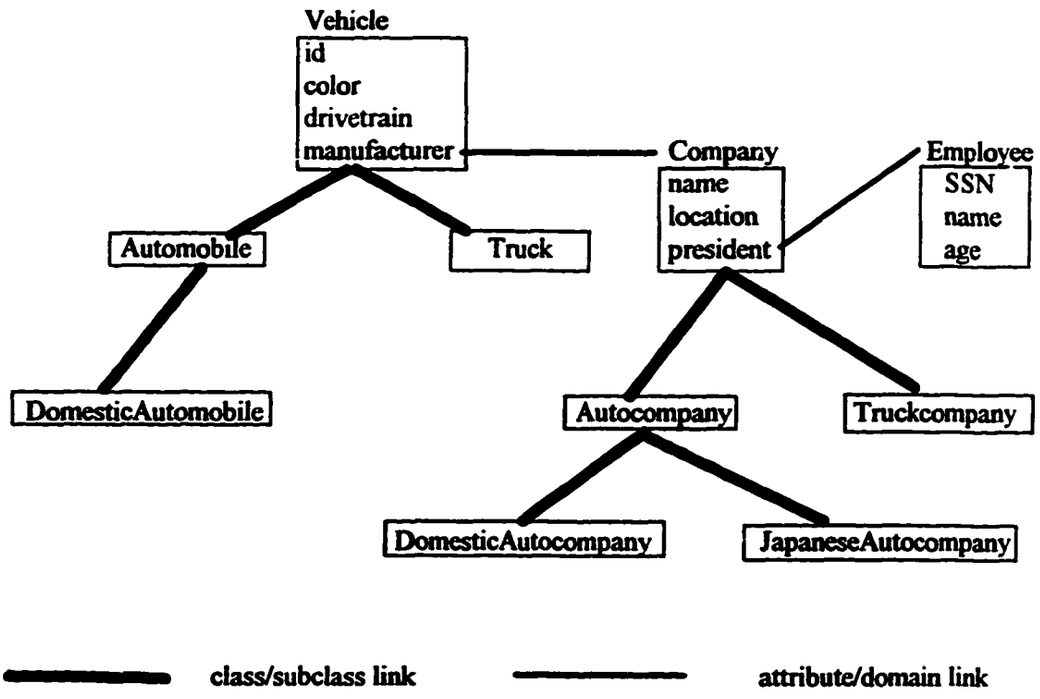


Figure 1-1. Illustrative OODB schema [Kim, 1990]

In OODBs, one of the main concerns is to increase concurrency among methods so that more transactions can run in parallel. Otherwise, aborting or blocking a transaction to meet database consistency may waste system resources or delay other transactions. Commutativity is a widely used criterion to determine whether a method can run concurrently with those in progress on the same object [Malt,1992]. Two methods commute if their execution orders do not affect the results of the methods. Two methods conflict with each other if they do not commute.

Two types of access to an object induce three different types of conflicts among accesses to a class: *conflicts between instance accesses*, *conflicts between class definition accesses*, and *conflict between instance access and class definition access*. For example, a conflict between instance accesses occurs if two instance methods are trying to modify an attribute value of the same instance at the same time. Also, updating the same class definition such as modifying the implementation code of the same instance method at the same time induces conflict between class definition accesses.

1.2.2. Class Hierarchy Locking

One of the major properties of an OODB is inheritance. That is, a subclass inherits definitions defined on its superclasses. Also, there is an *is-a* relationship between a subclass and its superclass. Thus, an instance of a subclass is a specialization of its superclasses (and conversely, an instance of a superclass is a generalization of its subclasses) [Garz,1988]. This inheritance relationship between classes forms a *class hierarchy*. There are two types of inheritance: single inheritance and multiple inheritance. In single inheritance, a class can inherit the class definition from one superclass. On the

other hand, a class can inherit the class definition from more than one class in multiple inheritance.

While there are some operations on only one class such as class definition read or instance write on one instance, there are two types of operations on a class hierarchy: *class definition write and instance access to all or some instances of a given class and its subclasses* (also called IACH, meaning Instance Access to Class Hierarchy). A query is an example of IACH where a *query* is defined as instance reads to a given class and its subclasses [Garz,1988]. Due to inheritance, while a class and its instances are being accessed, the definitions of the class' superclasses should not be modified. Also, due to the *is-a* relationship between classes, the search space for a query against a class, say C, may include the instances of all classes in the class hierarchy rooted at C as well as all instances of C. Thus, for locking_based concurrency control schemes, when a class definition write or query is requested on some class, say C, we need to get locks for all subclasses of C as well as C. We call MCA (Multiple Class Access) for class definition write and IACHs, and SCA (Single Class Access) for other operations such as class definition read and instance access to a single class.

For example, in Figure 1-1, changing any definition of a class vehicle may block any other incoming access to class *Automobile*, *Truck* or *DomesticAutomobile*. Also, due to the *is-a* relationship among classes, a query to a class *Autocompany* may access all instances of class *DomesticAutocompany* and *JapaneseAutocompany* as well as instances of class *Autocompany*.

1.2.3. Nested method invocations

In OODBs, objects can have nested structures. That is, an object can be composed of complex objects or atomic objects. For example, in Figure 1-1, an object *vehicle* can consist of three atomic objects (i.e., *id*, *color*, and *drivetrain*) and a complex object *manufacturer* . It is natural that, in OODBs, each class can define its own method and a method on a class can invoke another method on its subobject (also called *nested method invocation*) [Muth, 1993].

In OODBs, two different objects can share a common object in an underlying hierarchy [Herr,1990]. We call the common object a *referentially shared object* (RSO). Once again, in Figure 1, two different instance objects *vehicles* may share the same instance object *company* in an underlying nested object hierarchy. Thus, methods on different objects may not commute [Muth,1993]. The RSO (also called non-disjoint complex object) is a fundamental concern of OODB since new objects may be composed of existing objects in modular design as indicated in [Rese,1994]. Thus, a nested object hierarchy may result in referential sharing.

Existing works have many disadvantages as follows. For *conflicts among methods*, application programmers have a burden to provide commutativity relationships for instance access. That is, in order to provide better concurrency among methods, application programmers should know possible states of objects and results of each method. Also, for class definition access, existing works either provide less concurrency due to big locking granularity or incur too much run-time overhead for higher concurrency. For *class hierarchy locking*, existing studies, which can be classified into two types (i.e., *explicit locking* and *implicit locking*), incur too much locking overhead

and aim at a special type of access to class hierarchy (i.e., *explicit locking* aims at access to a higher-level class of the hierarchy while *implicit locking* aims at access to a class near the leaf-level). For *nested method invocations*, either concurrency is still limited since semantic information is not utilized or too much run-time overhead is incurred since locks are required for each atomic operation. Also, most existing studies do not consider referentially shared objects (non-disjoint complex objects) which is a necessary condition for modular design in an OODB [Rese,1994].

1.3. Research Objectives

This research has the following objectives.

An integrated concurrency control scheme is developed for an OODB. The proposed scheme is based on locking and deals with the following issues: *conflicts among methods* (i.e., conflicts among instance access methods, conflicts among class definition access methods, conflicts among instance access methods and class definition access methods), *class hierarchy locking* with single inheritance and multiple inheritance, and *nested method invocations*. Also, a proof to the correctness of the proposed concurrency control scheme is given

In order to test the performance of the proposed scheme, first, an analytical model is constructed for concurrency controls in OODBs. Based on the model, performance is evaluated for each type of access and for a mixture of all types of access to compare the proposed scheme and the existing works: Orion [Garz,1988] and Malta ([Malt,1991],[Malt,1993]).

Also, extensive simulation experiments are conducted to compare the proposed concurrency control scheme with the existing schemes, based on the 007 OODB benchmark. The results obtained are then analyzed, and guidelines for developing and selecting an OODB concurrency control technique are provided.

1.4. Organization of the dissertation

The rest of the dissertation is organized as follows.

In Chapter 2, related work for concurrency controls in OODBs is discussed. For each access type (conflict among methods, class hierarchy locking and nested method invocations), advantages and disadvantages of existing work is presented

In Chapter 3, an integrated concurrency control is proposed. The scheme deals with the three access types. The correctness of the proposed scheme is also provided.

An analytical model for the proposed concurrency control scheme and also two existing schemes in OODBs is introduced in Chapter 4. Using the analytical model, performance analysis is conducted comparing the proposed work and the existing work of Orion and Malta.

In chapter 5, a simulation model is introduced in order to compare performance among the proposed scheme and the existing work of Orion and Malta. For standard set of requirements, the 007 OODB benchmark is adopted. Based on the benchmark, results are obtained and then analyzed.

Finally, Chapter 6 gives conclusions and future research issues of the dissertation.

CHAPTER 2

LITERATURE REVIEW

In this chapter, a review of relevant research on concurrency control techniques in OODBs are presented. Techniques are discussed in terms of three access types (conflict among methods, class hierarchy locking and nested method invocations) for centralized OODBs.

2.1. Conflicts among methods

2.1.1. Conflicts between instance accesses

In many existing locking-based schemes, concurrency between instance accesses is limited since locking granularity is an entire instance object ([Garz,1988], [Cart,1990], [Wang,1990], [Malt,1991]). In ([Garz,1988], [Wang,1990], [Cart,1990], [Malt,1991]), and their locking granularity is a unit of instance object for an instance read and an instance write, respectively. Thus, two transactions accessing the same object may conflict with each other even though they access disjoint attributes of the instance object. This results in limited concurrency among instance accesses.

While the above schemes are based on only locking, a scheme in Gemstone OODB system [Serv,1990] is based on both optimistic and pessimistic (locking-based) concurrency control. Under optimistic concurrency control, a transaction T's possible conflicts can be detected by comparing its read and write sets with those of any other transactions T1's that already committed after transaction T began. That is, a conflict may occur either if T's write set and a T1's write set have common objects or if an object in

T's write set is in another transaction T1's read set and an object in T's read set in also in T1's write set. If a conflict is detected, T needs to be restarted. Under a locking_based concurrency control, the scheme provide three lock types: read, write and exclusive. Holding a read lock on an object means that any other transaction cannot get write or exclusive lock on the same object or commit if it has written the object. A holding a write lock on an object means that any other lock requesting transaction cannot get any kind of lock on the same object or commit if the lock requesting transaction has written on the object. This write lock is different from conventional exclusive lock. That is, other transaction may read a write-locked object optimistically and commit. If a transaction holds an exclusive lock on an object, any other lock requesting transactions cannot get any kind of lock on the same object or commit if the lock requesting transaction has written or read on the object. This lock prevents another transaction from reading the value of the object and then writing that derived information in other objects. In this approach, a lock granularity for an instance access is an entire instance object so that the concurrency provided is still limited.

In addition to the above techniques, several techniques have been proposed to increase concurrency among instance accesses ([Agra,1992], [Badr,1988], [Badr,1992], [Chry,1991]). In order to decide commutativity of instance access methods, they require application programmers to perform semantic analysis on the methods as follows.

The *right backward commutativity* is introduced to provide more concurrency among methods in [Agra,1992]. It assumes that, at any given time, the current state of an object consists of a committed state, and a set of operations belonging to active

transactions. Also, it assumes that, when an operation is executed on an object, a result *res* is returned. Like conventional commutativity relations, the right backward commutativity is used to define conflict or none-conflict relationship between a lock holder and a lock requester. But, the right backward relationship has the following characteristics [Agra,1992]: "an operation o_1 is said to have *right backward commutativity* with another operation o_2 on an object if for every state in which o_2 can be executed after o_1, executing o_2 followed by o_1 has the same state and result as executing o_1 followed by o_2". This is less restrictive than commutativity relation since right backward commutativity is included in commutavitity and commutativity does not necessarily include the right backward commutativity. But, in order to support *right backward commutativity*, application programmers need to know all possible outcomes of each method. For example, consider an instance object representing bank account in [Agra,1992]. The state of the instance object is the amount of money at given time. Assume that there are three operations defined on the object: *deposit*, *withdraw* and *balance*. The response to a *deposit* operation is always *OK*. The response to a *withdraw* operation is either *yes* or *no* depending on balance while the response to a *balance* operation is the amount of money in the account. Then, depending on the response to operation o, <o, res> may have different conflict relationships with other operations. For example, <withdraw (i), no> does not conflict with <balance, j> since <withdraw (i), no> has right backward commutativity relationship with all <balance, i> pairs where i and j are the amount of money. If the response of *withdraw* operation is not considered, the balance operation always conflicts with the *balance* operation.

In [Bard,1988], the concurrency is enhanced by taking attribute locking granularity instead of an instance object. They define the affected set of each method attributes accessed by the method so that two methods commute if the intersection of their affected set of attributes is disjoint. For example, two instance write methods do not conflict, as long as their affected set of attributes is disjoint. Thus, it can achieve higher concurrency than entire instance object is taken as a locking granularity. But, application programmers need to know the affected set of each method and structure of every object. For example, assume two methods M_1 and M_2 , and four attributes a_1, a_2, a_3 and a_4 are defined on object *o*. Again, assume that M_1 reads attributes a_1 and a_3 while M_2 modifies attribute values of a_2 and a_4. Then, two methods can run concurrently whereas they conflict with each other when the entire object lock granularity is considered.

Recoverability is used to enhance concurrency in [Badr,1992]. It is defined as follows: "an instance method m_1 is *recoverable* relative to another instance method m_2, if m_2 returns the same value whether or not m_1 is executed immediately before m_2". For commutativity-based schemes, a method which does not commute with other uncommitted methods will be blocked until those conflicting methods are aborted or committed. In the *recoverability-based* scheme, non-commuting but recoverable methods are allowed to execute concurrently. But, the commit order of the transactions invoking recoverable methods should be fixed. That is, it is based on the order in which they are invoked. If a lock requesting method does not have recoverability relationship with a lock holding method which is uncommitted, the lock requesting method is blocked. This

recoverability implies commutativity. This work requires application programmers to know all outcomes of each method for possible input parameters.

A formal methodology to define the commutativity relationships among methods is presented in [Chry,1991]. In this work, for each operation, the outcome and result should be provided by application programmers. The outcome of an operation is its status such as ok or nok (not ok) and other values it returns are called its result. For example, Pop operation in stack can have nok and top element as an outcome and result, respectively. Also, this technique adopts refined commutativity relations among operations: abort-dependency (AD) and commit-dependency (CD). In the traditional commutativity relationship among operations, a commutativity table contains binary relation with yes (commute) or no (not commute). Instead, in this work, an entry may contain three entries [Chry,1991]: ND (no-dependency), AD and CD. AD and CD have the following relationship: assume that an operation r follows an operation s. If s is a write and r is a read, the transaction has to abort when the first transaction aborts for some reason. This is due to the fact that the information used by r may not be valid. The second transaction can commit only if the first transaction commits. In this case, the second transaction is said to have an AD relationship with the first transaction. On the other hand, if s is a read and r is a write, the outcome and result of r are not affected by the effects of s. In this case, if both transactions commit, the first has to commit before the second in order to ensure serializability. That is, the second can commit only after the first commits or aborts. In this case, the second transaction is said to have a CD relationship with the first transaction. For example, consider a QStack object where a QStack combines the properties of a stack and

a queue [Chry,1991]. The operations defined on QStack are Enq (e)/Push (e) and Deq (e). Enq (e) or Push (e) add an element e to the back of the QStack. It returns ok if the QStack is not full, nok otherwise. Deq (e) deletes an element e from the front of the QStack. It returns e if the QStack is not empty, nok otherwise. Then, (Deq, Push) entry in commutativity table has the following relationships: {(AD, $Push_{out}$ = ok), (CD, $Push_{out}$ = nok)} where (Deq, Push) entry represents the situation that a Deq follows a Push on a QStack object. Finally, for each operation, its locality (i.e., a set of component objects accessed by an operation) is defined. This concept is similar to affected set of attributes in [Badr,1988]. As can be seen in the above description, application programmers need to know the outcomes of each method. In addition to this, dependency relation such as abort-dependency or commit-dependency between each pair of method should be provided by application programmers.

Forward and backward commutativity relations, which are introduced in ([Weih,1988],[Weih,1989]), are combined into a new relation called Forward-Backward commutativity (FBC) in [Guer,1995]. The backward commutativity is supported by the update in place (UIP) model. In the UIP model, any effects of active transactions are recorded immediately in the database. The backward commutativity (BC) is defined as follows [Guer,1995]: assume that the values of an object represent the state of the object. The state of an object can be accessed only by operations defined in the specifications of the object. The specification of an object represents the set of possible states and responses produced by this operation. Formally, for a given state s, a response and a state of an object are stated as follows: The return (op,s) represents return value by operation

op on state s. The state (op,s) represents the state produced after the execution of op. Then , "op1 BC op2 if and only if ∀ s such that state (op1,s), state (op2, state(op1,s)), return (op1,s) and return (op2, state (op1,s)) are defined, the followings three equalities hold":

(1) state (op1, state (op2,s)) = state (op2, state (op1,s))

(2) return (op1,s) = return (op1, state (op2,s))

(3) return (op2,s) = return (op2, state (op1,s))

On the other hand, Forward commutativity (FC) is supported by the deferred update model (DU). In the DU model, the effects of active transactions are not recorded until those transactions commit. Also, those effects can not be seen by any other transactions. FC can be defined as follows [Guer,1995]: "op1 FC op2 if and only if ∀ s such that state (op1,s), state (op2,s), return (op1,s) and return (op2,s) are defined, then three equalities (1), (2) and (3) defined above hold". Finally, FBC can be defined as follows [Guer,1995]: "op1 FBC op2 if and only if ∀ s such that state (op1, s), state (op2, s), state (op2, state (op1,s)) are defined and return (op2,s) = return (op2, state (op1,s)), the two equalities (2) and (3) hold". This FBC is less restrictive than both BC and FC since all FBC relationships imply BC and FC relationships, not vice versa. Thus, FBC relationship can provide more concurrency than both relationships. The three commutativity relations BC, FC and FBC define commutativity relationships between a lock requester and a lock holder. But, unlike conventional commutativity, the three commutativity relationships BC, FC and FBC utilize states and responses of objects in order to enhance concurrency. In order to define this commutativity relationship, possible

states and response of all operations in an object should be analyzed and then commutativity relation table is constructed. Note that FBC can be supported by the neither the UIP nor DU model since FBC requires both of UIP and DU model. In other words, in order to support FBC, each object needs to be biversioned. In the biversion object model, each object has two states: *current-state* and *committed-state*. The current-state has the value accessed by both active transaction and committed transactions. Whereas, the committed-state has the value accessed only by committed transactions.

Recently, a commutativity relation, called *general commutativity*, has been introduced in order to provide concurrency more than both forward commutativity and backward commutativity [Naka,1994]. This commutativity is based on both forward commutativity and backward commutativity defined in ([Weih,1988], [Weih,1989]). [Naka,1994] argues that two commutativity relationships are not subsets of each other, so that one cannot claim which one is better. Moreover, each commutativity requires a different recovery algorithm and a different implementation of an object. A general commutativity relation includes both commutativity (forward and backward) relations. This means that a general commutativity relation can achieve higher degree of concurrency. Like in [Guer,1995], possible states and responses of all operations should be analyzed in order to construct general commutativity. But, unlike [Guer,1995], the commutativity is based on multiversion objects which have both committed states and current states of objects at the same time, since this scheme requires histories of objects.

Those techniques presented so far require application programmers to define the commutativity relationships among methods. The construction of commutativity relations

is a burden on application programmers. Recently in [Malt,1993], the process of constructing commutativity relations from method contents is automated. It is based on the notion of *affected sets of attributes* [Badr,1988]. That is, even if two instance methods conflict in terms of read or write operations, as long as their access modes on individual attributes do not conflict, two methods can run in parallel. Commutativity of methods is determined at compile-time so that run-time commutativity checking is avoided. As a preliminary step to construct commutativity relations among methods, they construct an Direct Access Vector (DAV) for each method. A DAV is a vector whose field corresponds to each attribute defined in the class on which the method operates. Each value composing this vector denotes the most restrictive access mode used by the method when accessing the corresponding field. An access mode of an attribute can have one of three values, N (Null), R (Read) and W (Write) with N < R < W for their restrictiveness. Access mode information is syntactically extracted from the source code of the method at compile-time. After the construction of DAVs of methods, commutativity of methods can be constructed as follows : two methods commute if their corresponding DAVs commute. In turn, two DAVs commute if their access modes are compatible for each attribute. This commutativity relation is defined in the form of a table.

The above technique [Malt,1993] takes access mode information solely from the source code of a method and thus frees the user from determining commutativity relations. Also, this approach can provide finer concurrency by examining attribute level rather than object level locking granularity. Since a DAV of a method is the union of its own DAV and DAVs of all other methods defined in that method, deadlocks due to lock escalation

can be reduced by declaring the most exclusive access mode in a method. In other words, this scheme reduces possibility of lock conversions [Malt,1993], which is a main source of deadlocks. However, concurrency improvement offered by this technique is still limited since run-time information on attributes is not taken into account. Actual access modes of a method can be determined at run-time since every statement is not used due to some branch statements. Those access modes may be less restrictive than access modes in the corresponding DAV. Thus, by taking run-time information, concurrency can be increased.

2.1.2. Conflicts between class definition accesses

In the existing OODBs such as Orion, O_2 and Gemstone ([Garz,1988], [Cart,1990], [Serv,1990]), any class definition access requires a read or write lock (depending on class definition read or class definition write) on an entire class object. Thus, no matter what kind of update operation is performed on a class object, it blocks all other class definition access operations even if they need to access disjoint portions of the class object.

The only two lock modes on class definition access, RD (Read Definition) and MD (Modify Definition), are adopted in [Malt,1991]. They do not consider any finer granules on class definition access such as reading definition of attributes or updating definition of methods, etc. This results in limited concurrency among class definition writes and class definition reads. This is due to that more concurrency can be achieved by taking finer locking granularity such as attribute definition or method definition on class definition.

A higher concurrency among class definition writes is achieved in [Agra,1992] by

providing finer locking granularity rather than taking locking granularity as an entire class object. That is, they classify a class definition into definitions of attributes and methods. For class definition writes on methods, they classify them into three categories and corresponding lock types : 1) add a method m to a class, *add* (m), 2) delete a method m from a class, *del* (m), and 3) replace the implementation of a method m by a new implementation, *rep* (m). For updates on attributes, they classify them into two categories : 1) add an attribute *a* to a class, *add* (*a*), and 2) delete an attribute *a* from a class, *del* (*a*). Thus, as long as two class definition write methods access disjoint portions of a class definition, they can run concurrently. But, they do not consider class hierarchy relationships which are also class definition as well as attribute definition and method definition.

A locking-based technique, called O^2C^2, is proposed in order to increase concurrency between class definition accesses [Olse,1995]. This scheme is based on Orion [Gart,1988] and O_2 [Cart,1990] and has basically the same lock types for class definition access and instance access. Also, their locking granularity for a class definition and an instance access are an entire class object and an instance object, respectively. But, in order to provide more concurrency between class definition accesses, they extract method definition from a class definition. By doing this, there is concurrency between class definition write and method definition. This concurrency is not possible in Orion and O_2. But, they do not consider attribute definitions which are smaller definitions than an entire class object. Thus, they fail to provide further concurrency by taking finer granularity in a class object.

2.1.3. Conflicts between instance access and class definition access

In most concurrency control schemes dealing with class definition writes, a definition write on a class C blocks instance accesses as well as class definition reads on the same class C ([Garz,1988], [Cart,1990], [Malt,1991], [Serv,1990], [Lee,1996]). These techniques are discussed in the following paragraphs..

In [Garz,1988], they adopt S (shared) and X (exclusive) lock modes for class definition reads and writes, respectively. Also, an entire class object is taken for a lock granularity. Since X mode conflicts with all other lock modes, a class definition write blocks all other access to the same class. In their work, S and X modes are used for an instance read and instance write. This results in limited concurrency since a class definition read does not commute with any instance write in the scheme. Actually, a class definition read commutes with an instance write as described in [Cart,1990]. In [Cart,1990], as in [Garz,1988], only two lock modes are used for an entire class object: C_R (class definition read) and C_W (class definition write), respectively. Since C_W conflicts with C_R and any other instance access modes, concurrency between class definition access (class definition read and class definition write) and instance access is limited. As discussed in Section 2.1.2, two lock modes on a class object limits concurrency between class definition write and instance access [Malt,1991] since higher concurrency is possible by taking finer locking granularity in both class objects and instance objects. In [Malt,1991], MD (Modify Definition) blocks any other instance access as well as RD (Read Definition) and MD, since MD lock does not commute with any other lock modes. In [Serv,1990], an *exclusive lock* is required for a class definition write. It guarantees that other transactions (they call

sessions) cannot acquire any kind of lock on the object since an exclusive lock on a class does not commute with any other lock requesting transactions. This results in severe concurrency degradation. Similarly, in [Lee,1996], they adopt two locks on a class object: RS (Read Schema) and WS (Write Schema). Since WS lock is not compatible with any other lock modes, concurrency between a class definition access and an instance access is limited.

A limited concurrency between class definition write and instance access is provided in [Agra,1992] as follows. A lock granularity as individual attributes and individual methods instead of an entire class object is adopted. That is, as long as two class definition access methods or instance access methods access disjoint portions of a class definition, they can run concurrently. For example, updating the implementation code of a method, say M1, can run concurrently with an instance method, say M2. In their work, whenever an instance method M is invoked, the instance method M is read-method-definition-locked *use* (M), which is a read lock for method definition M, instead of a lock on entire class object. Also, attributes, say a_1, a_2, ..., a_k, accessed by that method are read-attribute-definition-locked *use* (a_1), *use* (a_2), ,,,, *use* (a_k). By taking fine lock granularity, two method definitions can be updated as long as they access a disjoint set of attributes. For example, assume that two methods M_1 and M_2 and four attributes a_1, a_2, a_3 and a_4. Also, assume that M_1 accesses three attributes a_1, a_2, a_3. If a transaction invokes M_1, use (M_1) lock for the method and use (a_1), use (a_2), and use (a_3) locks for attributes in M_1 are required. If another transaction is trying to delete attribute a_4, then del (delete attribute) lock is required so that del (a_4) is requested. In this case, two transactions access disjoint

set of attributes, they can run concurrently without any blocking. But, these attribute locks are required each time an instance access method is invoked, they incur large overhead.

In [Olse,1995], an instance write method can run concurrently with a class definition write method on the same class. This concurrency is based on the following argument: "the instance update operation is given a copy of old class definition that is publicly available. Once a class definition is updated, it becomes publicly available and all new instances use it. After all instance update operations that used an old class definition have either aborted or completed, the new class definition are applied to all instances of that class". Although they allow concurrency between instance access and class definition access, their lock granularity is still too big because an entire instance object is taken.

2.2. Class Hierarchy Locking

As discussed in Chapter 1, due to class hierarchy, class definition write and query-type access on a class may need to access more than one class on a class hierarchy. More specifically, updating a class definition at a high level in the hierarchy may require locks for classes at lower levels in the hierarchy. This is not issue in the relational database since locking tables or records are independent [Olse,1995]. That is, if a table needs to be updated, only locking on the table is needed and no more effects will be propagated to any other tables.

There are two approaches dealing with a class hierarchy locking: *explicit locking* ([Garz,1988], [Cart,1990], [Malt,1993]) and *implicit locking* ([Jaga,1993],[Malt,1991], [Lee,1996]).

In *explicit locking*, for an IACH (Instance Access to Class Hierarchy) involving a class, say C, and all of its subclasses, and for a class definition write on a class C, a lock is set not only on the class C, but also on each subclass of C on the class hierarchy. For other types of access (i.e., class definition read and instance access to a single class), a lock is set for only the class to be accessed (also called *target class*). Thus, for an MCA (Multiple Class Access), transactions accessing a class near the leaf level of a class hierarchy will require fewer locks than transactions accessing a class near the root of a class hierarchy. As another advantage of *explicit locking*, it can treat *single inheritance* where a class can inherit the class definition from one superclass, and *multiple inheritance* where a class can inherit the class definition from more than one superclass, in the same way. But, it increases the number of locks required by transactions accessing a class at a higher level in the class hierarchy.

In *implicit locking*, setting a lock on a class C requires extra locking on a path from C to its root as well as on C. *Intention* locks ([Kort,1991],[Date,1985]) are set on all ancestors of a class before the class (also called *target class*) is locked. An intention lock on a class indicates that some lock is held on a subclass of the class. For an MCA on a target class, locks are not required for every subclass of the target class. It is sufficient to put a lock on only the target class (in single inheritance) or locks on the target class and subclasses of the target class which have more than one superclass (in multiple inheritance) [Garz,1988]. (Note that, in [Garz,1988], for a query to some instances of a class and its subclasses, locks are required for instances of each subclass). This is due to that, for an MCA access, only lock on a target class is enough to detect any conflict in

subclasses of the target class. Thus, it can reduce lock overhead over explicit locking. But, *implicit locking* requires a higher locking cost when a target class is near the leaf level in the class hierarchy due to intention lock overhead.

For example, consider the following class hierarchy. In order to update the class definition of class C, each scheme works as in Figure. 2-1.

In Figure 2-1, for *implicit locking*, intention locks IWs corresponding to W (Write) locks are required for all superclasses on the path from C to the root A. Thus, if another transaction, say T, needs to update the class definition in A (i.e., it needs to get W lock on class A), it does not have to search each class through the class hierarchy for conflict checking by the help of the intention lock IW on class A. That is, since IW and W conflict with each other, T's lock request is blocked on class A. (Note that, in implicit locking, there is no conflicts between intention locks, and between an intention lock and an SCA (Single Class Access) lock. But, there can be conflicts between an intention lock and an MCA lock depending on the commutativity relationship ([Garz,1988],[Malt,1993]) On the other hand, an explicit locking does not require any intention locks. But, it requires a Cw (Class Write) lock on each subclass (i.e., class D and E) of the target class through the class hierarchy since any modification of the class definition in C may affect the definitions of its subclasses.

A new class hierarchy locking scheme, which is based on *implicit* locking, is introduced in [Jaga,1993]. They number the classes based on a topological sort with respect to the partial order of inclusion relationships so that an inclusion path goes from a higher-numbered class to a lower-numbered class. For any two classes C_1 and C_2, C_1 is

included in C_2 if every member of C_1 is also a member of C_2. When a transaction requests a lock on some class, it requests intention locks on the class' inclusion path in numeric order, i.e., from the highest number to the lowest number. For any preferred class such as frequently accessed one, locking cost is reduced by assigning it a low number. That is, the number of lock setting can be reduced by starting with a low number rather than a high number. But, even though this scheme can reduce locking overhead in some sense, it still has fundamental disadvantages of *implicit* locking. Their numbering scheme can be applied to only the multiple inheritance case since multiple numberings are possible in only multiple inheritance. For single inheritance, it works like the conventional *implicit* locking scheme.

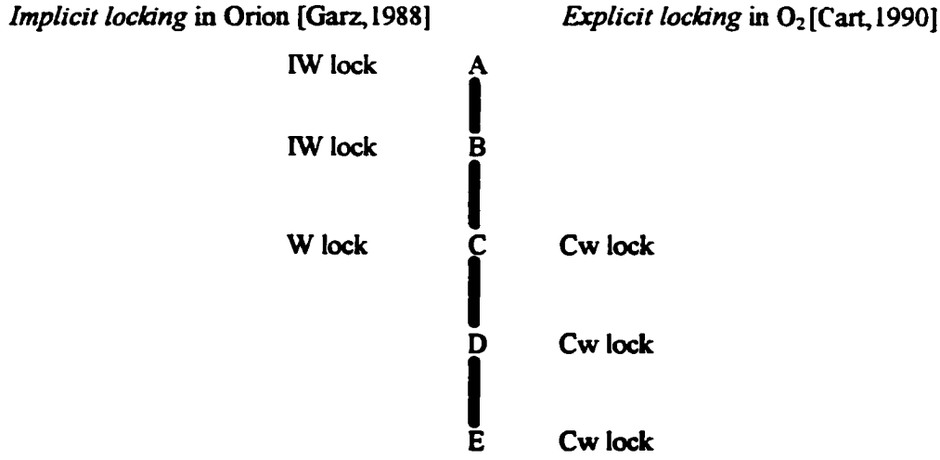


Figure. 2-1. An example of implicit and explicit locking

The class hierarchy locking scheme in [Wang,1990], which is also based on implicit locking, reduces locking overhead by adopting two boundaries: *attribute boundary* and *transaction boundary*. For each attribute in a class, the attribute boundary is

determined as follows. The boundary of attribute a_i in class C is (1) any class up to and including the first superclass where the attribute is redefined and (2) any class up to but not including the first subclass where the attribute is redefined. After all attribute boundaries are determined, the transaction boundary is determined by taking the intersection of individual attribute boundaries. Based on transaction boundary information, locks are granted on classes within the boundary from the highest class to the lowest class. Thus, if a class hierarchy has some attributes redefined, a transaction needs fewer locks than an implicit locking. But, this scheme depends on the number of attributes redefined so that it has basically same problem as in implicit locking. That is, if there is no attribute redefined, their scheme acts like implicit locking.

2.3. Locking on nested method invocations

As we discussed in Chapter 1, an object consists of many disjoint and/or non-disjoint subobjects and also nested method invocations are natural in OODB applications. In the literature, the following approaches are used to deal with nested method invocations.

In an earlier attempt for nested method invocations, a locking technique is developed for disjoint and non-disjoint complex objects in [Herr,1990]. They argue that the traditional approaches dealing with complex objects have the following problems: the granule-oriented problem, protocol oriented problem and authorization problem. Locking an entire complex object may decrease concurrency severely although it can reduce concurrency control overhead. On the other hand, locking individual objects can lead to

tremendous overhead (granule-oriented problem). In non-disjoint complex objects, updating shared object can lead to big overhead since all parent objects of the shared object should be locked (protocol-oriented protocol). Combining concurrency control and authorization components can achieve higher concurrency (authorization-oriented problem). For example, if a transaction does not have right to update some object, an exclusive lock is not required for the object. In order to solve three problems above, for a complex object type, they created the *general lock graph*, which is to solve the granule-oriented problem. In turn, for the general lock graph, the *corresponding object-specific lock graph* which is to solve the protocol-oriented problem and authorization-oriented problem, is constructed. Although their locking protocol considers non-disjoint subobjects or RSO (Referentially Shared Object), it does not exploit semantics in order to enhance concurrency.

As discussed in Chapter 1, OODBs can provide higher concurrency among methods using behavioral properties of methods. That is, although two methods do not commute with each other in terms of read and write operations, they commute using semantics associated with methods. For example, consider a bank object and a method deposit (x, m) where deposit (x, m) is to deposit m dollars after reading an initial balance x. Two deposit methods are considered two write operations, but their execution results are same regardless of their execution order. In existing OODBs such as Orion, O_2 and Gemstone ([Garz,1988], [Cart,1990], [Serv,1990]), they do not exploit any semantics of methods. Their locking schemes are based on read and write operations so that the concurrency provided is very limited.

The nested two-phase locking with *ordered sharing* is proposed in [Agra,1992]. Their work is based on nested two-phase locking in [Moss,1985]. In ordered sharing scheme, locks are required for each atomic operation. They provide better concurrency using *ordered sharing* between locks. Unlike commutativity relationships, when ordered sharing is adopted, a lock request is never delayed until a transaction holding a conflicting lock commits or aborts. That is, a lock request is always granted as follows [Agra,1992]: for a given operation o_1, the set of operations are divided into two categories: the set of operations that commute with o_1, and the set of operations that do not commute with o_1. If o_1, a lock requester, commutes with some operation o_2, a lock holder, (i.e., o_1 has a *shared relationship* with respect to o_2), the lock request is granted and the execution order between them is not important. But if o_1 does not commute with o_2 (i.e., o_1 has *an ordered shared relationship* with o_2), a lock request is granted but the execution order should be preserved by observing a so called *ordered commit rule*: "if a transaction T_1 is granted a lock with an ordered shared relationship with respect to a lock held by T_2 on an object and T_2 is a proper descendent of parent of T_1., then, T_1 cannot commit unless T_2 has committed or aborted. Otherwise, the commit order of T_2 and T_1 is violated and T_2 may reveal its intermediate results to T_1 before abortion or commitment". Also, like the nested transaction model in [Moss,1985], a transaction cannot commit or abort until all its children are terminated, and locks are inherited by its parent when it commits. Using ordered sharing rule, concurrency can be increased in a sense since any lock request is granted and access can be shared as long as commit rule is observed. Even though this scheme increases concurrency, they do not exploit semantics of methods.

In [Muth,1993], a locking-based concurrency control scheme for OODB is presented. They exploit the semantics of methods to increase concurrency. In their work, the conflict between lower level operations or methods can be ignored due to the commutativity of higher level invoked methods in nested method execution. In their work, a lock is required on an object whenever a method or operation is called on the object. Also, locks are converted to *retained locks* at the end of a subtransaction. If a top-level transaction commits, all the locks held are released. They use semantics of methods as follows: when two atomic operations conflict with each other, if they have ancestor which are compatible with each other and the ancestor of the lock holder commits, the lock request is granted. That is, the lock request is not delayed until the top-level transaction commits so that a higher degree of concurrency can be achieved. Similarly, when two methods conflict with each other, the same principle can be applied. But, these authors do not consider OODBs with RSOs. This is a weakness of their work because RSOs are a fundamental property of OODBs and are necessary for modular design as indicated in [Rese,1994].

A semantic two-phase locking protocol for OODB is presented in [Rese,1994]. They consider RSOs and nested method execution. Also, they use semantics of method in order to increase concurrency as follows: any two methods may commute with each other if application programmers decide that their execution order is not important, by using semantics of methods. Thus, two instance write methods commute semantically with each other if their execution order does not violate behavioral aspect of an object. Thus, by taking semantics into considerations, higher concurrency can be achieved. But, the

semantically commuting methods should be executed atomically. In this work, locks are required only for atomic operations shown in [Date,1985]. The protocol works as follows: a subtransaction or top-level transaction T cannot terminate until all of the children are terminated. When a subtransaction is committed, its locks are inherited by its parent. On the other hand, when a transaction is aborted or is top-level and committed, its locks are released. A lock request is granted if one of the three following conditions are met: (a) no other transaction holds a conflicting lock, (b) if conflicting locks are found after checking commutativity relationship table, such locks are held by its ancestors and (c) if conflicting locks are found after checking commutativity relationship table and these locks are held by non-ancestors of lock holders, then one of the ancestors of the lock holders (not including the lock holders) and some ancestors of the lock requester commute. By applying rule (c), two semantically commuting methods are guaranteed to be executed atomically. Locking for each atomic operation incurs an overhead which has a critical effect on OODBs where many transactions are long-lived. Also, locking for each atomic operation may incur the following problem: it is likely that lock conversion from less restrictive lock (i.e., read lock) to more restrictive lock (i.e., write lock) may occur. This lock conversion is known as main source of deadlocks [Malt,1993]. Also, [Rese,1994] assumes that the commutativity relationships between methods are well-defined and can be derived based on semantics as well as the specification of the class and its methods. But, [Rese,1994] fails to provide a formal way to construct commutativity relationships among methods.

A semantic locking, called ESL (Enhanced Semantic Locking), is introduced in [Kwon,1997]. Basically, their locking scheme is same as the locking scheme in

[Rese,1994]. They consider semantics of methods and RSO as follows. For semantics of methods, they argue that methods can provide rich semantics than read and write operations since methods usually represent behavior of objects. Semantics can be provided at the discretion of the application programmer. On the other hand, in order for ESL to support RSO, they adopt "in-place" conflict resolution policy as follows. That is, lock modes are not associated with methods. In this work, commutativity of methods is determined at the time of methods invoke shared subobject at the same time. This scheme requests a lock whenever an read or write atomic operation is invoked in a method as in [Rese,1994]. But, ESL is different from [Rese,1994] in that lock conversion for retained lock is prohibited. Basically, this scheme has the same problem as in [Rese,1994]. That is, it may incur big overhead since a lock is required for each atomic operation invoked in a method. Also, an entire instance object locking granularity is adopted so that two methods accessing a disjoint set of attributes may conflict each other.

Chapter 3

AN INTEGRATED CONCURRENCY CONTROL SCHEME

In this Chapter, an integrated concurrency control scheme for three access types is developed. The principle for an integrated concurrency control is as follows: for conflict among methods, the finer locking granularity is adopted for both instance access and class definition access so that higher concurrency is achieved. Especially, for instance access, DAV concept is used in order to adopt attribute locking granularity instead of an entire instance object. These DAVs are also used for automation of commutativity relationships among instance access methods so that application programmers are free from burdens. These DAVs can reduce locking overheads and possibility of deadlocks. Also, breakpoints are used to adopt run-time information so that further concurrency can be achieved. For class definition access, fine lock granularity is adopted so that two methods can run concurrently as long as they access disjoint portions of class objects. For class hierarchy locking, special classes are used in order to reduce locking overhead where a special class is defined as a class on which class definition writes are performed frequently. The proposed class hierarchy scheme incurs fewer locks than both existing schemes, for any type of access. Finally, for nested method invocations, semantic information is used in order to increase higher concurrency among methods. Also, parent/children parallelism is adopted for better response time. For RSOs, conflicts are detected on actual method invocation so that low concurrency due to static commutativity relationships can be avoided.

Basic approaches for each access type are introduced in Section 3.1. Based on these approaches, the complete concurrency control algorithm is constructed in Section 3.2. Finally, in Section 3.3, the correctness of an integrated concurrency control is proven.

3.1. Handling individual access types

3.1.1. Conflicts among methods

As discussed in Chapter 2, there are two types of access to an object: instance access and class definition access. Thus, there are three kinds of conflicts depending on a lock holder and a lock requester: conflicts among instance accesses, conflicts among class definition accesses and conflicts between instance access and class definition access. For each type of conflict, principles to increase concurrency are presented in following Sections.

3.1.1.1. Conflicts among instance accesses

The principles are based on [Malt,1993]. Their work has the following characteristics: in order to enhance concurrency among instance accesses, attribute level lock granularity is taken instead of an entire instance object granularity. Also, lock is required for each instance access method invocation instead of atomic operations in an instance access method. By doing this, locking overheads as well as possibility of deadlocks can be reduced significantly. But, in [Malt,1993], dynamic information is not utilized, thus concurrency provided is still limited. In the proposed scheme, further concurrency can be achieved by adopting run-time information.

The work in [Malt,1993] is summarized as follows: lock is required for an instance method invocation. Also, attribute level lock granularity is considered. Thus, although two

instance methods conflict in terms of read or write operations, as long as their access modes on individual attributes do not conflict, two methods can run concurrently. In order to build commutativity relationships among methods, they construct an DAV for each method. An DAV is a vector whose field represents access mode of each attribute. Each value consists of one of three access modes: N (Null), R (Read) and W (Write). After construction of DAVs of methods, commutativity of methods can be constructed as follows: two methods commute if their corresponding DAVs commute. In this work, in addition to further concurrency by taking finer locking granularity, the possibility of deadlocks can be reduced. This is due to that an DAV of an instance access method represents most restrictive access mode for each attribute, lock escalation, which is a main source of deadlock, can be reduced.

In the proposed scheme, four objectives will be pursued. First, it still automates the process of commutativity relation construction. Second, it provides more concurrency than read and write access modes on methods. Third, it reduces deadlocks due to lock escalation. Finally, it increases concurrency among methods by exploiting run-time information.

The above objectives can be achieved as follows. In the proposed scheme, DAVs are used so that attribute level locking granularity is possible. Also, the construction of the commutativity relationships among instance access methods is based on these DAVs. Since the construction of DAVs is solely based on access modes of attributes, automation of commutativity relationships is possible. Also, locks are required for each method instead of each atomic operation so that locking overhead and the possibility of deadlocks

can be reduced. In [Malt,1993], DAVs represent static access modes of attributes. Thus, concurrency is still limited since some attributes are not accessed during actual method execution. In order to provide further concurrency, actual access modes of attributes are reflected using breakpoints where a breakpoint represent a code segment executed regardless of execution path. In [Malt,1993], the commutativity table entries contains only method names. But, in the proposed scheme, breakpoints as wel! as method names are included in the commutativity table entries. After a method execution, locks are changed from method to breakpoint, which is less restrictive, so that further concurrency is possible.

Similar to [Malt,1993], a two-phase pre-analysis is needed. It consists of two steps : 1) construction of DAV for each method and 2) construction of a commutativity table of methods. In each method, a break point is inserted by a programmer or a compiler when a conditional statement is encountered. Every method has a special break point called *first break point* before the first statement in the method. There are three kinds of DAVs m each method : 1) a final DAV of the first break point, which is a DAV of the entire method as in [Malt,1993] 2) an initial DAV of the first break point, which is a union of access modes of each attribute used by statements between the first break point and the next break point and access modes of each attribute used by statements from the first statement to the last statement that are executed regardless of execution paths. A union operation '+' is equivalent to *max*, e.g., N + W = W, that is, take more restrictive mode among two operations. Note that this union operation is necessary to build worst case access mode of each attribute. and 3) an initial DAV of every other break point, which

contains access modes of all attributes used by statements between this break point and the next break point (or end of the method).

For example, assume that there are three methods M1, M2 and M3 and an object O1 with four attributes a_1, a_2, a_3 and a_4. A, A1, A2, and A3 are breakpoints of M1, B is a breakpoint of M2, and C, C1, and C2 are breakpoints of M3. Note that the operation '+' stands for union. The contents and DAVs of each method are given below.

method M1
[A]
read a_1

If (a_1 > 100) then
then
{[A1]
a_2 <= a_1
End if

read a_2 — (*)

If (a_2 > 100) then
[A2]
a_3 <= a_2
End if
read a_3 — (**)
If (a_3 > 100) then
{[A3]
call M2
End if

method M2
[B]
read a_1
read a_4
a_4 <= a_1

method M3
[C]
read a_1

If (a_1 > 100)

{[C1]
return a_1}
else
{[C2]
read a_2
return a_2}
end if

The DAVs constructed for method M1 are :

initial DAV of [A] : {DAV of [A]} + {DAV of (*)} + {DAV of (**)}
= [R,N,N,N] + [N,R,N,N] + [N,N,R,N]
= [R,R,R,N]

initial DAV of [A1] : [R,W,N,N]
initial DAV of [A2] : [N,R,W,N]
initial DAV of [A3] = final DAV of M2 = [R,N,N,W]

final DAV of [A] = initial DAV of [A] + initial DAV of [A1] + initial DAV of [A2] +
initial DAV of [A3] = [R,R,R,N] + [R,W,N,N] + [N,R,W,N] + [R,N,N,W] = [R,W,W,W]

Similarly, the DAVs for M2 are :

Final DAV of [B] : [R,N,N,W]
initial DAV of [B] : [R,N,N,W]

and DAVs for M3 are

Final DAV of [C] : [R,R,N,N]
initial DAV of [C] : [R,N,N,N]
initial DAV of [C1] : [R,N,N,N]
initial DAV of [C2] : [N,R,N,N]

While in the scheme proposed in [Malt,1993], the DAVs for the methods would be:

DAV of M1 : [R,W,W,W] DAV of M2 : [R,N,N,W] DAV of M3 : [R,R,N,N]

After the construction of the breakpoints' DAVs in all methods, a commutativity relation of methods is constructed. Unlike in [Malt,1993], entries in the commutativity table contain breakpoints as well as method names. The construction process is as follows ([Jun,1995-1],[Jun, 1995-2]).

In a commutativity table, a lock requester's entries contain names of the final DAVs of the first break points in all methods (represented as N_F where N is the name of the first break point in each method). For example, A_F represents a final DAV of the first break point A in method M1, which is [R,W,W,W]. A lock holder's entries contain names of the final DAV of the first break point (in the form of N_F), name of the initial DAV of the first break point (in the form of N_I) and names of the initial DAVs of other break points (represented as Ni where $1 \leq i \leq$ number of breakpoints -1) in each method. For example, in method M1, A_F, A_I, A1, A2 and A3 represent the following DAVs, [R,W,W,W], [R,R,R,N], [R,W,N,N], [N,R,W,N] and [R,N,N,W], respectively. Since we assume the worst case access mode for each attribute before execution, lock requesters

always have the most restrictive access modes (i.e., final DAVs of the first break points). But, after a method execution, a lock holder may have a less restrictive access mode. Two break points commute if their corresponding DAVs commute. Two DAVs commute if, for every attribute, its access mode in the two DAVs commute. Table 3-1 gives the commutativity tables constructed in the proposed scheme and in the scheme proposed in [Malt,1993].

The proposed concurrency control is based on two-phase locking [Eswa,1976]. When a transaction invokes a method on an object, it gets a lock containing the final DAV of the first break point in the method (represented as N_F where N is the name of the first break point in each method). As the transaction meets a break point during run-time, the break point is recorded. After the method execution, the lock is changed from N_F to N_I, N_J,...N_S where N_I is a name representing the initial DAV of the first break point and N_J...N_S are names representing the initial DAVs of the other break points encountered during the method execution. Since the union of DAVs of N_I, N_J,...N_S may be less restrictive than the DAV of N_F, this can give more concurrency to other transactions which request locks on the same object.

The commutativity table of the proposed scheme [Malt,1993] for object O1

		lock holders									
		A_F	A_I	A1	A2	A3	B_F	C_F	C_I	C1	C2
lock requester	A_F	N	N	N	N	N	N	N	Y	Y	N
	B_F	N	Y	Y	Y	N	N	Y	Y	Y	Y
	C_F	N	Y	N	Y	Y	Y	Y	Y	Y	Y

Commutativity table in for object O1

	M1	M2	M3
M1	N	N	N
M2	N	N	Y
M3	N	Y	Y

Table 3-1. Examples of commutativity tables constructed for the proposed scheme and for [Malt,1993]

Below is an example that illustrates how the proposed scheme works and how concurrency is improved over that proposed in [Malt,1993]. Consider the following concurrent transaction invocations T1, T2, T3 and T4 on class O1, which is defined previously, with three instances i1, i2 and i3. (Mi, I_J) is an invocation of method Mi on instance I_J. Assume that, starting at time t, each method call execution (Mi, I_J) takes 1 second and a transaction commits as soon as its last method call execution is finished.

(time)	t	t+1	t+2	t+3	t+4	t+5	t+6
(transactions)							
T1 :	(M1,i1)			(M1,i2)			
T2 :		(M2,i1)					
T3 :			(M3,i2)		(M3,i3)		(M3,i3)
T4 :						(M2,i3)	

Assume that, in the proposed work, the lock format on instance has the following form [trans-name, m-name(B_1)(B_2) ...(B_n)] where *trans-name* is a transaction holding a lock, *m-name* is a method invoked, B_1, B_2,...B_n are break points encountered during the method execution. In [Malt,1993], the lock format has the following form : [*trans-name*, *m-name*] where *trans-name* and *m-name* have same meaning as the lock table in proposed work.

The following example shows how the proposed scheme works and how the proposed scheme gives better transaction response time than the scheme in [Malt,1993] by adopting run-time information. Starting from time t, the locks on each instance are changed as follows.

The Proposed Scheme	Scheme in [Malt,1993]

t : i1 : [T1, M1:A_F]
// lock request is granted since no other lock is held
// Assume that break points encountered after execution
// are, A_1 and A_2
	i1 : [T1,M1] // lock request is granted
	// since other lock is held on i1

t+1 : i1 : [T1, M1:(A_I)(A_1)(A_2)] [T2, M2:(B_F)]
// B_F commutes each of A_I, A_1, A_2 so that lock is granted
// T2 is committed since T2 does not have any more
// method invocation

i1 : [T1,M1] blocked : [T2,M2]
// lock request M2 by T2 is blocked
// since M2 and M1 do not commute

t+2 : i1 : [T1, M1:(A_I)(A_1)(A_2)]
i2 : [T3, M3:(C_F)]
// lock request is granted since no lock is held on i2
// Assume that break points encountered after execution
// are C_I, C_1

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T3,M3]
// lock request is granted since no lock is
// since no lock is held on i2

t+3 : i1 : [T1, M1:(A_I)(A_1)(A_2)]
i2 : [T3, M3:(C_I)(C_1)] [T1, M1:(A_F)]
// Assume that break points encountered after execution
// are A_I and A_1.
// T1 is committed

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T3,M3] blocked : [T1,M1]

t+4 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3(C_F)]
t+4.5 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3(C_I)(C_2)]
// Assume that break points encountered after execution
// are C_I and C_1.

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T3,M3] blocked : [T1,M1]
i3 : [T3,M3]

t+5 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3:(C_I)(C_2)] [T4, M2:(B_F)]
t+5.5 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3:(C_I)(C_2)] [T4, M2:(B_I)]
// T4 is committed

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T3,M3] blocked : [T1,M1]
i3 : [T3,M3] [T4,M2]
// T4 is committed

t+6 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3:(C_I)(C_1)] [T3, M3:(C_F)]
t+6.5 : i2 : [T3, M3:(C_I)(C_1)]
i3 : [T3, M3:(C_I)(C_1)] [T3, M3:(C_I)(C_1)]
// Assume that break points encountered after execution
// are C_I and C_1.

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T3,M3] blocked : [T1,M1]
i3 : [T3,M3]
// T3 is committed

t+7 : T3 is committed (all transactions are committed)

i1 : [T1,M1] blocked : [T2,M2]
i2 : [T1,M1] // T1 is committed

t+8 :

i1 : [T2,M2] // T2 is committed

t+9 :

all transactions are committed

From the above example, it is concluded that the proposed scheme gives better transaction response time using run-time information.

In this approach, a method may have many break points depending on the method's logic. This requires larger commutativity tables and also incurs much run-time overhead for lock changes and commutativity checking. Thus, a way to reduce the number of break points in a method is necessary in order to reduce this overhead. Some strategies are presented to reduce the number of break points as follows.

- **Breakpoint optimization strategy 1**

It is known that the union of DAVs encountered after a method execution is at least as restrictive as the initial DAV of the first break point. Thus, if an initial DAV of some break point in a method has equal or less restrictive DAV than the initial DAV of the first break point, it is not necessary to keep track of it and to include it as a member of the commutativity table. This is due to that further concurrency can not be achieved by keep tracking of such break points. For example, the DAV of the break point C1 of method M3 in the previous example is [R,N,N,N], which is the same as the initial DAV of the first break point C. Thus, we do not have to include the initial DAV of C1 in the commutativity table.

- **Breakpoint optimization strategy 2**

The key idea in this strategy is to control concurrency level at the expense of run-time overhead. That is, if higher concurrency is necessary, keep track of more breakpoints during run-time. Otherwise, make DAVs more restrictive so that run-time overhead can be reduced at the expense of concurrency. For this purpose, define *the most restrictive access mode* (MRAM) for each method. MRAM can have one of two values, R (Read) or W

(Write). A method *m* has MRAM = R if it is an instance read method. On the other hand, a method *m* has MRAM = W if there is at least one attribute with a W mode in method *m*. Also, define *Access Mode Change Percentage* (AMCP), $0 \leq AMCP \leq 100$, for each break point. The AMCP of break point B_i in method m is defined as follows.

$$\text{AMCP of } B_i = \frac{\text{the number of attribute in Bi whose access mode is MRAM}}{\text{the number of attribute in m whose access mode is MRAM}}$$

For example, if the initial DAV of B_i and the final DAV of B_F in method m are defined as [R,R,W,N] and [W,W,W,R], respectively, then AMCP of B_i is 33% (=1/3) since MRAM is W and the number of attributes in B_i whose access mode = W (MRAM) is 1, and also the number of attributes in method m whose access mode = W is 3.

If a method has many breakpoints, large overhead may be incurred due to commutativity table searching for conflict checking and run-time breakpoint tracking Thus, to reduce the number of break points (i.e., reduce the commutativity table size), let a break point, say B, have an initial DAV and be an entry in the commutativity table only if AMCP of B is greater than P% where P ($0 \leq P \leq 100$) is defined. (The specific value of P can be chosen depending on whether concurrency is important or not). Otherwise, perform the following operation and do not include the initial DAV of B in the commutativity table.

the initial DAV of the first break point = the initial DAV of the first breakpoint + the initial DAV of B

For example, consider the following DAVs of methods M1, M2, and M3.

method M1	method M2	method M3
Final DAV of [A] : [R,W,W,W]	Final DAV of [B]: [R,N,N,W]	Final DAV of [C]: [R,R,N,N]
initial DAV of [A] : [R,R,R,N]		initial DAV of [C]: [R,N,N,N]
initial DAV of [A1]: [R,W,N,W]		initial DAV of [C1]: [R,N,N,N]
initial DAV of [A2]: [N,R,W,N]		initial DAV of [C2]: [R,R,N,N]
initial DAV of [A3] : [N,R,R,R]		

MRAM of M1 = W	MRAM of M2 = W	MRAM of M3 = R
from Final DAV of [A]	from Final DAV of [B]	from Final DAV of [C]

AMCP of each break point in each method is as follows.

For Final DAV of [A], AMCP = 100 (=3/3)
For initial DAV of [A1], AMCP = 66.6 (=2/3)
For initial DAV of [A2], AMCP = 33.3 (=1/3)
For initial DAV of [A3], AMCP = 0 (=0/3)

For Final DAV of [B], AMCP = 100 (= 1/1)

For Final DAV of [C], AMCP = 100 (=2/2)
For initial DAV of [C], AMCP = 50 (=1/2)
For initial DAV of [C1], AMCP = 50 (= 1/2)
For initial DAV of [C2], AMCP = 100 (=2/2)

For example, AMCP of breakpoint [A2] = 33.3%. This is because the number of attribute in [A2] whose access mode = W is 1, and the number of attributes in the final DAV of [A] whose access mode = W is 3. Also, AMCP of breakpoint [A3] = 0 since the number of attributes in [A3] whose access mode = W is zero. Suppose we define P as 30%. Then, the following break points participating as entries in the commutativity table can be obtained.

Final DAV of [A] : [R,W,W,W]	Final DAV of [B] : [R,N,N,W]	Final DAV of [C] : [R,R,N,N]
initial DAV of [A] : [R,R,R,R]		initial DAV of [C] : [R,N,N,N]
initial DAV of [A1] : [R,W,N,W]		initial DAV of [C2] : [R,R,N,N]
initial DAV of [A2] : [N,R,W,N]		
=> [A3] is added to the initial DAV of [A].		[C1] is removed due to the breakpoint optimization strategy 1.

Strategy 1 is to eliminate any breakpoint which is not helpful to increase concurrency. By removing those break points, we can reduce run-time overhead and storage overhead. Strategy 2 is to give trade-off between concurrency and run-time

overhead. That is, the higher AMCP is, the less run-time overhead is; but this results in less concurrency. On the other hand, with less AMCP value, more concurrency can be provided at the expense of run-time overhead since access modes of attributes can be less restrictive so that more transactions can run concurrently. Strategy 2 is to adjust the degree of concurrency depending on applications. For example, any application adopting long transactions may require small AMCP value since lock waiting time due to conflict may outweigh concurrency control overhead. The strategy 1 can be applied to any method without further information such as the frequency of method invocations. But, it provides a limited form of concurrency since methods may not have many conditional branch statements. On the other hand, in strategy 2, the level of concurrency for each method can be changed by increasing or decreasing its AMCP value. But, deciding the value of AMCP for each method may need additional cost since access frequency in breakpoints should be analyzed.

This optimization process is done at compilation time. Thus, it is not necessary to optimize break points for each method invocation, resulting in a reduction in run-time overhead. But, for OODBS whose schema is continuously evolving, the optimization incurs some overhead since method contents (also corresponding DAVs) may change frequently.

3.1.1.2. Concurrency among class definition access

In [Kim,1990], the taxonomy of class definition update, which is comprehensive among OODBs, is provided as follows. There are two types of changes to the schema of an OODB. One is to the definition of a class. This includes changes to the attributes and

methods defined for a class, such as changing the name or domain of an attribute, adding or dropping an attribute or a method. Another type of change is to the class-hierarchy structure. This includes adding or dropping a class, and changing the superclass/subclass relationship between a pair of classes. We represent MA (Modify Attribute), MM (Modify Method), and MCR (Modify Class Relationship) as *Modify definition of an attribute*, *Modify definition of a method, and Modify the superclass/subclass relationship*. respectively.

The definition of each class object can be classified into three groups : definition of the class itself, definition of attributes in the class, definition of methods in the class, respectively [Kim,1990]. The definition of a class includes the name of the class, set of all attributes defined for or inherited into the class, sets of superclasses and subclasses of the class, and a set of methods defined or inherited into the class. The definition of attributes contains the class to which the attribute belongs, the superclass on which the attribute is defined if the attribute is inherited, and the domain of the attribute. Likewise, the definition of methods includes the class to which the method belongs, the name of the method, the implementation code of the method, and the superclass in which the method is defined if the method is inherited. Assume that RA (Read Attribute), RM (Read Method) and RCR (Read Class Relationship) represent Read definition of attributes, Read definition of methods, Read definition of class itself, respectively.

Assume that updating definition of any method does not affect definition of any attribute. The following table 3-2 gives the commutativity relationships among class

definition updates and class definition reads, where Y and N stands for *commute* and *not commute*, respectively.

	MA	MM	MCR	RA	RM	RCR
MA	N	N	N	N	N	N
MM	N	N	N	Y	N	N
MCR	N	N	N	N	N	N
RA	N	Y	N	Y	Y	Y
RM	N	N	N	Y	Y	Y
RCR	N	N	N	Y	Y	Y

Table 3-2. Commutativity relationship among class definition access

Using the above commutativity relationship, for a class definition access method, a finer granularity lock can be obtained on class than conventional OODBS such as Orion and O_2 do since a class definition object is divided into three parts: attribute, method and class relationship. The lock granularity in the proposed work is one of MA, MM and MCR (for class definition update) and RA, RM. RCR (for class definition read) [Jun,1995-3]. Whenever a class definition access method is invoked, the commutativity needs to be checked between the lock holder and the lock requester using the commutativity table in Table 3-1 and grant a lock if they commute. Also, the lock format is [*trans-name*, *lock-type*] where *trans-name* is a transaction holding a lock and *lock-type* is a class definition access lock type ∈ { MA, MM, MCR, RA, RM, RCR}. For example, consider the following transactions on class O1. For example, consider the following two transactions T1 and T2. T1, at time t, is modifying the definition of a method and T2, at time t+1, is reading the definition of an attribute. The following diagram shows the possible execution of two transactions T1 and T2 by the proposed scheme and Malta's scheme.

<Malta's scheme>

T1 T2

---

t MM (delete a method)
t+1 RA (read definition of an attribute)

< Proposed Scheme >

t C1 : [T1,MM]
t+1 C1 : [T1,MM] [T2,RA]

Figure 3-1. Illustration of the Proposed scheme

3.1.1.3. Concurrency between class definition access and instance access

Fine concurrency between class definition access and instance access can be achieved based on following principle: take finer granularity for an instance access and a class definition access so that let transaction run concurrently as long as they access disjoint portion of objects.

In order to provide fine concurrency between attribute definition accesses and between attribute definition access and instance access, an *attribute access vector* (AAV) is created whenever an MA or RA or an instance access method has a lock (that is, no other active MA or RA or instance access method has the lock). Each field in AAV represents an attribute. For each attribute field, a value can have one of three values: W (update, set by MA), R (read, set by RA or instance access method) and N (null).

- The incoming MA or RA method checks commutativity using this vector and set R (for RA) or W (for MA) on the corresponding attribute in AAV if, for each attribute to be accessed by an incoming method, the lock modes of the requester and the holder are compatible.
- The incoming instance method checks commutativity comparing this vector with its DAV as follows. : For each attribute accessed by the instance access method, check if the attribute is W locked in AAV. If so, block the lock request by the instance access method.

Otherwise, set R lock (in AAV) to each attribute accessed by the instance access method and grant the lock. Note that we do not change the access modes of the AAV after the instance access method execution in order to avoid excessive runtime overhead.

• Whenever an MA or RA or an instance access method is committed by an invoking transaction, it resets the vector AAV (that is, if there is no active MA or RA or instance access method, it removes the AAV).

Second, in order to increase concurrency between method definition accesses, between method definition access and instance access, and between method definition access and attribute definition access, each individual method is taken as locking granularity instead of taking all methods of one transaction as locking granularity.

A *method access vector* (MAV) is created whenever an MM or RM or an instance access method has a lock (that is, no other active MM or RM or instance access method has the lock). Each field in MAV represents a method. For each method field, a value can have one of three values: W (update, set by MM), R (read, set by RM or instance access method) and N (null). The vector MAV is to give parallelism between method definition updates and between method definition updates and method definition read. For example, while the implementation code of method M1 is updated by a transaction T1, another transaction T2 can read the implementation code of method M2.

• The incoming MM or RM method checks commutativity using this vector and set R (for RM) or W (for MA) on the corresponding method field in MAV if the lock is granted. Also, check commutativity using AAV and set R on the corresponding attribute field in

AAV if, for each attribute to be accessed by an incoming method, the lock modes of the requester and holder are compatible.

• The incoming access instance method checks commutativity comparing this vector with its DAV as follows : Check if the method field is W locked in MAV. If so, block the lock request by the instance method. Otherwise, set R lock (in MAV) in the corresponding method's field. Also, check commutativity comparing AAV with its DAV and set a R lock on each field if compatible.

• Whenever an MM or RM or instance access method is committed by the invoking transaction, it resets the vector MAV (That is, if there is no active MM or RM or instance access method, it removes the MAV) and AAV.

The following table gives the commutativity relationships among class definition updates (MA,MM,MCR), class definition reads (RA,RM,RCR), and instance access methods (noted by I).

	MA	MM	MCR	RA	RM	RCR	I
MA	Δ	Δ	N	Δ	Δ	N	Δ
MM	Δ	Δ	N	Y	Δ	N	Δ
MCR	N	N	N	N	N	N	N
RA	Δ	Y	N	Y	Y	Y	Y
RM	Δ	Δ	N	Y	Y	Y	Y
RCR	N	N	N	Y	Y	Y	Y
I	Δ	Δ	N	Y	Y	Y	Δ

Table 3-3. Commutativity relationship between class definition access and instance access where Δ means that two methods commute as long as they are accessing disjoint portions of an object.

For example, with class CL1 defined in Section 3.1.1.1, consider the following method invocations by transactions T1, T2 and T3. At time t, T1 is reading definition of attribute a_3. At time t+1, T is invoking a method M2 on an instance I_1. At time t+2, T3 is invoking a method M3 on instance I_1. At time t+3, T1 is modifying the definition of method M1. Finally, at time t+4, T2 is reading the definition of an attribute a_2.

time	T1	T2	T3
t	CL1:RA(a_3)		
t+1		CL1:M2 on I_1	
t+2			CL1: M3 on I_1
t+3	CL1: MM(M1)		
t+4		CL1: RA(a_2)	

The followings show how lock are changed on class CL1 and instance I_1 by each transaction at a time.

t. CL1 : AAV : [a_1:N, a_2:N, a_3:R(T1), a_4:N]

// The method call RA on attribute a_3 by T1 needs to create an AAV since no other
// transaction has invoked MA, RA or an instance access method and set W on the a_3 field.

t+1. CL1 : AAV[a_1:R(T2), a_2:N, a_3:R(T1), a_4:R(T2)], MAV[a_1:N, a_2:R(T2), a_3:N]
I_1 : [M2(B_F), T2]

// For T2 invoking M2 on I_1, checks AAV if, for each attribute accessed by M2, there is an
// incompatible attribute access mode using DAV of M2. Also, check M2 field in MAV if
// some other transaction is updating M2.

t+2. CL1 : AAV[a_1:R(T2,T3), a_2:R(T3), a_3:R(T1), a_4:R(T2)], MAV[a_1:N, a_2:R(T2),
a_3:R(T3)]
I_1 : [M2(B_F), T2], [M3(C_F), T3]

// Perform the same task like T2 in step 2. Assume that break points C_1 and C_1 are met
// during execution of M3.

t+3. CL1 : AAV[a_1:R(T1,T2,T3), a_2:R(T1,T3), a_3:R(T1), a_4:R(T1,T2)],

MAV[a_1:W(T1), a_2:R(T2), a_3:R(T3)]
I_1 : [M2(B_F), T2], [M3(C_I, C_1), T3]

// Since any transaction is not invoking M1, T1's request is granted. Thus, set W on M1
// field in MAV and set R on AAV for each attribute used in M1.

t+4. CL1 : AAV[a_1:R(T1,T2,T3),a_2:R(T1,T2,T3),a_3:W(T1),a_4:R(T1,T2)],
MAV[a_1:W(T1), a_2:R(T2), a_3:R(T3)]
I_1 : [M2(B_F), T2], [M3(C_I, C_1), T3]

// Since any transaction is not modifying definition of an attribute a_2, T2's request is
// granted. Thus, set R on a_2 field in AAV.

It is possible that updating AAV and MAV whenever an instance access method is invoked incurs too much overhead. This is true especially for those OODB systems whose schema need not be changed frequently. In this case, the frequency of schema update is not high. Thus, the overhead by the technique used in this subsection outweighs the concurrency increased. For such OODB systems, take lock granularity as all attributes for RA or MA rather than individual attributes. For a method definition access, take lock granularity as all methods for RM or MM. Likewise, for instance access methods, use RA and RM locks on class, instead of using AAV and MAV. Thus, the following protocol can be used as an alternative.

- When a transaction invokes an instance access method, get RA and RM locks and check commutativity among instance access methods.
- When a transaction which has invoked an instance access method is committed, release RA and RM locks.

3.1.2. Class hierarchy locking

3.1.2.1. Basic Idea

The objective here is to develop a new class hierarchy locking scheme which can be used for any OODB applications with less locking overhead than both existing schemes, explicit locking and implicit locking. To achieve this, some classes in a class hierarchy are designated as *special classes*. Roughly, a *special class* (SC) is defined as a class on which class definition writes or IACHs (Instance Access to Class Hierarchy) are performed frequently. For the proposed scheme, how to determine if a class is a SC or not will be discussed in Section 3.1.2.6.

In this new class hierarchy locking scheme, intention locks are set on SCs only; thus, locking overhead is reduced compared to implicit locking which requires intention locks on every superclass of the target class. When a transaction needs to access an SC which is already intention locked, by invoking an MCA lock on it, a concurrency control can reduce conflict checking overhead due to the help of the intention locks. That is, every conflict can be detected by the help of commutativity relationships between intention locks and MCA (Multiple Class Access) locks on the SC. On the other hand, if a class has little or no possibility of being accessed by an MCA, there is no need to set an intention lock on that class since SCA (Single Class Access)s do not use intention locks to check for conflicts. As we discussed earlier, there is no conflict between an intention lock and an SCA lock and any conflict is determined only at the target class. Thus, unlike implicit locking, we do not have to set an intention lock on every class on the path from a target class to a root.

In order to have fewer locks required for an MCA than those required by explicit locking, the proposed scheme works as follows: for an SCA, a lock is set on only the

target class, like explicit locking. For MCAs, unlike explicit locking which requires locks on the target class and all its subclasses, locks are set on every class from the target class to the first SC through the subclass chain of the target class. If there is no such SC, then locks are set on the leaf classes. If the target class is an SC, then set a lock only on the target class. Thus, by choosing an SC as a class on which MCAs are performed frequently, the locking overhead can be reduced.

For example, consider the following class hierarchy in Figure. 3-2.a. Assume that a transaction T1 invokes an MCA lock on class C6. Let LS1 be a lock setting for T1. Assuming that classes C1, C4 and C7 are SCs, then Figure. 3-2.b, 3-2.c, and 3-2.d show how locks are set in explicit locking, implicit locking, and the proposed scheme. Note that C_1 and C_{10} are a root and a leaf class, respectively.

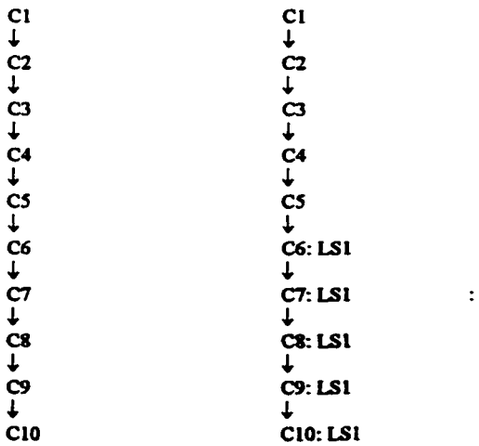


Figure. 3-2.a
Class hierarchy

Figure 3-2.b
Explicit locking

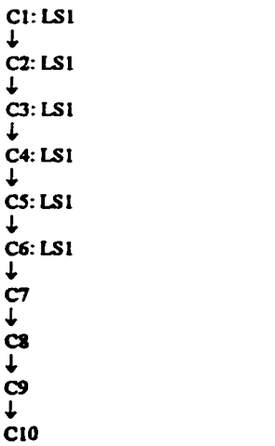


Figure 3-2. c
Implicit locking

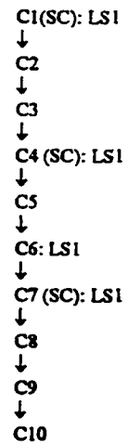


Figure. 3-2.d
the proposed scheme

3.1.2.2. Lock Modes

In order to illustrate the principle for reducing overhead for class hierarchy locking, the following locking granularity and lock modes are used. The principle is to keep low overhead for any type of access to a class hierarchy. At first, assume the lock granularity as follows: adopt instance level granularity for instance access and entire class object for class definition access like Orion [Kim,1990] and O_2 [Cart,1990]. Below are locks needed for different types of instance and class access ([Jun,1996],[Jun,1997-1],[Jun,1997-2]). For convenience, lower-case letters and upper-case letters are used to name locks on an instance and a class, respectively.

Operations	Locks needed
instance read	r (on target instance) TR, IMPR, INTSR, INTSPR, QR, PQR (on target class or its superclasses)
instance write	w (on target instance) TW, IMPW, INTSW, INTSPW, QW, PQW (on target class or its superclasses)
Class definition write	CW (on target class) INTSW (intention lock for each SC on the path from the target class to its root)
Class definition read	CR (on target class) INTSR (intention lock for each SC on the path from the target class to its root)

• instance read

- (for SCA) TR (Target Read) lock means that some (not all) instances of a target class are *r* locked. A TR lock is set on a *target class* whenever an *r* lock is set on its instance.

- (for SCA) IMPR lock (Implicit Read on target class) means that all instances are *read* locked *implicitly*. Like both explicit locking and implicit locking, we reduce locking

overhead by setting an IMPR lock on the target class, not on individual instances, if the majority of instances are accessed.

- (for MCA) QR (Query Read on a target class) means that all instances of a target class and its subclasses are *read* locked as in implicit locking. We reduce locking overhead by setting an QR lock on only the target class, not setting IMPR lock on all subclasses of the target class.

- (for MCA) PQR (Partial Query Read on a target class) means that some instances of a target class and its subclasses are read locked. For access to some instances of a target class and its subclasses, we put only a PQR lock on a target class and a *r* lock on each individual instance to be accessed. Thus, unlike QR lock, when a PQR lock is set on a class, say C, any instance write to some instances of a subclass of C may not conflict because actual conflicts can be detected only on individual instances.

- An intention lock INTSR (INTention Superclass Read) is set for every SC on the *superclass chain* from a target class to its root whenever an IMPR or a QR lock is set on the target class. It indicates that some instance read lock is held on a subclass of the class.

- An intention lock INTSPR (INTention Superclass Partial Read) is set for every SC on the superclass chain from a target class to its root when a TR or PQR lock is set on the target class.

• Instance write

- (for SCA) TW lock (Instance Write on target class) means that some (not all) instances of a target class are *w* locked. An TW lock is set on a *target class* whenever *w* lock is set on its instance.

- (for SCA) IMPW (Implicit Write) lock means that all instances of a target class are *w* locked *implicitly.*

- (for MCA) QW (Query Write on a target class) means that all instances of a target class and its subclasses are write locked.

- (for MCA) PQW (Partial Query Write on a target class) means that some instances of a target class and its subclasses are write locked. As in PQR lock, we set only a PQW lock on a target class and a *w* lock on each individual instance to be accessed.

- INTSW (INTention Superclass Write) lock is set for every SC on the *superclass chain* from the target class to its root whenever an IMPW or QW lock is set on an instance or class.

- An intention lock INTSPW (INTention Superclass Partial Write) is set for every SC on the superclass chain from a target class to its root when a TW or PQW lock is set on the target class

3.1.2.3. Commutativity Relation Table

In Tables 3-4 and 3-5, we provide commutativity relation among the lock modes introduced above.

a) instance

		lock holder	
		r	*w*
lock requester	*r*	Y	N
	w	N	N

Table 3-4. Commutativity relation for locks on an instance

b) Class

Conflicts are checked based on the principle of implicit locking as follows. For conflict checking between MCAs, if either the lock holder or requester requires locks only on a class, conflict relationships are determined directly by read-write commutativity relationship. Otherwise (i.e., both require locks on a class as well as instances), conflicts are determined on individual instances. For conflicts between MCAs and intention locks, conflicts are determined as if an intention lock were a real lock. For example, setting CW and INTSR on the same class will cause conflicts. Also, there is no conflict between SCAs and intention locks.

		lock holder												
		CW	CR	TR	IMPR	INTSR	INTSPR	QR	PQR	TW	IMPW	INTSW	INTSPW	QW	PQW
l	CW	N	N	N	N	N	N	N	N	N	N	N	N	N	N
o	CR	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	TR	N	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y
k	IMPR	N	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N	N
	INTSR	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N
r	INTSPR	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
e	QR	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
q	PQR	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	N	Y
q	TW	N	Y	Y	N	Y	Y	N	Y	Y	N	Y	Y	N	Y
e	IMPW	N	Y	N	N	Y	Y	N	N	N	N	Y	Y	N	N
s	INTSW	N	Y	Y	Y	Y	Y	N	N	Y	Y	Y	Y	N	N
t	INTSPW	N	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	N	Y
e	QW	N	Y	N	N	N	N	N	N	N	N	N	N	N	N
r	PQW	N	Y	Y	N	N	Y	N	Y	Y	N	N	Y	N	Y

Table 3-5. Commutativity table for locks on a class

Once again, the principle is to reduce locking overhead than both implicit locking and explicit locking using special class concept, for any type of access to a class hierarchy.

3.1.2.4. Class hierarchy locking algorithm for single inheritance

In this section, class hierarchy locking scheme is presented. It is to reduce locking overhead than both implicit locking and explicit locking using SC. Especially, based on access frequency information for each class, intention locks are set on only SC only so that locking overhead is reduced comparing implicit locking. Also, unlike explicit locking, locks are not required for the target class and all its subclasses. In the proposed scheme, locks are set only every class from the target class to the first SC through the subclass chain of the target class. This results in less locking overhead than explicit locking.

The proposed locking-based concurrency control scheme is based on two-phase locking which requires each transaction to obtain a read (or write) lock on a data item before it reads (or writes) that data item, and not to obtain any more locks after it has released some lock [Eswa,1976]. For a given lock request on a class, say C, we set locks on C and all classes on the class hierarchy to which the class C belongs as follows.

Step 1) locking on SCs

• For each *SC* (if any) through the superclass chain of C, check conflicts and set an *intention lock* if it commutes. If it does not commute, block the lock requester.

Step 2) Locking on a target class

•If the lock request is an SCA, check conflicts with locks set by other transactions and set one of TR, TW, IMPR, IMPW (depending on the lock request type) or CR (class definition read) on only the target class C if it commutes and set an *r* or *w* lock on the instance to be accessed (which we call *target instance*) if *a* method is invoked on the instance and commute. If it does not commute, block the requester.

•If the lock request is an MCA, then, from class C to the first *SC* (or leaf class if there is no SC) through the subclass chain of C, check conflicts and set either CW, QR, PQR, QW or PQW lock on each class if commute. If the class C is an SC, then set a lock only on C.

Note that a lock is set on the first SC so that other incoming transactions that access a subclass of the first SC can check conflicts since those transactions need to set intention locks on the first SC. Thus, every conflict can be detected. The reason we set a lock on each class (besides the first SC) from the class C to the first SC (not including the SC) is as follows: if a lock is set only on the first SC, then some conflict may not be detected. For example, if a requester accesses a subclass of a lock holder's class which is CW locked, then such a conflict may not be detected.

•If class C has more than one subclass, perform the same step 2) for each subclass of C.

Step 3) Locks are released only if a transaction is committed or aborted.

As an example, consider the following lock requests by two transactions T_1 and T_2 on a class hierarchy in Figure. 3-3.a

a) T_1: class definition write (CW) request on class C6

b) T_2: class definition read on class C5

Let LS_i be a lock set by transaction T_i. Assume that class C1, C4 and C7 are SCs. As seen in Figure 3-3.b, 3-3.c, and 3-3.d, 7, 12 and 11 locks are required for T_1 and T_2 by the proposed scheme, explicit locking, and implicit locking, respectively.

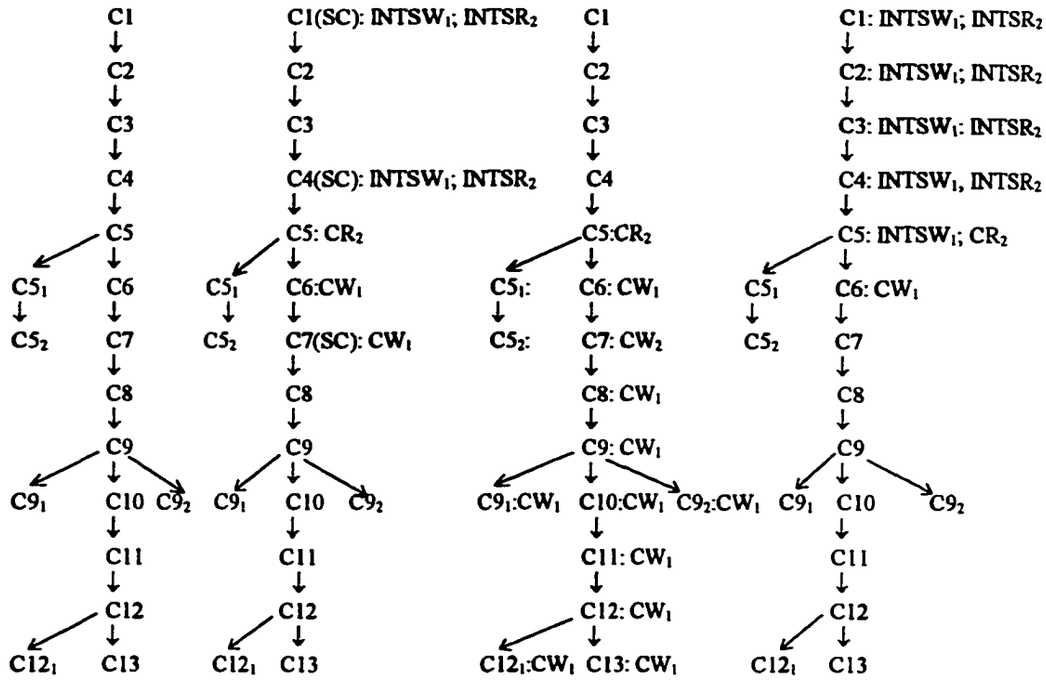


Fig 3-3.a A class hierarchy

3-3.b. Locks with the Proposed scheme

3-3.c. Locks with Explicit locking

3-3.d. Locks with Implicit locking

3.1.2.5. Considering Multiple Inheritance

The above protocol works for a single inheritance. But, it does not work for a multiple inheritance. Consider the following class hierarchy in which a class J has two superclasses, H and I in Figure. 3-4.a. Suppose a transaction T1 sets a QR lock on class F (Figure. 3-4.b). Suppose now another transaction T2 sets a CW lock on class G (Figure. 3-4.c). Even though T1 already sets a conflict lock mode QR on classes J and K implicitly, T2 can get a lock successfully. This is due to that the conflict can not be detected by intention locks on class A which is only common class by both transactions. That is, the above protocol does not work correctly.

In order to make the above protocol work correctly in *multiple inheritance*, a principle is adopted from Orion [Kim,1990]: when setting a QR, PQR, QW, PQW, or CW lock on a class C, also set those locks on subclasses of C which have more than one superclass. By doing this, any conflict on the subclasses of C can be detected. Then only those subclasses need to be examined for conflict checking. Also, intention locks are set on each class through only one superclass chain of the target class C. Although only one chain is selected for intention locks, possible conflicts are detected on either the target class C or subclasses of the target class C. In this example, using the proposed scheme, lock settings for QR lock on class F are changed as in Figure. 3-4.d.

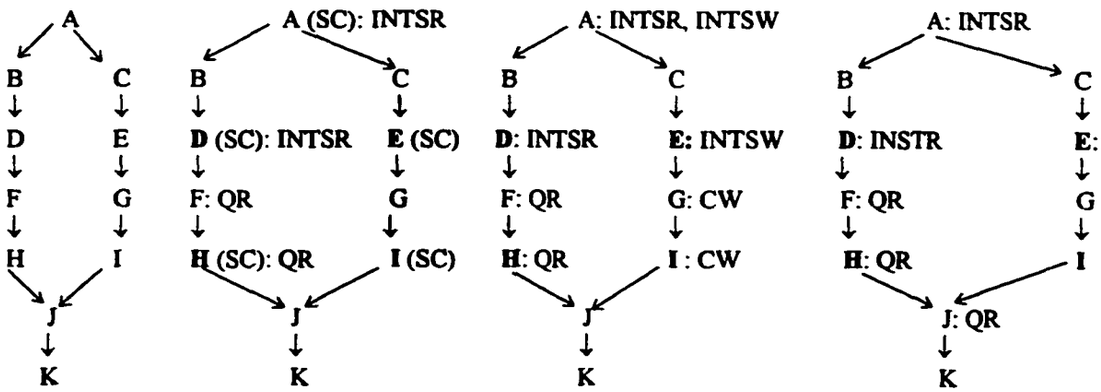


Fig. 3-4.a
class hierarchy

Fig. 3-4.b
QR lock on class F

Fig. 3-4.c
CW lock on class G

Fig. 3-4.d
Locks with multiple inheritance consideration in the proposed scheme

3.1.2.6. Special Class Assignment

Assume that we have information on the number of access to each class (by different transactions) in an OODB. For the proposed scheme, it is necessary to know only two types of access frequency to each class: SCA and MCA. With this access information for each class, whether the class is designated as an SC or not can be determined as

follows. The principle is to designate SC as a class on which class definition writes or IACHs are performed frequently. Note that the following SC assignment scheme is taken place at preprocessing so that any run-time overhead does not occur.

Starting from each leaf class until all classes are checked.

step 1) If a class is a leaf, then do not designate it as an SC.

If a class C has not been considered for SC assignment and all subclasses of C have been already considered for SC assignment, then do the followings:

for class C and all of the subclasses,

calculate the number of locks (N_1) when the class is designated as an SC

calculate the number of locks (N_2) when the class is not designated as an SC

// In calculation, we do not consider any superclass of C yet

step 2) Designate it as an SC only if $N_1 < N_2$. That is, the class can be an SC only if the number of locks can be reduced by doing so.

For example, consider a simple class hierarchy as in Fig 3-5.a and assume that number of access information on the hierarchy are defined as in Fig. 3-5.b. The numbers represent the numbers of access to the class by different transactions. For example, in Fig 3-5.b, 100 MCAs are performed on class C_1 and 300 SCAs on C_1 by different transactions accessing this class hierarchy. Note that, for MCAs, the numbers represent only access initiated at a given class. Thus, we do not count the number of MCA access initiated at its superclasses. In the SC assignment scheme, since C_4 and C_5 are leaf classes, they are not designated as SCs. At the class C_3, if C_3 is designated as an SC, the number of locks

needed for class C_3, C_4 and C_5 are 450, 300 and 400, respectively, resulting 1150 locks for three classes. That is, C_3 needs only 450 locks since any locks are not necessary for its subclasses as in implicit locking. On the other hand, any access to C_4 or C_5 needs intention locks on C_3, resulting 300 and 400 locks for C_4 and C_5, respectively. On the other hand, if C_3 is not designated as an SC, then the number of locks needed for classes C_3, C_4, and C_5 are 1200 locks. In this case, the proposed scheme works as in explicit locking. Thus, class C_3 become an SC. Similarly, two classes C_1 and C_2 become non-SCs. Fig. 3-5.c shows the result of the SC assignment scheme based on access frequency information.

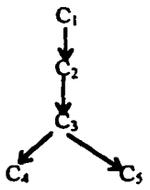


Fig. 3-5.a. Simple class hierarchy

C_1 : MCA:100, SCA: 300

C_2 : MCA: 200, SCA: 200

C_3 : MCA:400, SCA: 50

C_4: MCA: 100, SCA: 50

C_5: MCA:100, SCA: 100

Fig. 3-5.b. Number of Access for each class

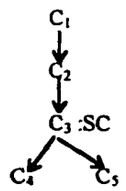


Fig. 3-5.c. Result of SC assignment

For multiple inheritance, the same SC assignment scheme can be applied. For example, consider a simple multiple class hierarchy as in Fig. 3-6.a. and assume that we have frequency information on the hierarchy as in Fig. 3-6.b. Assume that, when C_5 is locked, C_3 is chosen for intention lock setting. The result of the SC assignment scheme is shown in Fig. 3-6.c.

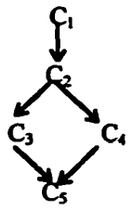


Fig. 3-6.a. Simple class hierarchy

C_1: MCA: 50, SCA: 100

C_2 : MCA: 600, SCA: 200

C_3 : MCA: 100, SCA:150

C_4 : MCA: 300, SCA: 100

C_5: MCA: 100, SCA: 50

Fig 3-6.b. Access numbers for each class

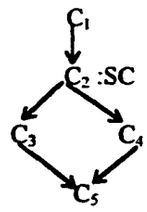


Fig. 3-6.c. Result of SC assignment

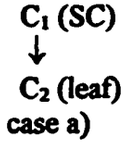
3.1.2.7. Performance evaluation of the proposed scheme

In this subsection, it is shown that the proposed scheme performs better than both explicit locking and implicit locking. That is, assuming that the number of access is stable for each class, it is shown that the proposed scheme incurs either equal or fewer number of locks than both explicit locking and implicit locking. The proof is based on induction.

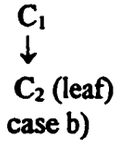
Claim: with a stable number of access for each class, the proposed class hierarchy scheme performs better than both explicit locking and implicit locking.

Proof) Induction is used on the number n in a given class hierarchy. Let n be the number of classes considered so far in the SC assignment scheme. Let N_E, N_I and N_P be the number of locks by explicit locking, implicit locking and proposed locking, respectively, for classes considered so far in SC assignment.

- n =1 : $N_E = N_I = N_P$
- n=2 : In this case, without loss of generality, two classes are formed as follows.



case a)



case b)

If C_1 (superclass) is an SC as in case a), then $N_P \leq N_E$ otherwise C_1 would not be an SC, and $N_P = N_I$. If C_1 is not an SC as in case b), then $N_P \leq N_I$, and $N_P = N_E$.

Assume that the proposed scheme works up to n = K

• n = K+1: without loss of generality, let (K+1)th class be a root of the classes considered for SC assignment. Let X be a root (i.e., K+1th class) and $Y_1,...,Y_n$ be the first SCs through the subclass chain of X as in Figure. 3-7. Also, Let N (X:SC) and N (X:non-SC) be number of locks required when a class X is designated as an SC, and when X is not designated SC, respectively in the proposed scheme (assume that all subclasses of X have been considered in the SC assignment scheme).

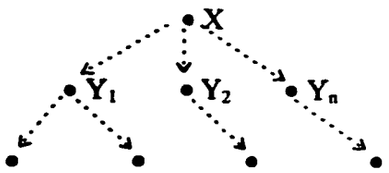


Figure. 3-7. The case where X is not SC

case a) Assume that X is not SC (i.e., N (X:SC) > N (X:non-SC))

At first, prove that $N_P \leq N_E$: for locks required for SCA to class X, both schemes need the same number of locks. For MCA to class X, for the proposed scheme, locks are required for each class from X to $Y_1,...,Y_n$ and subclasses of X which have more than one superclass, if multiple inheritance. On the other hand, locks are required from X to every subclass of X by explicit locking. Also, for locks required for access to classes other than X, $N_P \leq N_E$ by induction assumption and no intention locks on x are necessary by the proposed scheme. Thus, $N_P \leq N_E$ for any access.

Now, prove that $N_P \leq N_I$.

subcase a.1), $N_P \leq N(X:SC)$. Otherwise, x would be an SC by the proposed SC assignment scheme.

subcase a.2) prove that $N(X: SC) \leq N_I$. In implicit locking, every class is an SC since any access to class C needs intention locks on superclasses of C and any MCA access to C need no locks other than a lock on class C. This is corresponding to the proposed scheme where all classes are SCs. Thus, for locks required for access to X, both schemes incur the same number of locks. For locks required for access to classes other than X, intention locks (if necessary) are needed to be set on X by both schemes. But, for locks required for access to classes other than X, $N_P \leq N_I$ by induction assumption. Thus, $N (X:SC) \leq N_I$. This implies that $N_P \leq N_I$.

case b) Assume that x is SC (i.e., $N (X:SC) < N (X:non-SC)$)

$N_P \leq N_I$: same as subcase a.2

Now, we prove that $N_P \leq N_E$.

subcase b.1), $N_P \leq N (X:non-SC)$. Otherwise, X would not be a SC by the proposed SC assignment scheme.

subcase b.2) $N (X:non-SC) \leq N_E$: for locks required for SCA to class X, both schemes incur the same number of locks. For MCA to class X, locks are needed from X to $y_1,...,y_n$ as in Figure 3-6 and subclasses of X which have more than one superclass, if multiple inheritance, in the proposed scheme. But locks are required from x to every subclass of X

in explicit locking. For locks required for access to classes other than X, $N_P \leq N_E$ by induction assumption. Thus, $N_P \leq N_E$.

From case a) and b), with a stable number of access to a class hierarchy, it is concluded that the proposed scheme does not require more locks than explicit locking and implicit locking.

3.1.3. Nested method invocations

In this subsection, a way to handle nested method invocations is presented. It deals with all three aspects discussed earlier: *semantics of methods, nested method invocation, and referentially shared objects.*

In order to increase concurrency among methods, semantic information can be utilized. This semantic information can be extracted at the discretion of application programmers since methods represent behaviors of objects. Thus, although two methods conflicts in terms of read and write commutativity relationships, two methods can run concurrently using semantics. Also, better transaction response time can be achieved by using parent/children parallelism. Also, in order to deal with RSOs, conflicts on RSOs are not defined statically. This results in low concurrency. The conflict among methods are detected on actual method invocation on objects so that further concurrency can be achieved

3.1.3.1. Assumptions

Assume that objects are organized in a hierarchy and referential sharing is allowed. Also, adopt the following transaction model and method model: a transaction consists of a sequence of method invocations to objects ([Cart,1990], [Agra,1992]). A method execution consists of a partial order of method invocations and atomic operations [Hadz,1991]. Also, assume that a method in an object can invoke methods on objects which are lower in the hierarchy [Rese,1994].

Consider the following object hierarchy in Fig. 3-8.a. The database (DB) consists of class *Cars*. Each *car* instance is a tuple object composed of various atomic objects and of component class *Orders*. Each *order* instance is a tuple object composed of atomic objects. In the proposed scheme, referential sharing is allowed. That is, an instance of class *Order* can be shared by two different instances of class *Cars*. In this object hierarchy, assume that a customer can rent only one car at any time. But a customer can request multiple car rental orders so that the order is granted by any available car. Figure 3-8.b shows an example of a car rental order requested for two cars by a customer.

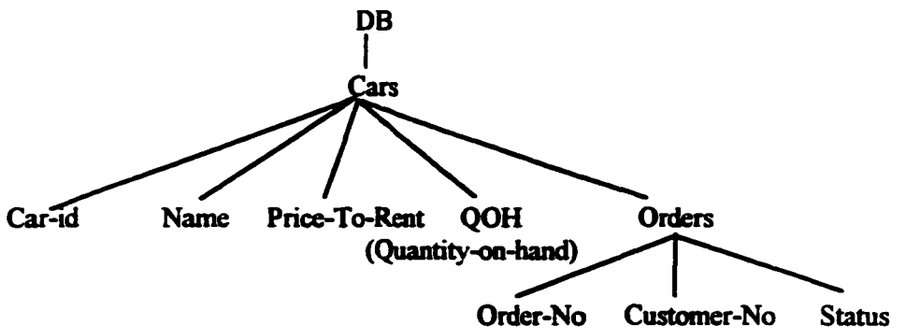


Figure. 3-8.a. An Object Hierarchy

Car objects:

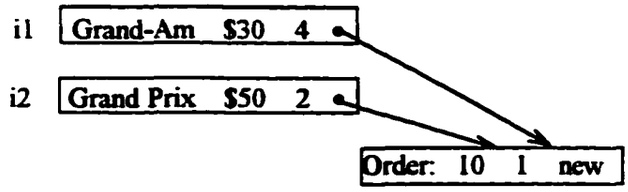


Figure. 3-8.b. An example of the object hierarchy

Assume that there are three methods *Adjust-Price*, *Check-Out-Rent* and *Pay-Rent*, for class *Cars*.

Adjust-Price(i)

// For a car instance i (Car-id), if QOH is greater than 10, price to rent a car is decreased by 10%

If i.QOH > 10 then

    i.Price-To-Rent <= i.Price-To-Rent * 0.9

End if

End

Check-Out-Rent(i,Order-No)

// For a car instance i, a rent-a-car request by Order-No o is granted if that order is not granted

// yet

If Test-status (o) = *new* then

    call Change-Status (o, granted)

    i.QOH <= i.QOH -1

end if

End

Pay-Rent (i,o)

// Pay rental fee for car i by Order-No o

read i.Price-To-Rent

read i.QOH

Change-Status (o, *paid*)

End

For class Orders, assume that two methods *Test-status* and *Change-status* have the following implementation code, respectively. There are three status for each order: *new*, *granted* and *paid*.

Test-status (o)

// test status of an instance o of class Orders

read (o.status)

return status

End

Change-status(o, value)

// change status of an instance o of class Orders to *value*

write (o.status, *value*)

End

3.1.3.2. Automation of commutativity for methods

In order to provide fine concurrency while automating commutativity of methods, the sample principle used in Section. 3.1.1 is adopted. That is, a two-phase pre-analysis is needed. It consists of two steps : 1) construction of DAV for each method and 2) construction of a commutativity table of methods. The construction of DAV for each method is summarized as follows: in each method, a break point is inserted by a

programmer or a compiler when a conditional statement is encountered. Every method has a special break point called *first break point* before the first statement in the method. There are three kinds of DAVs in each method : 1) a final DAV of the first break point, 2) an initial DAV of the first break point, and 3) an initial DAV of every other break point, which contains access modes of all attributes used by statements between this break point and the next break point (or end of the method). The commutativity table of methods is constructed as follows: a lock requester's entries contain names of the final DAVs of the first break points in all methods (represented as N_F where N is the name of the first break point in each method). When a transaction invokes a method on an object, it gets a lock containing the final DAV of the first break point in the method. As the transaction meets a break point during run-time, the break point is recorded. After the method execution, the lock is changed to names of breakpoints encountered during method execution.

For example, consider the object hierarchy in Figure 3-8.a. For convenience, for class *Cars*, let four attributes Car-id, Name, Price-To-Rent and QOH be a_1, a_2, a_3, and a_4, respectively. Similarly, for class *Orders*, let three attributes Order-No, Customer-No, and Status be b_1, b_2, and b_3, respectively. Assume that, for class Cars, A and A_1 are breakpoints of method *Adjust-Price*, B and B_1 are breakpoints of method *Check-Out-Rent* and C is a breakpoint of method *Pay-Rent*. Likewise, assume that, for class *Orders*, let D and E be breakpoints of methods *Test-Status* and *Change-Status*, respectively. Also, for simplicity, we call methods *Adjust-Price*, *Check-Out-Rent* and *Pay-Rent* as M1, M2 and M3, respectively. Similarly, we call methods *Test-status* and *Change-status* as N1 and N2, respectively, for class *Orders*.

Adjust-Price(i) (also called M1)

[A]

If i.QOH > 10 then

[A_1]

    i.Price-To-Rent := i.Price-To-Rent * 0.9

End if

End

Check-Out-Rent(i,Order-No) (also called M2)

[B]

If Test-status (o) = *new* then

[B_1]

    call *Change-Status* (o, *granted*)

    i.QOH := i.QOH - 1

end if

End

Pay-Rent (i,o) (also called M3)

[C]

read i.Price-To-Rent

Change-Status (o, *paid*)

End

Based on the definition of breakpoints and DAVs, for the object hierarchy in Figure 3-8.a, the following breakpoints and DAVs for the methods can be obtained. For convenience, let DAV (x) represent the initial DAV of a breakpoint x in some method M. Also, let DAV (M) represent the final DAV of the first break point in method M.

Note that, in this example, a method *Check-Out-Rent* or *Pay-Rent* includes another nested method invocation (*Test-status* or *Change-status*). But, this nested method invokes another subobject so that its DAV is not included in the DAV of method *Check-Out-Rent* or *Pay-Rent*.

The DAVs constructed for method M1 are:

DAV (M1) = [R,N,W,R]; DAV (A) = [R,N,N,R]; DAV (A1) = [R,N,W,N]

Similarly, the DAVs for M2 and M3 are:

DAV (M2) = [R,N,N,W]; DAV (B) = [R,N,N,N]; DAV (B1) = [R,N,N,W]

DAV (M3) = [R,N,R,R]

Similarly, for class *Orders*, we have DAVs of each break point in the method as follows.

Test-status (o) (also called N1)
[D]
read (o.status)
End

Change-status(o, value) (also called N2)
[E]
write (o.status, value)
End

DAV (N1) = [R,N,R]; DAV (N2) = [R,N,W]

Note that, for class *Orders*, two methods do not have conditional statements so that the DAVs of the methods are the same as the DAVs of the first breakpoints. In this

work, we do not include the DAVs of the first break point for such a case since further concurrency can not be achieved.

After the construction of the breakpoints' DAVs in all methods, a commutativity relation of methods in the form of a table as in Section 3.1.1.1 is created. For convenience, denote D(x) as DAV(x) where x is the name of a method or a break point. Table 3-6 gives the commutativity tables constructed in proposed scheme.

		lock holders						
		D(M1)	D(A)	$D(A_1)$	D(M2)	D(B)	$D(B_1)$	D(M3)
lock requester	D(M1)	N	Y	N	N	Y	N	N
	D(M2)	N	N	Y	N	Y	N	N
	D(M3)	N	Y	N	N	Y	N	Y

Table 3-6.a. A commutativity table for class *Cars*

		lock holders	
		D(N1)	D(N2)
lock requester	D(N1)	Y	N
	D(N2)	N	N

Table. 3-6.b. A commutativity table for class *Orders*

3.1.3.3. Considering semantics, nested method invocation and RSO (Referentially Shared Object)

In this subsection, a way of dealing with three aspects *semantics of methods*, *nested method invocation* and *RSO*) is presented [Jun, 1997-3].

At first, based on the automated commutativity relationships presented in Section 3.1.1, it is possible that application programmers may define commutativity relationships for some methods by making use of semantics of methods as in ([Muth,1993]. [Rese,1994]). Thus, though these two methods do not commute in terms of read and write

access modes, they may commute semantically at the discretion of an application programmer. For example, for class *Cars*, two methods *Check-Out-Rent* and *Pay-Rent* may commute semantically, that is, customers may check out first and then pay the rental fee or vice versa. If two methods, say, M1 (requester) and M2 (holder), commute semantically, then we give *S* commutativity relationship between M1 (and all breakpoints of M1) and M2 (and also all breakpoints of M2) where *S* means semantically commute. Then, a new commutativity table for class *Cars* is constructed as in Table 3-7. In the commutativity table, Y means commute (unconditionally). That is, if two methods (one is a lock requester and the other is a lock holder) have Y relationship, a lock requester can get a lock at any time. If two methods have N relationship, a lock requester can get a lock only if the lock holding transaction is committed or aborted. On the other hand, if two methods have S relationship, a lock requester can get a lock if a holder's method execution is finished. That is, the requester does not have to wait until the lock holding transaction is committed or aborted.

		lock holders						
		D(M1)	D(A)	$D(A_1)$	D(M2)	D(B)	$D(B_1)$	D(M3)
lock requester	D(M1)	N	Y	N	N	Y	N	N
	D(M2)	N	N	Y	N	Y	N	S
	D(M3)	N	Y	N	S	S	S	Y

Table 3-7. A commutativity table for class *Cars*

For nested method invocations, the following principles are applied: each method invocation is associated with a lock. Before any method invocation, a lock is requested and granted. Also, when a method execution is finished, the lock is inherited by its parent.

Then, the lock is said to be *retained* by its parent [Rese,1994]. If a transaction is finished, its locks are discarded. For two methods which commute semantically, they commute only if both execute atomically. That is, for such methods, a requester cannot get a lock until a holder's method execution is finished so that the requester can get a lock only if a holder's lock is inherited by its parent. Thus, unlike N commutativity relationship, a lock request is not delayed until the lock holding transaction commits.

Finally, for RSOs, method invocations on different objects may result in conflicts since those methods may invoke methods on the same subobject. In the proposed scheme, conflicts are determined dynamically for each subobject as in [Rese,1994] since such a conflict may not be detected before actual method invocation.

3.1.3.4. The proposed scheme for nested method invocations

The proposed scheme for nested method invocations is based on two-phase locking [Eswa,1976]. Based on the discussion in Section 3.1.3.3, the following scheme is constructed .

1. Lock is required only for method execution and is granted before method execution. After method execution, lock is changed (i.e., it reflects the breakpoints executed)
2. A method execution cannot terminate until all of its children are terminated. When a method execution ***m*** terminates:
   a. there exists a parent of ***m*** and ***m*** commits: locks held by ***m*** are inherited by its parent (i.e., locks are *retained* by its parent)

b. either there exists a parent of *m* and *m* aborts or there is no parent of m: locks held by *m* are discarded.

3. A lock can be granted if either of the following conditions is satisfied.

a. no other method holds or retains a conflicting lock

b. if conflicting locks are held, such locks are retained by ancestors of the requesting method

c. (for semantic commutativity) if conflicting locks are retained by non-ancestors, then one of the ancestors of the retainer (not including the retainer itself) and an ancestor of the requester commute.

In rule 3.b, when ancestor/descendent parallelism is allowed, a parent is not supposed to see uncommitted results of the child method. Otherwise, the parent may be aborted due to reading uncommitted value. For example, assume that a parent T initiates a method M, which accesses some data item X, and continues to do its own work. When T needs to access data item X so that it requires a conflicting lock on X, T can get a lock only if the lock held by M is retained by T.

In rule 3.c, we implement semantically commutativity relationships. As we discussed for the two methods which commute semantically, two methods commute only if both execute in an atomic way. Thus, we let a lock requester get a lock only if a holder's method execution is finished (i.e., its lock is inherited by an ancestor). In additions, for two methods commuting semantically, a requester's descendent can also get a lock if a holder's method execution is finished.

Figure 3-9 shows that two transactions T1 and T2 invoke the same method M1 on instance car1 of class *Car* and M2 (by T1) and M3 (by T2) on car2 (and on order2 of class Orders), and the same method M2 on car3 (and on order3 of class Orders). Assume that, only the first breakpoint [A] has been executed in two method invocations of M1 by T1 and T2 on an instance car1. Also, assume that breakpoints [B] and [B_1] have been executed on an instance car3 in method invocations of M2 by T1 and T2. Note that a line indicates a nested method invocation.

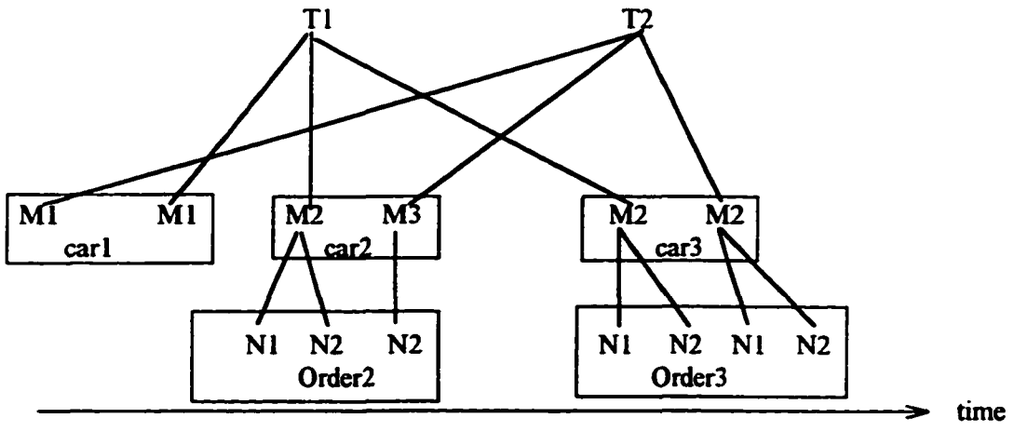


Figure 3-9. A possible execution of transactions in the proposed scheme

In the above example, two method invocations of M1 on car1 commute by adopting dynamic information. This commutativity would not be possible if we used static commutativity relationships for methods as in [Malt,1993]. Also, two methods M2 and M3 on instance car2 commute semantically so that the method invocation on M3 by T2 can be executed only after M2 invoked by T1 is finished, that is, after the lock held by M2 is inherited by T1. This guarantees atomic execution of method invocation M2 by T1. Without the semantics of methods, the method execution M3 by T2 is blocked until the

entire transaction T1 is committed. In the proposed scheme, the method execution is delayed only until method M2 invoked by T1 is committed. Thus, we can increase concurrency by adopting semantic information. Also, a lock request by method invocation M2 of T2 on car3 is not granted since a conflicting lock is held by T1. Thus, the method invocation M2 of T2 can be executed after T1 is committed. For method invocations on instance car3, consider the following execution which requires locks by atomic operations as in [Rese,1994]. This results in a deadlock situation as in Figure 3-10. In Figure 3-10, two transactions T1 and T2 invoke a method M2. In turn, M2 invokes two atomic operations Test-status and Change-status on instance Order3. When the two transactions invoke the method M2 such an order as in Figure 3-10, a deadlock can occur if the scheme in [Rese,1994] is adopted. In the proposed scheme, such a deadlock situation can be avoided by adopting locks for the execution of the methods.

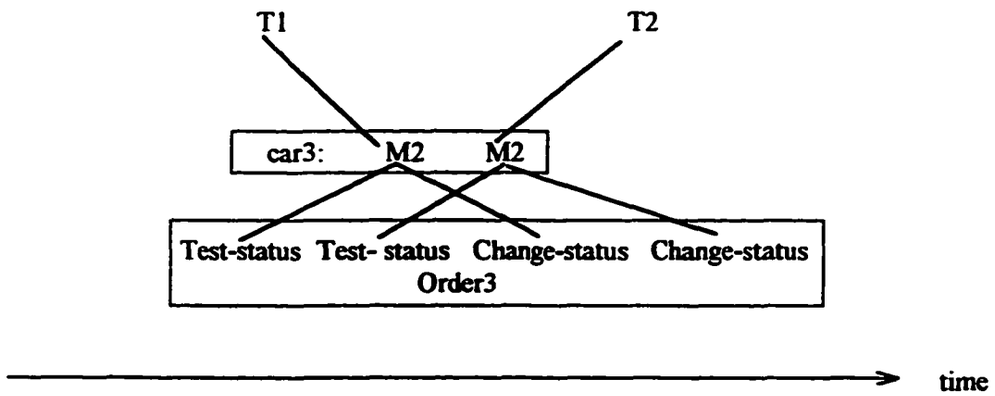


Figure 3-10. A possible execution by a scheme requiring locks for atomic operations

3.2. Integrated concurrency control scheme

3.2.1. Transaction and method model

Several transaction models have been proposed for OODBs depending on applications and their needs ([Ozsu,1994], [Bili,1992]). In this work, the transaction and method model are adopted from ([Cart,1990], [Agra,1992]). This model is simple and does not require any compensating work when a part of a transaction is aborted.

A transaction has the following format: *<trans-id, O>* where

trans-id : a unique transaction identifier

O : a set of operations representing the implementation of the transaction. These operations may include a sequence of method invocations ($oid_i.m_k;....oid_j.m_l$) where oid and m are an object name and a method name, respectively, a *begin_transaction* statement which indicates a new transaction is starting for bookkeeping purpose, a *commit* or an *abort* statement.

A method consists of *<Nm, Arg, OP >* where

Nm : name of the method

Arg : arguments of the method

.OP :a set of operations representing the implementation of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value. Also, a method can call another method defined on the same object or different objects during its execution.

3.2.2. Complete concurrency control algorithm

3.2.2.1. Lock format

A lock of an instance access on a class has the following format:

a) for a target class:

[*trans-name, method-name (break-points),* F_1, F_2] where *trans-name, method-name (break-points)* are transaction holding a lock and method invoked (including breakpoints encountered). F_1 and F_2 are Boolean fields. F_1 indicates whether an instance access method applies to some instance (represented as F) or all instances of the class (represented as T). F_2 indicates whether an instance method is an MCA (represented as T) or not (represented as F).

b) For an intention lock

[*trans-name,I, class-name*] where *trans-name* is the same as in a), I means an intention lock for instance access, and *class-name* is the class on which actual lock is held.

c) For a target instance

[*trans-name, method-name* (*break-points*)]

Also, a lock table for a class definition access has the following format:

a) For a target class

[*trans-name, method-name*] where *trans-name* is the same as before and method-*name* is one of the followings: {MCR, MM(method-names), MA(attribute-names), RCR, RM(method-names), RA(attribute-names)}.

b) For an intention lock

[*trans-name, method-name, I*] where *trans-name* is the same as before, *method-name* is the same as in a), and *I* means an intention lock for a class definition access

Finally, a lock of a nested method invocation has the following format:

a) For a class

[*trans-name, ancestors-ids, owner-id, RET, method (break-points), F_1, F_2*] where *trans-name* is the same as before, *ancestors-ids* are ancestors of current lock requester, *owner-id* is lock requester (method or transaction), *RET* is a Boolean field indicating the lock is retained or not, remaining fields are the same as above.

b) For an instance

[*trans-name, ancestor-ids, owner-id, RET, method (break-points)*] where each field is the same as above.

3.2.2.2. The Integrated Concurrency Control Algorithm

Based on the principles for conflict among methods, class hierarchy locking and nested method invocations discussed so far, an integrated concurrency control algorithms is constructed as follows. For each method invocation, the procedure **Main** is invoked first.

// Depending on method types, there are two branches for each method invocations

Main
If a method is a regular (non-nested) method invocation, then go to **Conflict-among-methods**
else go to **Nested-method-invocations**
end if
end Main

Conflict-among-methods
- go to **Decide-conflict-types** and return conflict-type

//There are four cases depending on the lock requester and the lock holder as follows
- If conflict-type is **type-a**, then go to **Check-if-instance-method**
go to **Class-hierarchy-locking**
else if conflict-type is **type-b**, then go to **type-b-conflict-check** and check conflict
If there is conflict, block the request
else go to **Check-if-instance-method**:

go to **Class-hierarchy-locking**;
end if
else if conflict-type is **type-c**, then go to **type-c-conflict-check** and check conflict
If there is conflict, block the request
else go to **Check-if-instance-method**;
go to **Class-hierarchy-locking**;
end if
else if conflict-type is **type-d**, then go to type-d-conflict-check and check conflict
If there is conflict, block the request
else go to **Check-if-instance-method**;
go to **Class-hierarchy-locking**;
end if
end if

Check-if-instance-method
// If the requesting method is an instance access method, then do extra work as follows.
// If the requesting method is a class definition access method, just grant lock.

If the method is an instance access method, perform the followings.

// before instance access method execution
set a lock of final breakpoint of a method;
// during instance access method execution
record break point encountered during the method execution
// after instance access method execution
change lock from an initial lock to lock of break points encountered during run-time.

else
grant the lock request
end if;
end Check-if-instance-method
end Conflict-among-methods

Class-hierarchy-locking

// For a given lock request on class, say Y, we set locks on class hierarchy as follows.
case i)
If the access is an MCA, then do the followings
// Set intention locks at every SC through superclass chain (if the class hierarchy
// forms multiple inheritance, use only one superclass chain
For each SC though superclass chain do
go to **Decide-conflict-types** and return conflict-type;
//There are four cases depending on the lock requester and the lock holder as follows
- If conflict-type is **type-a**, then go to **type-a-conflict-check** and check conflict
If there is conflict, block the request
else set an intention lock
end if

else if conflict-type is **type-b**, then go to **type-b-conflict-check** and check conflict
If there is conflict, block the request
else set an intention lock
end if
else if conflict-type is **type-c**, then go to **type-c-conflict-check** and check conflict
If there is conflict, block the request
else set an intention lock
end if
else if conflict-type is **type-d**, then go to type-d-conflict-check and check conflict
If there is conflict, block the request
else set an intention lock
end if
end if
end For
// Now, set locks for subclasses through subclass chain
For each class from the target class Y to up to first S **do**
// If there is no such SC, then up to leaf class through the subclass chain. If Y has more than one
// subclass, then, for each subclass of Y, do the following steps
go to **Decide-conflict-types** and return conflict-type;
//There are four cases depending on the lock requester and the lock holder as follows
- If conflict-type is **type-a**, then go to **type-a-conflict-check** and check conflict
If there is conflict, block the request
else set the lock
end if
else if conflict-type is **type-b**, then go to **type-b-conflict-check** and check conflict
If there is conflict, block the request
else set an the lock
end if
else if conflict-type is **type-c**, then go to **type-c-conflict-check** and check conflict
If there is conflict, block the request
else set the lock
end if
else if conflict-type is **type-d**, then go to type-d-conflict-check and check conflict
If there is conflict, block the request
else the lock
end if
end if
end For
end if
Case ii)
If the access is an SCA, then do the followings
// Set intention locks at every SC through superclass chain (if the class hierarchy
// forms multiple inheritance, use only one superclass chain
For each SC though superclass chain do
go to **Decide-conflict-types** and return conflict-type;
//There are four cases depending on the lock requester and the lock holder as follows
- If conflict-type is **type-a**, then go to **type-a-conflict-check** and check conflict
If there is conflict, block the request

```
                                    else set an intention lock
                                end if
        else if conflict-type is type-b, then go to type-b-conflict-check and check conflict
                                    If there is conflict, block the request
                                      else set an intention lock
                                    end if
            else if conflict-type is type-c, then go to type-c-conflict-check and check conflict
                                      If there is conflict, block the request
                                        else set an intention lock
                                      end if
                else if conflict-type is type-d, then go to type-d-conflict-check and check conflict
                                        If there is conflict, block the request
                                          else set an intention lock
                                        end if
    end if
   end For
// Now, set a lock for only on target class
go to Decide-conflict-types and return conflict-type;
   //There are four cases depending on the lock requester and the lock holder as follows
   - If conflict-type is type-a, then go to type-a-conflict-check and check conflict
                                If there is conflict, block the request
                                  else set a lock on the target class
                                end if
        else if conflict-type is type-b, then go to type-b-conflict-check and check conflict
                                    If there is conflict, block the request
                                      else set a lock on the target class
                                    end if
            else if conflict-type is type-c, then go to type-c-conflict-check and check conflict
                                      If there is conflict, block the request
                                        else set a lock on the target class
                                      end if
                else if conflict-type is type-d, then go to type-d-conflict-check and check conflict
                                        If there is conflict, block the request
                                          else set a lock on the target class
                                        end if
    end if
   end for
end if
end Class-hierarchy-locking
```

Decide-conflict-types

```
// Depending on lock types of lock requesters and holders, there are four types of conflicts as
// follows..

case a) Lock requester: intention lock; Lock holder: intention lock
return type-a;
case b) Lock requester: intention lock; Lock holder: regular lock
```

return **type-b**;
case c) Lock requester: regular lock; Lock holder: regular lock
return **type-c**;
case d) Lock requester: regular lock; Lock holder: intention lock
return **type-d**;
end Decide-conflict-types

Type-a-conflict-check
// There is no conflicts between intention locks
return **no-conflict**
end Type-a-conflict-check

Type-b-conflict-check
// Commutativity can be checked using the following table 3-8. Let [I,X] be intention lock for
// lock type X.

	SCA	MCA	RCR	RM	RA	MCR	MM	MA
[I,SCA]	Y	C_1	Y	Y	Y	N	C_2	C_3
[I,MCA]	Y	C_4	Y	Y	Y	N	C_2	C_5
[I,RCR]	Y	Y	Y	Y	Y	N	Y	Y
[I,RM]	Y	Y	Y	Y	Y	N	C_2	C_3
[I,RA]	Y	Y	Y	Y	Y	N	Y	C_6
[I,MCR]	N	N	N	N	N	N	N	N
[I,MM]	Y	C_2	Y	Y	Y	N	C_2	C_5
[I,MA]	Y	C_9	Y	Y	Y	N	C_7	C_6

Table 3-8. Commutativity table between intention lock and regular lock

// For symbols other than Y or N, conflict are decided as follows.
C_1: //check commutativity using instance access commutativity table
call **Conflict-resolution** (C_1)
C_2: // check commutativity using vector MAV
call **Conflict-resolution** (C_2)
C_3: // check using vector AAV and DAV of the requester
call **Conflict-resolution** (C_3)
C_4: // check redefinition using instance access commutativity table
call **Conflict-resolution** (C_4)
C_5: // check commutativity between DAV of lock requester's method (and each DAV of lock
// requester's method redefined through subclass chain of the requester's class) and AAV
call **Conflict-resolution** (C_5)
C_6: // check using vector AAV
call **Conflict-resolution** (C_6)
C_7: // check commutativity using DAVs of methods in MAV and vector AAV. If methods in
// MAV is redefined through subclass chain of MM's class, check commutativity for each
// redefined method using DAV of such method and AAV.
call **Conflict-resolution** (C_7)
C_8: // check commutativity between DAV of holder's method (instance access method or RM)
// and vector AAV

call **Conflict-resolution** (C_8)
C_9: // check commutativity between DAV of lock holder's method (and DAVs of lock holder's
// method redefined through subclass chain of the holder's class) and AAV
call **Conflict-resolution** (C_9)
end Type-b-conflict-check

Type-c-conflict-check
// Commutativity can be checked using the following table 3-9.

	SCA	MCA	RCR	RM	RA	MCR	MM	MA
SCA	C_1	C_1	Y	Y	Y	N	C_2	C_3
MCA	C_1	C_4	Y	Y	Y	N	C_2	C_5
RCR	Y	Y	Y	Y	Y	N	Y	Y
RM	Y	Y	Y	Y	Y	N	C_2	C_3
RA	Y	Y	Y	Y	Y	N	Y	C_6
MCR	N	N	N	N	N	N	N	N
MM	C_2	C_2	N	C_2	O	N	C_2	C_5
MA	C_8	C_9	N	C_8	C_6	N	C_7	C_6

Table 3-9. Commutativity table between (regular) locks

// For symbols other than Y or N, conflict are decided as follows.
C_1: //check commutativity using instance access commutativity table
call **Conflict-resolution** (C_1)
C_2: // check commutativity using vector MAV
call **Conflict-resolution** (C_2)
C_3: // check using vector AAV and DAV of the requester
call **Conflict-resolution** (C_3)
C_4: // check redefinition using instance access commutativity table
call **Conflict-resolution** (C_4)
C_5: // check commutativity between DAV of lock requester's method (and each DAV of lock
// requester's method redefined through subclass chain of the requester's class) and AAV
call **Conflict-resolution** (C_5)
C_6: // check using vector AAV
call **Conflict-resolution** (C_6)
C_7: // check commutativity using DAVs of methods in MAV and vector AAV. If methods in
// MAV is redefined through subclass chain of MM's class, check commutativity for each
// redefined method using DAV of such method and AAV.
call **Conflict-resolution** (C_7)
C_8: // check commutativity between DAV of holder's method (instance access method or RM)
// and vector AAV
call **Conflict-resolution** (C_8)
C_9: // check commutativity between DAV of lock holder's method (and DAVs of lock holder's
// method redefined through subclass chain of the holder's class) and AAV
call **Conflict-resolution** (C_9)
end Type-c-conflict-check

Type-d-conflict-check

// Commutativity can be checked using the following table 3-10.

	[I,SCA]	[I,MCA]	[I,RCR]	[I,RM]	[I,RA]	[I,MCR]	[I,MM]	[I,MA]
SCA	Y	Y	Y	Y	Y	Y	Y	Y
MCA	C_1	C_4	Y	Y	Y	N	C_2	C_5
RCR	Y	Y	Y	Y	Y	Y	Y	Y
RM	Y	Y	Y	Y	Y	Y	Y	Y
RA	Y	Y	Y	Y	Y	Y	Y	Y
MCR	N	N	N	N	N	N	N	N
MM	C_2	C_2	N	C_2	Y	N	C_2	C_5
MA	C_8	C_9	N	C_8	C_6	N	C_7	C_6

Table 3-10. Commutativity table between regular locks and intention locks

// For symbols other than Y or N, conflict are decided as follows.
C_1: //check commutativity using instance access commutativity table
call **Conflict-resolution** (C_1)
C_2: // check commutativity using vector MAV
call **Conflict-resolution** (C_2)
C_4: // check redefinition using instance access commutativity table
call **Conflict-resolution** (C_4)
C_5: // check commutativity between DAV of lock requester's method (and each DAV of lock
// requester's method redefined through subclass chain of the requester's class) and AAV
call **Conflict-resolution** (C_5)
C_6: // check using vector AAV
call **Conflict-resolution** (C_6)
C_7: // check commutativity using DAVs of methods in MAV and vector AAV. If methods in
// MAV is redefined through subclass chain of MM's class, check commutativity for each
// redefined method using DAV of such method and AAV.
call **Conflict-resolution** (C_7)
C_8: // check commutativity between DAV of holder's method (instance access method or RM)
// and vector AAV
call **Conflict-resolution** (C_8)
C_9: // check commutativity between DAV of lock holder's method (and DAVs of lock holder's
// method redefined through subclass chain of the holder's class) and AAV
call **Conflict-resolution** (C_9)
end Type-d-conflict-check

Conflict-resolution (p)

case p = C_1
If intention lock is held by holder, go to the class indicated by intention lock and use instance access commutativity table. Otherwise use instance access commutativity table on the target class.
case i) lock requester accesses only some instance of a class,
If lock holder has lock on a set of instance, then check commutativity with lock *on the class* using the instance access commutativity table;
return commute or no-commute

otherwise check commutativity with locks *on target instances* using instance access commutativity table;
return **commute** or **no-commute**;

case ii) lock requester accesses a set of instance of a class
check commutativity with lock (set by the lock holder) *on the class* using the instance access commutativity table;
return **commute** or **no-commute**;

case p = C_2
If the lock holder is MM (or intentioned MM) locked, then
  // Assume that an instance method M_1 is invoked by lock requester
   check if method field of M_1 is W locked in MAV;
   return **commute** or **no-commute**;
else
// Assume that an instance method M_1 is invoked or the definition of M_1 is being read by holder
  Check if the definition of M_1 is to be modified;
   return **commute** or **no-commute**;
end if

case p = C_3
// Assume that either an instance method M_1 is invoked by lock requester or the definition of M_1 is
// to be read by requester
use DAV of M_1 defined on target class;
For each attribute accessed in DAV of M_1 do
   check if there is at least one attribute with W locked in vector AAV;
   return **commute** or **no-commute**;
end do

case p = C_4
// Assume that an instance method M_r is invoked by the requester and another instance method M_h
// is being executed by the lock holder
Check commutativity using instance access commutativity table at target class.
If not commute, return **no-commute**.
Otherwise, do the following steps.
      For each subclass S of the lock requester's class on which either M_r or M_h is redefined do
        check commutativity with lock on the class using instance access commutativity table on S;
      end do
      If there is at least one class on which the commutativity is not satisfied, return no-commute;
      Otherwise, return commute;

case p = C_5
// Assume that an instance method M_1 is invoked by the lock requester
use DAV of M_1 on the target class;
For each attribute accessed in DAV of M_1 do
 check if there is at least on attribute with W locked in AAV;
 If so, return **no-commute**;
 Otherwise, do the following steps.
    For each subclass S of the target class on which M_1 is redefined **do**

```
      For each attribute accessed in DAV of M_l do
        check if there is at least one attribute with W locked in vector AAV;
        If so, return no-commute;
        Otherwise, continue;
      end do
    end do
end do
return commute;

case p = C_6
If CA lock is held, then
   for each attribute to be accessed by lock requester do
       check if there is at least one attribute with W locked in vector CA set by lock holder;
       If so, return no-commute;
       Otherwise, return commute;
   end do
else if a_i is being read by holder, then return no-commute // Assume a_i is to be modified//
else return commute
end if

case p = C_7
For each method M_i with R or W lock in MAV do
  use DAV of M_i on the target class;
  For each attribute accessed in DAV of M_i do
    check if there is at least one attribute with W locked in vector AAV or a_i is to be modified.
    If so, return no-commute;
    Otherwise, perform the following steps;
    For each subclass of the target class on which M_i is redefined do
      For each attribute accessed in DAV of M_i;
        check if there is at least one attribute with W locked in vector AAV or a_i is to be modified;
        If so, return no-commute;
      end do
    end do
  end do
end do
return commute;

case p = C_8
// Assume that an instance method M_l is being  invoked or a definition of M_l is being read by
// holder
// assume that attribute a_i is to be modified
use DAV of M_l on the class indicated by intention lock (if an intention lock is held);
Check if a_i is accessed in DAV of M_l;
If so, return no-commute;
Otherwise, return commute;

case p = C_9
// Assume that an instance method M_l is being invoked by the lock holder and attribute a_i is to be /
modified
```

```
use DAV of M_I on the class indicated by intention lock (if an intention lock is held);
For attribute a_i do
  check if a_i is accessed in DAV of M_I;
  If so, return no-commute;
  Otherwise, perform the following steps;
  For each subclass S of the lock holder's class on which M_I is redefined do
    For each attribute a_i do
      check if a_i is accessed in DAV of M_I;
      If so, return no-commute;
    end do
  end do
end do
return commute;

end Conflict-resolution ( )

Nested-method-invocations
// A lock is required for method execution and is granted before method execution.
// A method execution cannot terminate until all of its children are terminated. When a method
// execution m terminates as follows:
    // a. there exists parent of m and m commits : locks held by m are inherited by its parent (i.e.,
    //    locks are retained by its parent)
    // b. there exists parent of m and m aborts: locks held by m are discarded.
    // c. there is no parent of m: locks are discarded
    If a nested method satisfies one of three following condition, grant the lock request
        a. no other method or transaction holds or retain a conflicting lock
        b. if conflicting locks are held, such locks are retained by ancestors of the requesting
           method
        c. (for semantic commutativity) if conflicting locks are retained by non-ancestors, then one
           of the ancestors of the retainer (not including retainer itself) and an ancestor of the
           requester commute
    else
        block the request
    end if
end Nested-method-invocations
```

For an illustration of the concurrency control scheme, consider a class hierarchy and a composite object hierarchy in Figure 3-11. Assume that each method has the following break points.

m_1: A, A1
m_2: no break point
m_3: C, C1
m_5: E, E1
m_7: no break point
m_{10}: K, K1

m_{11}: L, L1
m_{12}: N, N1
m_{13}: O, O1

Let class Domestic_auto, Company and Employee have the following commutativity tables.

	$D(m_1)$	D(A)	D(A1)	$D(m_2)$	$D(m_3)$	D(C)	D(C1)	$D(m_5)$	D(E)	D(E1)	$D(m_7)$
$D(m_1)$	Y	Y	Y	N	N	Y	N	N	N	N	Y
$D(m_2)$	N	N	N	N	N	Y	N	N	N	N	N
$D(m_3)$	N	N	N	N	Y	Y	Y	N	Y	N	N
$D(m_5)$	N	N	N	N	N	N	N	Y	Y	Y	N
$D(m_7)$	Y	Y	Y	N	N	Y	N	N	N	N	Y

Table 3-11. Commutativity table for class *Domestic_auto*

	$D(M_{10})$	D(K)	D(K1)	$D(m_{11})$	D(L)	D(L1)
$D(m_{10})$	N	Y	N	S	S	S
$D(m_{11})$	S	S	S	N	N	N

Table 3-12. Commutativity table for class *Company*

	$D(m_{12})$	D(N)	D(N1)	$D(m_{13})$	D(O)	D(O1)
$D(m_{12})$	N	Y	N	N	Y	N
$D(m_{13})$	N	N	N	N	Y	N

Table 3-13. Commutativity for class Employee

Consider the following transactions on the class hierarchy in Figure 3-11.

T1: change implementation code of method m_5 on class Passenger

T2: Domestic_auto.I1.m_3

T3: Domestic_auto.I1.m_1, Company.I1.m_{10} (nested method invocation)

T4: Company.I1.m_{11}

Assume that the break point C is encountered by T2. Figure 3-12 shows the lock table after four transactions are finished but not committed. Assume that classes Vehicle_on_land and Domestic_auto are only SCs.

3.3. The correctness of the proposed concurrency control scheme

The proposed scheme has three features of access: conflicts among methods, class hierarchy locking and nested method invocations. For the first two types of access, that is, conflicts among access and class hierarchy locking, the standard two phase locking is adopted [Eswa,1976]. For the standard two phase locking, the serializability is guaranteed. That is, as long as a concurrency control produces a schedule based on two-phase locking, the schedule satisfies the serializability.

But, for nested method invocations, semantic commutativity is adopted. For notion of serializability where all conflicts are preserved, it is called *conflict serializability* [Rese,1994]. But, for the serializability where some conflicts can be ignored based on the semantics of operations at higher level, it is called *semantic serializability*. For semantic serializability of semantic concurrency control, its proof is given in [Rese,1994]. For their concurrency control, lock is required for each atomic operation while locks are required for method invocation for the proposed semantic concurrency control scheme. In this subsection, the proof of the correctness is shown for class hierarchy locking and nested method invocations, respectively.

3.3.1. The correctness of class hierarchy locking

In this subsection, it is proven that the proposed algorithm is correct, that is, it satisfies serializability [Eswa,1976]. The proof is based on that, for any lock requester, its conflict with a lock holder (if any) is always detected. With this proof, since the class hierarchy locking scheme is based on two-phase locking, it is guaranteed that the proposed scheme satisfies serializability [Eswa,1976]. This type of proof technique is adopted in earlier work in [Liou,1991]. Also, for simplicity, we prove only for single inheritance. For multiple inheritance, the correctness can be proved similarly. If a lock requester is an SCA, then its lock holders (whose lock modes need to checked for conflict with lock requester) consist of transactions holding locks on the target class and all special classes in the superclass chain of the target class. If a lock requester is an MCA, then its lock holders include those defined above plus transactions holding locks on each class from the target class to the first special class in the subclass chain of the target class.

There are four cases depending on the types of lock requesters and holders.

case 1) *the lock holder is an SCA*
the lock requester is an SCA

If a lock holder (L_H) and a lock requester (L_R) access different classes, there is no conflict. If a lock holder and a lock requester access the same class, there is no conflict on all SCs through the superclass chain of the target class because intention locks on SCs are compatible with L_R. Thus conflicts can be detected on the target class.

a = attribute
i = instance
m = method

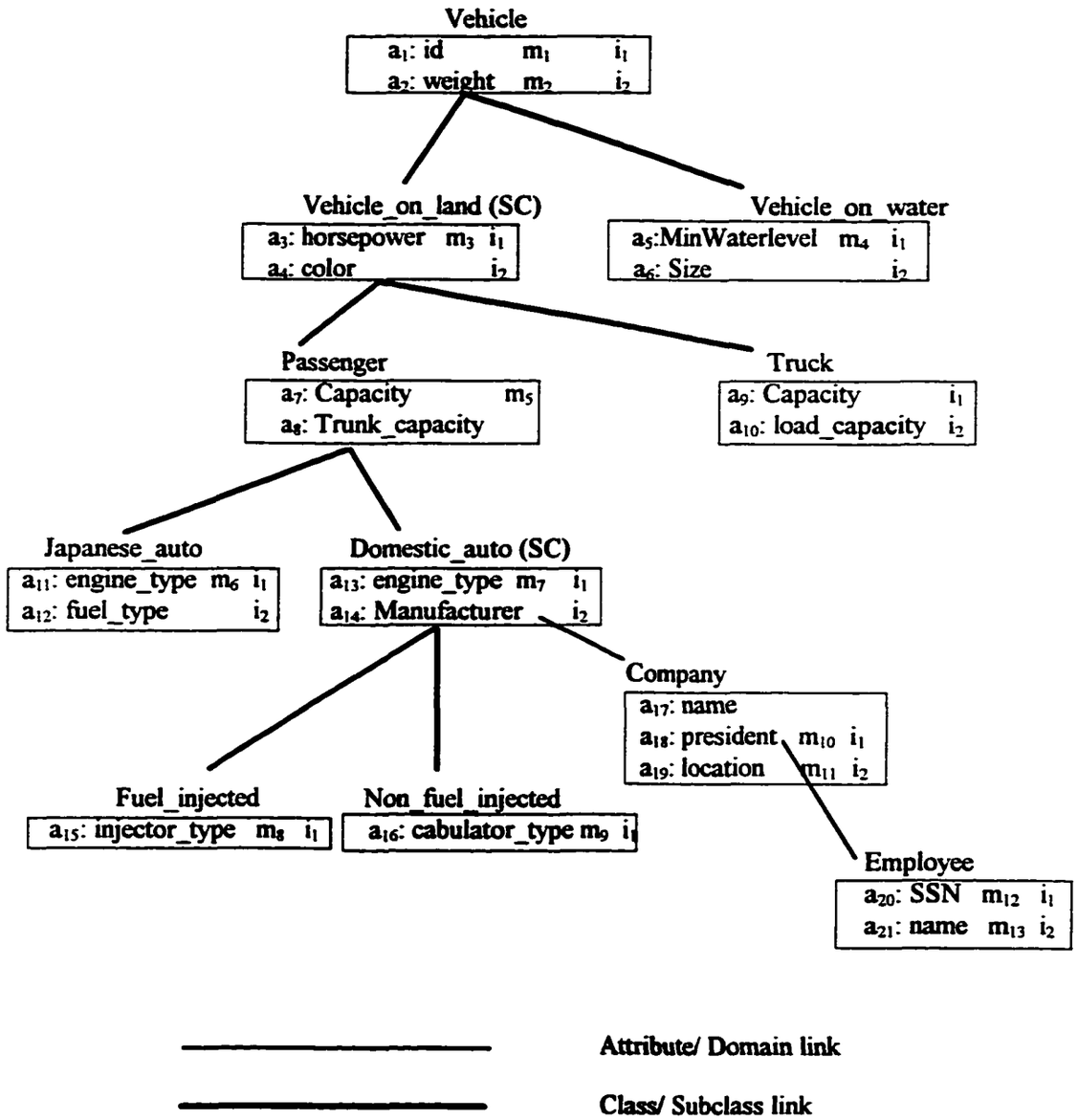


Figure 3-11. An illustrative class hierarchy and composite object hierarchy example

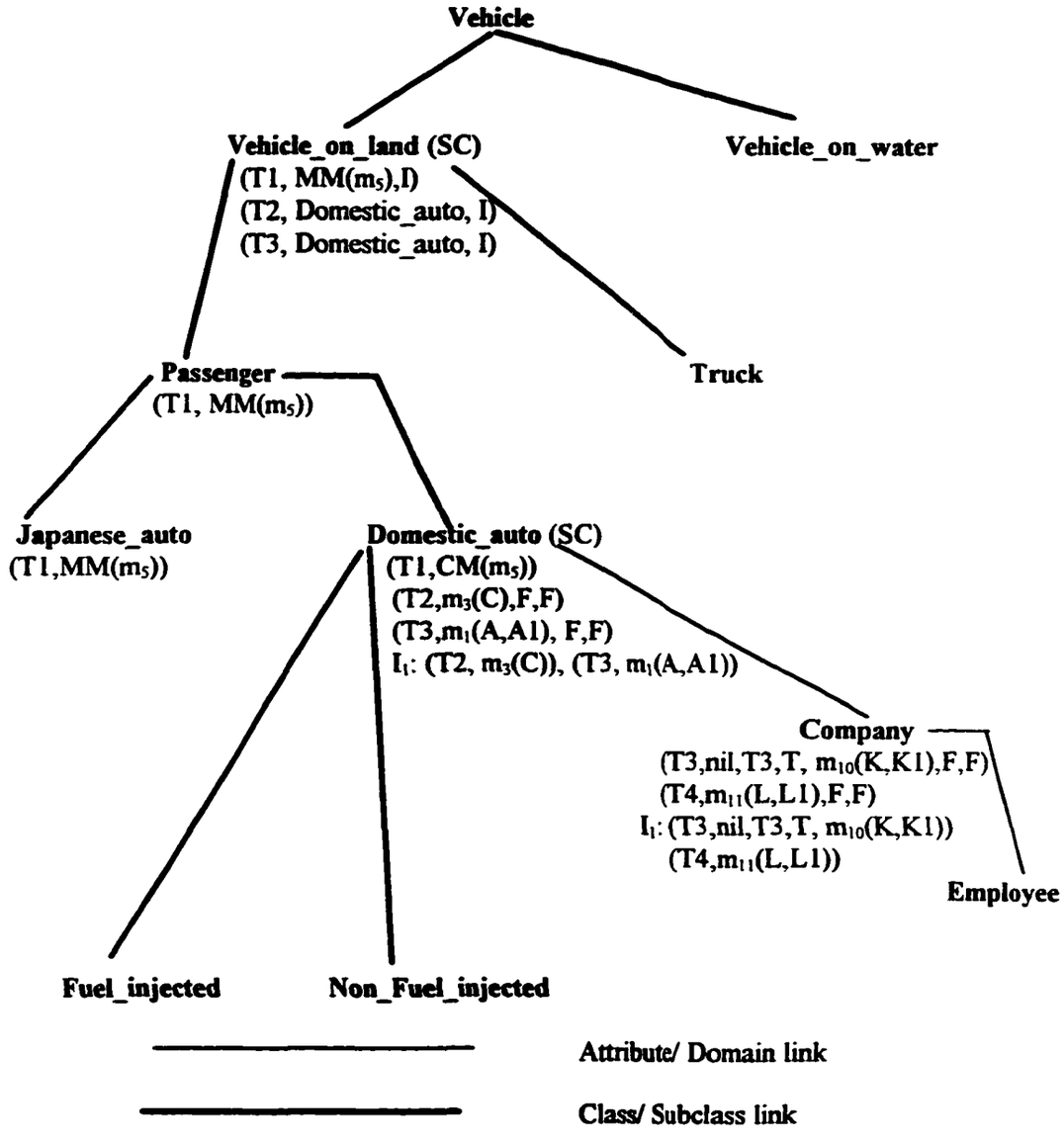


Figure. 3-12. Transaction executions on class hierarchy

case 2) the lock holder is an SCA
the lock requester is an MCA

If the L_H is holding a lock on a superclass of the L_R's class, there is no conflict since the L_R does not access the L_H's class. If the L_H is holding a lock on the L_R's class or subclass, then there are two subcases. If there exists an SC between L_H and L_R, then conflict is detected on the nearest SC through the subclass chain of the L_R's class (case 2.1). Otherwise, the conflict is detected on the class of L_H (case 2.2). Let R and H be two classes on which the L_R requests a lock and the L_H holds a lock, respectively. In case 2.1, as shown in Figure 3-13.a, a conflict (if any) is checked on SC1, which is the nearest *special class* of the L_R's class through its subclass chain, since the L_H has an intention lock on SC1 and the requester requests CW, QR, PQR, QW or PQW on SC1. On the other hand, in case 2.2, for subcase a, a conflict (if any) is checked on H as in Figure. 3-13.b since L_H does not have any intention locks through the superclass chain of R and L_R needs to set an MCA lock on H. For subcase b, a conflict (if any) is checked on H as in Figure 3-13.c since intention locks on all special classes through the superclass chain of H are compatible and the requester needs to set an MCA lock on H.

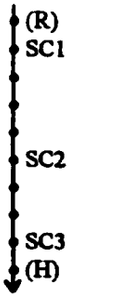


Fig 3-13. a. Case 2.1

Fig. 3-13.b. Subcase a of case 2.2

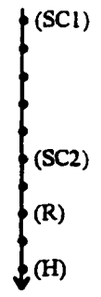


Fig. 3-13.c. Subcase b of case 2.2

case 3) the lock holder is an MCA
the lock requester is an SCA

If L_H is holding a lock on a subclass of L_R, there is no conflict. If L_H is holding a lock on the class of L_R or on a superclass of L_R, then there are two cases in which conflicts will be detected. If there exists some SCs between L_R and L_H, the conflict is detected on the first SC to L_H through the subclass chain of L_H such as SC2 in Figure. 3-14.a (case 3.1). Otherwise, the conflict is detected on the class of L_R as in Figure. 3-14.b (case 3.2).

Figure. 3-14. a. Case 3.1

Figure. 3-14.b. Case 3.2

case 4) the lock holder is an MCA
the lock requester is an MCA

If L_H accesses the same class or superclass of L_R's class, the conflict is detected as in either case 3.1 or case 3.2. On the other hand, if L_H accesses a subclass of the L_R' class, the conflict is detected as in either case 2.1 or case 2.2.

From cases 1), 2), 3) and 4), for any lock requester, it is guaranteed that its conflict with a lock holder (if any) is always detected. Also, since the proposed scheme is based on two-phase locking, serializability is guaranteed [Eswa,1976].

3.3.2. The correctness of nested method invocations

The proof for nested method invocations is similar to that in [Rese,1994], which is based on two techniques: ***substitution*** and ***commutativity-based reversals***. The

substitution can be further divided into two techniques: reduction and expansion. Reduction is to transform a separated transaction t whose children are leaves to another transaction t' which is the same as t except that t is substituted by the method corresponding to transaction t. Note that a transaction is said to be separated if no operations of other transactions are interleaved with its leaves. Expansion is the inverse of the reduction. The commutativity-based reversal is also to transform a transaction t which has two commuting consecutive leaves to another transaction t' which is the same as t except that these consecutive leaves are reversed.

The proof is based on the following principle [Rese,1994]: create a sequence of equivalent intermediate executions starting from an original execution of the proposed scheme and finally create a serial execution consisting only top-level transactions. From original execution, a node is selected and separated by necessary commutativity-based reversals, and finally reduced. These separations and reductions are repeated until only top-level transactions are created. Thus, it is enough to show that, for any execution produced by the proposed semantic locking, equivalent serial execution can be obtained by iterations of separations and reductions from the original execution.

In each intermediate execution step, for separation and reduction, any node t is selected. The node should satisfy the following property: its right most child is the leftmost among nodes having only leaves as children. The proof is based on that it is always possible to separate t. It is said that an operation o interleaves within operation t if there are some children of t on the leftside as well as on the rightside of the operation o [Rese,1994]. For an initial execution, say E_O, it is not possible that a method m can be

interleaved within t where m conflicts with a child of t. This is due to the proposed scheme that requires m to wait until t is terminated. Thus, the concern is focused on any intermediate execution E_i.

For contradiction, assume that, in an intermediate execution E_i, t cannot be separated where t's rightmost child is the leftmost among nodes that have only leaves as children. This is because a child of t, say t_L, conflicts with and precedes v and in turn v conflicts with and precedes, another child of t, say t_R. Since t_L conflicts with v, there exists descendants t_L' and v' of t_L and v, respectively in original execution E_O so that t_L' conflicts with and precedes v'.

At first, it is concluded that t_L' and v' conflict with each other on the same object since conflicts on different objects can not be defined due to the proposed scheme. Assume that there is no semantic commutativity between t_L' and v'. Since there is a conflict between t_L' and v' by assumption, the execution of v' can be resumed only after the lock set by t_L' is released. That is, the execution of v' can be started only after all descendants of t are finished. Thus, y can not be interleaved within the children of x. Assume that there is semantic commutativity between method t_L' and v'. In this case, there exist some ancestor (t_L') and ancestor of (v) and they commute with each other where ancestor (t_L') and ancestor (v) are ancestor of t_L' and the proper ancestor of v, respectively. Then, v' can get a lock as long as the execution of ancestor (t_L') is finished. Since the method commutativity is defined on each object, the ancestor (t_L') and ancestor (v) are defined on the same object. But, since ancestor of (t_L') and ancestor (v) are the ancestor of t_L' and the proper ancestor of v, respectively, the method t_L is defined on

higher object than object on which v is defined. Similarly, it is proven that the method v is defined on higher object than object on which t is defined. This violates the assumption that a nested method is called from a higher object to a lower object. Thus, it is concluded that, for any execution produced by the proposed semantic locking scheme, an equivalent serial execution can be obtained by iterations of separations and reductions from the original execution.

Chapter 4

Performance Analysis by Analytical Models

In this chapter, an analytical model is constructed to measure the performance of the proposed concurrency control techniques. The rationales for adopting an analytical model for performance evaluation are as follows: an analytical model is an abstraction of a system that avoids unnecessary details [Lazo,1984]. Thus, an analytical modeling is to extract and test essential parts to the system behavior from mass of details that is the system itself, with less time and cost. On the other hand, simulation gives accurate result, in a more extensive and real environment. But, it may take tremendous time to complete. Also, modeling gives guidelines for simulation as to which parameters are necessary, which system components are important, which performance metrics are needed, and which testing cases are should be prepared.

In order to analyze the performance of the proposed technique, two existing representative technique are selected for comparison: Orion [Kim,1990] and Malta's ([Malt,1991], [Malt,1993]). The reasons to choose these two existing schemes are as follows. These schemes include all of three access types in OODBs, that is, conflict among methods, class hierarchy locking and nested method invocations. Also, those schemes have different characteristics for each access type. First, consider conflict among methods. Orion and Malta adopt an entire class object as locking granularity for class definition access. But, for instance access, Malta adopts attributes as locking granularity while Orion adopts an entire instance object.

Also, Orion does not provide any concurrency between an instance access and a class definition access while Malta does. Second, for class hierarchy locking, Orion adopts implicit locking while explicit locking is adopted in Malta. For nested method invocations, both Orion and Malta do not consider parent/child parallelism. But, locks are required for every atomic operations in Orion while locks are required for each method invocation in Malta.

This Chapter is organized as follows. In Section 4.1, an analytical model for concurrency control for OODB is introduced. In Section 4.2, data structures to implement three schemes are introduced. Also, an analytical parameters are identified in Section 4.2. In Section 4.3, for each access type in OODB (that is, conflict among access, class hierarchy locking and nested method invocation), necessary analytical parameter values are obtained in order to measure the performance of the concurrency control techniques. Finally, an OODB benchmark, called 007 benchmark ([Care,1993], [Care,1994]), and analysis results based on this benchmark are presented in Section 4 4.

4.1. Analytical Model

4.1.1. A Basic Model

The analytical model is based on [Yu,1993]. In that work, an analytical model is introduced for two-phase locking concurrency control with the following simple assumptions: each transaction can have only exclusive (write) access mode to data items. Also, every transaction has the same number of granule access. For access patterns, they assume uniform access over a set of granules. That is, each granule has equal probability

to be accessed. Finally, they assume the same execution time for each granule in the database .

The transaction model in [Yu,1993] is as follows: each transaction consists of N_L + 2 states where N_L is the random number of granules accessed by the transaction (granule is the unit of data to which concurrency control is applied) and L is the number of granules in the database. The state 0, called *initial setup phase* is to generate transaction-id, granules and lock types for each transaction. Also, states 1 to N_L-1 are called *execution phase*. In these states, before access, conflicts are checked. If there is a conflict, the lock request is denied. Otherwise, a lock is set and the data item is accessed. In state N_L, called *commit phase*, each transaction releases its locks and is committed. Each state i, $1 \leq i \leq N_L-1$, is divided into two substates i_1 and i_2. In substate i_1, the transaction holds i-1 locks and is waiting for its *i*th lock request to be satisfied. Let b denote the mean time in substate i_1 and where $b = P_W * R_W$ where P_W and R_W are lock contention probability and lock waiting time, respectively . In substate i_2, the transaction holds i locks and is executing. Let *a* denote the mean time in substate i_2.

Based on the transaction model, the transaction response time is calculated as follows.

$R = R_{INPL} + R_E + N_L * P_W * R_W + T_{commit}$ where R is mean response time, R_{INPL} is the execution time in state 0, R_E is the sum of execution times in states 1,..., N_L, P_W is lock contention probability, R_W is lock waiting time for a transaction, and T_{commit} is commit time for a transaction. Note that R_{INPL}, R_E, N_L and T_{commit} are constants. Thus, once the values of P_W and R_W are found, the mean response time R can be obtained.

At first consider R_W. In order to get R_W, let G be the sum of lock holding times for each granule over N_L granules by a transaction. Then

$$G = \left\{ \sum_{i=1}^{N_L} (ia + (i-1)b) \right\} + N_L c \qquad (4.1)$$

where c is commit time (= T_{commit})

Then, G/N_L becomes mean lock holding time overhead over N_L granules. Also, R_W can be defined as follows.

$$R_w = \sum_{i=1}^{N_L} \left(\frac{(i-1)b}{G} \left\{ \frac{R_w}{f_1} + a + s_i \right\} + \frac{ia}{G} \left\{ \frac{a}{f_2} + s_i \right\} \right) + \frac{N_L c}{G} \left\{ \frac{c}{f_3} \right\} \qquad (4.2)$$

where (i-1)b/G and ia/G are the conditional probability that a lock request contends with a transaction in substate i_1 and i_2, respectively, given that lock contention occurs, $1 \le i \le N_L$, and $N_L c/G$ is the similar expression for state N_L+1.

The quantity s_i is the mean time from leaving state i until the end of commit and given by

$s_i = (N_L - i)(a + b) + c$

Also, R_W/f_1 and a/f_2 are the mean remaining time in substate i_1 and i_2, respectively, given that the transaction blocking the lock request was in that state, $1 \le i \le N_L$. c/f_3 is the mean remaining time in the commit phase given that the transaction blocking the lock request was in that state. Based on these expressions, the following expressions can be induced.

$\left\{ \frac{R_w}{f_1} + a + s_i \right\}$ = lock waiting time (in substate i_1) + execution time (in substate i_2) + mean

time from leaving state i until the end of commit

$\left\{\frac{a}{f_2}+s_i\right\}$ = execution time (in substate i_2) + mean time from leaving state i until the end of commit

$\left\{\frac{c}{f_3}\right\}$ = commit time (i.e., = T_{commit})

R_W can be simplified as follows.

$$R_w = \frac{(a+b)^2(N_L+1)(N_L-1)}{6} \Big/ \left\{a\left(\frac{N_L+1}{2}\right)+c+b\left(\frac{N_L-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \frac{(N_L+1)((a^2/f_2)+ab+ac+bc)}{2} \Big/ \left\{a\left(\frac{N_L-1}{2}\right)+c+b\left(\frac{N_L-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \left\{\frac{c^2}{f_3}-ab-bc\right\} \Big/ \left\{a\left(\frac{N_L+1}{2}\right)+c+b\left(\frac{N_L-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\} \qquad (4.3)$$

Now, consider P_W. P_W can be expressed as follows.

P_W = (arrival rate of lock request for a granule) * (mean lock holding time)

$= \{\lambda * (N_L/L)\} * \{G/N_L\}$ (4.4)

where λ is transaction arrival rate. Note that *utilization factor(or traffic intensity)* = *arrival rate * mean service time* [Lazo, 1984]. That is, the probability of conflict is the same as the utilization factor of the data item.

Based on expressions (4.3) and (4.4), values of R_W and P_W can be obtained repeatedly as follows: knowing that $b = R_W * P_W$, start with $b = 0$ to get values of $R_W = f(b) = f(0) = x(1)$ and of $P_W = f(b) = f(0) = x(2)$ where x(i), $i \geq 1$, are temporary variables.

Then, b = x(1)*x(2). Once again, get the value of R_W = f(b) = x(3) and of P_W = f(b) = x(4). These iterations are continued until approximation of b is reached. For example, If b =x(2n-1)*x(2n) ≅ x(2n+1)*x(2n+2) , then choose values of R_W and P_W at stage n+1.

4.1.2. An Extended Model

For performance comparisons of the three techniques, the following less restrictive assumptions are necessary: there are only two types of transactions : IA (instance access) and CDA (class definition access). Also each transaction can have variable execution time for each granule. In addition, multiple access modes as well as exclusive access mode are allowed for locks. For the above assumptions, the basic model in Section 4.1.1 can be extended as follows [Yu, 1993].

Since there are two types of transactions, R_W and P_W can be divided further as follows. Let P_{IA} be the probability that a transaction is an IA transaction. Likewise, let P_{CDA} be the probability that a transaction is a CDA transaction where $P_{IA} + P_{CDA} = 1$. Also, let P_{LI} and P_{LC} be the probabilities of lock contention with IA transactions and CDA transactions, respectively, assuming that a lock requester is an IA transaction. Likewise, let $P_{C,I}$ and $P_{C,C}$ be the probabilities of lock contention with IA transactions and CDA transactions, respectively, assuming that a lock requester is a CDA transaction. Also, let R_I and R_C be the mean waiting times given that there was a contention with a IA transaction and a CDA transaction, respectively.

Assuming that P_{WI} represents the overall contention prob. of an IA transaction and R_{WI} represents mean waiting time of an IA transaction for all types of conflict given that there was a contention. Then, P_{WI} and R_{WI} can be expressed as follows.

$$P_{WI} = P_{L,I} + P_{L,C} \qquad (4.5)$$

$$R_{WI} = (P_{L,I}*R_I + P_{L,C}*R_C) / P_{WI} \qquad (4.6)$$

Let $b_I = P_{L,I}*R_I + P_{L,C}*R_C$ and

$$G_I = \left\{\sum_{i=1}^{N_L}(ia_I + (i-1)b_I)\right\} + N_L*c_I$$

where a_I is the execution time for each granule for an IA transaction and c_I is commit time for an IA transaction.

Let $s_{i,IA}$ be the mean time from leaving state i_2 until the end of commit for an IA transaction. Then,

$$s_{i,IA} = (N_L - i)*(a_I + b_I) + c_I \qquad (4.7)$$

and

$$R_I = \sum_{i=1}^{N_L}\left(\frac{(i-1)b_I}{G_I}\left\{\frac{R_m}{f_1} + a_I + s_{i,u}\right\} + \frac{ia_I}{G_I}\left\{\frac{a_I}{f_2} + s_{i,u}\right\}\right) + \frac{N_L c_I}{G_I}\left\{\frac{c_I}{f_3}\right\} \qquad (4.8)$$

R_I can be simplified as follows:

$$R_I = \frac{(a_I+b_I)^2(N_L+1)(N_L-1)}{6} \Bigg/ \left\{a_I\left(\frac{N_L+1}{2}\right) + b_I\left(\frac{N_L-1}{2}\right) + c_I\right\}$$

$$+ \frac{(N_L+1)((a_I^2/f_2) + b_I R_m + a b_I + a_I c_I + b_I c_I)}{2} \Bigg/ \left\{a_I\left(\frac{N_L+1}{2}\right) + b_I\left(\frac{N_L-1}{2}\right) + c_I\right\}$$

$$+ \left\{\frac{c_I^2}{f_3} - \frac{b_I R_m}{f_1} - a b_I - b_I c_I\right\} \Bigg/ \left\{a_I\left(\frac{N_L+1}{2}\right) + b_I\left(\frac{N_L-1}{2}\right) + c_I\right\} \qquad (4.9)$$

Let P_{WC} be the overall contention prob. of a CDA transaction and R_{WC} be mean waiting time of a CDA transaction for all types of conflict given that there was a contention. P_{WC} and R_{WC} are expressed as follows.

$$P_{WC} = P_{C,I} + P_{C,C} \qquad (4.10)$$

$$R_{WC} = (P_{C,I}*R_I + P_{C,C}*R_C) / P_{WC} \qquad (4.11)$$

Let $b_C = P_{C,I}*R_I + P_{C,C}*R_C$ and

$$G_C = \left\{\sum_{i=1}^{N_L}(ia_C + (i-1)b_C)\right\} + N_L*c_C$$

where a_C is execution time for each granule for a CDA transaction and c_C is commit time for a CDA transaction.

Let $s_{i,CDA}$ be the mean time from leaving state i_2 until the end of commit for a CDA transaction. Then,

$$s_{i,CDA} = (N_L - i)*(a_C + b_C) + c_C \qquad (4.12)$$

and

$$R_C = \sum_{i=1}^{N_L}\left(\frac{(i-1)b_C}{G_C}\left\{\frac{R_{WC}}{f_1} + a_C + s_{i,CDA}\right\} + \frac{ia_C}{G_C}\left\{\frac{a_C}{f_2} + s_{i,CDA}\right\}\right) + \frac{N_L c_C}{G_C}\left\{\frac{c_C}{f_3}\right\} \qquad (4.13)$$

R_C can be simplified as follows:

$$R_C = \frac{(a_C+b_C)^2(N_L+1)(N_L-1)}{6} \Big/ \left\{a_C\left(\frac{N_L+1}{2}\right) + b_C\left(\frac{N_L-1}{2}\right) + c_C\right\}$$

$$+ \frac{(N_L+1)((a_C{}^2/f_2) + b_C R_{WC} + a_C b_C + a_C b_C + b_C c_C)}{2} \Big/ \left\{a_C\left(\frac{N_L-1}{2}\right) + b_C\left(\frac{N_L-1}{2}\right) + c_C\right\}$$

$$+\left\{\frac{c_c^2}{f_3}-\frac{b_c R_{mc}}{f_1}-a_c b_c-b_c C_c\right\}\Bigg/\left\{a_c\left(\frac{N_L+1}{2}\right)+b_c\left(\frac{N_I-1}{2}\right)+c_c\right\} \qquad (4.14)$$

The values of b_I and b_C can be obtained repeatedly as follows:

step 1) Start with $b_I = b_C = 0$

$R_I = f(b_I, b_C) = f(0,0)$
$R_C = f(b_I, b_C) = f(0,0)$
$P_{L,I} = f(b_I) = f(0)$
$P_{C,I} = f(b_I) = f(0)$
$P_{L,C} = f(b_C) = f(0)$
$P_{C,C} = f(b_C) = f(0)$

Then, get the new values of $b_I = P_{L,I} * R_I + P_{L,C} * R_C$ and $b_C = P_{C,I} * R_I + P_{C,C} * R_C$

step 2) With the new values of b_I and b_C obtained in step (1), compute:

$R_I = f(b_I, b_C)$
$R_C = f(b_I, b_C)$
$P_{L,I} = f(b_I)$
$P_{C,I} = f(b_I)$
$P_{L,C} = f(b_C)$
$P_{C,C} = f(b_C)$

Get the new values of $b_I = P_{L,I} * R_I + P_{L,C} * R_C$ and $b_C = P_{C,I} * R_I + P_{C,C} * R_C$

step n) With the new values of b_I and b_C obtained in step (n-1), compute:

$R_I = f(b_I, b_C)$
$R_C = f(b_I, b_C)$
$P_{L,I} = f(b_I)$
$P_{C,I} = f(b_I)$
$P_{L,C} = f(b_C)$
$P_{C,C} = f(b_C)$

Get the new values of $b_I = P_{L,I} * R_I + P_{L,C} * R_C$ and $b_C = P_{C,I} * R_I + P_{C,C} * R_C$

If $b_I \cong b_I$ (of step n-1) and $b_C \cong b_C$ (of step n-1), then choose the values of b_I and b_C of step n.

Let RES_I and RES_C be the response times of IA transaction and CDA transaction, respectively. Then,

$$RES_I = t_{int} + N_L * a_I + N_L * (P_{L,I} * R_I + P_{L,C} * R_C) + c_I \qquad (4.15)$$

$$RES_C = t_{int} + N_L * a_C + N_L * (P_{C,I} * R_I + P_{C,C} * R_C) + c_C \qquad (4.16)$$

Then, the overall response time RES is expressed as follows

$$RES = \frac{P_{IA} * RES_I + P_{CDA} * RES_C}{P_{IA} + P_{CDA}} \qquad (4.17)$$

4.2. Analytical parameters

In this section, in order to get necessary analytical parameter values such as mean lock waiting time and mean conflict probability, data structures used for implementation are presented. Also, a complete analytical parameter table is constructed. Finally, analytical parameter values are obtained.

4.2.1. Lock tables

For an implementation of the lock table, the following data structure is assumed (in Figure 4-1): each class has B buckets for maintaining lock tables for instances where the parameter B is chosen by the application programmer. The reason for adopting buckets is to reduce search time for a particular instance. Assume that, for each class and each instance, two pointers X and Y are used and each pointer takes one MM (main memory) word to point transactions holding locks where X and Y are pointers for the first lock holding transaction and for the last lock holding transaction, respectively. Also, assume

that, for each bucket, two pointers X' and Y' are used to point the first instance accessed and the last instance accessed, respectively.

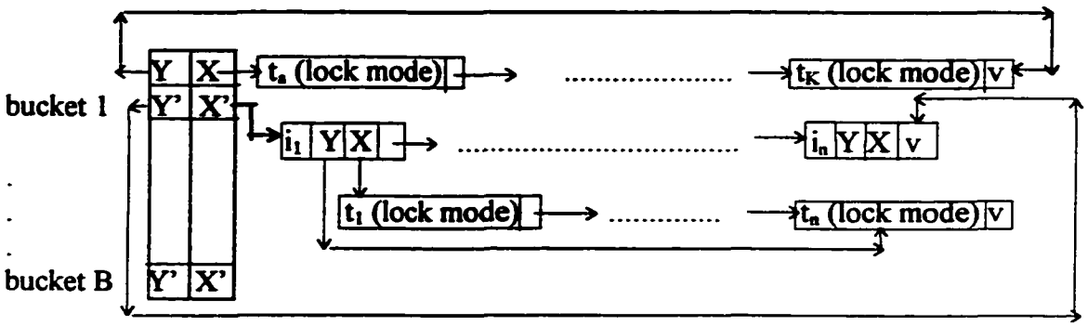


Figure 4-1. illustrative lock table structure for three technique implementations

In Figure 4-1, for each class and each instance, two pointers are used for the lock holding transactions so that searching for any particular transaction can be done either forward traversal or backward traversal. By doing this, the search time can be reduced significantly than sequential search. Likewise, for each bucket, two pointers are used for the same purpose. A lock format of each lock holding transaction consists of a transaction-id, lock mode and pointer to the next lock holding transaction. Note that, in Figure 4-1, t_i and v represent a transaction-id and nil pointer, respectively.

For the commutativity table, Orion and the proposed scheme use the same ones introduced in Chapter 3. But, for Malta's technique, the following commutativity tables are used.

-instance (Assume that there are Nm methods in each class)

	M_1	M_2	.....	N_{Nm}
M_1	Y	N		Y
M_2	N	Y	.....	N
.				
.				
M_{Nm}	Y	N	.....	Y

Table 4-1. Commutativity table for instance access in Malta's work

Also, for the commutativity table for both instance access and class definition access is as follows (Table 4-2). Let I denote an instance access method. Let RD and MD denote *read class definition* and *modify class definition*, respectively. Δ means that commutativity depends on instance method commutativity.

	I	RD	MD
I	Δ	Y	N
RD	Y	Y	N
MD	N	N	N

Table 4-2. Commutativity table for instance access and class definition access in Malta's work

The lock table is same as the Orion, but each lock mode is replaced either by a method name (if the lock requester is an instance method) or by a lock mode ∈ {RD, MD} (if the lock requester is a class definition access method).

4.2.2. Analytic Parameters

Table 4-3 shows the complete analytical parameter table for the analytical model. In order to compare three techniques using real values of analytical parameters, a representative benchmark, called 007 ([Care, 1993], [Care,1994]).

Parameters	Description	Default value
AT (A)	Number of atomic operations in a method	4

b	mean lock waiting time	calculated
B	Number of buckets	5
BASIC_OP	Time to perform one basic operation	0.000007 ms [Huan, 1995]
D	class hierarchy depth	3
L	Number of instances in a database	42100
L_C	levels in composite object hierarchy	3
MM_ ACCESS	One main memory word access time	0.0922 ms [Huan, 1995]
M_P	Multiprogramming number	$\lambda*(t_{int}+N_C*x)$
N_A	Number of attributes in a class	5
N_C	Number of classes in a class	10
N_G	Number of granule accesses per transaction	3.95
N_I	Number of instances in a class	4210
N_{COM}	Number of objects accessed by a nested method	4
NUM_INST	Number of instances in a bucket	$M_P*N_G/(2*B*N_C)$
NUM_ $TRANS_{CLASS}$	Average number of transactions holding locks on a class	M_P*N_G/N_C
NUM_ $TRANS_{INST}$	Average number of transactions holding locks on an instance	$M_P*N_G/(N_C*N_I)$
Num_br	number of breakpoints in an instance method	1
P_{IA}	prob. of instance access transaction	0.9
P_{CDA}	prob. of class definition access transaction	0.1
P_{IR}	prob. of instance read	0.72
P_{IW}	prob. of instance write	0.18
P_{CDR}	prob. of class definition read	0.05
P_{CDW}	prob. of class definition write	0.05
P_{CDR_RC}	prob. of class definition read for class relationship	$P_{CDR}*0.25$
$P_{CDR\text{-}RM}$	prob. of class definition read for method	$P_{CDR}*0.5$
P_{CDR_RA}	prob. of class definition read for attribute	$P_{CDR}*0.25$
$P_{CDW\text{-}WC}$	prob. of class definition write for class relationship	$P_{CDW}*0.25$
P_{CDW_WM}	prob. of class definition write for method	$P_{CDW}*0.5$
P_{CDW_WA}	prob. of class definition write for attribute	$P_{CDW}*0.25$
λ	transaction arrival rate	500 (200-700)
x	execution time for each granule	2 ms
S	number of attributes accessed in an instance method	4
t_{lock}	time to get a lock	calculated
t_{commit}	time to commit a transaction	calculated
$t_{breakpoint}$	time to record breakpoints in a method	calculated
t_{int}	time to initialize a transaction	0.0072ms

x	execution time for each granule	2 ms

Table 4-3. Analytical parameter table

The parameters t_{lock}, t_{commit}, and $t_{breakpoint}$ are calculated based on each algorithm shown in Section 4.2.1 and the lock table and analytical parameters shown in this section. Detail steps to calculate each parameter are found in appendix.

t_{lock} (Orion)

$$=[12+ 10*NUM\_TRANS_{CLASS} + P_{IA}*[7+2*NUM\_INST + 7*NUM\_TRANS_{INST}]]*MM\_ACCESS$$
$$+[6+17*NUM\_TRANS_{CLASS} + P_{IA}*[9+2*NUM\_INST + 11*NUM\_TRANS_{INST}]]*BASIC\_OP$$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$ where NUM_INST is number of buckets, NUM_TRANS_{CLASS} is the average number of transactions holding locks on class, NUM_TRANS_{INST} is the average number of transactions holding locks on instance, and M_P is the average number of transactions in the system. Let $t_{lock, IA}$ (Orion) and $t_{lock, CDA}$ (Orion) be times to get a lock by instance access transaction and class definition access transaction, respectively. Then,

$t_{lock, IA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 1$

$t_{lock, CDA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 0$

t_{commit} (Orion)=

$$N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2] +$$
$$[AT-1]*P_{IA}*[NUM\_TRANS_{INST}*2+3]]*MM\_ACCESS$$
$$+N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2* NUM\_TRANS_{INST}+ P_{IA}*[9+NUM\_INST*2]+$$
$$[AT-1]*P_{IA}*(NUM\_TRANS_{INST}*2]]*BASIC\_OP$$

where NUM_INST $\cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

Let c_I (Orion) and c_C (Orion) be commit times taken by an instance access transaction and a class definition access transaction, respectively, in the Orion technique.

c_I(Orion) = t_{commit} (Orion) where $P_{IA} = 1$
c_C(Orion) = t_{commit} (Orion) where $P_{IA} = 0$

Time to get a lock in the Malta's technique:

t_{lock} (Malta) =

$=[12+8*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+NUM\_TRANS_{INST}*[10+N_M/26]]*MM\_ACCESS+[6+13*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+NUM\_TRANS_{INST}*[17+N_M/26*2]]*BASIC\_OP$

where NUM_INST $\cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$

$t_{lock, IA}$ (Malta) = t_{lock} (Malta) where $P_{IA} = 1$

$t_{lock, CDA}$ (Malta) = t_{lock} (Malta) where $P_{IA} = 0$

t_{commit} (Malta)=
$=N_G*[6+2*NUM\_TRANS_{CLASS}+2*P_{IA}*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*MM\_ACCESS + N_G*[6+2*NUM\_TRANS_{CLASS}+2*P_{IA}*NUM\_TRANS_{INST}+P_{IA}*[9+NUM\_INST*2]]*BASIC\_OP$

where NUM_INST $\cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

c_I(Malta) = t_{commit} (Malta) where $P_{IA} = 1$

c_C(Malta) = t_{commit} (Malta) where $P_{IA} = 0$

In the proposed technique, t_{lock} (Proposed) is represented as follows. $t_{lock, IA}$ (Proposed) = $t_{init\text{-}lock}$ (Proposed) + $t_{breakpoint}$ + $t_{changelock}$ where $P_{IA} = 1$, $t_{init\text{-}lock}$, $t_{breakpoint}$ and $t_{changelock}$ represent time to get initial lock in instance access, time to record a breakpoint during method invocation, time to change locks after method invocation, respectively.

Also, $t_{lock,CDA}$ (Proposed) = $t_{init\text{-}lock}$ (Proposed) where $P_{IA} = 0$

$t_{init\text{-}lock}$, $t_{breakpoint}$ and $t_{changelock}$ have the following values.

Thus, $t_{init\text{-}lock}$ (Proposed)
$= [12+12*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+ NUM\_TRANS_{INST}*[10+N_M/26]]]*$
$MM\_ACCESS +[6+[19+t*2]*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+NUM\_TRANS_{INST}$
$*[15+N_M/26*2+t*2]]*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$, $t = num\_br*P_I+N_A*P_C$

$t_{breakpoint} = [5*num\_br + 3]*MM\_ACCESS + [num\_br*7 + 5]*BASIC\_OP$

$t_{changelock} = [5 +$
$2*NUM\_TRANS_{CLASS}+2*NUM\_TRANS_{INST}+2*NUM\_INST]*MM\_ACCESS$
$+$
$[15+2*NUM\_TRANS_{CLASS}+2*NUM\_INST+2*NUM\_TRANS_{INST}]*BASIC\_OP$

Thus, t_{lock} (Proposed) = t_{lock}(Proposed) + $t_{breakpoint}$ + $t_{changelock}$

$= [28+14*NUM\_TRANS_{CLASS}+4*NUM\_INST+ NUM\_TRANS_{INST}*[12+N_M/26]+5*num\_br]*$
$MM\_ACCESS+[35+[21+t*2]*NUM\_TRANS_{CLASS}+4*NUM\_INST+NUM\_TRANS_{INST}*[17+N_M/$
$26*2+t*2+num\_br*7]*BASIC\_OP$

Also, t_{commit} (Proposed) has the following value.

t_{commit} (Proposed)=
$= N_G*[6+2*NUM\_TRANS_{CLASS}+2*P_{IA}*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*$
$MM\_ACCESS +N_G*[6+2*NUM\_TRANS_{CLASS}+ 2*P_{IA}*NUM\_TRANS_{INST} +$
$P_{IA}*[9+NUM\_INST*2]]*BASIC\_OP$

where NUM_INST $\cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

Thus,

c_I(Proposed) = t_{commit} (Proposed) where $P_{IA} = 1$

c_C(Proposed) = t_{commit} (Proposed) where $P_{IA} = 0$

4.3. Analysis for each access type

In this section, for each access type (i.e., conflict among methods, class hierarchy locking, and nested method invocations), mathematical formulas for transaction response time are obtained for each technique.

4.3.1. Analysis for Conflict among methods

4.3.1.1. Response time without blocking

If there is no blocking, then $P_W = R_W = 0$. Let $R_{X, IA}$ and $R_{X, CDA}$ represent response times of an instance access (IA) transaction and a class definition access (CDA) transaction of technique X, respectively. Then, each technique has the following response time for an IA transaction. Note that C_I (X) is defined in Section 4.2.2.

$R_{ORION, IA} = t_{int} + AT*N_L*(x + t_{lock, IA}(Orion)) + c_I(Orion)$

$R_{MALTA, IA} = t_{int} + N_L*(x + t_{lock, IA}(Malta)) + c_I(Malta)$

$R_{PROPOSED, IA} = t_{int} + N_L*(x + t_{lock, IA}(Proposed)) + c_I(Proposed)$

Also, each technique has the following response time for a CDA transaction. Also, note that C_C (X) and $t_{lock, CDA}$ (Proposed) are defined Section 4.2.2.

$R_{ORION, CDA} = t_{int} + N_L*(x + t_{lock, CDA}(Orion)) + c_C(Orion)$

$R_{MALTA,\,CDA} = t_{int} + N_L*(x + t_{lock,\,CDA}\,(Malta)) + c_C\,(Malta)$

$R_{PROPOSED,\,CDA} = t_{int} + N_L*(x + t_{lock,\,CDA}\,(Proposed)) + c_C\,(Proposed)$

Finally, each transaction has the following response time for both types of transactions.

$R_{ORION} = P_{IA}* R_{ORION,\,IA} + P_{CDA} *R_{ORION,\,CDA}$

$R_{MALTA} = P_{IA} * R_{MALTA,\,IA} + P_{CDA} * R_{MALTA,\,CDA}$

$R_{PROPOSED} = P_{IA}* R_{PROPOSED,\,IA} + P_{CDA}* R_{PROPOSED,\,CDA}$

4.3.1.2. Response time with blocking

If blocking is possible, the response time for each technique is as follows

a) Orion

$P_{I,I} = \{\lambda *G_I/N_I*N_C\}* P_{IA}* \{2*P_{IR}*P_{IW} + P_{IW}*P_{IW}\}/(P_{IR}+P_{IW})^2$

$P_{I,C} = \{\lambda*G_C / N_C\}*P_{CDA} * \{P_{CDW} + P_{CDR}* P_{IW}\}/ (P_{CDA} + P_{IA})^2$

$P_{C,I} = \{\lambda*G_I/N_C\}* P_{IA} * \{P_{CDW} + P_{CDR}*P_{IW}\}/ (P_{CDA} + P_{IA})^2$

$P_{C,C} = \{\lambda*G_C/ N_C\}* P_{CDA}*\{2*P_{CDR}*P_{CDW} + P_{CDW}*P_{CDW}\} / (P_{CDR} + P_{CDW})^2$

$$G_I = \frac{N_L*(N_L+1)}{2} *a_I\,(Orion) + \frac{N_L*(N_L-1)}{2} *b_I + N_L*c_I$$

$$G_C = \frac{N_L*(N_L+1)}{2} *a_C\,(Orion) + \frac{N_L*(N_L-1)}{2} *b_C + N_L*c_C$$

where a_I(Orion) and a_C(Orion) represent the execution times of each method by an IA transaction and a CDA transaction, respectively.

Let RES_I and RES_C be the response times of an IA transaction and a CDA transaction, respectively. Then

$RES_I = t_{int} + N_L*a_I\,(Orion) + N_L*(P_{I,I}*R_I + P_{I,C}*R_C) + c_I\,(Orion)$

$RES_C = t_{int} + N_L*a_C\,(Orion) + N_L*(P_{C,I}*R_I + P_{C,C}*R_C) + c_C\,(Orion)$

The overall response time RES is

$$RES = \frac{P_{IA} * RES_I + P_{CDA} * RES_C}{P_{IA} + P_{CDA}}$$

b) Malta

$$P_{L,I} = \{\lambda * G_I / N_I * N_C\} * P_{IA} * \{2 * P_{IR} * P_{IW} * (S/A)^2 + P_{IW} * P_{IW} * (S/A)^2\} / \{(P_{IR} + P_{IW})^2 * (S/A)\}$$

$$P_{L,C} = \{\lambda * G_C / N_C\} * P_{CDA} * P_{CDW} / (P_{CDA} + P_{IA})^2$$

$$P_{C,I} = \{\lambda * G_I / N_C\} * P_{IA} * P_{CDW} / (P_{CDA} + P_{IA})^2$$

$$P_{C,C} = \{\lambda * G_C / N_C\} * P_{CDA} * \{2 * P_{CDR} * P_{CDW} + P_{CDW} * P_{CDW}\} / (P_{CDR} + P_{CDW})^2$$

$$G_I = \frac{N_L * (N_L + 1)}{2} * a_I \text{ (Malta)} + \frac{N_L * (N_L - 1)}{2} * b_I + N_L * c_I \text{ (Malta)}$$

$$G_C = \frac{N_L * (N_L + 1)}{2} * a_C \text{ (Malta)} + \frac{N_L * (N_L - 1)}{2} * b_C + N_L * c_C \text{ (Malta)}$$

where a_I (Malta) and a_C(Malta) are the execution times of each method by an IA transaction and a CDA transaction, respectively.

Let RES_I and RES_C be response time of IA transaction and CDA transaction, respectively. Then

$$RES_I = t_{int} + N_L * a_I \text{ (Malta)} + N_L * (P_{L,I} * R_I + P_{L,C} * R_C) + c_I \text{ (Malta)}$$

$$RES_C = t_{int} + N_L * a_C \text{ (Malta)} + N_L * (P_{C,I} * R_I + P_{C,C} * R_C) + c_C \text{ (Malta)}$$

The overall response time RES

$$RES = \frac{P_{IA} * RES_I + P_{CDA} * RES_C}{P_{IA} + P_{CDA}}$$

c) The proposed scheme

$$P_{L,I} = \{\lambda * G_I / N_I * N_C\} * P_{IA} * \{2 * P_{IR} * P_{IW} * (S/A)^2 / (0.9 * num\_br) + P_{IW} * P_{IW} * (S/A)^2 / (0.9 * num\_br)\} / (P_{IR} + P_{IW})^2 * (S/A)$$

$$P_{L,C} = \{\lambda * G_C / N_C\} * P_{CDA} * \{P_{CDW\text{-}WC} + P_{CDW\text{-}WM} * (1/N_M^2) + P_{CDW\text{-}WA} * (S/A)\} / (P_{CDA} + P_{IA})^2$$

$$P_{C,I} = \{\lambda * G_I / N_C\} * P_{IA} * \{P_{CDW\text{-}WC} + P_{CDW\text{-}WM} * (1/N_M^2) + P_{CDW\text{-}WA} * (S/A)\} / (P_{CDA} + P_{IA})^2$$

$$\begin{aligned} P_{C,C} = {} & \{\lambda * G_C / N_C\} * P_{CDA} * [P_{CDR\_RC} * P_{CDW\_WC} + P_{CDR\_RM} * \{P_{CDW\_WC} + P_{CDW\_WM} * \\ & (1/N_M{**}2)) + P_{CDW\_WA} * (S/A)\} + P_{CDR\_RA} * \{P_{CDW\_WC} + P_{CDW\_WA} * (1/A{**}2)\} + \\ & P_{CDW\_WC} * \{P_{CDR\_RC} + P_{CDR\_RM} + P_{CDR\_RA} + P_{CDW\_WC} + P_{CDW\_WM} + P_{CDW\_WA}\} + \\ & P_{CDW\_WM} * \{P_{CDW\_WC} + P_{CDW\_WM} * (1/N_M{**}2)) + P_{CDW\_WA} * (S/A) + P_{CDR\_RM} * \\ & (1/N_M{**}2)\} + P_{CDW\_WA} * \{P_{CDW\_WC} + P_{CDW\_WM} * (S/A) + P_{CDW\_WA} * (1/A{**}2) \\ & + P_{CDR\_RM} * (S/A) + P_{CDR\_RA} * (1/A{**}2)\}] / (P_{CDR} + P_{CDW})^2 \end{aligned}$$

$$G_I = \frac{N_L * (N_L + 1)}{2} * a_I \text{ (Proposed)} + \frac{N_L * (N_L - 1)}{2} * b_I + N_L * c_I \text{ (Proposed)}$$

$$G_C = \frac{N_L * (N_L + 1)}{2} * a_C \text{ (Proposed)} + \frac{N_L * (N_L - 1)}{2} * b_C + N_L * c_C \text{ (Proposed)}$$

where a_I(Proposed) and a_C(Proposed) are the execution times of each method by an IA transaction and a CDA transaction, respectively.

Let RES_I and RES_C be the response times of an IA transaction and a CDA transaction, respectively

$$RES_I = t_{int} + N_L * a_I \text{ (Proposed)} + N_L * (P_{L,I} * R_I + P_{L,C} * R_C) + c_I \text{ (Proposed)}$$

$$RES_C = t_{int} + N_L * a_C \text{ (Proposed)} + N_L * (P_{C,I} * R_I + P_{C,C} * R_C) + c_C \text{ (Proposed)}$$

The overall response time RES is

$$RES = \frac{P_{IA} * RES_I + P_{CDA} * RES_C}{P_{IA} + P_{CDA}}$$

4.3.2. Analysis for Class hierarchy locking

For the analysis for class hierarchy locking, we have the following assumptions

- the fan-out of each class (the number of subclasses of a class) is F (default)

- the class hierarchy depth (level) is D. Thus, the average level in the class hierarchy requested by a transaction is $A_D = \lceil (D+1)/2 \rceil$ (i.e, in the middle of the class hierarchy).

$t_{lock\text{-}CHL}$ and $t_{commit\text{-}CHL}$, for each class hierarchy locking, can be obtained as follows: Let $C_{I\text{-}CHL}(X)$ and $C_{C\text{-}CHL}(X)$ be the commit time of an IA transaction and a CDA transaction, respectively, for class hierarchy locking technique X.

a) Orion (Implicit locking)

Assume that N is the number of locks required (including intention locks).

$t_{lock\text{-}CHL}$ (Orion) =
$=[12+10*N*NUM\_TRANS_{CLASS}+P_{IA}*[7+2*NUM\_INST+7*NUM\_TRANS_{INST}]]*N*MM\_ACCESS+[6+17*N*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+11*NUM\_TRANS_{INST}]] * N*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$

$t_{lock\text{-}CHL,\ IA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 1$; $t_{lock\text{-}CHL,\ CDA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 0$

$t_{commit\text{-}CHL}$ (Orion)=
$N_G*[6+2*N*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]+[AT-1]*P_{IA}*[NUM\_TRANS_{INST}*2+3]]*N*MM\_ACCESS + N_G*[6+2*N*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[9+NUM\_INST*2]+[AT-1]*P_{IA}*[NUM\_TRANS_{INST}*2]]*N*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

$c_{I\text{-}CHL}$(Orion) = $t_{commit\text{-}CHL}$ (Orion) where $P_{IA} = 1$; $c_{C\text{-}CHL}$(Orion) = $t_{commit\text{-}CHL}$ (Orion) where $P_{IA} = 0$

b) Malta (Explicit locking)

Let N be the number of locks on classes (including locks on subclasses)

$t_{lock\text{-}CHL,\ IA}$ (Malta) =
$=[12+8*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+NUM\_TRANS_{INST}*[10+N_M/26]]]*$
$MM\_ACCESS+[6+13*NUM\_\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+NUM\_TRANS_{INST}*$
$[17+N_M/26*2]]]*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$ where $P_{IA} = 1$

$t_{lock\text{-}CHL,\ CDA}$ (Malta) =
$=[[12+8*NUM\_TRANS_{CLASS}+P_{IA}*(8+2*NUM\_INST+NUM\_TRANS_{INST}*[10+N_M/26]]]*$
$[P_{CDR}+P_{CDW}*N]/P_{CDA}]*MM\_ACCESS$
$+[6+13*NUM\_\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+2*$
$NUM\_TRANS_{INST}*[17+N_M/26*2]]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$ where $P_{IA} = 0$

$c_{I\text{-}CHL}$ (Malta)=
$N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*$
$MM\_ACCESS+N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[9+$
$NUM\_INST*2]]*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$ where $P_{IA} = 1$

$c_{C\text{-}CHL}$ (Malta)=
$N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]$
$*[P_{CDR}+P_{CDW}*N]/P_{CDA}*MM\_ACCESS+N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*$
$NUM\_TRANS_{INST}+P_{IA}*[9+NUM\_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}*BASIC\_OP$

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$ where $P_{IA} = 0$

c) The proposed scheme

Let N be all locks required (including intention locks and locks on subclasses). In this analysis, only locks on subclasses are assumed for simplicity.

$t_{lock\text{-}CHL,\ IA}$(Proposed)=
= [28+14*NUM_TRANS$_{CLASS}$+4*NUM_INST+NUM_TRANS$_{INST}$*[12+ N_M/26]
+5*num_br]*MM_ACCESS +[35+[21+t*2]*NUM_TRANS$_{CLASS}$+4*NUM_INST+
NUM_TRANS$_{INST}$ *[17+N_M/26*2+t*2]+num_br*7*N*BASIC_OP

where P_{IA}=1 and NUM_INST ≅ M_P*N_G/(2*B*N_C), NUM_TRANS$_{CLASS}$ ≅ M_P*N_G/N_C,

NUM_TRANS$_{INST}$ ≅ M_P*N_G/(N_c*N_I), t = num_br*P_I+N_A*P_C

$t_{lock\text{-}CHL,\ CDA}$ (Proposed) =
[12+12*NUM_TRANS$_{CLASS}$+P_{IA}*[8+2*NUM_INST+NUM_TRANS$_{INST}$*[10+N_M/26]]]*[
P_{CDR}+P $_{CDW}$*N]/P_{CDA} *MM_ACCESS
+[6+[19+t*2]*NUM_TRANS$_{CLASS}$+P_{IA}*[9+2*NUM_INST+
NUM_TRANS$_{INST}$*[15+N_M/26*2+t*2]]]* [P_{CDR}+P_{CDW}*N]/P_{CDA}*BASIC_OP

where NUM_INST ≅ M_P*N_G/(2*B*N_C), NUM_TRANS$_{CLASS}$ ≅ M_P*N_G/N_C,

NUM_TRANS$_{INST}$ ≅ M_P*N_G/(N_c*N_I), t = num_br*P_I+N_A*P_C where P_{IA} = 0

$c_{I\text{-}CHL}$(Proposed) =
N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2*NUM_TRANS$_{INST}$+P_{IA}*[4+NUM_INST*2]]*MM
reads
+N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2*NUM_TRANS$_{INST}$+P_{IA}*[9+NUM_INST*2]]*BA
SIC_OP

where NUM_INST ≅ M_P*N_G/(2*B*N_C), NUM_TRANS$_{CLASS}$ ≅ M_P*N_G/(2*N_C),

NUM_TRANS$_{INST}$ ≅ M_P*N_G/(2*N_c*N_I)

$c_{C\text{-}CHL}$(Proposed) =
N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2*NUM_TRANS$_{INST}$+P_{IA}*[4+NUM_INST*2]]*(P_{CDR}+
P_{CDW}*N]/P_{CDA}*MM_ACCESS+N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2*NUM_TRANS$_{INST}$+
P_{IA}*[9+NUM_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA} *BASIC_OP

where NUM_INST $\cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_i)$

4.3.3. Analysis for nested method invocations

For analysis for nested method invocations, consider the following assumptions:

- There are L_C levels in the composite object hierarchy.
- For each instance access method accessing a composite object, there are F_C number of method invocations to subobjects in the composite object hierarchy. Thus, for each instance access method invocation on the top-level composite object, there are $N_{COM} = 1+ Fc + (F_C)^2 + ... + (F_C)^{Lc}$ number of objects accessed
- For the analysis of the proposed technique, semantic commutativity is not considered for simplicity.
- Instance access methods are only considered in order to simplify analysis.

Assume that the same lock table is used as in Figure 4-1. In calculating the analytical parameter, $t_{lock\text{-}NMI}$ and $t_{commit\text{-}NMI}$ where $t_{lock\text{-}NMI}$ and $t_{commit\text{-}NMI}$ are time to get a lock by a nested method invocation transaction and time to commit by a nested method invocation transaction, respectively, the detail steps to calculate each parameter are found in the appendix.

a) Orion

$t_{lock\text{-}NMI}$ (Orion) =
= [18 + 2*NUM_INST + 7*NUM_TRANS_{INST}]*MM_ACCESS
+ [15 + 2*NUM_INST + 11*NUM_TRANS_{INST} + N_M/26*2]*BASIC_OP

where NUM_INST $M_P*N_{COM}/(2*B*N_C)$, and $NUM\_TRANS_{INST} \cong M_P*N_{COM} / (N_C*N_I)$

$t_{commit\text{-}NMI}$ (Orion)=
$N_{COM}*[9+2*NUM\_TRANS_{CLASS}+2*NUM\_TRANS_{INST}+2*NUM\_INST+[AT-1]*[2*NUM\_TRANS_{INST}+3]]*MM\_ACCESS$
$+ N_{COM}*[15+2*NUM\_TRANS_{CLASS}+ 2*NUM\_TRANS_{INST} + 2*NUM\_INST+[AT-1]*[2*NUM\_TRANS_{INST}]]*BASIC\_OP$

where $NUM\_INST \cong M_P/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P/(2*N_c*N_I)$

b) Malta's

$t_{lock\text{-}NMI}$ (Malta) =
$= [18 + 2*NUM\_INST + NUM\_TRANS_{INST} *[10+ N_M/26]]*MM\_ACCESS$
$+ [15 + 2*NUM\_INST + NUM\_TRANS_{INST} [17+ N_M/26*2]]*BASIC\_OP$

where NUM_INST $M_P*N_{COM}/(2*B*N_C)$, and $NUM\_TRANS_{INST} \cong M_P*N_{COM} / (2*N_C*N_I)$

$t_{commit\text{-}NMI}$ (Malta)=
$N_{COM}*[9+2*NUM\_TRANS_{CLASS}+2*NUM\_TRANS_{INST}+2*NUM\_INST]*MM\_ACCESS$
$+N_{COM}*[15+2*NUM\_TRANS_{CLASS}+2* NUM\_TRANS_{INST}+ 2*NUM\_INST]*BASIC\_OP$

where $NUM\_INST \cong M_P/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$,

$NUM\_TRANS_{INST} \cong M_P/(2*N_c*N_I)$

c) The proposed scheme

In the proposed scheme, $t_{lock\text{-}NMI}$ (Proposed) = $t_{init\text{-}lock\text{-}NMI} + t_{breakpoint\text{-}NMI} + t_{changelock\text{-}NMI}$. Each

of $t_{init\text{-}lock\text{-}NMI}$, $t_{breakpoint\text{-}NMI} + t_{changelock\text{-}NMI}$ is as follows.

$t_{init\text{-}lock\text{-}NMI}$ =
$= [18 + 2*NUM\_INST + NUM\_TRANSA_{INST} *[10+N_M/26]]*MM\_ACCESS$
$+ [15 + 2*NUM\_INST + NUM\_TRANS_{INST} *[17+ N_M/26*2]]*BASIC\_OP$

$t_{breakpoint\text{-}NMI}$ =
$= [9*num\_br+(num\_br+1)*N_{COM}+12]*MM\_ACCESS$
$+ [9*num\_br + (2*num\_br+2)*N_{COM}+7]*BASIC\_OP$

$t_{changelock\text{-}NMI}$ =
= [5 + 2*NUM_TRANS$_{CLASS}$+2*NUM_TRANS$_{INST}$+2*NUM_INST]*MM_ACCESS
+ [15+2*NUM_TRANS$_{CLASS}$+2*NUM_INST+2*NUM_TRANS$_{INST}$]*BASIC_OP

$t_{commit\text{-}NMI}$ (Proposed)=
N_{COM}*N_{COM}/2*[9+2*NUM_TRANS$_{CLASS}$+2*NUM_TRANS$_{INST}$+2*NUM_INST]
*MM_ACCESS
+N_{COM}*N_{COM}/2*[15+2*NUM_TRANS$_{CLASS}$+2* NUM_TRANS$_{INST}$+ 2*NUM_INST]
*BASIC_OP

where NUM_INST $\cong N_{COM}$* M_P*N_C/(2*B*N_C), NUM_TRANS$_{CLASS} \cong N_{COM}$* M_P/(2*N_C),

NUM_TRANS$_{INST} \cong N_{COM}$* M_P/(2*N_c*N_I)

In order to find the response time for each technique, it is sufficient to find P_W and R_W of each technique. The same notations are adopted as in Section 4.1.

a) Orion

$$G = \left\{\sum_{i=1}^{N_{com}}(ia+(i-1)b)\right\} + N_{com}*c$$

where a = x (i.e., execution time for each granule) and c = $t_{commit\text{-}NMI}$ (Orion)

$$R_w = \frac{(a+b)^2(N_{com}+1)(N_{com}-1)}{6} \Big/ \left\{a\left(\frac{N_{com}+1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \frac{(N_{com}+1)((a^2/f_3)+ab+ac+bc)}{2} \Big/ \left\{a\left(\frac{N_{com}-1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \left\{\frac{c^2}{f_3}-ab-bc\right\} \Big/ \left\{a\left(\frac{N_{com}+1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$P_W = \{\lambda*(N_{COM}/L)\}*\{G/N_{COM}\}*(2*P_{IR}*P_{IW} + P_{IW}*P_{IW})/(P_{IR}+P_{IW})^2$

Thus, $Res_{orion} = t_{init} + (x+AT*t_{lock\text{-}NMI}(Orion))*N_{COM} + N_{COM}*P_W*R_W + t_{commit\text{-}NMI}(Orion)$

b) Malta's

$$G = \left\{\sum_{i=1}^{N_{com}}(ia+(i-1)b)\right\} + N_{com}*c$$

where a = x (i.e., execution time for each granule) and c = $t_{commit\text{-}NMI}$ (Malta)

$$R_w = \frac{(a+b)^2(N_{com}+1)(N_{com}-1)}{6} \Big/ \left\{a\left(\frac{N_{com}+1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \frac{(N_{com}+1)((a^2/f_2)+ab+ac+bc)}{2} \Big/ \left\{a\left(\frac{N_{com}-1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \left\{\frac{c^2}{f_3}-ab-bc\right\} \Big/ \left\{a\left(\frac{N_{com}+1}{2}\right)+c+b\left(\frac{N_{com}-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$P_W = \{\lambda^* (N_{COM}/L)\}*\{G/N_{COM}\}*\{2*P_{IR}*P_{IW}*(S/A)^2 + P_{IW}*P_{IW}*(S/A)^2\}/\{(P_{IR}+P_{IW})^2 *(S/A)\}$

Thus, $Res_{malta} = t_{init} + (x+t_{lock\text{-}NMI}(Malta))*N_{COM} + N_{COM}*P_W*R_W + t_{commit\text{-}NMI}(Malta)$

c) The proposed scheme

$$G = \left\{\sum_{i=1}^{Lc}(ia+(i-1)b)\right\} + N_{com}*c$$

where a = x (i.e., execution time for each granule) and c = $t_{commit\text{-}NMI}$ (Proposed)

$$R_w = \frac{(a+b)^2(Lc+1)(Lc-1)}{6} \Big/ \left\{a\left(\frac{Lc+1}{2}\right)+c+b\left(\frac{Lc-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \frac{(Lc+1)((a^2/f_2)+ab+ac+bc)}{2} \Bigg/ \left\{a\left(\frac{Lc-1}{2}\right)+c+b\left(\frac{Lc-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$+ \left\{\frac{c^2}{f_3}-ab-bc\right\} \Bigg/ \left\{a\left(\frac{Lc+1}{2}\right)+c+b\left(\frac{Lc-1}{2}\right)\left(1-\frac{1}{f_1}\right)\right\}$$

$$P_W = \{\lambda^* (N_{COM}/L)\}^*\{G/N_{COM}\}\ \{2^*P_{IR}^*P_{IW} * (S/A)^2/(0.9^*num\_br) + P_{IW}^*P_{IW}^*(S/A)^2/(0.9^*num\_br)\}\ /(P_{IR}+P_{IW})^2*(S/A)$$

$$Res_{PROPOSED} = t_{init} + (x+t_{lock\text{-}NMI}\ (Proposed))^*L_C + L_C^*P_W^*R_W + t_{commit\text{-}NMI}\ (Proposed)$$

4.4. Analysis

In this section, the performance of the three techniques are evaluated based on the analytical results using the 007 benchmark. The performance for access type as well as the overall performance are presented.

4.4.1. Conflict among methods

As can be seen in Figure 4-2, as instance write ratio increases, transaction response time increases since the probability of conflict becomes higher. The possible conflicts include conflicts between instance reads and instance writes (in all schemes) and between instance write and class definition accesses (in only Orion). Analysis for each technique is as follows: in Figure 4-2, as instance write ratio increases, Orion performs worst. This is due to the fact that, in Orion, their locking granularity for instance access is an instance object and also a lock is required whenever an atomic action is invoked during an instance

access method invocation. In Malta's work, they adopt attribute as locking granularity for an instance access and a lock is required for an instance access method invocation. As in Malta's, the proposed technique adopts attribute locking granularity and a lock is required for each instance access method invocation. In addition, dynamic information is adopted to increase concurrency among methods in the proposed scheme. As Figure 4-2 shows, since Orion's performance falls dramatically as instance write ratio increases, relative performance of Malta's and the proposed scheme is insensitive.

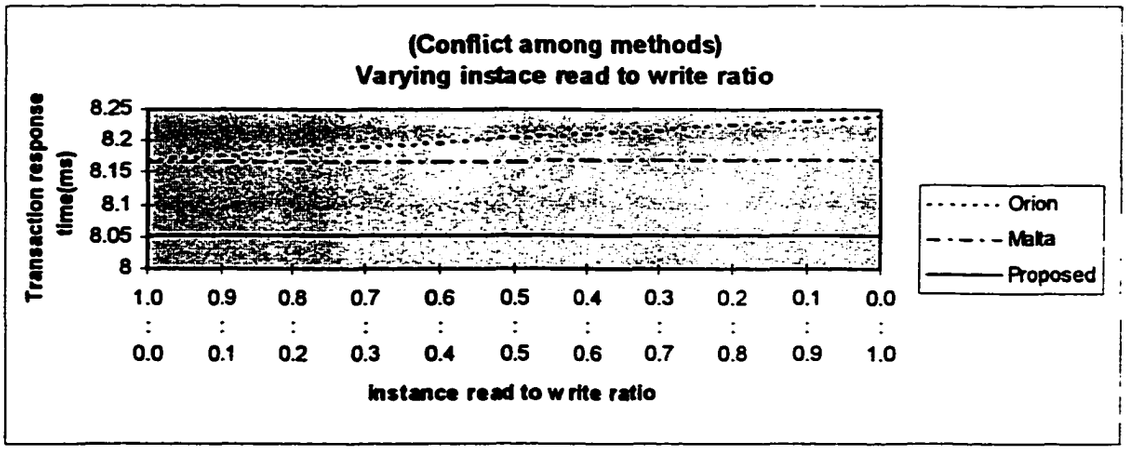


Figure 4-2. Varying instance read to write (Conflict among methods)

In Figure 4-3, as the class definition write ratio increases, the proposed scheme performs best and Orion does worst. This is due to the fact that, in Orion, class definition reads conflicts with instance writes. In Malta's work, they provide concurrency between instance writes and class definition reads, but their locking granularity for class definition access is still an entire class object. In the proposed technique, unlike other work, class

definition write access may run concurrently with other class definition access or instance access as long as they access disjoint sets of objects. As class definition write ratio increases, the difference become increases.

Figure 4-4 shows testing case of varying arrival rate. The higher arrival rate means the bigger load in the system. In turn, the bigger system load results in more conflicts on shared resources such as CPU or data items among transactions so that transaction response times become higher. As shown in Figure 4-4, when the system load is light, there is no clear winner among the three techniques. But, when the load is heavy, Orion and Malta perform worse. The proposed technique is immune to arrival rate due to higher concurrency than other schemes. Especially, Orion's performance becomes worst due to its big locking granularity for instance access and class object and low concurrency among its lock types.

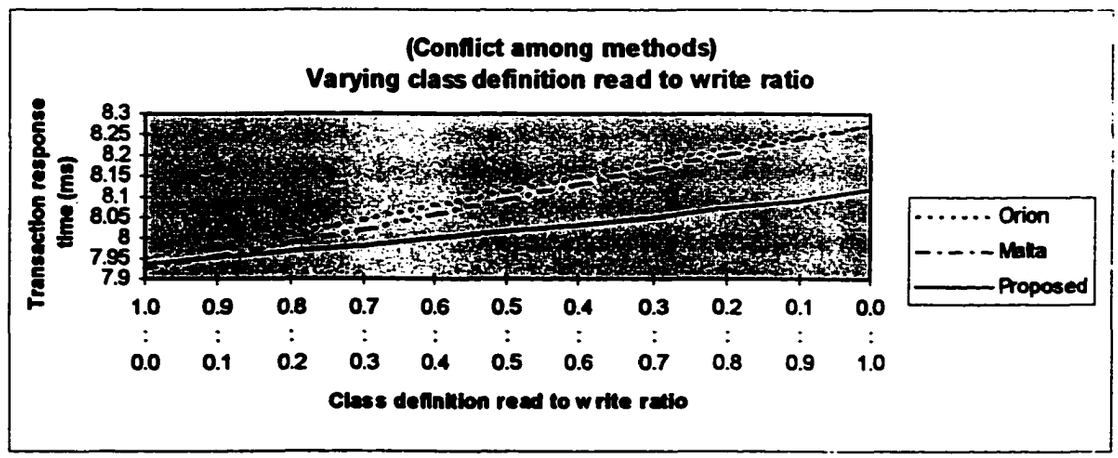


Figure 4-3. Varying class definition read to write ratio (Conflict among methods)

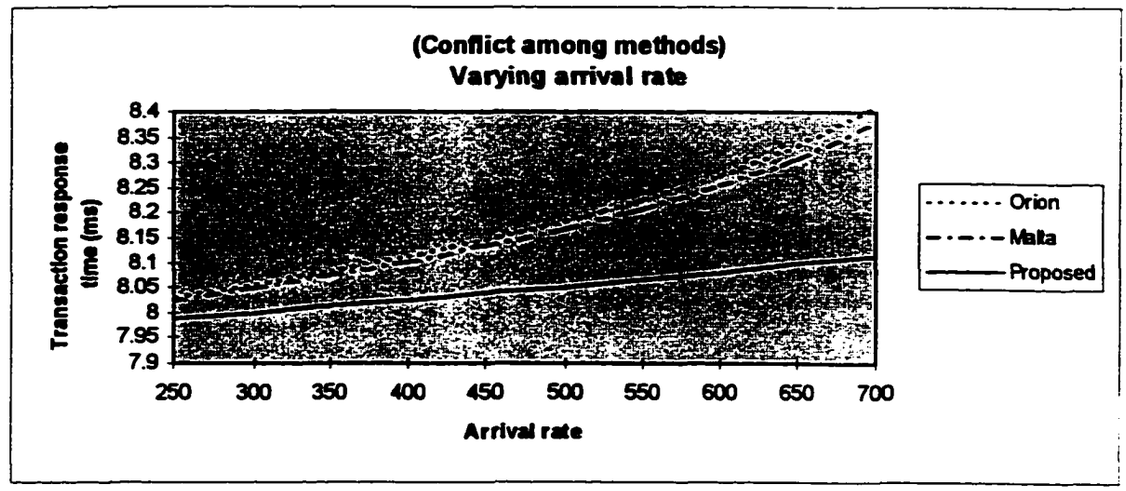


Figure 4-4. Varying arrival rate (Conflict among methods)

4.4.2. Class hierarchy locking

Figure 4-5 shows testing case of varying class definition read to write ratio. As the ratio increases, transaction response times become higher. This is due to the fact that, as the ratio is increased, the conflicts among transactions are also increased. When class higher locking is considered, as the ratio is increased, Orion, which adopts explicit locking, incurs higher overhead since locking overhead for MCA access is increased. Likewise, Malta's scheme, which adopts implicit locking, incur higher overhead since locking overhead due to intention locks is increased. Thus, both schemes is very sensitive to the ratio change. On the other hand, the proposed scheme incurs less locking overhead than both explicit locking and implicit locking. Thus, transaction response time in the proposed scheme is immune to the ratio change.

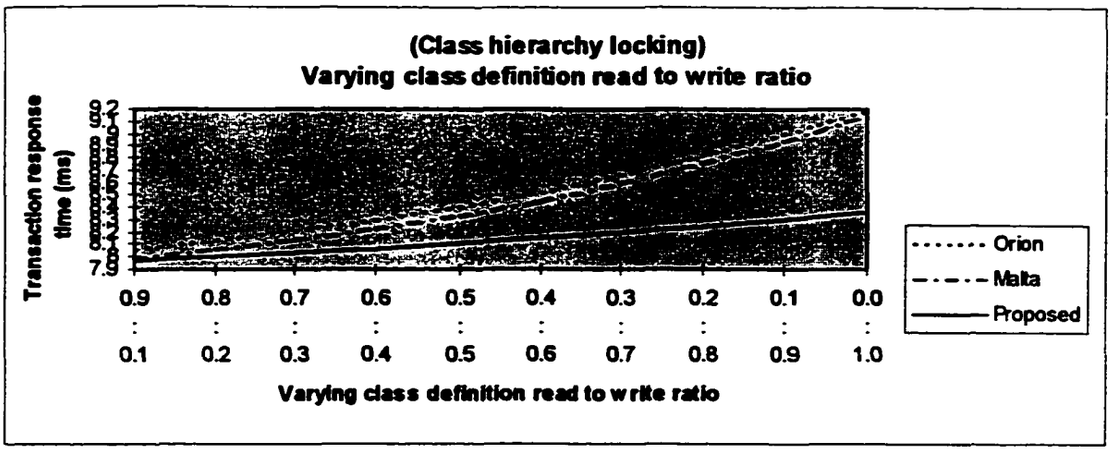


Figure 4-5. Varying class definition read to write ratio (Class hierarchy locking)

In Figure 4-6, as access to class hierarchy goes down from root to bottom, Orion takes more response time. This is due to the fact that implicit locking is adopted in Orion. It requires more intention locks as access to the class hierarchy is near leaf level. On the other hand, Malta adopts explicit locking. In explicit locking, access to root requires more locks for class definition write and query type access. Thus, as access to a class hierarchy toward the bottom, it requires fewer locks. But, in the 007 benchmark, there are only 3 levels in the class hierarchy. Thus, the performances of Malta's technique and the proposed scheme (whose locking scheme takes less overhead than Malta's) do not change much.

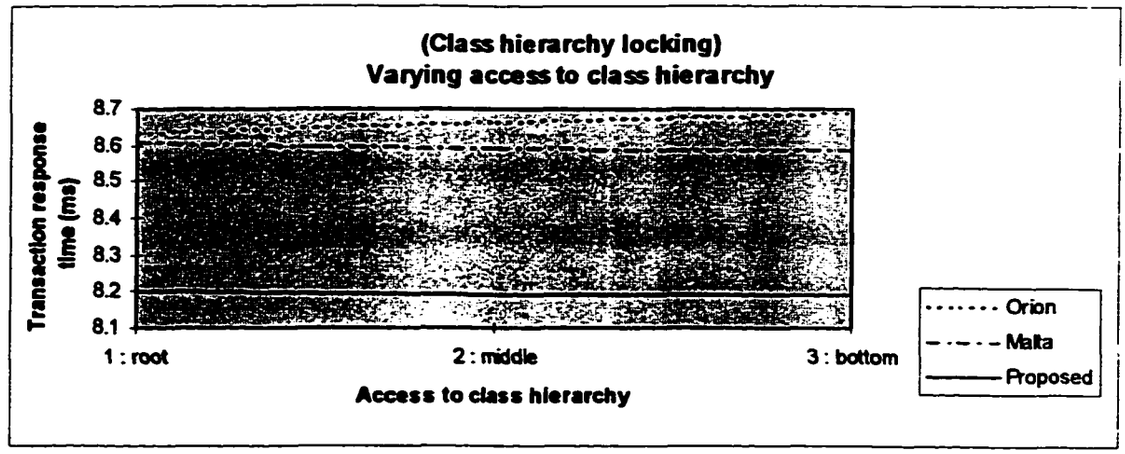


Figure 4-6. Varying access to class hierarchy (Class hierarchy locking)

The Figure 4-7 shows the performance of the three techniques when varying arrival rate. As arrival rate increases, the transaction response time increases in all three techniques. This is due the fact that, the system load increases, conflicts on data items among transactions may be increased. Especially, in the proposed scheme, the class hierarchy locking is based on a hybrid of implicit locking and explicit locking. It is based on special class and provides fewer number of locks than both implicit locking and explicit locking. Orion requires locks in class hierarchy locking. Thus, its performance is worst.

4.4.3. Nested method invocations

Figure 4-8 shows the performance of the three techniques when varying instance read to write ratio. The performances of both Orion and Malta's scheme are worse than the proposed work. This is mainly due to the fact that parent/children parallelism is not allowed in both Orion and Malta's work. Also, the small difference between Orion and Malta's scheme is due to the non-parallelism between parent/children. Thus, as the depth of a composite object hierarchy increases, the differences become clear. The other reasons for big difference between the proposed scheme and other two schemes are their locking

granularity and concurrency degree as in testing case of conflict among methods. Figure 4-9 shows testing case of varying arrival rate. As in Figure 4-8, the proposed scheme is better than both Orion and Malta's scheme. This difference is due to mainly the parallelism. Also, Malta's work is slightly better than Orion in their performance.

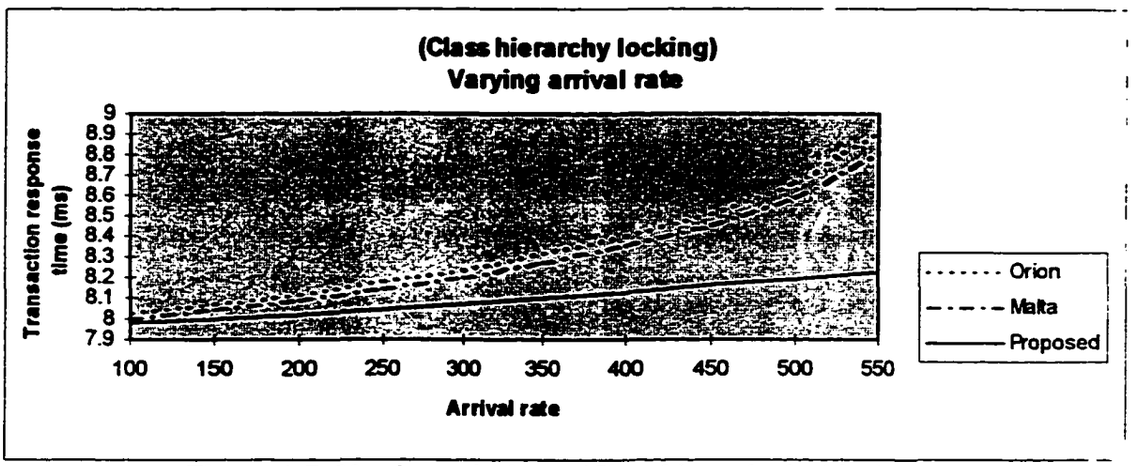


Figure 4-7. Varying arrival rate (Class hierarchy locking)

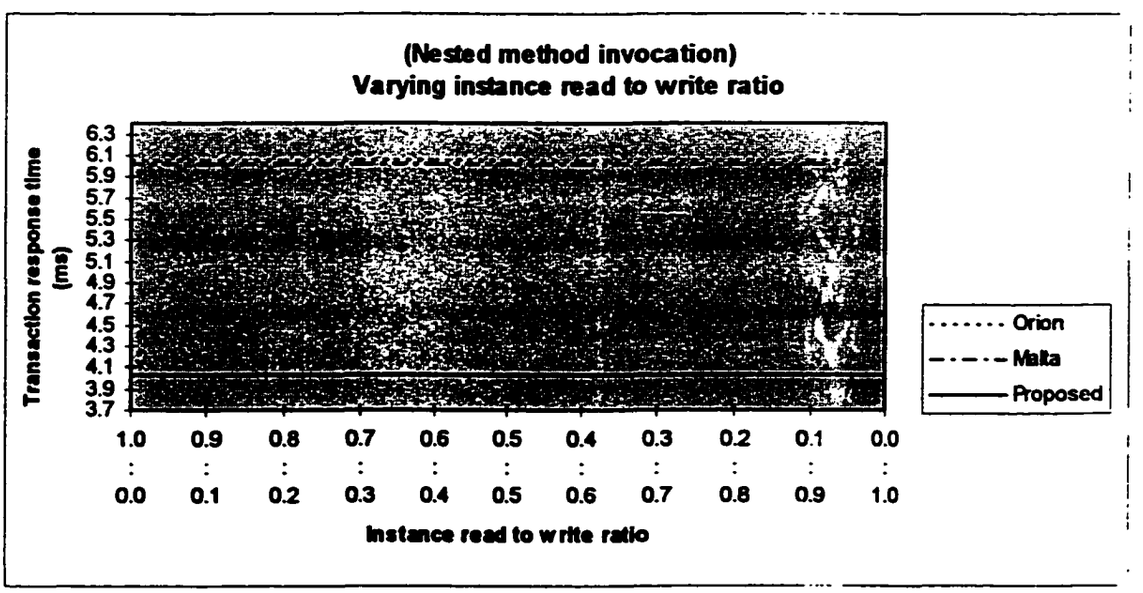


Figure 4-8. varying instance read to write ratio (Nested method invocations)

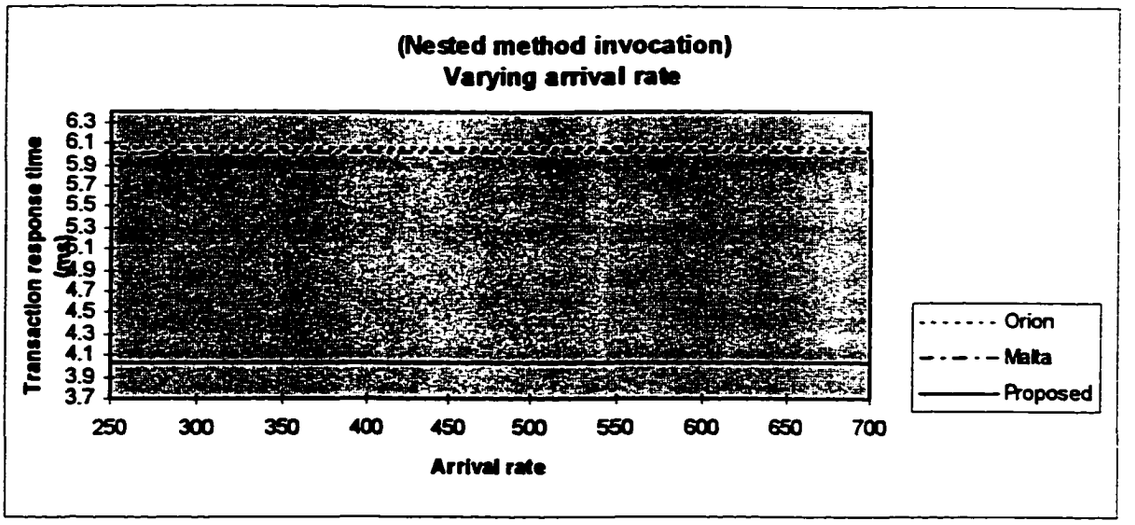


Figure 4-9. Varying arrival rate (Nested method invocations)

4.4.4. Overall performance

In order to get the overall performance for both class hierarchy locking and nested method invocation, the following principle is adopted: for each technique, let RES_{REG} and RES_{NEST} be the response time of regular transaction (i.e., transactions concerning with only conflict among methods and class hierarchy locking) and response time of transactions invoking nested method, respectively. Also, let P_{REG} and P_{NEST} be the probability of regular transaction and probability of nested method invoking transaction, respectively where $P_{REG} + P_{NEST} = 1$. Then, the overall performance can be measured as follows: $RES_{OVERALL} = P_{REG} * RES_{REG} + P_{NEST} * RES_{NEST}$

Figure 4-10 shows the performance of three techniques when the ratios of P_{REG} and P_{NEST} is varied. As in Figures 4-5 to 4-9, the response times in class hierarchy locking are much higher than the response times in nested method invocations, for all three techniques. Thus, as the nested method invocation ratio decreases, the overall transaction

response times is increased. Also, as in Figure 4-8 and 4-9, the difference between the proposed work and the other works is bigger. Thus, as the nested method invocation ratio increases, the difference between the proposed scheme and the other schemes becomes higher.

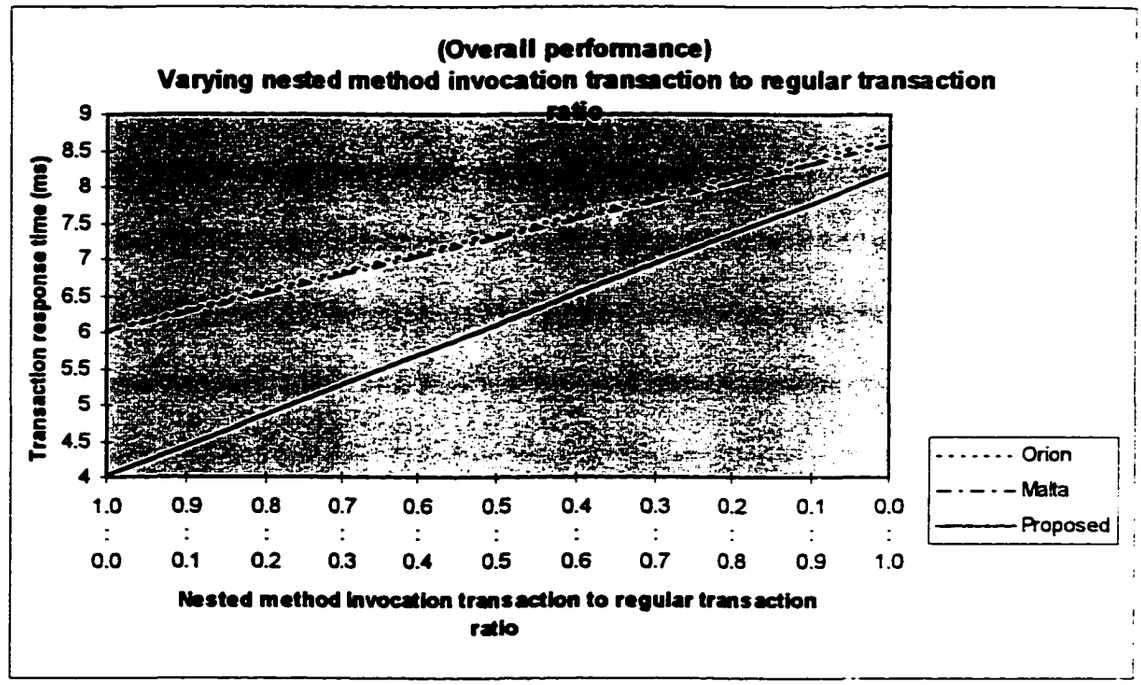


Figure 4-10. Varying nested method invocation to regular transaction ratio (Overall Performance)

Chapter 5

Performance Evaluation and Analysis by Simulation

5.1. Introduction

In Chapter 4, performance evaluation and analysis are done by mathematical modeling, under restricted environments. In order to evaluate the proposed technique in more general environments, in this chapter, a simulation model is constructed and extensive simulation experiments are conducted. In Section 5.2, a simulation model is introduced. In Section 5.3, the simulation parameters and simulation methodology are discussed. In Section 5.4, simulation results from various testing cases and analysis are presented.

5.2. Simulation Model

The simulation model is constructed from models used in existing works for concurrency control performance evaluation ([Kim, 1991]). Also, the simulation model is implemented using SLAM II simulation language [Prit,1986]. Figure 5.1 shows a general diagram of the simulation model.

5.2.1 Simulation Component Descriptions

The simulation model has six major components: transaction generator, transaction manager, CPU scheduler, concurrency control manager (lock manager), deadlock manager, buffer manager. Also, it consists of two physical resources : CPU, memory.

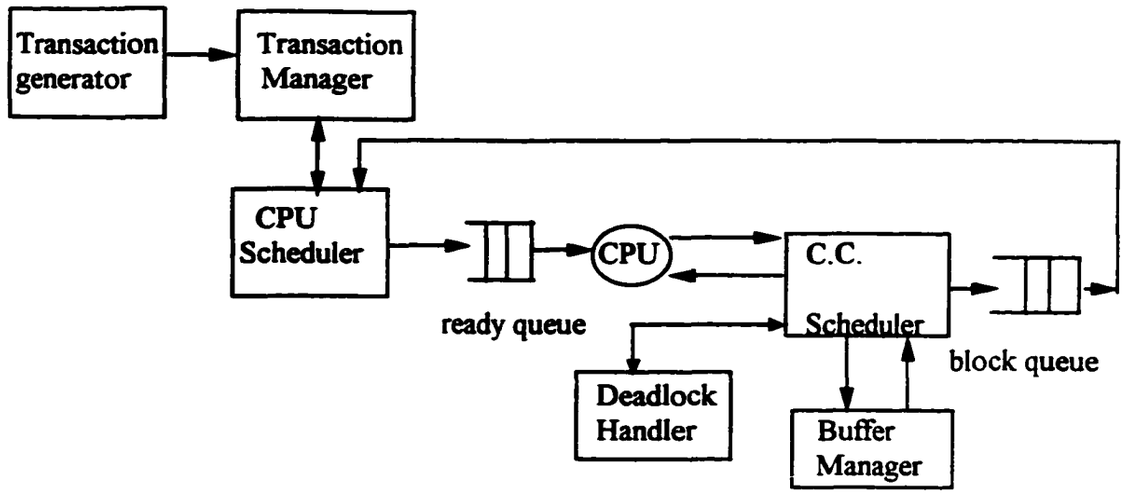


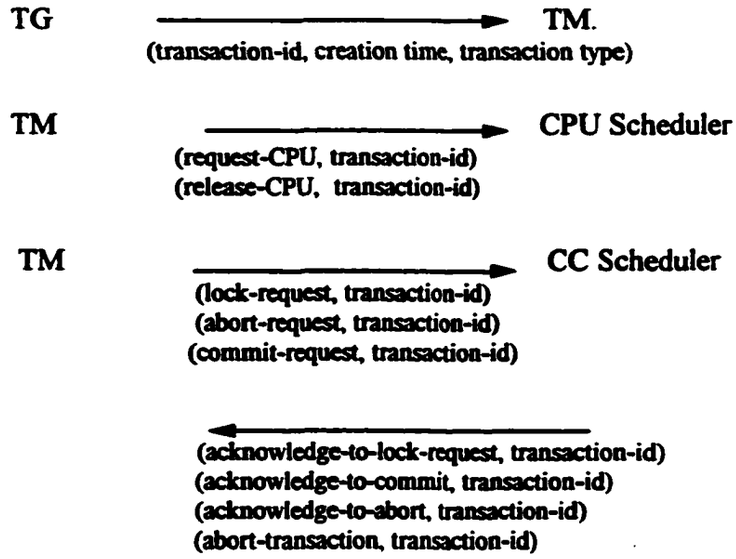
Fig. 5.1, Simulation Model ([Kim, 1991])

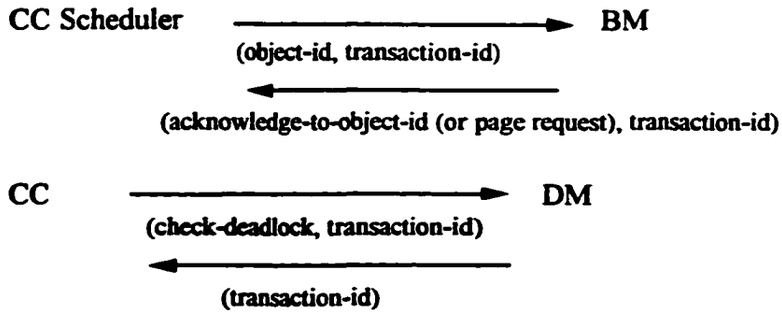
The transaction generator (TG) creates each transaction with its creation time, unique transaction identifier and transaction type. Each transaction consists of a sequence of (method, object-id) pairs. The transaction manager (TM) is responsible for scheduling and executing all transactions. It sends lock/unlock requests as well as abort/commit messages to the concurrency control (CC) scheduler. It also restarts aborted transactions. The CPU scheduler performs various CPU-related operations such as executions of methods. The CPU can be released by a transaction as a result of a lock conflict or for an I/O operation. The FIFO (First-in First-out) is chosen for CPU scheduling scheme. That is, any transaction arriving to CPU first has the higher priority. Also, any transaction holding CPU can not be preempted by other transactions. The concurrency control scheduler (CC scheduler) synchronizes data access requests of transactions. The CC scheduler orders the

data accesses based on the concurrency control protocol executed. An access request of a transaction is either granted or results in blocking or abort of the transaction. If access request is granted, the transaction attempts to read the data item from the MM (main memory). The data access to MM is done by buffer Manager (BM). Since main-memory database is assumed in this simulation, there is no page fault. The FIFO strategy is used in the management of memory buffer. The deadlock manager (DM) detects any deadlock occurred during data item access. If a transaction is blocked for specific time period, DM is invoked to check a deadlock using WFG (wait-for graph). If a cycle is detected, then the transaction will be aborted and restarted.

5.2.2. Message interface among simulation modules

In this subsection, messages among components are defined. Note that an arrow represents direction for message, and contents within parenthesis represent messages.





5.2.3. Algorithms of Simulation Modules

In this subsection, the algorithm of each simulation module is presented.

a). Transaction Generator (TG)

For each transaction:
Generate transaction-id, creation time, transaction type;
// Each transaction has the following structure: ([method$_1$,object-id$_1$]...[method$_N$,object-id$_N$]) ;
Pass the transaction to the transaction manager (TM);

b). Transaction Manager (TM)

1) accept (transaction-id and its transaction type) from TG

request multiprogramming resource (MP) ;
send (request-CPU, transaction-id) to CPU Scheduler;
While *there is still access remained in that transaction* do
    send (request-CPU, transaction-id) to CPU Scheduler;
    send (lock-request, transaction-id) to CC Scheduler;
    send (release-CPU, transaction-id) to CPU scheduler;
End while;
send (request-CPU, transaction-id) to CPU scheduler;
send (commit-request, transaction-id) to CC Scheduler;
send (release-CPU, transaction-id) to CPU Scheduler;

2) accept (abort-transaction, transaction-id) from the CC Scheduler
// Due to deadlock and conflict resolution scheme

send (request-CPU, transaction-id) to CPU Scheduler;
send (abort-request, transaction-id) to CC Scheduler;
send (release-CPU, transaction-id) to CPU Scheduler;

3) accept (acknowledge-to-lock-request, transaction-id) from CC scheduler through CPU Scheduler

go to step 1) // process next (transaction-id, transaction-type)

4) accept (acknowledge-to-abort, transaction-id) from CC scheduler through CPU Scheduler
// restart an aborted transaction

send (request-CPU, transaction-id) to CPU Scheduler
do any bookkeeping work(or statistics) for the aborted transaction;
  While *there is still access remained in that transaction* do
    send (request-CPU, transaction-id) to CPU Scheduler;
    send (lock-request, transaction-id) to CC Scheduler;
    send (request-CPU, transaction-id) to CPU scheduler;
    End while;
  send (request-CPU, transaction-id) to CPU scheduler;
  send (commit-request, transaction-id) to CC Scheduler;
  send (release-CPU, transaction-id) to CPU Scheduler;

5) accept (acknowledge-to-commit, transaction-id) from CC scheduler through CPU Scheduler
// Do any bookkeeping work(statistics) for the committed transaction
send (request-CPU, transaction-id) to CPU Scheduler;
do any bookkeeping work(statistics) for the committed transaction;
send (release-CPU, transaction-id) to CPU Scheduler;
free MP;
go to step 1); // process next (transaction-id, transaction-type)

c). CPU Scheduler

1) accept (request-CPU, transaction-id) from TM

  If CPU-busy = yes then
    put (request-CPU, transaction-id) into ready queue
  else
    assign CPU to the transaction with transaction-id
  end if

2) accept (release-CPU, transaction-id) from TM

  release CPU;
  If there is waiting transaction(s) in the ready queue then
    pick the first transaction in the ready queue and assign CPU
  end if

d). CC Scheduler

1) accept (lock-request, transaction-id) from TM through CPU Scheduler

  send (request-CPU, transaction-id) to CPU Scheduler

If lock is conflict then // depending on schemes
put (transaction-id, method-id, object-id) in the block queue;
send (release-CPU, transaction-id) to CPU Scheduler;
else
get a lock; // depending on schemes
send (object-id, transaction-id, method-id) to BM;
wait-for (acknowledge-to-object-id, transaction-id, method-id) from BM;
send (acknowledge-to-lock request, transaction-id, method-id) to TM;
end if

2) accept (abort-request, transaction-id) from TM

send (request-CPU, transaction-id) to CPU Scheduler;
remove all entry with transaction-id from lock table;
wake any blocked transaction and send it to TM;
send (acknowledge-to-abort, transaction-id) to TM;
send (release-CPU, transaction-id) to CPU Scheduler;

3) accept (commit-request, transaction-id) from TM

send (request-CPU, transaction-id) to CPU Scheduler;
remove all entry with transaction-id from lock table;
send (acknowledge-to-commit, transaction-id) to TM through;
wake up any blocked transaction and send it to TM;
send (release-CPU, transaction-id) to CPU Scheduler;

4) accept (transaction-id) from DM
send (request-CPU, transaction-id) to CPU Scheduler;
send (abort-transaction, transaction-id) to TM;

e). Deadlock Manager

accept (check-deadlock, transaction-id) from CC scheduler;
check cycle in WFG (wait-for-graph);
send (transaction-id) to CC scheduler;

f). Buffer Manager

1) accept (object-id, transaction-id) from CC scheduler
send (request-CPU, transaction-id) to CPU Scheduler;
get *page number* corresponding to *object-id*;// use buffer table
send (acknowledge-to-page-request, transaction-id) to CC scheduler;
send (release-CPU, transaction-id) to CPU scheduler;

5.3. Simulation parameter and methodology

5.3.1. OO7 Benchmark Descriptions

The 007 benchmark is chosen in order to evaluate the proposed locking scheme in OODB There have been a number of benchmarks in OODB environments ([Cart,1992], [Ande,1990], [Berr,1991]). But, existing benchmarks are not comprehensive so that wide range of OODB features can not be tested accordingly. For example, HyperModel [Ande,1990] does not include object queries and repeated object updates. Also, it is difficult for testers to implement the model from their specifications.

The 007 benchmark ([Care,1993],[Care,1994-1],[Care,1994-2]) provides a comprehensive test of OODB performance than its predecessors. Especially, it provides wide range of pointer traversal including sparse traversals and dense traversals, a rich set of updates and queries including sparse updates and the creation and deletion of objects. Also, its implementation details are open to public so that OODB testers can implement the benchmark easily.

There are three sizes of the 007 benchmark: *small*, *medium* and *large*. Table 5-1 shows the parameters of the 007 benchmark. There are ten classes in the 007 benchmark. Among those ten classes, classes DesignObj and Assembly serve as abstract superclass in which provide class definitions but not instance object. The DesignObj is the root of the class hierarchy and is (direct) superclass of classes AtomicPart, CompositePart, Assembly and Module, respectively. Also, the Assembly class is (direct) superclass of classes ComplexAssembly and BaseAssembly, respectively.

Parameters	Small	Medium	Large

NumAtomicPerComp	20	200	200
NumConnPerAtomic	3,6,9	3,6,9	3,6,9
DocumentSize (bytes)	2000	20000	20000
Manual Size (bytes)	100K	1 M	1 M
NumCompPerModule	500	500	500
NumAssmPerAssm	3	3	3
NumAssmLevels	7	7	7
NumCompPerAssm	3	3	3
NumModules	1	1	1

Table 5-1. 007 Benchmark parameters [Care,1994-1]

The 007 benchmark consists of two components: the design library and assembly hierarchy. The key component of the design library is a set of composite parts, forming CompositePart class. Each composite part is associated with document object (Document class). Also, each composite part consists of a set of atomic parts, forming AtomicParts class. In small 007, 20 atomic parts form a composite part. The connections between atomic parts are supported by the a Connection object between each pair of atomic parts.

The Assembly Hierarchy provides higher structure to the Design Library. Especially, each assembly is either consisted of composite part (the assembly is called a BaseAssembly class) or it is consisted of other assembly objects (the assembly is called a ComplexAssembly class). There are 7 levels in the assembly hierarchy. The bottom level of the assembly hierarchy consists of base assembly objects. Each base assembly object is

associated with composite part object bi-directionally. The higher level consists of complex assemblies. Each complex object is associated with either base assemblies (if the complex object has level two) or other complex object (if the complex object has higher level). Each assembly hierarchy forms a module which is the largest unit. Each module is associated with a Manual object.

5.3.2. Simulation Parameters

All the parameters used in simulation are summarized in Tables 5-2 and 5-3. Note that the OODB benchmark 007 ([Care,1993], [Care,1994-1], [Care,1994-2]) is adopted to define database and transaction-related parameters. Also, all the parameters related to machine and disk are derived from the DEC 3000 Model 400/400S AXP Alpha workstation [DEC, 1993] and Micropolis 22000 disk drivers [SCSI, 1993], respectively. The following notations are used to classify the simulation parameters accordingly.

M : Machine related parameters
D : Disk related parameters
TR : Transaction related parameters
DB : Database related parameters

Parameters	Default Value [Reference]
M: CPU power	140 MIPS [Dec, 1993]
M: time to process one operation	0.000007 ms [Huan,1995]
M: mean time to set a lock by an instance access transaction	0.3641 ms (Orion) [calculated] 0.3537 ms (Malta) [calculated] 0.3572 ms (Proposed) [calculated]
M: mean time to release a lock by instance access transaction	0.0035 ms (Orion) [calculated] 0.0019 ms (Malta) [calculated] 0.0019 ms (Proposed) [calculated]
M: mean time to set a lock by class definition access transaction	0.3522 ms (Orion) [calculated] 0.3522 ms (Malta) [calculated] 0.3522 ms (Proposed) [calculated]

M: mean time to release a lock by class definition access transaction	0.0011 ms (Orion) [calculated] 0.0011 ms (Malta) [calculated] 0.0011 ms (Proposed)[calculated]
M: number of bytes per word	4 [Dec, 1993]
M: Memory word access time	0.00018 ms [Dec, 1993]
M: number of memory buffer	20 [DEC, 1993]
D: Size of disk(block) page	2048 bytes [SCSI, 1993]
D: Avg. disk seek time	10 ms [SCSI, 1993]
D: Avg. disk latency time	5.56 ms [SCSI, 1993]
D: Disk page transfer time	0.0064 ms [SCSI, 1993]
D: Number of pages in Database	1997 pages [Calculated]
DB: NumAtomicPerComp	20 [Care,1994]
DB: NumConnPerAtomic	3,6,9 [Care,1994]
DB: NumCompPerModule	500 [Care,1994]
DB: NumAssmPerAssm	3 [Care, 1994]
DB: NumAssmLevels	7 [Care, 1994]
DB: NumCompPerAssm	3 [Care, 1994]
DB: NumModules	1 [Care, 1994]
DB: Number of instances in class Module	1 [Care, 1994]
DB: Number of instances in class Manual	1 [Care, 1994]
DB: Number of instances in class ComplexAsssembly	364 [Care, 1994]
DB: Number of instances in class BaseAssembly	729 [Care, 1994]
DB: Number of instances in class CompositePart	500 [Care, 1994]
DB: Number of instances in class Document	500 [Care, 1994]
DB: Number of instances in class AtomicPart	10000 [Care, 1994]
DB: Number of instances in class Connection	30000 [Care, 1994]

Table 5-2. Static Parameters of the Simulation Model

Parameters	Default value (Range)
M: multiprogramming level	10 (5 - 15)
TR: Prob. of Traversal	0.45 (0 - 1)
TR: Prob. of Query	0.45 (0 - 1)
TR: Prob. of Structural Modification	0.1 (0 - 1)
TR: Prob. of Traversal T1 (Traversal type)	0.08 (0 - 1)
TR: Prob. of Traversal T6 (Traversal type)	0.08 (0 - 1)
TR: Prob. of Traversal T2 (Traversal type)	0.08 (0 - 1)
TR: Prob. of Traversal T3 (Traversal type)	0.08 (0 - 1)
TR: Prob. of Traversal T8 and T9 (Traversal type)	0.08 (0 - 1)
TR: Prob. of Traversal CU (Traversal type)	0.05 (0 - 1)
TR: Prob. of Query Q1 (Query type)	0.09 (0 - 1)

TR: Prob. of Query Q2, Q3 and Q7 (Query type)	0.09 (0 - 1)
TR: Prob. of Query Q4 (Query type)	0.09 (0 - 1)
TR: Prob. of Query Q5 (Query type)	0.09 (0 - 1)
TR: Prob. of Query Q8 (Query type)	0.09 (0 - 1)
TR: Prob. of Insert (Structural Modification)	0.05 (0 - 1)
TR: Prob. of Delete (Structural Modification)	0.05 (0 - 1)
TR: Transaction interarrival time	500 (100 - 1000)

Table. 5-3. Dynamic Parameters of the Simulation Model

5.3.2. Simulation Methodology

The 007 benchmark has three database sizes: small, medium and large. Each has different number of instance per class. For the simulation, small size is selected for simplicity. It is assumed that the transaction arrivals are based on the Poisson distribution. In the Poisson distribution, any transaction arrival time is totally random [Freu,1987]. In this simulation, each transaction has equal probability to be generated. Also, in order to prevent system overload, the total number transactions in the system at any moment is limited by the parameter *Multiprogramming level*.

For the simulation, the following methodology is adopted. For CPU scheduling, transaction arrived earlier to CPU ready queue has higher priority. Unless a transaction is blocked due to I/O or conflict by lock request, the transaction can hold CPU without any preemption. Also, the FIFO policy is also adopted for the buffer management for simplicity, although this policy may not produce the best performance. In order to manage deadlock, the following principle is adopted: If a transaction's lock request is denied and thus the transaction is kept blocked for some time, abort and restart the transaction if blocking the transaction creates a cycle in the WFG (wait-for graph). As the

performance metrics, average response time and average lock waiting time are adopted. The response time of a transaction and lock waiting time are defined as follows.

Transaction response time = transaction commit time - transaction arrival time

Lock waiting time of a transaction = $\sum_{i=1}^{N_G}(L_i - I_i)$ where N_G is the number of granules accessed in a transaction, and I_i and L_i represent the initial lock requesting time to granule i and the lock granted time to granule i.

5.4. Analysis

In order to evaluate the performance of the proposed concurrency control technique, extensive simulation testing cases are performed. As in performance evaluation by mathematical modeling in Chapter 4, the proposed scheme and two existing concurrency control techniques are tested for each access type. Also, analysis is done based on the simulation results.

5.4.1. Conflict among methods

Three testing cases are chosen for conflict among methods: varying arrival rate, varying instance read to write ratio and varying class definition read to write ratio.

Figure 5-2. shows the testing case of varying arrival rate. The purpose of this testing case is to examine how those techniques work under various system load. Orion performs the worst. Malta's scheme works better than Orion. The proposed scheme works the best for the entire range. The average lock waiting times of Orion, Malta's scheme and the proposed work are 34.81 ms, 26.3 ms and 15.18 ms, respectively. Thus, the proposed

scheme incurs the least lock waiting time. The differences among each scheme are based on concurrency they provide. That is, Orion provides the worst concurrency while the proposed scheme provides the best concurrency. Orion adopts an entire instance object locking granularity so that the degree of concurrency is very limited. Malta's scheme and the proposed scheme adopt attribute locking granularity so that concurrency is enhanced. Especially, in the proposed work, further concurrency is achieved by adopting run-time information on access modes of attributes. Also, unlike Orion, in the Malta's scheme and the proposed scheme, locks are required for each instance method instead of atomic operation so that locking overhead is reduced. On the average, the proposed technique works better than Malta's scheme by 7.3% and better than Orion by 37%. On the other hand, Malta's work is better than Orion by 27.7%.

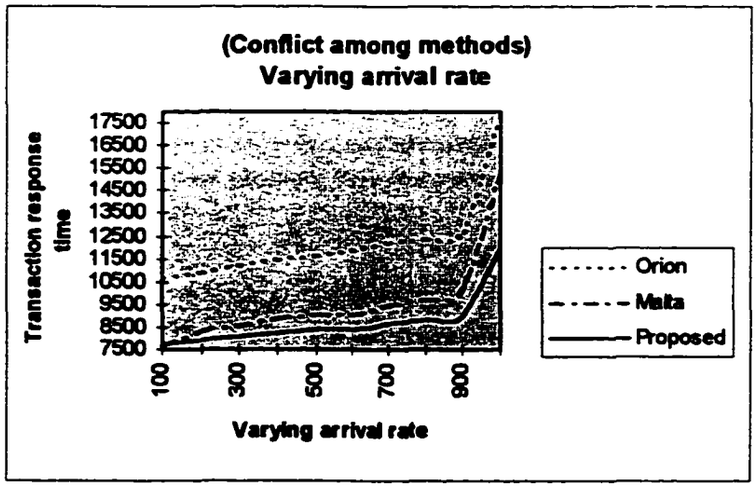


Figure 5-2. Varying arrival rate

Figure 5-3 shows the testing case of varying class definition read to write ratio. Orion performs the worst. Since Orion takes an entire class object as the lock granularity

for class definition access and also there is no concurrency between class definition read and class definition write, Orion results in the worst performance. Although the entire class object is taken as the lock granularity in Malta's scheme, still limited concurrency is provided in their work. That is, there is no conflict between an instance write and a class definition read method. On the other hand, in the proposed scheme, high concurrency is achieved by taking small granularity in class definition access. Thus, even though two class definition write methods conflict with each other if the entire class object is adopted for lock granularity, two methods may not conflict if the small granularity is taken. The average lock waiting time of Orion, Malta's scheme and the proposed scheme are 38.57 ms, 25 ms and 18.49 ms, respectively. On the average, the proposed scheme works better than Orion by 30.14% and better than Malta's work by 13.74%. Also, Malta's scheme works better than Orion by 14..42%.

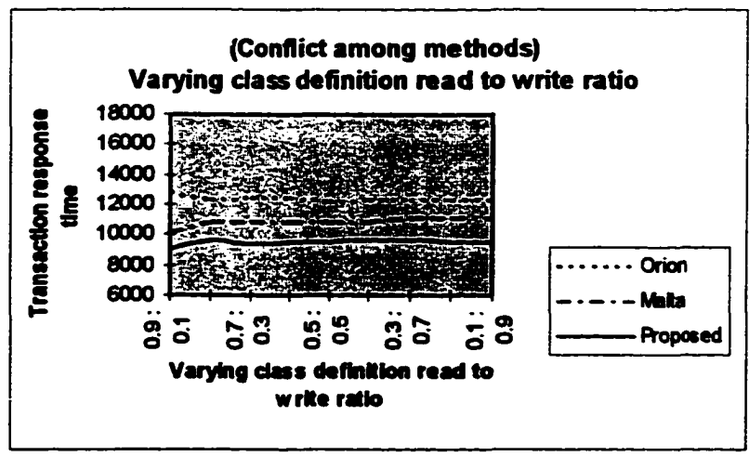


Figure 5-3. Varying class definition read to write ratio

Figure 5-4 shows the testing case of varying instance read to write ratio. Orion performs the worst. Especially, as the instance read to write increases, transaction

response time is increased dramatically. Since the lock granularity is an entire instance object for instance access in Orion, Orion does not provide any concurrency between an instance read method and an instance write method. Malta provides some concurrency among instance access methods but the concurrency is still limited since access modes of attributes are static. The proposed scheme works the best. This is mainly due to high concurrency among instance access methods resulted from dynamic information for attribute access is adopted. Also, in Malta's scheme and the proposed scheme, lock requests are based on methods while Orion is based on atomic operations in the method. This results in less overhead as well as less chance of deadlock due to lock escalation, in both Malta's scheme and the proposed scheme. On the average, the proposed scheme works better than Malta's scheme by 7.8% and better than Orion by 66.8%. Malta's scheme works better than Orion by 53.9%. Also, the average lock waiting time of Orion, Malta's scheme and the proposed work are 43.2 ms, 24.9 ms and 17.1 ms, respectively.

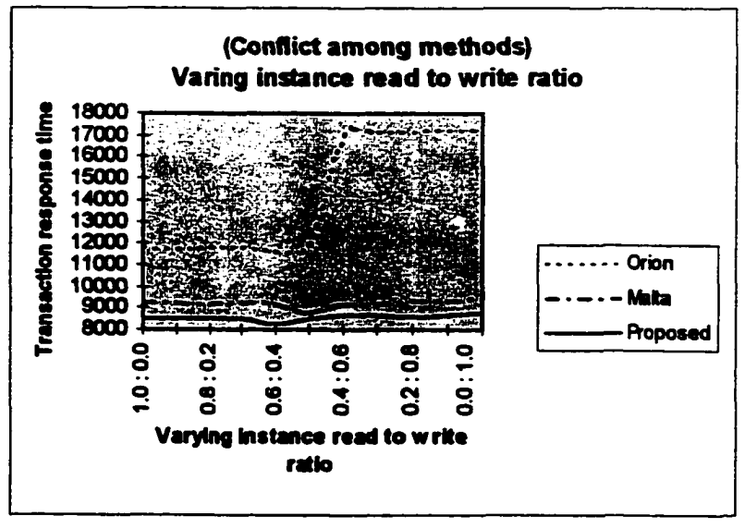


Figure 5-4. Varying instance read to write ratio

5.4.2. Class hierarchy locking

Three techniques have different class hierarchy locking schemes. In order to test their class hierarchy locking, two testing cases are performed as follows.

Figure 5-5 shows the testing case of varying arrival rate. Orion performs the worst. Orion adopts an implicit locking which requires intention locks for superclasses of a target class, for any kind of access. This results in lock overhead. Malta's scheme adopts explicit locking which does not require any intention locks. But, for class definition access and queries, it may incur much overhead than implicit locking. On the other hand, in the proposed scheme, lock overhead is less than both implicit and explicit locking, using frequency information of each class. On the average, the proposed scheme works better than Orion by 52.6% and better than Malta's scheme by 32.9%. Malta's scheme works better than Orion by 26.5%. Also, the average lock waiting time of Orion, Malta's work and the proposed scheme are 31.2 ms, 23.9 ms and 11.4 ms, respectively.

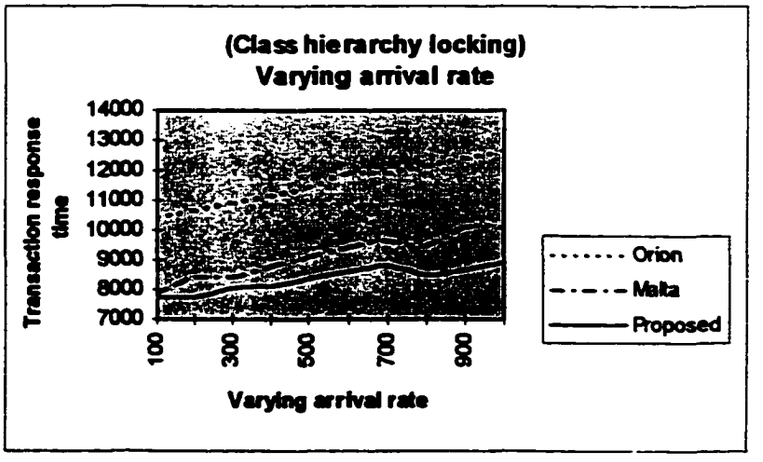


Figure 5-5. Varying arrival rate

Figure 5-6 shows the testing case of varying access to class hierarchy. In this testing case, transactions access from the root class to the leaf class in the class hierarchy. The purpose of this testing case is to measure the performance of class hierarchy locking technique used by each scheme as transactions access classes in the different levels of the class hierarchy.

As transactions access classes near the root in the class hierarchy, implicit locking has less locking overhead while explicit locking incurs much locking overhead for class definition writes and queries. On the other hand, if transactions access the leaf level in the class hierarchy, the implicit locking incurs higher locking overhead due to intention locks while explicit locking takes less overhead. No matter where transactions access to class hierarchy, the proposed scheme performs better than both works. As shown in Figure 5-6, there is not much difference as access to class hierarchy varies. This shows that locking overhead incurred by each scheme does not affect the performance significantly. On the average, the proposed scheme works better than Orion by 40.91% and better than Malta by 5.9%. Malta's scheme works better than Orion by 33%. Also, average lock waiting time of Orion, Malta's work and the proposed scheme are 29.5 ms, 19.9 ms and 14.6 ms, respectively.

5.4.2. Nested method invocations

In order to test three techniques for nested method invocation access, two testing cases, varying arrival rate and the ratio nested method invocation and non-nested method invocation, are performed as follows.

Figure 5-7 shows the testing case of varying arrival rate. In this case, transactions invoke only nested methods so that the performance is evaluated only for nested method invocations.

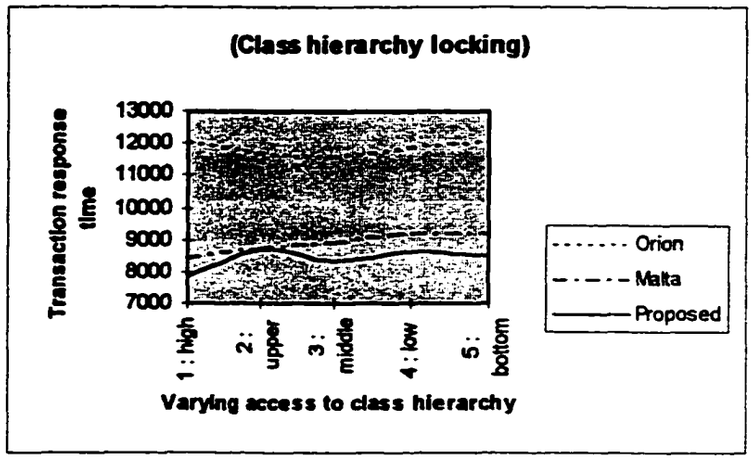


Figure 5-6. Varying access to class hierarchy

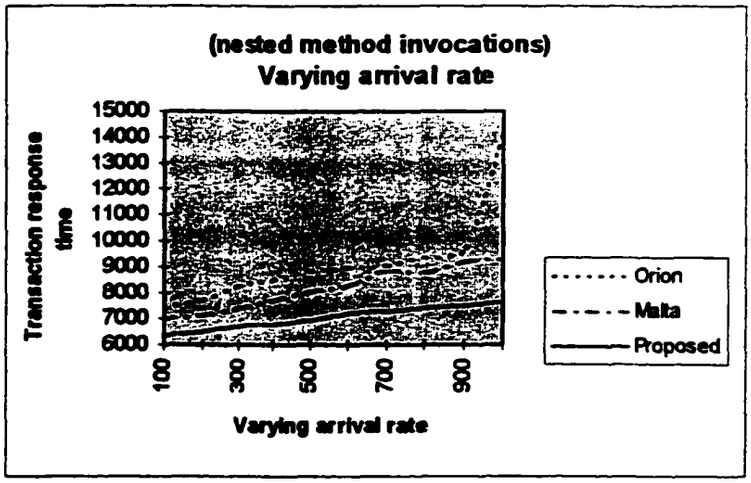


Figure 5-7. Varying arrival rate (Transaction response time)

The reason Orion performs the worst is as follows: Like in the conflict among method testing case, the concurrency provided by Orion is lower than those provided by Malta's scheme and the proposed scheme. Also, for nested method invocations, parent/children parallelism is not utilized in Orion and Malta's scheme. Malta's scheme provides higher concurrency than Orion but less than the proposed scheme. Also, parent/children parallelism is not considered. On the other hand, the proposed scheme provides the highest concurrency by using the small lock granularity and parallelism between parent and children method invocations using concurrent execution of parent and children methods. On the average, the proposed technique performs better than Orion by 30.8% and better than 16% by Malta's scheme. Malta's scheme works better than Orion by 12.8%. Also, the average lock waiting time of Orion, Malta's scheme and the proposed scheme are 16.2 ms, 18.9 ms and 13.1 ms, respectively.

Figure 5-8 shows the testing case of varying nested method invocation to non-nested method invocation ratio. In Figure 5-8, as nested method invocation method ratio is increased, the performance in Orion decreases. The reason is as in the testing case of varying arrival rate. Malta's scheme and the proposed scheme is relatively insensitive to the ratio of nested method invocation and non-nested method invocation. This is due to higher concurrency provided by both schemes. On the average, the proposed scheme works better than Orion by 28.4% and better than Malta's scheme by 11.2%. Malta's scheme works better than Orion by 15.5%. Also, the average lock waiting times of Orion, Malta's work and the proposed scheme are 17.8%, 23.4% and 14.2%, respectively.

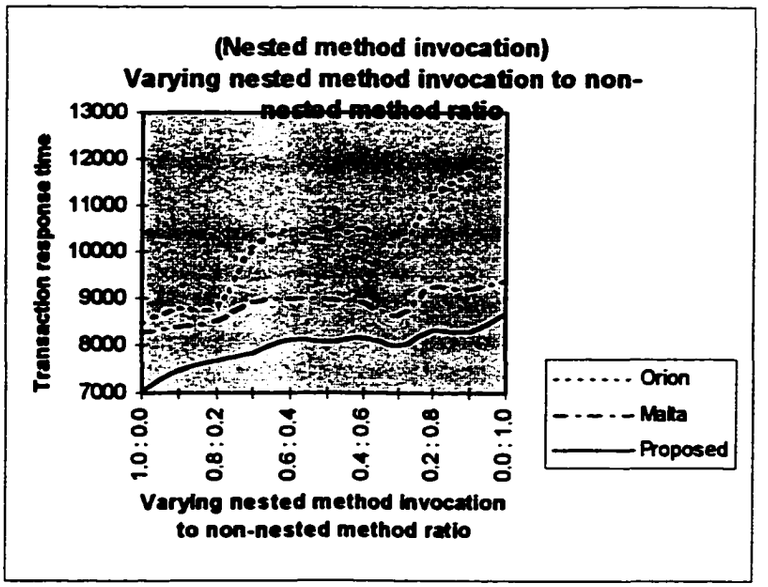


Figure 5-8. Varying nested method invocation to non-nested method invocation ratio

5.5. Conclusions

Through the simulation study for the performance evaluations of three schemes, the following conclusions are reached: for any testing case, the proposed scheme performs the best. Malta's work performs the second best. Finally, Orion performs the worst. Especially, the proposed scheme performs better than Orion by 28.4% to 66.75% and better than Malta's scheme by 5.9% to 32.9%. Malta's scheme performs better than Orion by 14.42% to 53.9%.

In the simulation study, the transaction response time is much larger than locking overhead. In other words, reducing locking overhead in concurrency control scheme in OODB does not affect the performance significantly. Thus, it is concluded that providing high degree of concurrency is the key factor in order to obtain the best performance in transaction response time.

Chapter 6

Conclusions and Future Research

6.1. Summary and Conclusions

Concurrency control is a mechanism used to coordinate accesses to the multi-user database so that the consistency of the database is maintained. OODBs have been adopted for non-standard applications requiring advanced modeling power to handle complex data and relationships among such data. Thus, concurrency control schemes in OODBs are more complicated than conventional databases. Also, transactions in OODBs usually requires long-duration running time. Thus, it is very important that concurrency control schemes not incur large overhead while increasing concurrency among users so that the performance should not be degraded.

In this research, three important issues of concurrency controls in OODBs are discussed: *conflict among methods*, *class hierarchy locking* and *nested method invocations*. The previous works for each issue were presented. Techniques were proposed to overcome the shortcomings of the previous works. Finally, an integrated concurrency control which includes all three issues was proposed.

The proposed technique was based on the following principles for each access type. For conflict among methods, finer locking granularity is adopted for both instance access and class definition access so that higher concurrency is achieved. Especially, for instance access, DAVs and breakpoints are adopted in order to provide higher concurrency using fine locking granularity and run-time information. Also, locks are

required for instance method invocations so that locking overhead is reduced and the possibility of deadlocks is also reduced. For class hierarchy locking, special classes are used in order to reduce locking overhead. The proposed class hierarchy scheme incurs fewer locks than both existing schemes. Finally, for nested method invocations, semantic information is used in order to provide higher concurrency among methods. Also, parent/children parallelism is adopted for better response time. In order to test the performance of the proposed technique, a mathematical model was constructed and extensive simulation experiments were conducted. Through the mathematical model and simulation, the performance evaluation of the proposed scheme and two existing works Orion and Malta's were conduced and results were analyzed.

As shown in the simulation results, there are clear differences among three techniques. The simulation results for each access type are as follows. For conflict among methods, the proposed scheme performs better than Orion by 37.03% and Malta's work by 7.3%. For class hierarchy locking, the proposed scheme performs better than Orion by 52.6 % and Malta's work by 32.9%. For nested method invocation, the proposed scheme performs better than Orion by 30.8% and Malta's scheme by 16%. In overall performance, the proposed scheme performs better than Orion by 37.03% and Malta's scheme by 7.3%.

Through extensive performance evaluations by mathematical modeling and simulation, the following conclusions are made: for conflicts among instance accesses, the proposed scheme utilized run-time information of attributes and adopted attribute level locking granularity so that it provides the better response time than both Orion and Malta. For conflicts among class definition accesses and conflicts among class definition access

and instance access, the proposed scheme provides the better response time by adopting small locking granularity instead of an entire instance object and an entire class object. For class hierarchy locking, the proposed scheme gives the better response time than both implicit locking and explicit locking, by utilizing access information of the classes. Finally, for nested method invocations, the proposed scheme utilized parent/children parallelism, run-time information of attributes and reduced locking overheads so that its response time is better than both existing works.

Based on the performance evaluations, the guideline for using concurrency control schemes in OODBs is as follows: for conflict among methods, the better response time can be achieved by utilizing run-time information of attributes and taking smaller lock granularity instead of an entire instance object and an entire class object. For class hierarchy locking, the locking overheads can be reduced by special classes which are based on access information on classes. Finally, for nested method invocations, utilizing parent/children parallelism and run-time information of attributes gives the better response time. Also, locks for method invocations instead of atomic operations give the better response time.

6.2. Directions for Future Research

The proposed scheme aims at centralized environments. For centralized OODBs, the proposed aims at stable OODB systems. But, if an OODB system whose schemas are continuously evolving, modifying DAVs and SCs may incur overheads. Thus, the future research is to deal with evolving OODB systems.

The proposed OODB concurrency control technique can be extended to distributed OODBs as follows: although considerable concurrency control schemes have been proposed for distributed databases ([Bern,1981], [Bern,1987], [Ozsu,1991], [Levy,1994]) only a few techniques have been proposed for a distributed OODB ([Daya,1994], [Naka, 1994]). But, the following issues are not discussed in the previous research. First, replication in distributed OODB makes a concurrency control scheme complicated. Consider the following partial replicated OODB such that a class D with two subclasses, say A and C, is stored in site 1 and a class D is alone stored in site 2. In this case, locking on D in site 2 requires locks on A and C on site 1 if explicit locking is used. Thus, how to manage replicated objects with less locking overhead is of concerns. Second, in a distributed OODB with composite objects, composite objects may reside in different sites. Thus, locking on a composite object requires locking on all subcomposite objects in different sites, resulting in much overhead. The concern is to provide an efficient locking on distributed composite objects. Third, it is necessary to consider how to distribute lock managers effectively to reduce communication cost.

References

[Ande,1990] T. L. Anderson, A. J. Berre, M. Mallison, H. H. Porter III, B. Scheider, "The HyperModel Benchmark", Proc. of the 2nd Int. Conf. on Extending Database Technology, Lecture Notes on Computer Science 416, Springer-Verlag, Berlin, pp. 317 - 331.

[Agra,1992] D. Agrawal and A. E. Abbadi, "A Non-restrictive Concurrency Control for Object-Oriented Databases", *3rd Int. Conf. on Extending Data Base Technology*, Vienna, Austria, Mar. 1992, pp. 469 - 482.

[Badr,1988] B. R. Badrinath and K. Ramamritham, "Synchronizing Transactions on Objects", *IEEE Transactions on Computers*, *37*(5),1988, pp. 541 - 547.

[Badr,1992] B. R. Badrinath and K. Ramamritham, "Semantic-Based Concurrency Control : Beyond Commutativity", *ACM Transactions of Database Systems*, *17*(1), 1992, pp. 163 - 199.

[Bern,1981] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, *13* (2), 1981, pp. 185 - 221.

[Bern,1987] P.A. Berstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Berr,1991]. A. J. Berre and T. L. Anderson, "The HyperModel Benchmark for evaluating object-oriented databases, In Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, edited by R. Gupta and E. Horowitz, Englewood Cliffs, Jew Jersey, Prentice-Hall, pp. 75 - 91.

[Bili,1992] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish and K. Ramamritham, "A Flexible Transaction Facility for an Object-Oriented Database", Tech. Report, AT&T Bell Labs, 1992.

[Care,1987] M. J. Carey, "Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions", IEEE Trans. on Software Engineering, Vol. 13, No. 6, Jun. 1987, pp. 746 - 751.

[Care,1993] M. J. Carey, D. J. Dewitt and J. F. Naughton, "The 007 Benchmark", Proc. of the 1993 ACM SIGMOD Conference on Management of Data, Washington D.C., May, 1993, pp. 12 - 21.

[Care,1994] M. J. Carey, D. J. Dewitt, C. Kant and F. Naughton, "A Status Report on the 007 OODBMS Benchmarking Effort", Proc. of OOPSLA, Portland, Oregon, 1994, pp. 414 - 426.

[Care,1994-1] M. J. Carey, D. J. Dewitt, C. Kant and F. Naughton, "A Status Report on the 007 OODBMS Benchmarking Effort", Proc. of OOPSLA, Portland, Oregon, 1994, pp. 414 - 426.

[Care,1994-2] M. J. Carey, D. J. Dewitt and J. F. Naughton, "The 007 Benchmark", Tech. Report, Dept. of Computer Science, Univ. of Wisconsin, 1994.

[Cart,1990] M. Cart and J. Ferrie, "Integrating concurrency control into an object-oriented database system", *2nd Int. Conf. on Extending Data Base Technology*, Venice, Italy, Mar. 1990, pp. 363 - 377.

[Catt,1992] R. Cattell and J. Skeen, "Object Operations Benchmark", ACM Transactions on Database Systems, Vol. 17, No. 1, Mar. 1992, pp. 1 - 31.

[Chry,1991] P. K. Chrysanthis, S. Raughuram, and K. Ramamritham, "Extracting Concurrency from Objects : A Methodology", *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, 1991, pp. 108 - 117.

[Date,1985] C. J. Date, *An Introduction to Database Systems*, Vol. II, Addison-Wesley, 1985.

[Daya,1994] U. Dayal, "An Activity/Transaction Model for a Distributed Multi-Service System" in *Distributed Object Management* edited by M. T. Ozsu, U. Dayal and P. Valduriez, Morgan-Kaufmann, 1994.

[DEC,1993] DECdirect Workshop Solutions Catalog, winter 1993.

[Denc,1994] D. Dench and B. Prior, *Introduction to C++*, Chapman& Hall, 1994.

[Eswa,1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system", *Communication of ACM, 19*(11), 1976, pp. 624 - 633.

[Freu, 1987] J. E. Freund and R. E. Walpole, Mathematical Statistics, 4th Edition, Prentice-Hall, Englewood Cliff, NJ, 1987.

[Garz,1988]. J. F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System", *ACM SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, Jun. 1988, pp. 37 - 45.

[Goya,1993] P. Goyal, T. S. Narayanan and F. Sadri, "Concurrency Control for Object Bases", Journal of Information Systems, Vol. 18, No. 3, 1993, pp. 167 - 180.

[Guer,1993] R. Guerraoui, "Toward Modular Concurrency Control for Object-Oriented Distributed Systems", Proc. of the 4th Workshop on Future Trends of Distributed Computing System, Lisbon, Portugal, 1993, pp. 240 - 246.

[Guer,1995] M. Guerni, J. Ferrie, J. Pons, "Concurrency and Recovery for Typed Objects using a New Commutativity Relation", Lecture Notes in Computer Science 1013, 4th Int. Conf. on Deductive and Object-Oriented Databases, Singapore, 1995, pp. 411 - 428.

[Hadz,1991] T. Hadzilacos and V. Hadzilacos, "Transaction Synchronization in Object Bases", Journal of Computer and System Sciences, Vol. 43, No. 1, pp. 2 - 24.

[Hali,1989] U. Halici and A. Dogac, "Concurrency Control in Distributed Databases Through Time Intervals and Short-Terim Locks" IEEE Trans. on Software Engineering, Vol. 15, No. 8, Aug. 1989, pp. 994 - 1003.

[Hali,1991] U. Halici and A. Dogac, "An Optimistic Locking Technique For Concurrency Control in Distributed Databases", IEEE Trans. on Software Engineering, Vol. 17, No. 7, Jul. 1991, pp. 712 - 724.

[Herr,1990] U. Herrmann, P. Dadam, K. Kuspert, E. A. Roman and G. Schlageter, "A Lock Technique for Disjoint and Non-Disjoint Complex Objects", *2nd Int. Conf. on Extending Data Base Technology*, Venice, Italy, Mar. 1990, pp. 219 - 237.

[Huan,1995] J. Huang, "Recovery Techniques in Real-Time Main Memory Databases", Ph.D. Dissertation, Univ. of Oklahoma, Dept. of Computer Science, 1995.

[Hung,1992] S. L. Hung and K. Y. Lam, "Locking protocols for Concurrency Control in Real-Time Database Systems", *ACM SIGMOD RECORD*, 21(4), 1992, pp. 22 - 27.

[Jaga,1993] H. V. Jagadish and Daniel F. Lieuwen, "Multi-Granularity Locks in an Object-Oriented Database", AT&T Tech. Report, 1993.

[Jose,1991] J. V. Joseph, S. M. Thatte, C. W. Thompson, D. L. Wells, "Objected-Oriented Database: Design and Implementation", Proc. of IEEE. Vol. 79, No. 1, Jan. 1991, pp. 42 - 64.

[Jun,1995-1] W. Jun and L. Gruenwald, "Supporting Fine Concurrency in Object-Oriented Databases", 1995 Arkansas Computer Conferences, Mar. 1995, pp. 28-29.

[Jun,1995-2] W. Jun and L. Gruenwald, "Automation of Fine Concurrency in Object-Oriented Database", Int. Conf. on Computer Applications on Engineering and Medicine, Dec. 1995, pp. 72 - 75.

[Jun,1995-3] W. Jun and L. Gruenwald, "An Integrated Concurrency Control in Object-Oriented Databases", Mid-America Symposium on Emerging Computer Technologies, Sep. 1995, pp. 51-58.

[Jun,1996] W. Jun and L. Gruenwald, "A Flexible Class Hierarchy Concurrency Control Technique in Object-Oriented Database Systems", Int. Conf. on Computers and Their Applications, San Francisco, CA, Mar. 1996, pp. 191 - 196.

[Jun,1997-1] W. Jun and L. Gruenwald, "A Class Hierarchy Concurrency Control Technique in Object-Oriented Database Systems", The Third Int. Conf. on Computer Science and Informatics, Mar. Raleigh, NC, pp. 293 - 296.

[Jun,1997-2] W. Jun and L. Gruenwald, "An Efficient Class Hierarchy Concurrency Control Technique in Object-Oriented Database Systems", Journal of Information and Software Technology, Submitted in Nov. 1997.

[Jun,1997-3] W. Jun and L. Gruenwald, "Semantic-Based Concurrency Control in Object-Oriented Databases", Journal of Object-Oriented Programming, accepted, appear in Jan. 1998 issue.

[Kim,1990] W. Kim, *Introduction to Object-Oriented Databases*, The MIT Press, 1990

[Kim,1991] W. Kim, T. M. Chan and J. Srivastava, "Processor Scheduling and Concurrency Control in Real-Time Main Memory Databases", IEEE Symposium on Applied Computing, Kansas City, Missouri, USA, Apr., 1991, pp. 12 - 21.

[Kort,1983] H. F. Korth, "Locking Primitives in a Database System", *Journal of ACM*, *30*(1), 1983, pp. 55 - 79.

[Kort,1991] H. F. Korth and A. Silberschartz, *Database System Concepts*, 2nd Edition, McGraw Hill, Singapore, 1991.

[Kung,1981] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Trans. on Database Systems*, *6*(2), 1981, pp. 213 - 226.

[Kwon,1997] K. Kwon and S. Moon, "Semantic Multigranularity Locking for Object-Oriented Database Management Systems", Journal of Database Management Systems, Vol. 8, No. 2, 1997, pp. 23 - 33.

[Lazo,1984] E. D. Lazowska, *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*, Prentice-Hall, New Jersey, 1984.

[Lee,1996] S. Lee and R. Liou, "A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems", IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 1, Feb. 1996, pp. 144 - 156.

[Levy,1994] E. Levy, H. F. Korth and A. Silberschartz, "An Optimistic Commit Protocol for Distributed Transaction Management", Proc. of ACM SIGMOD Int. Conf. on Management of Data, Denver, Colorado, May 1991, pp. 88 - 97.

[Liou,1991] R. L. Liou, *A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems*, Master Thesis, Dept. of Computer Science and Information Engineering, National Chiao Tung University, Jun. 1991.

[Malt,1991] C. Malta and J. Martinez, "Controlling Concurrent Accesses in an Object-Oriented Environment", *2nd Int. Symp. on Database Systems for Advanced Applications*, Tokyo, Japan, Apr. 1991, pp. 192 - 200

[Malt,1992] C. Malta and J. Martinez, "Limits of commutativity on abstract data types", *3rd Int. Conf. on Information Systems and Management of Data*, Bangalore, India, Jul. 1992, pp. 261 - 270.

[Malt,1993] C. Malta and J. Martinez, "Automating Fine Concurrency Control in Object-Oriented Databases", *9th IEEE Conf. on Data Engineering*, Vienna, Austria, Apr. 1993, pp. 253- 260.

[Moss,1985] J. E. B. Moss, *Nested Transactions : An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, 1985.

[Muth,1993] P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse, "Semantic Concurrency Control in Object-Oriented Database Systems", *Proc. of the 9th IEEE Int. Conf. on Data Engineering*, Apr. 1993, pp. 233 - 242.

[Naka,1994] T. Nakajima, "Commutativity Based Concurrency Control and Recovery for Multiversion Objects", in *Distributed Object Management* edited by M. T. Ozsu, U. Dayal and P. Valduriez, Morgan-Kaufmann, 1994.

[Olse,1995] D. H. Olsen and S. Ram, "Towards a Comprehensive Concurrency Control Mechanism for Object-Oriented Databases", Journal of Database Management, Vol. 6, No. 4, 1995, pp. 24 - 35.

[Ozsu,1991] M. T. Ozsu and Patrick Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1991.

[Ozsu,1994] M. T. Ozsu, "Transaction Models and Transaction Management in Object-Oriented Database Management Systems", in *Advanced in Object-Oriented Database Systems* edited by A. Dogac, M. T. Ozsu, A. Biliris and T. Sellis, Springer-Verlag, 1994, pp. 147 - 184.

[Prit,1986] A. Alan B. Pritsker, "Introduction to Simulation and SLAM II", Systems Publishing Corporation, 1986.

[Rese,1994] R. F. Resende, D. Agrawal, and A. E. Abbadi, "Semantic Locking in Object-Oriented Database Systems", *Proc.of OOPSLA 94*, Portland, OR, USA, Oct. 1994, pp. 388 - 402.

[SCSI,1993] 22000 Series - SCSI Micropolis Disk Drive Information, 1993.

[Serv,1990] Servio Logic Corp : "Chap. 16 : Transactions and Concurrency Control", in Gemstone Product Overview, Alameda, CA., 1990.

[Shar,1996] P. Shah and J. Wong, "Concurrency Control in an Object-Oriented Data Base System", Journal of Systems and Software, Vol. 35, No. 3, Dec. 1996, pp. 169 - 183.

[Sing,1985] M. Singha, P. D. Nanadikar and S. L. Mebndirratta, "Timestamp Based Certification Schemes for Transaction in Distributed Database System", *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1985, pp. 402 - 411.

[Ulos,1992] O. Ulosoy, "Concurrency Control in Real-Time Database Systems", Ph. D. Dissertation, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, 1992.

[Wang,1990] S. Wang, "Improvement of Concurrency Control within Object-Oriented Database Systems", ACM Symposium on Applied Computing, Apr. 1990, pp. 68 - 70.

[Weih,1988] W. E. Weihl, "Commutativity-Based Concurrency Control for Abstract Data Types", *IEEE Trans. on Computers*, *37*(12), 1988, pp. 1488 - 1505.

[Weih,1989] W. E. Weihl, "The impact of Recovery on Concurrency Control", Proc. of the ACM Symposium on Principles of Database Systems, Philadelphia, Mar. 1989, pp. 259 - 269.

[Wolf,1993] V. F. Wolfe and Lisa B. Cingiser, "Object-Based Semantic Real-Time Concurrency Control", *Proc. of 1993 IEEE Real-Time Systems Symposium*, 1993.

[Yu,1993] P. S. Yu, D. M. Dias and S. Lavenberg, "On the Analytical Modeling of Database Concurrency Control", Journal of ACM, Vol. 40, No. 4, Sep. 1993, pp. 831 - 872.

Appendix

I. Algorithm for three techniques for conflict among methods

I.1. Orion

• Whenever a transaction requests a lock on class or instance, do the following steps.

(1) - Read the lock requester's mode and transaction_id
(2) - Find the lock entry for the class
(3) - Find the lock holder on the class
    - while there is different transaction holds a lock do
(4) - Find the commutativity table
(5) - Read the holder's mode
     - Check the commutativity table for commutativity of the lock holder's mode and the lock requester's mode
     - If no commute, go to step 12)
    end while
(6) - If the requester is a CDA transaction, then go to step 12)
(7) - Find lock entry for an instance
    - while there is a different transaction holding lock on instance do
(9)- Find a lock holder on instance
    - while there is different transaction holds a lock do
(10) - Read the holder's mode
      - Check the commutativity table for commutativity of the lock holder's mode and the lock requester's mode
      - If no commute, the go to step 12)
    end while
(11) Set a lock in the lock table and stop
(12) Block the requester and stop

•If transaction is committed or aborted, release all the locks . For each locks held, do as follows.

(1) Find the lock entry for the class
(2) -Find the lock holder on the class
    - while there is the same transaction holding lock on the class do
(3) - Release the lock on the class
(4) - If the committed transaction is a CDA transaction, then stop
(5) - Find the lock entry for the instance
(6) - Find a lock holder on the instance
     -while the same transaction holds a lock do
(7) - Release the lock on the instance, and stop.

end while

end while

I.2. Malta

• Whenever a transaction requests a lock on class or instance, do the following steps.

(1) - Read the lock requester's method name and transaction_id
(2) - Find the lock entry for the class
(3) - Find the lock holder on the class
- while there is a different transaction holding lock do
(4) - Find the commutativity table
(5) - Read the holder's mode
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, then go to step 12)
end while
(6) - If the requester is a CDA transaction, then go to step 11)
(7) - Find the lock entry for the instance
(8) - Find the lock holder on the instance
- while there is a different transaction holding lock do
(9) - Find the method commutativity table for instance
(10) - Read the holder's method
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 12)
- end while
(11) Set a lock in the lock table and stop
(12) Block the requester and stop

• If transaction is committed or aborted, release all the locks

(1) Find the lock entry for the class
(2) -Find the lock holder on the class
- while there is the same transaction holding lock on the class do
(3) - Release the lock on the class
(4) - If the committed transaction is a CDA transaction, then stop
(5) - Find the lock entry for the instance
(6) - Find the lock holder on the instance
-while the same transaction holds a lock do
(7) - Release the lock on instance, and stop.

end while

end while

I.3. The proposed scheme

• Initial lock request
Whenever a transaction requests a lock on some granule (class or instance), do the following steps.

(1) - Read the lock requester's method name and transaction_id
(2) - Find the lock entry for the class
(3) - Find the lock holder on the class
- while there is a different transaction holding a lock do
(4) - Find the commutativity table for class
(5) -Read the holder's method name
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, then go to step 12)
- end while
(6) - If the requester is a CDA transaction, then go to step 11)
(7) - Find the lock entry for the instance
(8) - Find the lock holder on the instance
- while there is a different transaction holding locks do
(9) - Find the method commutativity table for the instance
(10) - Read the holder's mode
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 12)
end while
(11) Set a lock in the lock table
(12) Block the request

• During method execution (if the method invoked is an instance method)
Record all the breakpoints encountered during method execution

• After method execution (if the method invoked is an instance method)
Change method level lock to breakpoint level using all the breakpoints recorded at run-time

(1) - Find the lock entry for the class
(2) - Find the lock holder for the class
- while the same transaction holding lock do
(3) Change method level lock to breakpoint level lock
(4) Find the lock entry for the instance
(5) Find the lock holder for on the instance

- while the same transaction holding lock do
(6) Change method level lock to breakpoint level lock
end while
end while

• If transaction is committed or aborted, release all the locks

(1) - Find the lock entry for the class
(2) - Find the lock holder on the class
- while there is the same transaction holding lock on the class do
(3) - Release the lock on class
(4) - If the committed transaction is a CDA transaction, then stop
(5) - Find the lock entry for the instance
(6) - Find the lock holder on the instance
-while the same transaction holds a lock do
(7) - Release the lock on instance, and stop.
end while
end while

II. How to calculate t_{lock} and t_{commit} ?

II.1 Orion

step 1)
=> 2 MM read //we assume that transaction_id, lock mode occupy one word, respectively.//

step 2)
Assume that transaction_id, lock mode and pointer takes one word respectively.

- get address of lock table => 1 MM read
- get address of lock entry for class =>
(address. of lock table) +(class-id - 1)*S_Lock // S_Lock is size of lock entry for each class where S_Lock = B*2+4. Assume that this number is stored in S_Lock

load reg1, address of lock table
load reg2, T-cid (target class id)
SUB reg2, 1
load reg3, S_Lock => 1 MM read
MUL reg2, reg3
ADD reg1, reg2

Total : 2 MM read and 6 operations

step 3)

=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{CLASS} times in a class
$NUM\_TRANS_{CLASS} \cong M_P * N_G / N_C$ where M_P is the multiprogramming level

Total: NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 4)
=> 1 MM read

step 5) - Read the holder's mode
=> 1 MM read; load reg3, lockmode

- Check commutativity table for commutativity of the lock holder's mode and the lock requester's mode
  - sequential search for row address of conflict table
  => average 3 MM read; compare (load reg1, lock mode; comp reg1, reg3) for each read
  - sequential search for column address of conflict table
  => average 3 MM read; compare (load reg2, lockmode; comp reg2, reg4) for each read
  - check commutativity
  => 1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp (reg1, 'y')

For each lock holder, 8 MM reads + 15 operations

Total: NUM_TRANS_{CLASS}*(8 MM reads + 15 operations)

step 6)
=> 1 MM read // Reading transaction type

step 7)
-get bucket address with prob. P_{IA}
load reg1, T-iid (target instance id)
load reg1, reg2
load reg3, B (number of buckets) => 1 MM read
DIV reg2, reg3
MUL reg2, reg3
SUB reg1, reg2
MUL reg1, 2

SUB reg1, 2
ADD reg1, 4
=> P_{IA}*(1 MM read + 9 operations)

- Find target instance
1 MM read (for x or pointer)
1 MM read (for T-iid)
Compare (check if that is target instance) => load reg1, iid; comp reg1, T-iid (target instance)

Assume that above steps are repeated NUM_INST times with prob. P_{IA}
NUM_INST can be approximated as follows.
NUM_INST $\cong M_P*N_G/(2*B*N_C)$ where M_P is the number of transactions in the system, $M_P \cong (t_{int} + N_G*x)*\lambda$, by applying Little's formula [Freu, 1987]

Total: P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations

step 8)
=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong M_P*N_G / (N_C*N_t)$

Total: P_{IA}*NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 9) - Read the holder's mode
=> 1 MM read; load reg3, lockmode
- Check commutativity table for commutativity of the lock holder's mode and the lock requester's mode
- sequential search for row address of conflict table
=> average 1.5 MM read; compare (load reg1, lock mode; comp reg1, reg3) for each read
- sequential search for column address of conflict table
=> average 1.5 MM read; compare (load reg2, lockmode; comp reg2, reg4) for each read
- check the commutativity
=> 1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp (reg1, 'y')

For each lock holder, 5 MM reads + 9 operations

Total: P_{IA}* NUM_TRANS_{INST}*(5 MM reads + 9 operations)

step 10)

If there are some transactions holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <- nil)
1 MM write (change Y)
1 MM write (change Nil to newly inserted transaction)
=> 6 MM access

If there is no transaction holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <-nil)
2 MM writes (change Y and X)
=> 6 MM access

If the lock requester is IA transaction, perform the same step as above.
=> P_{IA}*6 MM access

Total: (6+6*P_{IA}) MM access

step 11)
=> same as step 10)

Total: (6 + 6*P_{IA}) MM reads

Thus, t_{lock} (Orion) = 2 MM reads (step 1)
+ 2 MM reads + 6 operations (step 2)
+ NUM_TRANS_{CLASS}*(2 MM reads + 2 operations)
+ 1 MM read (step 4)
+ NUM_TRANS_{CLASS}*(8 MM reads + 15 operations) (step 5)
+ 1 MM read (step 6)
+ P_{IA}*[1+NUM_INST*2) MM read + (9+NUM_INST*2) operations] (step 7)
+ P_{IA}*NUM_TRANS_{INST}*(2 MM read + 2 operations) (step 8)
+ P_{IA}*NUM_TRANS_{INST}*(5 MM reads + 9 operations) (step 9)
+ (6+6*P_{IA}) MM reads (step 10 or 11)

=[12+10*NUM_TRANS_{CLASS}+P_{IA}*[7+2*NUM_INST + 7*NUM_TRANS_{INST}]] *MM_ACCESS
+ [6+17*NUM_TRANS_{CLASS}+ P_{IA}*[9+2*NUM_INST + 11*NUM_TRANS_{INST}]]*BASIC_OP
where NUM_INST $\cong$ M_P*N_G/(2*B*N_C), NUM_TRANS_{CLASS} $\cong$ M_P*N_G/N_C, NUM_TRANS_{INST} $\cong$ M_P*N_G/(N_c*N_I)

$t_{lock, IA}$ (Orion) = t_{lock}(Orion) where P_{IA} = 1

$t_{lock, CDA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 0$

How to find t_{commit} (Orion)?

step 1)
=> same as step 1) in t_{lock} : 2 MM reads + 6 operations

step 2)
=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{CLASS} times in a class.
$NUM\_TRANS_{CLASS} \cong M_P * N_G / (2 * N_C)$ where M_P is the multiprogramming level
Assume that only half of NUM_TRANS_{CLASS} transactions is searched.

Total: NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 3)
1 MM read (read next record of committed transaction)
1 MM read (read previous record of committed transaction)
1 MM write (the pointer of previous record indicates the next record of committed transaction)
=> 3 MM access

step 4)
=> 1 MM read // reading transaction type

step 5)
=> same as step 7) in t_{lock}
=> P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations

step 6)
=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong M_P * N_G / (2 * N_C * N_I)$

Assume that only half of NUM_TRANS_{INST} transactions is searched

Total: P_{IA}*NUM_TRANS$_{INST}$*(2 MM read + 2 operations)

step 7)
=> P_{IA}*step 3)
=> 3*P_{IA} MM reads
Total:[6+2*NUM_TRANS$_{CLASS}$+2*P_{IA}*NUM _TRANS$_{INST}$+P_{IA}*[4+NUM_INST*2]] MM reads
[6+2*NUM_TRANS$_{CLASS}$+ P_{IA}*2*NUM_TRANS$_{INST}$ + P_{IA}*[9+NUM_INST*2]] operations

Since each transaction accesses N_G granules and each instance access method requests AT locks per instance, step 6) and 7) need to be performed additional (AT-1) times for instance access method.

t_{commit} (Orion)=
N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2*NUM_TRANS$_{INST}$+P_{IA}*[4+NUM_INST*2] +
[AT-1]*P_{IA}*[NUM_TRANS$_{INST}$*2+3]]*MM_ACCESS
+N_G*[6+2*NUM_TRANS$_{CLASS}$+P_{IA}*2* NUM_TRANS$_{INST}$+ P_{IA}*[9+NUM_INST*2]+
[AT-1]*P_{IA}*[NUM_TRANS$_{INST}$*2]]*BASIC_OP
where NUM_INST $\cong$ M_P*N_G/(2*B*N_C), NUM_TRANS$_{CLASS}$ $\cong$ M_P*N_G/(2*N_C),
NUM_TRANS$_{INST}$ $\cong$ M_P*N_G/(2*N_c*N_I)

c_I(Orion) = t_{commit} (Orion) where P_{IA} = 1
c_C(Orion) = t_{commit} (Orion) where P_{IA} = 0

II.2 Malta

step 1)
=> 2 MM read // Assume that transaction_id, lock mode (or method name) occupy one
// word, respectively

step 2)
=>The lock table has basically same structure. But, in Malta's work, the lock is requested by the unit of method, if the requester is an instance method, so that the lock mode is replaced by the method name.
=> 2 MM read and 6 operations

step 3)
=> same as step 3) in Orion
=> NUM_TRANS$_{CLASS}$*(2 MM read + 2 operations)

step 4)
=> 1 MM read

step 5) - Read the holder's mode
=> 1 MM read; load reg3, lockmode
- Check commutativity table for commutativity of the lock holder's mode and the lock requester's mode

- sequential search for row address of conflict table
  => average 2 MM read; compare (load reg1, lock mode; comp reg1, reg3)
- sequential search for column address of conflict table
  => average 2 MM read; compare (load reg2, lockmode; comp reg2, reg4)
- If no commute => step 10)
- 1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp (reg1, 'y')

Thus, one conflict checking requires: NUM_TRANS_{CLASS}* (6 MM reads + 11 operations)

step 6)
=> 1 MM read // Reading transaction type

step 7)
=> same as step 7) in Orion
=> P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations

step 8)
=> P_{IA}*NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 9)
=> P_{IA} MM read

step 10)- Read the holder's mode
=>1 MM read; load reg3, lockmode
- Check the commutativity table for the commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 12)

• In Malta's work, the lock is requested for each instance access method. In real OODBs, there are many methods defined in each class so that searching a particular method in a commutativity table takes so much overhead. Thus, the following strategy is adopted: assign each method to the unique number so that the method can be searched directly in a commutativity table. There are two steps to do it.

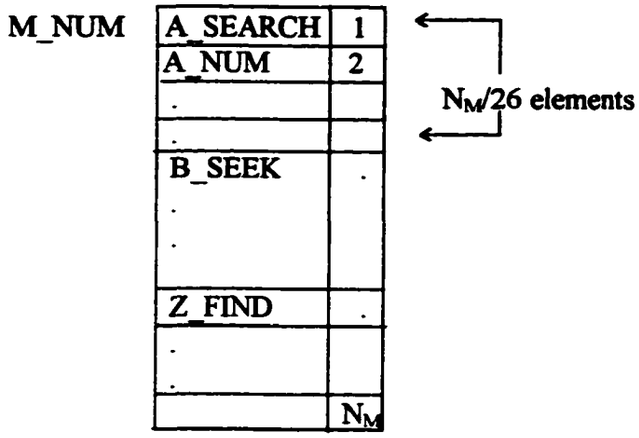
• substep 1) The hash function is adopted to map each instance method to integer. Assume that each method is named based on alphabet. Whenever a method is invoked, read the first character of the method. Based on order of this character in alphabet, the character is mapped into number.

h(first character of each method) = I where $1 \leq I \leq 26$

For example, for method ASCEND_ORDER and DESCEND_ORDER has the following hash function.

h(ASCEND_ORDER) = 1
h(DESCEND_ORDER) = 4

• substep 2) Then, assume that array M_NUM stores the unique number for each method. The array is has total N_M (number of methods) elements, and divided into 26 sectors. Thus, each sector has $N_M/26$ elements.



- calculate hash function => load reg1, first_Char; 1 MM read for reading first character
  DIV reg1, 26
- calculate the first entry of address in array M_NUM
  load reg2, 0
  load reg3, N_M => 1 MM read
  DIV reg3, 26
  MUL reg1, reg3
- search the method name
average $N_M/(26*2)$ MM read
compare; load reg1, method_name; comp reg1, T_method // Assume that T_method is to
  // be searched
- read the integer number => 1 MM read
Thus, total $(N_M/(26*2)+3)$ MM read + $(N_M/(26*2)*2+6)$ operations

- read the method name of the requester in the commutativity table => same as read method name of the holder

- read the commutativity relation
1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp (reg1, 'y')

Total: P_{IA}*NUM_TRANS_{INST}*[$(N_M/26+8)$ MM reads + $(N_M/26*2+15)$] operations

step 11)

=> same as step 10) in Orion
=> (6+6*P_{IA}) MM reads

step 12)
=> same as step 11) in Orion
=> (6+6*P_{IA}) MM access

t_{lock} (Malta) = 2 MM read (step 1)
+ 2 MM read and 6 operations (step 2)
+ NUM_TRANS_{CLASS}*(2 MM read + 2 operations) (step 3)
+ 1 MM read (step 4)
+ NUM_TRANS_{CLASS}* (6 MM reads + 11 operations) (step 5)
+ 1 MM read (step 6)
+ P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations (step7)
+ P_{IA}*NUM_TRANS_{INST}*(2 MM read + 2 operations) (step 8)
+ P_{IA} MM read (step 9)
+ P_{IA}* NUM_TRANS_{INST} *[(N_M/26+8) MM reads + P_{IA}*(N_M/26*2+15)] operations (step 10)
+ (6 + 6*P_{IA}) MM access (step 11 or 12)

=
[12+8*NUM_TRANS_{CLASS}+P_{IA}*[8+2*NUM_INST+NUM_TRANS_{INST}*[10+N_M/26]]*MM_ACCESS+[6+13*NUM_TRANS_{CLASS}+P_{IA}*[9+2*NUM_INST+NUM_TRANS_{INST}* [17+N_M/26*2]]*BASIC_OP
where NUM_INST ≅ M_P*N_G/(2*B*N_C), NUM_TRANS_{CLASS} ≅ M_P*N_G/N_C, NUM_TRANS_{INST} ≅ M_P*N_G/(N_e*N_I)

$t_{lock, IA}$ (Malta) = t_{lock} (Malta) where P_{IA} = 1
$t_{lock, CDA}$ (Malta) = t_{lock} (Malta) where P_{IA} = 0

How to calculate t_{commit} (Malta)?

=> same as Orion, but step 6) and 7) are not repeated (AT-1) times
Thus, t_{commit} (Malta)=
= N_G*[6+2*NUM_TRANS_{CLASS}+2*P_{IA}*NUM_TRANS_{INST}+P_{IA}*[4+NUM_INST*2]]* MM_ACCESS
N_G*[6+2*NUM_TRANS_{CLASS}+2*P_{IA}*NUM _$TRANS_{INST}$+P_{IA}*[9+NUM_INST*2]]* BASIC_OP where NUM_INST ≅ M_P*N_G/(2*B*N_C), NUM_TRANS_{CLASS} ≅ M_P*N_G/(2*N_C), NUM_TRANS_{INST} ≅ M_P*N_G/(2*N_e*N_I)

c_I(Malta) = t_{commit} (Malta) where P_{IA} = 1
c_C(Malta) = t_{commit} (Malta) where P_{IA} = 0

II.3. The proposed scheme

a) initial lock request

step 1)
=> same as Malta
=> 2 MM reads

step 2)
=> same as in Malta
=> 2 MM read and 6 operations

step 3)
=> NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 4)
=> 1 MM read

step 5) -Read the holder's mode
- read method name of the holder => 1 MM read; load reg3, lockmode
- Check commutativity table for commutativity of the lock holder's mode and the lock requester's mode
- If no commute, then go to step 10)

- sequential search for row address of conflict table => average 4 MM read; compare (load reg1, lock mode; comp reg1, reg3)
- sequential search for column address of conflict table => average 4 MM read; compare (load reg2, lockmode; comp reg2, reg4)
- 1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp reg1, 'y')
Thus, one conflict checking requires: NUM_TRANS_{CLASS}*(10 MM reads + (17+t*2)) operations. Assume that compare operations are performed t times where t = num_br*P_I+N_A*P_C, P_I and P_C denote probability of instance access transaction and probability of class definition access transaction

step 6)
=> 1 MM read

step 7)
=> same as Malta's work
=> P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations

step 8)
=> same as Malta's work
=> P_{IA}*NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 9)
=> P_{IA} MM read

step 10)

- calculate hash function => load reg1, first_Char; 1 MM read for reading first character
  DIV reg1, 26
- calculate the first entry of address in array M_NUM
  load reg2, 0
  load reg3, N_M => 1 MM read
  DIV reg3, 26
  MUL reg1, reg3
- search the method name
average $N_M/(26*2)$ MM read
compare; load reg1, method_name; comp reg1, T_method // Assume that T_method is to
  // be searched
- read the integer number => 1 MM read
Thus, total $(N_M/(26*2)+3)$ MM read + $(N_M/(26*2)*2+6)$ operations

- read the method name of the requester in the commutativity table => same as read method name of the holder

- read the commutativity relation
1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp reg1, 'y')

Total: P_{IA}* NUM_TRANS$_{INST}$ * [$(N_M/26+8)$ MM reads + $(N_M/26*2+13+t*2)$] operations

step 11)
=> same as Malta's work
=>$(6+6*P_{IA})$ MM reads

step 12)
=> same as Malta's work
=> $(6 + 6*P_{IA})$ MM access

Thus, t_{lock} (Proposed) => 2 MM reads (step 1)
+ 2 MM read and 6 operations (step 2)
+ NUM_TRANS$_{CLASS}$*(2 MM read + 2 operations) (step 3)
+ 1 MM read (step 4)
+ NUM_TRANS$_{CLASS}$* [10 MM reads + (17+t*2)] operations (step 5)
+ 1 MM read (step 6)
+ P_{IA}*(1 + NUN_INST*2) MM reads + P_{IA}*(9 + NUM_INST*2) operations (step 7)
+ P_{IA}*NUM_TRANS$_{INST}$*(2 MM read + 2 operations) (step 8)
+ P_{IA} MM read (step 9)
+ P_{IA}*NUM_TRANS$_{INST}$ * [$(N_M/26+8)$ MM reads + $(N_M/26*2+13+t*2)$] operations (step 10)

$+ (6 + 6*P_{IA})$ MM access (step 11 or 12)

$= [12+12*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+ NUM\_TRANS_{INST}*(10+N_M/26)]]*$ MM_ACCESS

$+[6+[19+t*2]*NUM_TRANS_{CLASS}+P_{IA}*[9+2*NUM_INST+NUM_TRANS_{INST}*[15+ N_M/26*2+t*2]]*$BASIC_OP

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$, $t = num\_br*P_I+N_A*P_C$

Thus, $t_{lock, IA}$ (Proposed) = t_{lock} (Proposed)+ $t_{breakpoint}$ + $t_{changelock}$ where $P_{IA} = 1$, where $t_{breakpoint}$ and $t_{changelock}$ will be calculated later.

$t_{lock,CDA}$ (Proposed) = t_{lock} (Proposed) where $P_{IA} = 0$

How to calculate t_{commit} (Proposed)?

=> same as Malta's work

Thus, t_{commit} (Proposed)=

$= N_G*[6+2*NUM\_TRANS_{CLASS}+2*P_{IA}*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*$ MM_ACCESS

$N_G*[6+2*NUM\_TRANS_{CLASS}+ 2*P_{IA}*NUM\_TRANS_{INST} + P_{IA}*[9+NUM\_INST*2]]*$ BASIC_Op

where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

c_I(Proposed) = t_{commit} (Proposed) where $P_{IA} = 1$

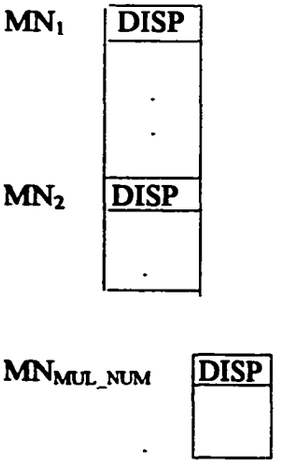
c_C(Proposed) = t_{commit} (Proposed) where $P_{IA} = 0$

<u>b) Recording breakpoints during method execution</u>

- Assume that there are MUL_NUM (active) transactions in the system at the same time (i.e., The multiprogramming level is MUL_NUM).
- Whenever a transaction is entered into system, the transaction is assigned multiprogramming number (MN) and MN is stored in some attribute of the transaction.

- Use array for maintaining breakpoints

- Assume that an array A(11*MUL_NUM) is used for maintaining breakpoints.
- The first element DISP stores displacement of the last element (breakpoint) of the transaction.
- The remaining elements are used to store the breakpoints.
- Assume that each method has maximum 10 breakpoints



a) For each breakpoint encountered during method execution, the following steps are performed.

- Find MN for the transaction
  1 MM read
- Find location of DISP
  A + (MN - 1)*11 => 5 CPU operations
  => load reg1, A; load reg2, MN; SUB reg2, 1; MUL reg2, 11; ADD reg1, reg2

- Put breakpoint into end of the list
  1 MM read (for DISP) + 1 MM write (for breakpoint) + adding DISP (2 operations: load reg1, DISP; ADD reg1, 2)) + 1 MM write (for DISP) => 2 CPU operations + 3 MM accesses

b) After method execution is finished, the following steps are performed

- Read all breakpoints
  1 MM read (for finding MN)
  A + (MN-1)*11 (finding DISP) => 5 operations
  1 MM read (for DISP)
  num_br* MM read (for each breakpoint)
  => (num_br + 2) MM accesses + 5 CPU operations

- Release memories
  1 MM write (DISP <- nil)

From a) and b), for recording all breakpoints in a method, the following overhead is obtained.

a) num_br * (4 MM accesses + 7 CPU operations)

b) (num_br+3) MM accesses + 5 CPU operations

Total: $t_{breakpoint}$ = [5*num_br + 3]*MM_ACCESS + [num_br*7 + 5]*BASIC_OP

c) Lock change (After method execution)

step 1)
=> 2 MM read and 6 operations

step 2)
=> NUM_TRANS_{CLASS}*(2 MM read + 2 operations)
// Assume that half of transactions on the class are searched

step 3)
=> 1 MM access (write breakpoints)

step 4)
=> (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)
=> NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> 1 MM access

Total: $t_{changelock}$= [5 + 2*NUM_TRANS_{CLASS}+2*NUM_TRANS_{INST}+2*NUM_INST]
*MM_ACCESS
+ [15+2*NUM_TRANS_{CLASS}+2*NUM_INST+2*NUM_TRANS_{INST}]
*BASIC_OP

Thus, t_{lock} (Proposed) = t_{lock}(Proposed) + $t_{breakpoint}$ + $t_{changelock}$

= [28+14*NUM_TRANS_{CLASS}+4*NUM_INST+
NUM_TRANS_{INST}*[12+N_M/26)+5*num_br]* MM_ACCESS
+ [35+[21+t*2]*NUM_TRANS_{CLASS}+4*NUM_INST+NUM_TRANS_{INST}*
[17+N_M/26*2+t*2]+num_br*7]*BASIC_OP

III. Analysis for Class hierarchy locking

III.1. Assumption

- The fan-out of each class : F (default)
- For class hierarchy, the depth (level) is D. Thus, average level requested by overall transactions is $A_D = \lceil (D+1)/2 \rceil$ (i.e, in the middle of class hierarchy).

III.2. Analysis

Assume that 2 locks required for each level in implicit locking.

1) average level of access to class hierarchy = 1 (root class)
- Orion (Implicit locking): 1 + (D-2)*2 = 2*D - 3 (intention locks) for every access.
- Malta (Explicit locking): F^{D-1} extra locks for CDA transactions
- The proposed work: 1+(D-2)*2 = 2*D - 3 extra locks for every access.

2) average level of access to class hierarchy = A_D (middle class)
- Orion (Implicit locking): $(A_D-1)+2*(D-A_D) = 2*D-A_D-1$ locks (intention locks) for every access
- Malta (Explicit locking): F^{Ad} extra locks for CDA transactions
- The proposed work: $\min[F^{Ad}, 2*D-A_D-1]$ - D (one each level) extra locks for transactions. Assume that all locks are required for special classes in class hierarchy.

3) average level of access to class hierarchy = D (leaf class)
- Orion (Implicit locking): (D-1) extra locks (intention locks) for every access
- Malta (Explicit locking): 0 extra locks for CDA transactions where 2 locks (level 5)
- The proposed work: 0 extra locks for CDA transactions

t_{lock} and t_{commit} for each technique can be obtained as follows.

<u>a) Orion (Implicit locking)</u>

Assume that N is the number of locks required (including intention locks).

$t_{lock,}$ (Orion) =
$= [12+10*N*NUM\_TRANS_{CLASS}+P_{IA}*[7+2*NUM\_INST+7*NUM\_TRANS_{INST}]]*N*$
MM_ACCESS
$+[6+17*N*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+11*NUM\_TRANS_{INST}]]*N*$
$BASIC\_OP$ where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$

$t_{lock, IA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 1$; $t_{lock, CDA}$ (Orion) = t_{lock}(Orion) where $P_{IA} = 0$

t_{commit} (Orion)=
$N_G*[6+2*N*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS-$
$_{INST}+P_{IA}*[4+NUM\_INST*2]+[AT-1]*P_{IA}*[NUM\_TRANS_{INST}*2+3]]*N*MM\_ACCESS$
$+N_G*[6+2*N*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+$
$P_{IA}*[9+NUM\_INST*2]+[AT-1]*P_{IA}*[NUM\_TRANS_{INST}*2]]*N*BASIC\_OP$
where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

c_I(Orion) = t_{commit} (Orion) where $P_{IA} = 1$; c_C(Orion) = t_{commit} (Orion) where $P_{IA} = 0$

b) Malta (Explicit locking)

Let N be locks (including locks on subclasses)

$t_{lock, IA}$ (Malta) =
$=12+8*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+NUM\_TRANS_{INST}*[10+N_M/26]]]*MM\_ACCESS+[6+13*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+NUM\_TRANS_{INST}*[17+N_M/26*2]]]*BASIC\_OP$
where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$ where $P_{IA} = 1$

$t_{lock, CDA}$ (Malta) =
$=[[12+8*NUM\_TRANS_{CLASS}+P_{IA}*[8+2*NUM\_INST+NUM\_TRANS_{INST}*[10+N_M/26]]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}]*MM\_ACCESS+[6+13*NUM\_TRANS_{CLASS}+P_{IA}*[9+2*NUM\_INST+2*NUM\_TRANS_{INST}*[17+N_M/26*2]]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}*BASIC\_OP$
where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$ where $P_{IA} = 0$

How to calculate t_{commit} (Malta)?

c_I (Malta)=
$N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*MM\_ACCESS$
$+N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*(9+NUM\_INST*2]]*BASIC\_OP$
where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$ where $P_{IA} = 1$

c_C (Malta)=
$N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[4+NUM\_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}*MM\_ACCESS$
$+N_G*[6+2*NUM\_TRANS_{CLASS}+P_{IA}*2*NUM\_TRANS_{INST}+P_{IA}*[9+NUM\_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA}*BASIC\_OP$
where $NUM\_INST \cong M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$ where $P_{IA} = 0$

c) The proposed work

Let N be all locks required (including intention locks and locks on subclasses). In this analysis, only locks on subclasses are assumed for simplicity.

$t_{lock, IA}$(Proposed)=
$= [28+14*NUM\_TRANS_{CLASS}+4*NUM\_INST+NUM\_TRANS_{INST}*[12+ N_M/26]$

+5*num_br]*MM_ACCESS
+[35+[21+t*2]*NUM_TRANS_{CLASS}+4*NUM_INST+NUM_TRANS_{INST}*[17+N_M/26*2+t*2]+ num_br*7]*N*BASIC_OP where P_{IA}=1 and NUM_INST ≅ $M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$, t = num_br*$P_f$+$N_A$*$P_C$

$t_{lock, CDA}$(Proposed) =
[12+12*NUM_TRANS_{CLASS}+P_{IA}*[8+2*NUM_INST+NUM_TRANS_{INST}*[10+N_M/26]]]*[P_{CDR}+P_{CDW}*N]/P_{CDA} *MM_ACCESS
+[6+[19+t*2]*NUM_TRANS_{CLASS}+P_{IA}*[9+2*NUM_INST+NUM_TRANS_{INST}*[15+N_M/26*2+t*2]]]* [P_{CDR}+P_{CDW}*N]/P_{CDA} *BASIC_OP where NUM_INST ≅ $M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/N_C$, $NUM\_TRANS_{INST} \cong M_P*N_G/(N_c*N_I)$, t = num_br*$P_f$+$N_A$*$P_C$ where P_{IA} = 0

c_f(Proposed) =
N_G*[6+2*NUM_TRANS_{CLASS}+P_{IA}*2*NUM_TRANS_{INST}+P_{IA}*[4+NUM_INST*2]]*MM_ACCESS
+N_G*[6+2*NUM_TRANS_{CLASS}+P_{IA}*2*NUM_TRANS_{INST}+P_{IA}*[9+NUM_INST*2]]*BASIC_OP
where NUM_INST ≅ $M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

c_C(Proposed) =
N_G*[6+2*NUM_TRANS_{CLASS}+P_{IA}*2*NUM_TRANS_{INST}+P_{IA}*[4+NUM_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA} *MM_ACCESS
+N_G*[6+2*NUM_TRANS_{CLASS}+P_{IA}*2*NUM_TRANS_{INST}+P_{IA}*[9+NUM_INST*2]]*[P_{CDR}+P_{CDW}*N]/P_{CDA} *BASIC_OP where NUM_INST ≅ $M_P*N_G/(2*B*N_C)$, $NUM\_TRANS_{CLASS} \cong M_P*N_G/(2*N_C)$, $NUM\_TRANS_{INST} \cong M_P*N_G/(2*N_c*N_I)$

IV. Analysis for nested method invocations

Assumptions:

- For composite object hierarchy, assume that there are L_C levels.
- For each instance access method accessing composite object, there are F_C number of method invocations to subobject in composite object hierarchy. Thus, for each instance access method invocation on top-level composite object, there are $N_{COM} = 1 + Fc + (F_C)^2 + ... + (F_C)^{Lc}$ number of objects accessed
- For the proposed work, semantic commutativity and commutativity between parents and children (due to inheritance) are not considered for simplicity.
- Conflicts among instance access methods are only considered in order to simplify analysis.

IV.1. Orion

• Whenever a transaction requests a lock on instance, do the following steps.

(1) Read the lock requester's mode and transaction_id
(2) Find the lock entry for the class
(3) Set a lock on the class
(4) Find the lock entry for the instance
(5) - Find a lock holder on the instance
  - while there is a different transaction holding lock do
(6) - Find the commutativity table for the instance
(7) - Read the holder's mode
  - Check the commutativity table for commutativity of the lock holder's mode and the lock requester's mode
  - If no commute, go to step 9)
  end while
(8) Set a lock in the lock table and stop
(9) Block the request and stop

• If a transaction is committed or aborted, release all the locks. For each locks held, do as follows.

(1) - Find the lock entry for the class
(2) - Find a lock holder on the class
  - while the same transaction holds a lock do
(3) - Release the lock on the class
  end while
(4) - Find the lock entry for the instance
(5) - Find a lock holder on the instance
  - while the same transaction holds a lock do
(6) - Release the lock on the instance, and stop
  end while

IV.2. Malta's scheme

• Whenever a transaction requests a lock on instance, do the following steps.

(1) Read the lock requester's method name and transaction_id
(2) Find the lock entry for the class
(3) Set a lock on the class
(4) Find the lock entry for the instance
(5) - Find a lock holder on the instance
  - while there is a different transaction holding lock do
(6) - Find the method commutativity table for instance
(7) - Read the holder's method

- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 9)

end while

(8) Set a lock into lock table and stop
(9) Block the request and stop

• If a transaction is committed or aborted, release all the locks. For each locks held, do as follows.

(1) - Find the lock entry for the class
(2) - Find a lock holder on the class
- while there is a different transaction holding lock do
(3) - Release the lock on class
(4) - Find the lock entry for the instance
(5) - Find a lock holder on the instance
- while the same transaction holds a lock do
(6) - Release the lock on instance
end while
end while

IV.3. The proposed scheme

• Whenever a transaction requests a lock on instance, do the following steps.

(1) Read the lock requester's method name and transaction_id
(2) Find the lock entry for the class
(3) Set a lock on the class
(4) Find the lock entry for the instance
(5) - Find a lock holder on the instance
- while there is not a same subtransaction holding lock do
(6) - Find the method commutativity table for instance
(7) - Read the holder's method
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 9)

end while
(8) Set a lock in the lock table and stop
(9) Block the request and stop

• If a transaction is committed or aborted, release all the locks. For each locks held, do as follows.

(1) - Find the lock entry for the class
(2) - Find a lock holder on the class

- while there is a different transaction holding lock do
(3) - If the commiting method is top-level, release the lock
else inherit locks to parents
end while
(4) - Find the lock entry for the instance
(5) - Find a lock holder on the instance
- while there is not same subtransaction holds a lock do
(6) - If the commiting transaction is top-level, release the lock
else inherit locks to parents
end while

IV.1. Orion

How to calculate t_{lock} in Orion?

step 1)
=> 2 MM read // Assume that transaction_id, lock mode occupy one word, respectively.

step 2)
- Each class has B buckets for maintaining lock tables for instances
- The class has 4 MM words (2 words for blocked transactions, 2 words for locking transactions): Assume that transaction_id, lock mode and pointer takes one word, respectively.

- get address of lock table => 1 MM read
- get address of lock entry for class =>
(address of lock table) +(class-id - 1)*S_Lock // S_Lock is size of lock entry for each class where S_Lock = B*2+4. Assume that this number is stored in S_Lock.

load reg1, address of lock table
load reg2, T-cid (target class id)
SUB reg2, 1
load reg3, S_Lock => 1 MM read
MUL reg2, reg3
ADD reg1, reg2

Total : 2 MM read and 6 operations

step 3)
If there are some transactions holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <- nil)
1 MM write (change Y)

1 MM write (change Nil to newly inserted transaction)
=> 6 MM access

If there is no transaction holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <-nil)
2 MM writes (change Y and X)
=> 6 MM access

step 4)
load reg1, T-iid (target instance id)
load reg1, reg2
load reg3, B (number of buckets) => 1 MM read
DIV reg2, reg3
MUL reg2, reg3
SUB reg1, reg2
MUL reg1, 2
SUB reg1, 2
ADD reg1, 4
=> (1 MM read + 9 operations)

- Find target instance
1 MM read (for x or pointer)
1 MM read (for T-iid)
Compare (if that is target instance) => load reg1, iid; comp reg1, T-iid (target instance)

Assume above steps are done NUM_INST times. NUM_INST can be approximated as follows. NUM_INST $\cong M_P * N_{COM}/(2*B*N_C)$

Total: (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)
=> For each lock holder, the following overhead is required.
1 MM read (for pointer X)
1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong M_P * N_{COM} / (N_C * N_t)$

Total: NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> 1 MM read

step 7) - Read the holder's mode
=> 1 MM read; load reg3, lockmode
- Check commutativity table for commutativity of the lock holder's mode and the lock requester's mode
- sequential search for row address of conflict table
=> average 1.5 MM read; compare (load reg1, lock mode; comp reg1, reg3) for each read
- sequential search for column address of conflict table
=> average 1.5 MM read; compare (load reg2, lockmode; comp reg2, reg4) for each read
- check the commutativity
=> 1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp (reg1, 'y')

For each lock holder, 5 MM reads + 9 operations

Total: NUM_TRANS_{INST}*(5 MM reads + 9 operations)

step 8)
=> same as step 3) => 6 MM access

step 9)
=> same as step 3) => 6 MM access

t_{lock} (Orion) = 2 MM reads (step 1)
+ 2 MM reads and 6 operations (step 2)
+ 6 MM access (step 3)
+ (1+NUM_INST*2) MM reads + (9+NUM_INST*2) operations (step 4)
+ NUM_TRANS_{INST} *(2 MM read + 2 operations) (step 5)
+ 1 MM read (step 6)
+ NUM_TRANS_{INST} *(5 MM reads +9 operations) (step 7)
+ 6 MM access (step 8 or 9)
= [18 + 2*NUM_INST + 7*NUM_TRANS_{INST}]*MM_ACCESS
+ [15 + 2*NUM_INST + 11*NUM_TRANS_{INST} + N_M/26*2]*BASIC_OP where
NUM_INST $M_P*N_{COM}/(2*B*N_C)$, and $NUM\_TRANS_{INST} \cong M_P*N_{COM} / (N_C*N_I)$

How to find t_{commit} in Orion?

step 1)
=> same as step 2) in t_{lock} : 2 MM reads + 6 operations

step 2)
For each lock holder, the following overhead is required.
1 MM read (for pointer X)

1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{CLASS} times in a class
$NUM\_TRANS_{CLASS} \cong M_P / (2*N_C)$ where M_P is the multiprogramming level. Assume that only half of NUM_TRANS_{CLASS} transactions is assumed.

Total: NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 3)
1 MM read (read next record of committed transaction)
1 MM read (read previous record of committed transaction)
1 MM write (pointer of previous record => next record of committed transaction)
=> 3 MM access

step 4)
=> same as step 4) in t_{lock}
=> (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)
For each lock holder, the following overhead is required.
1 MM read (for pointer X)
1 MM read (for a transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong M_P / (2*N_C*N_I)$

Assume that only half of NUM_TRANS_{INST} transactions is searched.

Total: NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> same as step 3)
=> 3 MM reads

Note that, in Orion, since each method requires AT locks, step 5) and 6) need to be performed additional (AT-1) times.

Total: N_{COM}*[9+2*NUM_TRANS_{CLASS}+2*NUM _$TRANS_{INST}$+2*NUM_INST+[AT-1]*[2*NUM_TRANS_{INST}+3]]*MM_ACCESS
+ N_{COM}* [15+2*NUM_TRANS_{CLASS}+ 2*NUM_TRANS_{INST} + 2*NUM_INST+[AT-1]*[2*NUM_TRANS_{INST}]]*BASIC_OP

Since each transaction accesses N_{COM} granules,
t_{commit} (Orion)=
N_{COM}*[9+2*NUM_TRANS$_{CLASS}$+2*NUM _TRANS$_{INST}$+2*NUM_INST+[AT-1]*[2*NUM_TRANS$_{INST}$+3]]*MM_ACCESS
+ N_{COM}*[15+2*NUM_TRANS$_{CLASS}$+ 2*NUM_TRANS$_{INST}$ + 2*NUM_INST+[AT-1]*[2*NUM_TRANS$_{INST}$]]*BASIC_OP where NUM_INST $\cong M_P/(2*B*N_C)$,
NUM_TRANS$_{CLASS} \cong M_P*N_G/(2*N_C)$, NUM_TRANS$_{INST} \cong M_P/(2*N_c*N_I)$

IV.2. Malta's work

How to calculate t_{lock} in Malta's?

step 1)
=> same as Orion: 2 MM read

step 2)
=> same as Orion: 2 MM read and 6 operations

step 3)
=> same as Orion: 6 MM access

step 4)
=> same as Orion: (1+NUM_INST*2) MM reads + (9+NUM_INST*2) operations

step 5)
=> same as Orion
step 6)
=> same as Orion: 1 MM read

step 7) - Read the holder's mode
=> 1 MM read; load reg3, lockmode
- Check the commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 8)

• In Malta's work, the lock is requested for each instance access method. In real OODBs, there are many methods defined in each class so that searching a particular method in a commutativity table takes so much overhead. Thus, the following strategy is adopted: assign each method to the unique number so that the method can be searched directly in a commutativity table. There are two steps to do it.

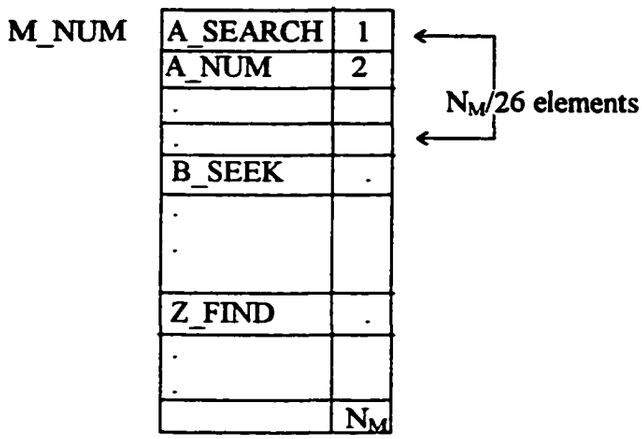
• substep 1) The hash function is used to map each instance method to integer. Also, assume that each method is named based on alphabet. Whenever a method is invoked,

read the first character of the method. Based on order of this character in alphabet, the character is mapped into number.

h(first character of each method) = I where $1 \leq I \leq 26$

For example, for method ASCEND_ORDER and DESCEND_ORDER has the following hash function.
h(ASCEND_ORDER) = 1
h(DESCEND_ORDER) = 4

• substep 2) Then, assume that array M_NUM stores the unique number for each method. The array is has total N_M (number of methods) elements, and divided into 26 sectors. Thus, each sector has $N_M/26$ elements.



- calculate hash function => load reg1, first_Char; 1 MM read for reading first character
  DIV reg1, 26
- calculate the first entry of address in array M_NUM
  load reg2, 0
  load reg3, N_M => 1 MM read
  DIV reg3, 26
  MUL reg1, reg3
- search the method name
average $N_M/(26*2)$ MM read
compare; load reg1, method_name; comp reg1, T_method // Assume that T_method is to
  // be searched
- read the integer number => 1 MM read
Thus, total $(N_M/(26*2)+3)$ MM read + $(N_M/(26*2)*2+6)$ operations

- read method name of the requester in the commutativity table => same as read method name of the holder

- read the commutativity table
1 MM read (y or n) + 1 compare (load reg1, conflict(row,column); comp reg1, 'y')

Total: NUM_TRANS_{INST}* [(N_M/26+8) MM reads + (N_M/26*2+15) operations]

step 8)
=> same as step 3) => 6 MM access

step 9)
=> same as step 3) => 6 MM access

t_{lock} (Malta) = 2 MM reads (step 1)
+ 2 MM reads and 6 operations (step 2)
+ 6 MM access (step 3)
+ (1+NUM_INST*2) MM reads + (9+NUM_INST*2) operations (step 4)
+ NUM_TRANS_{INST} (2 MM read + 2 operations) (step 5)
+ 1 MM read (step 6)
+ NUM_TRANS_{INST}*[(N_M/26+8) MM reads + (N_M/26*2+15) operations] (step 7)
+ 6 MM access (step 8 or 9)
= [18 + 2*NUM_INST + NUM_TRANS_{INST} *[10+ N_M/26]]*MM_ACCESS
+ [15 + 2*NUM_INST + NUM_TRANS_{INST} *[17+ N_M/26*2]]*BASIC_OP where
NUM_INST M_P*N_{COM}/(2*B*N_C), and $NUM\_TRANS_{INST} \cong M_P$*$N_{COM}$ / (2*N_C*N_I)

How to find t_{commit} in Malta's?

step 1)
=> same as Orion: 2 MM reads + 6 operations

step 2)
=> same as Orion: NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 3)
=> same as Orion: 3 MM access

step 4)
=> same as Orion: (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)

=> same as Orion: NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> same as Orion: 3 MM reads

Total:[9+2*NUM_TRANS$_{CLASS}$+2*NUM _TRANS$_{INST}$+2*NUM_INST] MM reads
[15+2*NUM_TRANS$_{CLASS}$+ 2*NUM_TRANS$_{INST}$ + 2*NUM_INST] operations

Since each transaction accesses N_{COM} granules,
t_{commit} (Malta)=
N_{COM}*[9+2*NUM_TRANS$_{CLASS}$+2*NUM_TRANS$_{INST}$+2*NUM_INST]* MM_ACCESS
+N_{COM}*[15+2*NUM_TRANS$_{CLASS}$+2* NUM_TRANS$_{INST}$+ 2*NUM_INST]*BASIC_OP
where NUM_INST $\cong M_P/(2*B*N_C)$, NUM_TRANS$_{CLASS} \cong M_P*N_G/(2*N_C)$,
NUM_TRANS$_{INST} \cong M_P/(2*N_c*N_I)$

IV.3. The proposed work

How to find t_{lock} in the proposed work?

1) initial lock request

step 1)
=> same as Malta's work
=> 2 MM reads

step 2)
=> same as in Malta
=> 2 MM read and 6 operations

step 3)
If there are some transactions holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <- nil)
1 MM write (change Y)
1 MM write (change Nil to newly inserted transaction)
=> 6 MM access

If there is no transaction holding locks =>
1 MM read (read Y)
3 MM writes (for transaction_id, lock, pointer <-nil)
2 MM writes (change Y and X)
=> 6 MM access

step 4)
load reg1, T-iid (target instance id)
load reg1, reg2
load reg3, B (number of buckets) => 1 MM read
DIV reg2, reg3
MUL reg2, reg3
SUB reg1, reg2

MUL reg1, 2
SUB reg1, 2
ADD reg1, 4
=> (1 MM read + 9 operations)

- Find target instance
1 MM read (for x or pointer)
1 MM read (for T-iid)
Compare (check if that is target instance) => load reg1, iid; comp reg1, T-iid (target instance)

Assume above steps are done NUM_INST times. NUM_INST can be approximated as follows.
NUM_INST $\cong N_{COM}*M_P/(2*B*N_C)$

Total: (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)
=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong N_{COM}*M_P/ (N_C*N_I)$

Total: NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> 1 MM read
step 7) - Find a lock holder on instance
- Check commutativity table for commutativity of the lock holder's method and the lock requester's method
- If no commute, go to step 9)
=> same as Malta's work
Total: NUM_TRANS_{INST}* [(N_M/26+8) MM reads + (N_M/26*2+15) operations]

step 8)
=> same as step 3) => 6 MM access

step 9)
=> same as step 3) => 6 MM access

t_{init} (Proposed)= 2 MM reads (step 1)

+ 2 MM read and 6 operations (step 2)
+ 6 MM access (step 3)
+ (1+ NUN_INST*2) MM reads + (9 + NUM_INST*2) operations (step 4)
+ NUM_TRANS_{INST}*(2 MM read + 2 operations) (step 5)
+ 1 MM read (step 6)
+ NUM_TRANS_{INST} *[(N_M/26+8) MM reads + (N_M/26*2+15) operations] (step 7)
+ 6 MM access (step 8 or 9)

= [18 + 2*NUM_INST + NUM_TRANSA_{INST} *[10+N_M/26]]*MM_ACCESS
+ [15 + 2*NUM_INST + NUM_TRANS_{INST} *[17+ N_M/26*2]]*BASIC_OP

2) Recording breakpoints during method execution

- Assume that there are MUL_NUM (active) transactions in the system at the same time (i.e., The multiprogramming level is MUL_NUM).
- Whenever a transaction is entered into system, the transaction is assigned multiprogramming number (MN) and MN is stored in some attribute of the transaction.

- Use array for maintaining breakpoints

- Assume that there are N_{COM} number of subtransaction of a given transaction
- Assume that an array A(13*MUL_NUM, N_{COM}) is used for maintaining breakpoints.
- The first and second element store object-id and method_name, respectively.
- The second element DISP stores displacement of the last element (breakpoint) of the transaction.
- The remaining elements are used to store the breakpoints.
- Assume that each method has maximum 10 breakpoints.

MN_1

object_id	object_id	.....	
method_name	method_name	.....	
DISP	DISP	.....	
. .			

MN_{MUL_NUM}

object_id	object_id	.....	
method_name	method_name	.....	
DISP	DISP	.....	

a) For each breakpoint encountered during method execution, the following steps are done.

- Read MN for the transaction, object_id and method name
  1 MM read + 1 MM read (o_id) + 1 MM read (method name) => 3 MM reads
- Find location of MN for the transaction in array
  A + (MN - 1)*13* N_{COM}

=> 7 CPU operations

=> load reg1, A (1 MM read); load reg2, MN; SUB reg2, 1; MUL reg2, 13; load reg3, N_{COM} (1 MM read); MUL reg2, reg3, ADD reg1, reg2;

Time to find (object_id, method_name) pair

0.5 * N_{COM} * {2 MM reads (object_id and method name) + 2 compare} where 2 compares take 4 operations as follows: load reg1, object_id; comp reg1, target_oid; load reg2, method_name; comp reg2, target_method name)

=> (5 + N_{COM}) MM access + (7 + 2*N_{COM}) operations

- Put breakpoint into end of the list
  1 MM read (for DISP) + 1 MM write (for breakpoint) + adding DISP (2 operations:
  load reg1, DISP; ADD reg1, 2)) + 1 MM write (for DISP) => 2 CPU operations + 3 MM accesses

b) After method execution is finished, the following steps are done.

- Read all breakpoints
  3 MM read (for finding MN, object_id and method_name)
  (5 + N_{COM}) MM access + (7 + 2*N_{COM}) operations (finding DISP)
  1 MM read (for DISP)
  num_br MM read (for each breakpoint)
  => (9+ N_{COM} +num_br) MM accesses + (7 +2*N_{COM}) CPU operations

- Release memories
  1 MM write (DISP <- nil)
  2 MM writes (object_id, method name <- nil)

=> 3 MM access

From a) and b), for recording all breakpoints in a method, we have the following overhead.

a) num_br*(8 + N_{COM}) MM accesses + num_br*(9 + 2*N_{COM}) CPU operations
b) (N_{COM} + num_br+12) MM accesses + (7+2*N_{COM}) CPU operations

Total: $t_{breakpoint}$ = [9*num_br+[num_br+1]*N_{COM}+12]*MM_ACCESS +

$[9*num\_br + [2*num\_br+2]*N_{COM}+7]*BASIC\_OP$

c) lock change (After method execution)

step 1)
=> 2 MM read and 6 operations

step 2)

=> NUM_TRANS_{CLASS}*(2 MM read + 2 operations)
// Assume that half of transactions on the class are searched

step 3)
=> 1 MM access (write breakpoints)

step 4)
=> (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)

=> NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> 1 MM access

Total: $t_{changelock}$= [5 + 2*NUM_TRANS_{CLASS}+2*NUM_TRANS_{INST}+2*NUM_INST]*
MM_ACCESS
+ [15+2*NUM_TRANS_{CLASS}+2*NUM_INST+2*NUM_TRANS_{INST}]
*BASIC_OP

Thus, t_{lock} (Proposed) = t_{lock}(Proposed) + $t_{breakpoint}$ + $t_{changelock}$

How to find t_{commit} in the proposed work?

step 1)
=> same as step 1) in t_{lock} : 2 MM reads + 6 operations

step 2)
=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (check if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{CLASS} times in a class $NUM\_TRANS_{CLASS} \cong N_{COM}* M_P/ (2*N_C)$ where M_P is the multiprogramming level. Assume that only half of NUM_TRANS_{CLASS} transactions is searched.

Total: NUM_TRANS_{CLASS}*(2 MM read + 2 operations)

step 3)

inherit locks into parents
1 MM read (ancestor_id)
1 MM write (owner_id)
1 MM write (RET<- True)
=> 3 MM access

Release the lock on class
1 MM read (read next record of committed transaction)
1 MM read (read previous record of committed transaction)
1 MM write (pointer of previous record => next record of committed transaction)
=> 3 MM access

step 4)
=> same as step 4) in t_{lock}
=> (1 + NUN_INST*2) MM reads + (9 + NUM_INST*2) operations

step 5)

=> For each lock holder, the following overhead is required.
1 MM read (for X)
1 MM read (for transaction)
1 Compare (if this is other transaction or not) => load reg1, t-id; comp reg1, T-tid (target transaction)

Assume that above steps are done NUM_TRANS_{INST} times in an instance
$NUM\_TRANS_{INST} \cong N_{COM}*M_P / (2*N_C*N_I)$

Also, assume that we search only half of NUM_TRANS_{INST} transactions.

Total: NUM_TRANS_{INST}*(2 MM read + 2 operations)

step 6)
=> same as step 3)
=> 3 MM reads

Total:[9+2*NUM_TRANS$_{CLASS}$+2*NUM _TRANS$_{INST}$+2*NUM_INST]*MM_ACCESS

[15+2*NUM_TRANS$_{CLASS}$+ 2*NUM_TRANS$_{INST}$ + 2*NUM_INST]*BASIC_OP where NUM_TRANS$_{CLASS}$ ≅ N_{COM}* M_P/ (2*N_C) and NUM_TRANS$_{INST}$ ≅ N_{COM}*M_P / (2*N_C*N_I), NUM_INST ≅ N_{COM}*M_P/(2*B*N_C)

Since there are N_{COM} subtransactions and each subtransaction has average N_{COM}/2 numbers of granules accessed,
t_{commit} (Proposed)=
N_{COM}*N_{COM}/2*[9+2*NUM_TRANS$_{CLASS}$+2*NUM_TRANS$_{INST}$+2*NUM_INST]*
MM_ACCESS
+N_{COM}*N_{COM}/2*[15+2*NUM_TRANS$_{CLASS}$+2* NUM_TRANS$_{INST}$+
2*NUM_INST]*BASIC_OP where NUM_INST ≅ N_{COM}* M_P*N_C/(2*B*N_C),
NUM_TRANS$_{CLASS}$ ≅ N_{COM}* M_P/(2*N_C), NUM_TRANS$_{INST}$ ≅ N_{COM}* M_P/(2*N_c*N_I)

V. How to calculate Database size (number of pages in database)?

V.1. Storage format

Assume that storage format for instances is adopted as follows [Kim,1990]

uid	object length	attribute count	attribute vector	values offset vector	values

uid (unique object identifier): 5 bytes
object length (the total length of the object): 4 bytes (assumed)
attribute count (the number of attributes): 4 bytes (assumed)
attribute vector (identifiers of all attributes): 5 bytes*(number of attributes)
values offset vector (offsets of the values of the attributes): 4 bytes*(number of attributes)
values (values of attributes): depends on attribute size

1) Module class
size of values = 4+10+4+4+1093*4+4 = 4398 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*6 = 30 bytes
size of values offset vector size: 4*6 = 24 bytes
size of values : 4398 bytes
=> total : 4464 bytes

2) Manual class
size of values = 40+4+80+4+4= 132 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes

size of attribute vector size : 5*5 = 25 bytes
size of values offset vector size: 4*5= 20 bytes
size of values : 132 bytes
=> total : 189 bytes

3) CompositePart class
size of values = 4+10+4+4+4*4+4*20+4 = 122 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*7 = 35 bytes
size of values offset vector size: 4*7= 28 bytes
size of values : 122 bytes
=> Total: 197 bytes

4) Document class
size of values = 40+4+80+4 = 128 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*4 = 20 bytes
size of values offset vector size: 4*4 = 16 bytes
size of values : 128 bytes
=> total : 176 bytes

5) AtomicPart class
size of values = 4+10+4+4*2+4+4*3+4*3+4 =58 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*9 = 45 bytes
size of values offset vector size: 4*9 = 36 bytes
size of values : 58 bytes
=> total : 151 bytes

6) Connection class
size of values = 10+4+4+4 = 22 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*4 = 20 bytes
size of values offset vector size: 4*4 = 16 bytes
size of values : 22 bytes
=> total : 70 bytes

7) ComplexAssembly class
size of values = 4+10+4+4+4+4*3 = 38 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*6 = 30 bytes
size of values offset vector size: 4*6 = 24 bytes
size of values : 38 bytes
=> total : 104 bytes

8) BaseAssembly class
size of values = 4+10+4+4+4+4*3 = 38 bytes
size of uid : 4 bytes
size of object length : 4 bytes
size of attribute count : 4 bytes
size of attribute vector size : 5*6 = 30 bytes
size of values offset vector size: 4*6 = 24 bytes
size of values : 38 bytes
=> Total 104 bytes

V.2. Disk and Page layout in OODB [Kim,1990]

Assumptions:
A raw disk can be divided into a set of partitions (analogous to cylinder groups).
Each partition consists of number of segments.
Each segment in turn consists of a number of blocks or pages.

In each partition, the disk header contains information such as the number of partitions, the address and size of each partition and the recovery log file

Segments in a partition are described by a segment table in which the address and sizes of the page tables for the segments are stored.

Each page table in a segment records information about the size of each page in number of blocks

The partition table (in disk header) format

Class	Partition number	partition address	partition size
C_1	1	.....	.....
.....	.....	.....	.....
C_n	15	.....	.....

The segment table format in partition P_i (indicated by partition address in partition table).

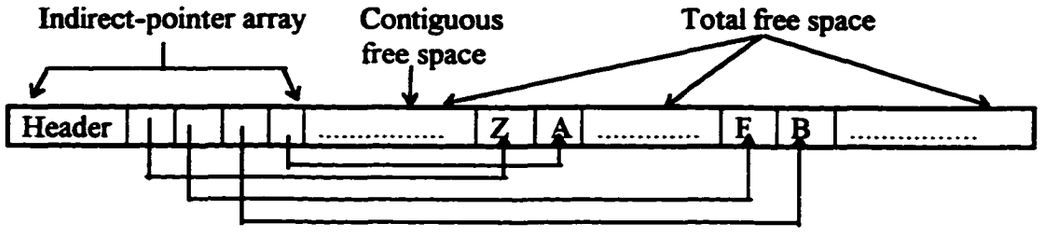
Class	Segment number	segment address	segment size
C_1	S_1	.....	.....

.....	.....	.....	.....
C_n	S_k	.....	.....

The page table format in segment S_i (indicated by segment address in segment table).

Class	Page number	Page address	Page size
C_1	P_1	.....	.....
.....	.....	.....	.....
C_n	P_m	.....	.....

A page has the following format [Kim, 1990]



Header provides information about the page such as number of objects, total free space, contiguous free space, offset to free space, etc.
Indirect pointer array: offset to the page location where the object is placed.
The physical address of an object: concatenation of the page that holds the object and the byte offset within the page.
For simplicity, assume that size of header is negligible and there is no free space. Then, for each class C_i , the amount of memory required for all of its instances, say S_i, can be calculated as follows. Let I_i be the number of instances in C_i.

S_i = size to store indirect pointer array + size to store all of instances in C_i
= I_i*size to store each address +I_i*size to store each instance
= I_i*(size to store each address + size to store each instance)

Then, for 007 benchmark, we can calculate DBsize as follows.
Module class = 4 + 4464 = 4468 bytes
Manual class = 4 + 189 = 193 bytes
CompositePart class = 500*(4+197) = 100500 bytes
Document class = 500*(4+176) = 90000 bytes
AtomicPart class = 500*20*(4+151) = 1550000 bytes
Connection class = 500*20*3*(4+70) = 2220000 bytes
ComplexAssemby class = 364*(4+104) = 39312 bytes
BaseAssembly class = 729*(4+104) = 78732 bytes

Pages needed for each class can be calculated as follows. Note that size of each page is 2048 bytes.
Module = $\lceil 4468/2048 \rceil$ = 2 pages

Manual = ⌈193/2048⌉ = 1 page
CompositePart = ⌈100500/2048⌉ = 50 pages
Document = ⌈90000/2048⌉ = 44 pages
AtomicPart = ⌈1550000/2048⌉ = 757 pages
Connection = ⌈2220000/2048 ⌉ = 1084 pages
ComplexAssembly = ⌈39312/2048⌉ = 20 pages
BaseAssembly = ⌈78732/2048⌉ =39 pages

Therefore, DBsize = 2+1+50+44+757+1084+20+39 = 1997 pages

Assume that there are 10 partitions (cylinders). Then, each partition has about 200 pages. Also, assume that only one class is stored in the same segment of disk pages. Thus, total 8 segment are required to store 007 benchmark database.