Improved Scheduling in Object Bases using Statically Derived Information[*]

Peter Graham and Ken Barker
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2
{pgraham,barker}@cs.umanitoba.ca

April 26, 1994

Abstract

Traditional approaches to scheduling in both object and data bases arrive at scheduling decisions based on dynamic, read/write behaviour or transaction submission order alone. Such approaches are based on exact but extremely limited information. Using static analysis, it is possible to derive more extensive but possibly inexact information. By combining the statically and dynamically derived information at scheduling time it is possible to increase the concurrency attained in transaction execution while decreasing the negative side-effects of conventional concurrency control techniques including unnecessary overhead and the possibility of deadlock.

In this paper, we describe a model and framework in which our approach may be applied. We then present the basic static analyses required to derive useful information for our concurrency control protocol and develop the scheduling algorithm itself. We conclude by arguing the effectiveness of our approach and then briefly discuss future work being undertaken.

1 Introduction

Concurrency control in existing object base systems is typically provided using object-level locking. This approach has several drawbacks:

- Locking on a per-object basis is too coarse and therefore limits potential concurrency. This is particularly true of large, complex, and nested objects.
- Lock based algorithms are subject to deadlock which is usually dealt with by rollback. Rollback is a high price to pay for concurrency control.
- Determining a serialization order using dynamic techniques such as locking may result in sub-optimal orders. This may result in decreased concurrency.

[*]This research was partially supported by the Natural Science and Engineering Research Council (NSERC) of Canada under Operating Grants (OGP-0105566 and OGP-0121527).

- The *significant* overhead of obtaining and releasing locks must be incurred even when it can be determined *à priori* that no conflict will occur.
- No object specific information is incorporated into the scheduling process.

Finer granularity concurrency control can be provided by performing attribute level locking. Unfortunately, while such fine granularity does offer the potential for greater concurrency, the overhead of managing so many locks negates any potential benefit since the cost of synchronization exceeds the savings offered by the additional concurrency.

Locking, at any level, and other concurrency control techniques in existing object base systems enforce an *arbitrary* serialization order. This in turn affects the attainable concurrency at any given point in the execution of a set of transactions. There is a "best" serialization order for any given set of concurrently executing transactions – the one which yields the greatest concurrency while maintaining execution correctness. Given all the possible interleavings and hence serialization orders, it is highly unlikely that the best one will be chosen arbitrarily. To methodically achieve the greatest possible concurrency, some *semantic* information concerning object behaviour and inter-object relationships must be considered. Such information may be conservatively determined prior to transaction execution using static analysis performed at class-compile and object-instantiation times.

We propose an approach which uses statically determined information to guide dynamic concurrency control enforcement in a nested transaction environment. By exploiting static information dynamically, intelligent (serialization) ordering decisions may be made which have the effect of increasing potential concurrency while decreasing concurrency control overhead and the potential of deadlock.

The rest of this paper is organized as follows: Related work is described in Section 2. Section 3 describes our model and assumptions. The static analysis required to derive the semantic information needed to apply our technique is described in Section 4. In Section 5 the process of scheduling method invocations using static information is discussed. Finally, Section 6 makes some concluding remarks and proposes directions for future research.

2 Related Work

Related work includes nested transactions and object base concurrency control.

2.1 Nested Transactions

Flat, structure-less, transactions as seen in conventional database systems, do not suit the processing of the highly structured data stored in an object base. One approach to adding structure to transactions is via *nested transactions* as described by Moss[1] [Mos85] and subsequently refined by many other researchers (e.g., Weihl[Wei89], Harder and Rothermel [TR93]). Nested transactions provide two fundamental benefits; they allow concurrency within a transaction, and they provide greater protection against failures by supporting recovery at a level of granularity finer than the entire transaction.

Two different forms of transaction nesting have been proposed; closed and open nesting [Mos85]. In both forms, the updates to data items made by one sub-transaction are hidden from all other transactions until the sub-transaction commits. Thus, atomicity is

[1]Earlier work on nested transactions was actually done by Reed[Ree78] but Moss' work is considered seminal.

provided at the sub-transaction level. But, in closed nesting, after a sub-transaction commits, only other sub-transactions of the same parent are allowed to see the changes. In open nesting, sub-transaction updates are visible to all other transactions following commitment. This has the affect of increasing potential concurrency but only at the expense of more complicated recovery.

Nested transactions offer clear benefits over flat transactions. Unfortunately, the manual subdivision of a transaction into sub-transactions and the subsequent management of their concurrent execution is an undesirable burden on the transaction programmer. Using the object nesting structure to implicitly determine sub-transaction boundaries, as described here, avoids this problem.

2.2 Object Base Concurrency Control

Object base concepts and design strategies have been discussed in several papers [ABD$^+$89, Kim90, BM91, HPC93]. Many prototype and commercial systems have been constructed including; Cactis [HK89], Exodus [CDG$^+$90], Gemstone [BOS91], Iris [WLH90], ObjectStore [LLOW91], ORION [KGBW90], and O_2 [Deu90] among others. The bulk of these systems provide only simple concurrency control mechanisms which are often based on object-level locking.

The issue of concurrency control in object bases has also been discussed in many research papers. Much of this work concentrates on issues associated with concurrency control for *nested objects*. Rakow, Gu, and Neuhold [RGN90] define object serializability for *open* nested transactions in object bases. The concurrency control algorithm they describe exploits the semantics and nesting of operations to increase concurrency. Concurrency control is based on commutativity (see Weihl [Wei88]). It is assumed that commutativity tables for all methods on all objects have been manually generated.

Transaction synchronization in object bases is also addressed by Hadzilacos and Hadzilacos [HH91] who define transaction management algorithms for *closed* nested transactions. The correctness of nested two-phase locking and nested timestamp ordering applied to object bases is shown and a new algorithm is described where a graph constructed according to a method-local partial order and inter-object ordering information is acyclic only if the history is equivalent to a serial history of the same method invocations with the same nesting structure. Unfortunately, this algorithm corresponds to testing to "see" that methods in a concurrent execution see the same views as the methods in a serial execution and hence is inherently inefficient.

Resende and El Abbadi [RE92] describe a graph testing concurrency control protocol for object bases using the closed nested transaction model of Hadzilacos and Hadzilacos. The *optimistic* protocol they suggest specifies how to dynamically construct a graph for each method invocation so that a cyclic graph occurs only if serializability is violated.

Recently several papers have attempted to exploit some *automatically derived* semantic information to improve the level of concurrency in object base systems. Graham, Zapp, and Barker [GZB92] address the problem of providing *method-level* scheduling. Conflict between methods is defined (and derived automatically at method compile time) based on which object attributes are accessed. Some of the problems associated with inter-object method invocations are considered and a collection of algorithms is presented which take advantage of *method dependence* information.

Hakimzadeh and Perrizo [HP93] modify the ROLL concurrency control scheme of Perrizo [Per91] to produce OC-ROLL (Object Centered ROLL). Their approach provides object-local scheduling at the attribute level and utilizes statically derived bit-strings which summarize attribute access information in much the same way as Graham, *et al* [GZB92]. Inter-object serializability is provided by strictly ordering transaction method invocations by the order of arrival of the transactions in the system. While *order preservation* simplifies the problem of inter-object serializability, it may also decrease potential concurrency.

Malta and Martinez [MM93] describe an approach to *relatively* fine granularity concurrency control in object bases. They too derive attribute access information statically and use it dynamically to increase concurrency. Using the information derived during compilation, additional lock modes are defined which incorporate *semantic* information into the concurrency control process. Their algorithm does not support method invocations from within methods and thereby precludes the benefits of nested objects.

Finally, Zapp and Barker [ZB93c, ZB93a, ZB93b] describe a model, architecture, and concurrency control algorithm for closed nested transactions in object bases which furthers the work of Hadzilacos and Hadzilacos. This is the starting point for the work presented in this paper. Their algorithm provides object-local serialization of transaction operations and global, inter-object serialization of transactions as a whole. Serialization errors may occur when object-local serialization orders for two transactions conflict either directly or indirectly. Such errors are detected by the global scheduler when the object-local schedulers report their selected serialization orders. Serialization errors result in roll-back and re-execution of at least some part of the transactions involved[2]. Once serialization orders are established globally they are enforced by blocking entire transactions. Blocking entire transactions rather than just their conflicting components (i.e. sub-transactions) decreases potential concurrency.

Our algorithm differs from that of Zapp and Barker in that static analysis is used to determine, *à priori*, when transaction conflicts definitely will, and possibly may, occur. This permits the global scheduler to instruct object-local schedulers to follow specific serialization orders for the operations of particular transactions. The application of such information enhances concurrency by allowing the selection of reasonable serialization orders and decreases overhead due to unnecessary roll backs. Additionally, only conflicting operations need to be postponed order to ensure serializability and this is done locally at each object.

3 The Model

3.1 Fundamental Concepts

In our model, an object base consists of a set of uniquely identified persistent objects that each contain structural and behavioural components. The structural component is a set of uniquely identified data items referred to as *attributes* whose values define the object's state. The behavioural component is a set of procedures, usually called *methods*, that are the only means of accessing the structural components and thereby modifying the object's state. This paper denotes the j^{th} attribute of object O_i as a_{ij}. Similarly, an object's method(s) are identified using the notation m_{ij}. The set of attributes and methods are specified in a *class* definition which is used to instantiate objects.

Definition 1 An *object* $O_i = (S_i, B_i)$ where:

1. i is the unique identifier/name of the object,

[2]As described by Zapp and Barker, *entire* transactions are rolled back.

2. S_i is the object's structure which is composed of attributes a_{ij},
3. B_i is the object's behaviour which is composed of methods m_{ij}. ■

Definition 2 An *object base* $\mathcal{OB} = \{O_1, O_2, ..., O_k\}$ is a set of uniquely identified objects upon which method invocations may be made. ■

Object base users execute transactions on objects by invoking methods that manipulate their attributes. We denote an invocation of method k on object j made by a transaction T^i by m^i_{jk}. For simplicity we assume that a user transaction consists of only a single object method invocation. A method may invoke many other object methods to accomplish work of arbitrary complexity.

Objects may be nested (either explicitly by means of encapsulation or implicitly by object reference). A method in one object may invoke a method in the same or another object. This results in a nested transaction model where method invocations made from within methods are executed as *sub*-transactions.

In what follows, we denote the set of all the operations (i.e. direct of indirect method invocations) of transaction T^i by OS^i (the transaction's Operation Set). We denote by N^i the transaction's termination condition ($N^i \in \{Commit, Abort\}$).

Definition 3 A *nested object transaction* is a partial order $T^i = (\Sigma^i, \prec^i)$ where:

1. $\Sigma^i = OS^i \cup \{N^i\}$,
2. for any two $m^i_{jk}, m^i_{jl} \in OS^i$ which conflict, either $m^i_{jk} \prec^i m^i_{jl}$ or $m^i_{jl} \prec^i m^i_{jk}$,
3. $\forall m^i_{jk} \in OS^i, m^i_{jk} \prec^i N^i$,
4. the termination conditions of all $m^i_{jk} \in OS^i$ are consistent and equal to N^i. ■

3.2 Correctness

The definition of a nested object transaction captures the *structure* of a single transaction execution. It does not address the question of which concurrent executions of a transaction or set of transactions are semantically valid. This problem is addressed by the development of a *correctness criterion*. The correctness criterion in this paper is *serializability*.

Abstractly, a concurrent execution of a set of transactions is serializable if the effects of its execution on the object base are equivalent to the effects of the same set of transactions executed in *some* serial order.

The traditional forms of serializability based on reads and writes of data items in a flat database are inadequate for application in an object base with nested objects and transactions. We thus define intra-transaction and inter-transaction serializability which correspond directly to intra-transaction and inter-transaction concurrency. Intra-transaction concurrency arises from the concurrent execution of sub-transactions. Inter-transaction concurrency occurs when more than one user transaction (and their sub-transactions) execute concurrently with one another.

Correctness based on conventional serializability may be reasoned about using transaction execution histories. This is also true of intra-transaction and inter-transaction serializability. We now define histories in our model.

Definition 4 A *transaction execution history* for a set of transactions $\mathcal{T} = \{T^1, T^2, ..., T^n\}$ is a partial order $H^{\mathcal{T}} = (\Sigma^{\mathcal{T}}, \prec^{\mathcal{T}})$ where:

1. $\Sigma^{\mathcal{T}} = \bigcup_{1 \leq i \leq n} \Sigma^i$,
2. $\prec^{\mathcal{T}} \supseteq \prec^i \ where 1 \leq i \leq n$,
3. for any two $m^i_{jk}, m^p_{jr} \in \Sigma^{\mathcal{T}}, i \neq p$ which conflict, either $m^i_{jk} \prec^{\mathcal{T}} m^p_{jr}$ or $m^p_{jr} \prec^{\mathcal{T}} m^i_{jk}$. ■

Definition 5 A transaction execution history for a set of transactions $\mathcal{T} = \{T^1, T^2, ..., T^n\}$ is *serial* iff, $\forall m^i_{jk}, m^p_{jr} \in \Sigma^{\mathcal{T}}, i \neq p$, if $m^i_{jk} \prec^{\mathcal{T}} m^p_{jr}$ then $\forall m^i_{lm}, m^p_{lt} \in \Sigma^{\mathcal{T}}, m^i_{lm} \prec^{\mathcal{T}} m^p_{lt}$. ■

Definition 6 Two histories are *equivalent* if they are over the same set of transactions and they order conflicting operations identically. ■

Since we are using a nested transaction model, the definitions of histories apply to multiple user transactions and to sub-transactions of the same user transaction. This is not true of serializability.

Inter-transaction serializability is analagous to conventional serializability in that we are concerned about ordering conflicting operations between user transactions so the resulting history of the transactions' execution will be equivalent to the history of *some* serial execution of the same set of transactions.

Definition 7 An execution of a set of transactions $\mathcal{T} = \{T^1, T^2, ..., T^n\}$ is *inter-transaction serializable* iff its history is equivalent to some serial history of the transactions. ■

The definition of serializability for intra-transaction (i.e. sub-transaction) concurrency is somewhat different from that of inter-transaction serializability. The fundamental difference between intra-transaction and inter-transaction serializability is that there is a *single, required* serialization order among sub-transactions arising from the same parent transaction.

Serializability means execution equivalence to a serial execution. The serial execution of a single transaction has a single, prescribed ordering between all operations in the transaction and this includes operations which result in sub-transactions. To be correct, any concurrent execution must be equivalent to that specific serial execution order. Thus, any conflicting sub-transactions must execute in the order prescribed by serial execution.

Definition 8 An execution of the set of sub-transactions $\{ST^{i1}, ST^{i2}, ..., ST^{in}\}$ of some transaction T^i is *intra-transaction serializable* iff its history is equivalent to the serial history of the transaction T^i. ■

In the presence of both intra-transaction and inter-transaction concurrency, correct executions must be both intra-transaction and inter-transaction serializable. This paper focuses on ensuring inter-transaction serializability.

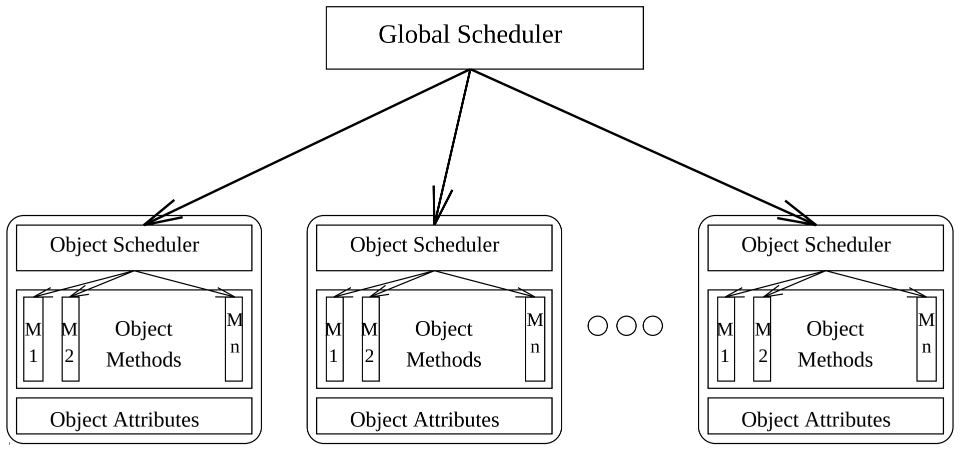

Figure 1: Two Level Scheduling

3.3 Assumptions

This paper assumes a "vanilla-flavoured" model of objects. For the most part, objects are treated as being no more than abstract data types. The fundamental issues which distinguish objects from ADTs are either not directly relevant to the work presented (e.g. inheritance) or are ignored (e.g. dynamic polymorphism). The term "object-based" rather than "object-oriented" has become prevalent in the literature [BM91]. The reader may choose to see this as work dealing with *object-based* environments.

A fundamental premise of this work is that concurrency control should be distributed across objects as suggested by Hadzilacos and Hadzilacos [HH91]. Full distribution appears to be impractical and as such, this paper adopts a "two-level" approach to concurrency control. Each object provides local serializability while global serializability is provided at a higher level. This structure is illustrated in Figure 1.

While we recognize that recovery and concurrency control are intimately linked, it is beyond the scope of this paper to discuss issues related to recovery.

4 Static Analysis

In this section the derivation of useful semantic information is discussed. This includes class-specific, and hence object-specific, information as well as information about inter-object relationships.

4.1 Capturing Object-Local Semantics

Since only conflicting operations need to be serialized and since serializability is the correctness criterion for our approach, the object method semantic information of interest is the *method-conflict* relationship. Providing concurrency control on a per-method basis is natural (methods are object "operations") and provides a compromise between the coarse granularity of object-level concurrency control and the high overhead of attribute level concurrency control.

We choose to define the method conflict relation using static method analysis at class compile time. While this is only one of several possible approaches (including the use of programmer specified commutativity tables [Wei89]) it is simple and *automatic*.

Conflicts between methods are defined using attribute-level conflicts as a basis. If two methods contain operations which access object-local data in a conflicting manner then the two methods are defined to conflict. The advantages of pre-analyzing methods to determine method level conflicts over the simple application of attribute level locking are:

1. Decreased locking overhead, particularly for large objects containing many attributes.

2. Potentially greater concurrency than simple object-level locking since non-conflicting methods are allowed to execute concurrently.

Definition 9 The *attribute read set* ($\mathcal{ARS}$) of method m_{ij} (in object O_i) consists of all those attributes a_{ix} which may be read by m_{ij} during its execution. ■

Definition 10 The *attribute write set* ($\mathcal{AWS}$) of method m_{ij} (in object O_i) consists of all those attributes a_{ix} which may be written by m_{ij} during its execution. ■

The definition of both the attribute read and write sets are "conservative" since compile time analysis is imperfect. The sets are potentially supersets of the attributes that will actually be referenced in any given execution of the given method. Inaccuracy cannot be avoided due ot the existence of conditionally executed code (i.e. `IF`, `WHILE`, etc. statements) where the conditions are not compile time evaluatable.

Definition 11 Two object methods m_{ij} and m_{ik} are said to *conflict* if any of the following three conditions are true:

$$\begin{gathered}\mathcal{ARS}(m_{ij}) \cap \mathcal{AWS}(m_{ik}) \neq \emptyset \\ \mathcal{AWS}(m_{ij}) \cap \mathcal{ARS}(m_{ik}) \neq \emptyset \\ \mathcal{AWS}(m_{ij}) \cap \mathcal{AWS}(m_{ik}) \neq \emptyset\end{gathered}$$

■

We assume the existence of a function $Conflicts(m_{ij}, m_{ik})$ which returns "TRUE" if methods j and k of object O_i conflict and which returns "FALSE" otherwise. The static enumeration of $Conflicts(m_{ij}, m_{ik})$ for all method pairs in an object permits the construction of a method compatibility matrix. Malta and Martinez [MM93] use this information to develop enhanced lock modes.

4.2 Capturing Inter-Object Relationships

The fundamental inter-object relationship of interest to concurrency control is the *part-of* (aggregation) relationship. Logically, any object referenced from another object is a "part-of" the first. This is true whether or not the sub-object is actually compiled as a *physical* part of the object. Thus the part-of relationship is captured by the inter-object calling relationship which can be represented by an object call graph.

Definition 12 The *object call graph* of method m_{ij} is a directed graph $\mathcal{OCG}(m_{ij}) = (V, E)$ containing a root vertex $v_{root} \in V$ corresponding to m_{ij} and directed edges $e \in E$ from v_{root} to those vertices $v_x \in V$ which are the root nodes of the object call graphs corresponding to the methods invoked by m_{ij}. ■

The $\mathcal{OCG}$ is most easily understood by considering it to be a threading of the objects in the object base in a way which reflects potential inter-object calls.

Logically traversing the object call graph permits the construction of sets of referenced object methods for each object. This gives rise to the following definition of the set of methods referenced (i.e. called) by an object method.

Definition 13 The *object method reference set* of method m_{ij} is:
$$\mathcal{OMRS}(m_{ij}) = \bigcup_{\forall m_{xy} \ referenceable \ by \ m_{ij}} \mathcal{OMRS}(m_{xy}). \qquad \blacksquare$$

The construction of the object method reference set for a given object method cannot generally be performed at compile time. Instead the construction of the $\mathcal{OMRS}$ must be performed at object instantiation time. While not as desirable as compile time analysis, analysis at object instantiation time is still prior to object access time. Furthermore, object instantiation takes place only once and is typically followed by many object accesses. Thus, any overhead incurred at instantiation time is amortized over the object accesses. Finally, the cost of the needed instantiation time analysis is minimal since objects must be declared before they are used (an object cannot be referenced until it exists). This means that the construction of the $\mathcal{OMRS}$ for an object is a single step process. As suggested by Definition 13 there is no need to search down chains of object references in the conceptual $\mathcal{OCG}$ since object methods which are referenced *indirectly* by the object method being analyzed are already summarized in the $\mathcal{OMRS}$s of the object methods referenced directly.

The calculation of the "static" information needed for our technique is either done at compile time when great effort can be expended freely or is done at instantiation time with little overhead.

5 Scheduling

This section discusses the scheduling of object transactions using static information to enhance concurrency. Due to space limitations, it does not specify an algorithm formally. It only provides a narrative describing the key concepts in such an algorithm.

The goal of exploiting statically derived semantic information is to increase the run time knowledge of transaction behaviour to enhance concurrency and reduce overhead. Within a single transaction, the application of static analysis to guide the concurrent execution of sub-transactions is ideal. A single transaction is completely self-contained and static analysis of the *entire* set of sub-transaction can be easily performed prior to run time. Dealing with *multiple* concurrently executing transactions is more problematic.

The set of transactions which will be concurrently executed at any time is unpredictable. Since we cannot pre-determine the transaction mix, we cannot statically pre-analyze the possible transaction *interactions*. We can however record information for each transaction and then analyze the information at run time to determine interactions. Our approach applies the statically derived information dynamically to select a "good" serialization order given only the transaction being scheduled and the set of currently executing transactions.

Our technique applies the derived object-local semantic information within the object schedulers and the inter-object semantic information in the global scheduler. The application of the object-local semantic information serves to decrease the detection of false conflicts through the use of method-level conflict analysis. The application of the inter-object information permits the selection of a reasonable serialization order which is then followed by *all* object schedulers to ensure serializability. The selection of a good serialization order (1)

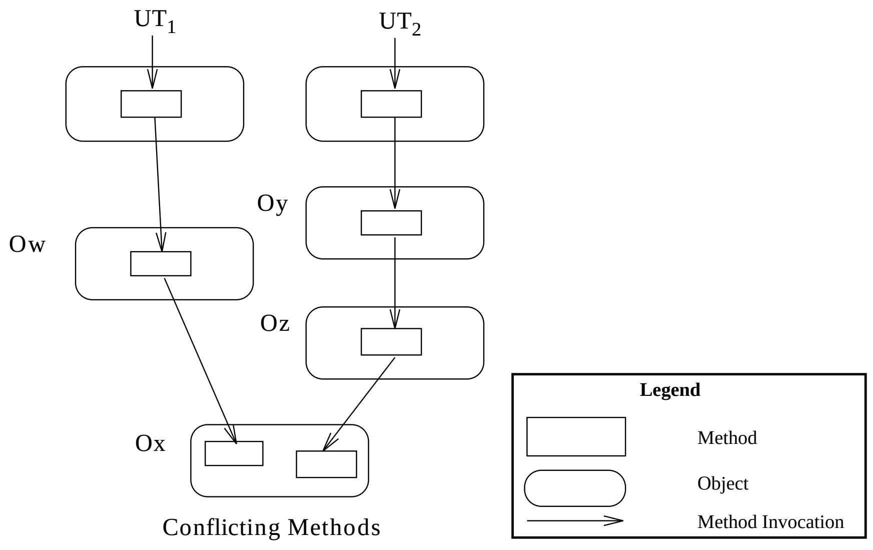

Figure 2: Sub Transaction Concurrency

enhances concurrency since unnecessary conflicts can be avoided, (2) decreases overhead by avoiding unnecessary rollbacks and by avoiding scheduling overhead for all but conflicting operations and, and (3) precludes deadlock by explicitly avoiding situations giving rise to it.

Zapp and Barker [ZB93c, ZB93a, ZB93b] describe a two-level concurrency control algorithm with enforcement of serializability based on access to an object's attributes done locally at each object by an object scheduler[3]. Ensuring inter-object serializability requires each object scheduler to report its selected serialization orders to a global (*method*) scheduler which maintains a global graph of inter-transaction orderings (the *DAX*). When two transactions access multiple common objects in conflicting ways, then inter-object serialization may be violated. Since serialization orders are determined locally at each object, conflicting orders may be determined at different objects. In this case, conflicting orders will be reported to the global scheduler and the serialization error will be detected. Once detected, one of the transactions will be rolled back and re-executed.

The DAX is always maintained in an acyclic form. When a new ordering is indicated by an object local scheduler, the arc is *conditionally* added to the DAX. A check for acyclicity is then performed and if a cycle is found, the conditional edge is removed (and the corresponding transaction will be rolled back). The extension to the algorithm we propose in this paper is to use the derived static information to predict ahead of time where conflicts will arise and determine what orderings may result in a cycle in the DAX. Undesirable orderings (those leading to serialization errors or deadlocks) are avoided by propogating *required* serialization orders to the object local schedulers where conflicts will occur. This approach is better than simply blocking entire transactions whose execution *may* result in errors because non-conflicting sub-transaction concurrency may still be exploited. Consider the calling sequence in Figure 2 where blocking UT_2 due to the conflict at O_x would result in lost sub-transaction concurrency between O_w and O_y and between O_w and O_z.

The advantages of our approach to concurrency control are clear. The effectiveness of the approach hinges on its efficient implementation. No benefit will be gained if the cost

[3]The use of conventional two-phase locking is suggested.

Initial $\mathcal{ATRS}$	$\{O_1, O_1, O_4, O_{7211}\}$
New Transaction's $\mathcal{OMRS}$	$+ \{O_4, O_{77}\}$
Final $\mathcal{ATRS}$	$= \{O_1, O_1, O_4, O_4, O_{77}, O_{7211}\}$

Figure 3: Adding an $\mathcal{OMRS}$ to the $\mathcal{ATRS}$

of using the static information outweighs the benefits of decreased overhead and increased concurrency.

To be able to analyze the potential interactions of the transaction being scheduled with the currently active transactions, a data structure must be maintained which captures object accesses by each transaction and vice versa. Since this data structure will be queried and updated frequently, it must be time efficient. Complicating this goal is the fact that the set of the objects accessed by any transaction is small compared to the size of the entire object base and is also likely to be sparse. Further, since different transactions may access vastly different sets of objects, the sets are likely to be close to disjoint for each pair of transactions.

To pre-detect potential conflicts, for each active transaction, the global scheduler must know which objects it will access. This information is captured by the $\mathcal{OMRS}$ for each transaction. For each object, the global scheduler must also know which transactions (or at least *how many* transactions) are accessing it. This information cannot be captured simply by the $\mathcal{OMRS}$s or any simple set amalgam of them. What is required is a multi-set (a set where a single element may occur multiple times) which summarizes transaction accesses to objects. The efficient implementation of a multi-set using a hashing scheme is straightforward.

Definition 14 The *active transaction reference set* $\mathcal{ATRS}$ is a multi-set containing elements representing objects such that for each active transaction accessing some object O_x, the element corresponding to O_x occurs in the multi-set once. ■

To maintain the $\mathcal{ATRS}$ so that it always accurately represents the current object accesses for the active transactions, steps must be taken when transactions are initially scheduled and when they terminate. When a transaction is scheduled, its $\mathcal{OMRS}$ (a degenerate multi-set where each element occurs only once) must be "added" to the $\mathcal{ATRS}$. When a transaction commits or aborts, its $\mathcal{OMRS}$ must be "subtracted" from the $\mathcal{ATRS}$. The addition of multi-sets is illustrated in Figure 3.

When a new transaction begins, its $\mathcal{OMRS}$ may be intersected with the $\mathcal{ATRS}$ and thethe result determines whether or not the new transaction may conflict with any of the active ones. New transactions that do not conflict may be scheduled immediately without further checking. Those that may conflict require *speculative* DAX checking to ensure that they will not cause deadlock and to possibly determine an appropriate serialization order with other transactions at those objects where conflicts may occur.

Determining possible serialization violations may be accomplished by *conditionally* adding arcs to the DAX and performing cycle detection as is done when a new serialization order is determined from an object scheduler in the original algorithm. This process is referred to as *speculative* DAX checking. As long as there are relatively few objects at which conflicts may occur (the expected case), exhaustive testing of all possible serialization orders is feasible. The process of determining *which* transactions conflict with the new

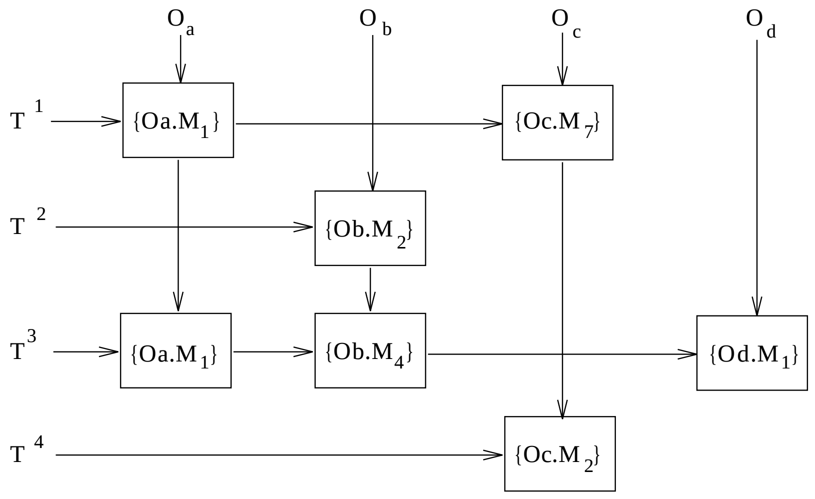

Figure 4: Scheduling Data Structure

transaction requires knowledge of which objects are in conflict and, for each such object, which other transactions are accessing it. These requirements and the required multi-set operations may be met by a simple scheduling structure composed of $\{object\_id, method\_id\}$ pairs which are threaded both by object and transaction (see Figure 4).

A decided benefit of scheduling *à priori* to avoid serialization errors is the impossibility of deadlock. In our approach, locking is *not* used as it is in the original algorithm. Object schedulers schedule methods based on method-level conflicts, allowing non-conflicting method executions to proceed concurrently and imposing an order on conflicting ones. This is subject to inter-object serialization orders imposed by the global scheduler to avoid rollback, etc. Since there is no waiting with held resources, only re-ordering, and because serializability precludes the necessary “circular wait” condition, deadlock cannot occur.

When the global scheduler decides that a prescribed serialization order is beneficial, it must ensure that the selected order is adhered to wherever the relevant transactions conflict. To accomplish this, the global scheduler and the object schedulers must communicate. Such communication is already a part of the original algorithm although it is not used to prescribe serialization orders.

In the simplest case, a prescribed serialization order will indicate that the new transaction must be serialized after an existing one. This is easily enforced since the global scheduler can send ordering information *before* actually issuing the object method invocations. A more interesting situation arises when the new transaction must serialize before an already active transaction. In this case, due to message latency, it is possible that the global scheduler will send a message prescribing an order that can no longer be met (because a conflicting transaction operation has already been performed). In this case, *partial* rollback (of only the *known* conflicting operations) may still be required. If the conflicting active transaction has not yet executed the conflicting operation, then no error occurs and the desired ordering is achieved as long as the object schedulers always process incoming serialization-order messages before scheduling new local operations.

Another complication arises because of the conservative nature of the static information. In the presence of non-compile-time evaluatable conditionals, the $\mathcal{OMRS}$s are constructed to reflect *all* possible methods being invoked. Of course, at run time, not all such methods may be executed. This means that a prescribed ordering between an operation from one transaction and a non-executed operation from another transaction may be sent by the global scheduler to some object scheduler. If the prescribed ordering serializes the operation which will never be executed first then the second operation will wait indefinitely for the first. To avoid this problem, the generated code for methods containing conditional method invocations may be easily augmented to send "non-execute" messages to relevant object schedulers in the event that the invocations are not made. Such messages should also be sent to the global scheduler so it may update the $\mathcal{ATRS}$ to reflect the new set of fewer accesses made by the transaction. This may lead to less-restrictive concurrency control.

This paper has been primarily concerned with inter-transaction serializability. It is also assumed that intra-transaction serializability is maintained through mechanisms beyond the scope of this paper (See Graham and Barker [GB94] for details).

6 Conclusions and Future Work

We have presented an approach to scheduling nested transactions in object bases which extends and improves the work of Zapp and Barker. Our goals were threefold; to increase concurrency, to decrease concurrency control overhead, and to avoid undesirable side-effects related to concurrency control.

Concurrency is increased over conventional object-level locking strategies by refining conflicts to the method level. This offers a compromise between the conflicting goals of maximizing concurrency and minimizing overhead that is not achieved by attribute level locking. Our method conflict criterion is based on *simple* semantics which are derivable *automatically* by the class method compiler. (Support for nested transactions is also automated by the compiler. This ensures the simplest possible programming environment for the transaction programmer.) Concurrency may also be increased due to the intelligent selection of serialization orders. This was not discussed in detail in the paper since benefits and liabilities are difficult to assess without a working prototype and well defined application environment.

Concurrency control overhead is reduced by shifting some of the effort to compile time and object instantiation time when it can be better tolerated. It is also reduced since our approach incurs overhead only when it is necessary. Unlike locking schemes which incur overhead (lock acquisition and release) for all data items, our approach incurs overhead (message sending) only when it is known that conflicts may otherwise occur. Concurrency control overhead is also reduced in the handling of rollback in the *infrequent* cases where it is required. The availability of *à priori* knowledge of where (at which objects) conflicts between transactions will occur permits rollback and re-execution to be easily performed on a *sub-transaction* basis. Rather than losing the work of an entire transaction, only the work of its conflicting sub-transactions is lost.

Finally, the fundamental concurrency control anomaly, deadlock, is precluded when using our scheme. To the best of our knowledge, this has only been provided before in algorithms which were strictly order-preserving. Order preservation has the potential consequence of decreased concurrency which our scheme does not.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40 – 57, 1989.

[BM91] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4):33 – 47, 1991.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64 – 77, 1991.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufmann, 1990.

[Deu90] O. Deux *et al*. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91 – 108, 1990.

[GB94] P.C.J. Graham and K. Barker. Intra Transaction Concurrency in Object Bases. In *Proceedings of the International Conference on Computers and Information*, Peterborough, Canada, May 1994. *Accepted*.

[GZB92] P.C.J. Graham, M.E. Zapp, and K. Barker. Applying Method Data Dependence to Transactions in Object Bases. Technical Report TR 92-7, University of Manitoba, Dept. of Computer Science, 1992.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2 – 24, 1991.

[HK89] S.E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, 14(3):291 – 321, 1989.

[HP93] H. Hakimzadeh and W. Perrizo. Instance Variable Access Locking for Object-Oriented Databases. *International Journal for Micro and Mini Computer Applications*, 1993. *in print*.

[HPC93] A.R. Hurson, S.H. Pakzad, and J.-B. Cheng. Object-Oriented Database Management Systems: Evolution and Performance Issues. *IEEE Computer*, 26(2):48 – 60, 1993.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109 – 124, 1990.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50 – 63, 1991.

[MM93] C. Malta and J. Martinez. Automating Fine Concurrency Control in Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pages 253 – 260, 1993.

[Mos85] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[Per91] W. Perrizo. Request Order Linked List (ROLL): A Concurrency Control Object for Centralized and Distributed Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 278 – 285, 1991.

[RE92] R.F. Resende and A. El Abbadi. A Graph Testing Concurrency Control Protocol for Object Bases. In *Proceedings of the 4th International Conference on Computing and Information*, pages 316 – 319, 1992.

[Ree78] R. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Dept. of Computer Science, Massachusetts Institute of Technology, 1978.

[RGN90] T.C. Rakow, J. Gu, and E.J. Neuhold. Serializability in Object-Oriented Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 112 – 120, 1990.

[TR93] T. Härder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39 – 74, January 1993.

[Wei88] W.E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488 – 1505, 1988.

[Wei89] W.E. Weihl. Theory of Nested Transactions. In S. Mullender, editor, *Distributed Systems*, chapter 12, pages 237 – 262. ACM Press, 1989.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implemntation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63 – 75, 1990.

[ZB93a] M.E. Zapp and K. Barker. Modular Concurrency Control Algorithms for Object Bases. In *International Symposium on Applied Computing: Research and Applications in Software Engineering, Databases, and Distributed Systems*, pages 28–36, Monterrey, Mexico, October 1993.

[ZB93b] M.E. Zapp and K. Barker. On Concurrency Control in Object Bases. In *Mid-Continent Information Systems Conference (MISC'93)*, pages 91–97, Fargo, USA, May 1993.

[ZB93c] M.E. Zapp and K. Barker. The Serializability of Transaction in Object Bases. In *Proceedings of the International Conference on Computers and Information*, pages 428–432, Sudbury, Canada, May 1993.