Semantic Synchronization in a Persistent Object System Library

Andrea H. Skarra and Naser S. Barghouti

Software and Systems Research Laboratory
AT&T Bell Laboratories
Murray Hill, NJ 07974 USA

({ahs,naser}@research.att.com)

Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

(alw@cs.colorado.edu)

University of Colorado
Department of Computer Science
Technical Report CU-CS-711-94 April 1994

A version of this report to appear in
The Proceedings of the 6th International Workshop
on Persistent Object Systems (POS6), September 1994

ABSTRACT

This paper describes a synchronization scheme that exploits the semantics of collections and indexes to improve concurrency in a persistent object system library. The library implements a common execution model in which data (rather than operations or queries) are shipped between a transaction and the database at a granularity no smaller than an object. The paper describes the impact of the execution model on the extent to which semantics can be used in synchronization, and it develops several techniques that yield higher concurrency levels in this common though restrictive model. Finally, it examines the process and tradeoffs involved in designing a semantic synchronization protocol.

1 Introduction

Applications typically interact with a database in the context of transactions to maintain consistency. In particular, a transaction is a sequence of operations that satisfies all consistency constraints on a database, and the database management system (DBMS) synchronizes concurrent transactions to produce an interleaved history that is *serializable* (i.e., equivalent to some serial execution of the transactions). A database remains consistent across repeated and concurrent access when an application uses transactions.

Most commonly, a DBMS uses a two-phase locking protocol to enforce serializability. Each transaction requests locks according to the protocol, and the DBMS evaluates each request for conflict with the locks held by other transactions. It grants requests that do not conflict, it delays requests that do conflict, and it denies requests whose delay would result in deadlock. A traditional protocol uses just the two lock types Read (Share) and Write (Exclusive) with the usual conflict semantics. It associates every data access with one or the other of the locks automatically, regardless of the kind of data or the context. If a DBMS instead uses a synchronization protocol that exploits the semantics of the data and the application, however, the number of transactions that can execute concurrently increases [14].

A semantic protocol and its lock types may be "hardwired" into the DBMS synchronization algorithm. For example, the multigranularity locking protocol [1, 7] employs a fixed set of five lock types that capture the access semantics of physically nested objects such as records within files. A DBMS that supports the protocol may implement just those lock types and no others.

Alternatively, the DBMS may support semantic protocols with an *extensible* synchronization algorithm, namely one whose behavior is modified by semantic specifications that it takes as input from the database administrator. The form of the specification is typically a compatibility matrix over a set of transaction types [5, 16] or a set of synchronization primitives such as events, semantic lock types, or abstract type operations [9, 13, 25]. The synchronization algorithm uses the specifications to decide conflict among concurrent transactions in much the same way that a traditional algorithm uses the conflict relation over Read and Write locks.

The correctness criterion in semantic protocols that corresponds to serializability is *semantic serializability.* A concurrent transaction history satisfies the criterion when its effect is equivalent to that produced by some serial execution of the transactions *as discernible through the data's functional interface.*

For example, a data type Coll_Auto (collection of automobiles) defines the following operations:

Add(a:Auto):OK inserts a.

Rem(a:Auto):OK deletes a.

List():Sorted_Auto_List returns the automobile identifiers in numerical order.

Member(a:Auto):Boolean returns true if a is a member; otherwise false.

Under semantic serializability, transactions that add automobiles to a Coll_Auto object can be arbitrarily interleaved. The physical representation of the resulting collection may be different from any produced by a serial execution of the transactions, but the difference is not visible through the interface. List() returns a sorted list that hides the insertion order.

We designed and implemented a synchronization protocol that guarantees semantic serializability in Persi [26], a persistent object system library that supports a transaction execution model commonly used in object-oriented database systems (OODBs). The objective of the protocol is to maintain consistency across the database while maximizing concurrency among transactions, whether they navigate to objects or use associative retrieval. The contribution of the work is that it develops semantics-based techniques that improve concurrency in a common though restrictive transaction execution model. In addition, it highlights the dependence of a semantic protocol on the underlying execution model.

We first describe the transaction execution model in Persi and other OODBs, and we compare it to those that are required by the currently proposed semantic synchronization schemes. We explain why the model restricts the extent to which a synchronization algorithm can use semantics to improve concurrency. We then develop several synchronization techniques that exploit the semantics of collections and indexes to an extent, and in a manner, that is compatible with the execution model. We describe the relevant portions of Persi, and we detail the synchronization protocol within a standard library interface. We then illustrate our approach with an example. Finally, we conclude with a discussion about the process of designing a semantic scheme and the tradeoffs that are involved.

2 Execution models

Persi implements a common execution model: a transaction's operations execute within the private address space of the process that contains the transaction. That is, a transaction does not ship operations or queries to a database execution engine and await the result. Rather, it acquires a copy of the data from the database system and performs the computations itself. Moreover, the unit of data transfer between a transaction and the database is at least the size of an object; a transaction does not read or write individual attributes alone. The commercial object-oriented DBMSs [4, 15, 19, 21, 24] support the same model, and henceforth we call it the *OO execution model.*

In contrast, the currently proposed models of semantic concurrency control (*cf.* Section 7) imply one of the following two execution models:

- *Finely-grained Read/Write access*

  The DBMS supports Read and Write access to small portions of the database. As a result, transactions can lock smaller regions and execute with greater concurrency.

  In the multigranularity protocol, for example, a transaction can read and write individual tuples in a relation. It does not have to write an entire relation in order to insert a single tuple. Thus, concurrent transactions can lock slots in disjoint segments of the same relation and insert tuples without risk of overwriting each other.

- *Encapsulated shared memory (ESM)*

The DBMS supports a single shared copy of each object, and it allows transactions to access an object only through the abstract interface. A transaction does not cache objects in its private address space; operations on an object execute only on the shared copy. ESM may be viewed as a generalization of finely-grained Read/Write access. An operation typically touches only part of an object, and it does so with well-defined semantics that a synchronization scheme can use to improve concurrency.

In a client-server setting, the DBMS implements ESM with an object server that executes operations on behalf of transactions. Transactions send operations to the server, and the server returns the results. All the operations on an object execute in the server on the same physical object. In other settings, a DBMS can implement ESM using the shared memory primitives provided by the operating system. Transactions load the interface operations for each object they access, and they execute the operations on the objects in shared memory under a suitable access protocol (e.g., one that preserves operation atomicity).

To illustrate, consider an object c of type Coll_Auto and transactions T_1 and T_2 that insert automobiles into c. Suppose c contains (auto1) initially, T_1 adds auto2 and auto3, and T_2 adds auto3 and auto4. In a serial execution, c contains (auto1,auto2,auto3,auto3,auto4) after T_1 and T_2 commit (in either order).

In a concurrent execution, T_1 and T_2 can be arbitrarily interleaved under semantic serializability. If the execution model supports finely-grained Read/Write access to the slots in c, the insertions can proceed independently at different slots, and regardless of the interleaving, c is correct after the transactions commit. Alternatively, if c resides in ESM and each Add() operation is atomic, c also ends up with the correct membership for any interleaving.

In contrast, transactions cannot add automobiles to the same Coll_Auto object concurrently under the OO execution model. Transactions execute with private copies of an object, and thus, an arbitrary interleaving of T_1 and T_2 may yield an incorrect result. Specifically, if T_1 and T_2 begin execution and fetch c at the same time, each receives the initial version of c in its entirety. Each then does its computation in isolation, and if the DBMS allows both to commit, c contains either (auto1,auto2,auto3) or (auto1,auto3,auto4) instead of the correct value. One transaction overwrites the other.

We conclude that this common execution model restricts the extent to which semantic synchronization can be implemented. In particular, it restricts the data access semantics available to the synchronization algorithm to simply Read and Write on the finest-grained access unit, where the unit is no smaller than an object. Transactions cannot invoke abstract type operations on fragments of an object. Thus, a multigranularity locking protocol cannot be used to improve concurrency at a coarse-grained object (unlike using the protocol on a relation and its tuples), since a change to any part of an object requires writing the whole object back to the database with a Write lock. Semantic serializability reduces to standard Read/Write serializability in the OO execution model.

3 A semantic approach

We propose a synchronization scheme that uses two techniques to improve concurrency within the OO execution model: *delay* and *re-fetch*.

Intuitively, an operation's execution can be delayed (along with the acquisition of the associated locks) when its return value does not depend on the object's initial state. The operation simply returns the value immediately, and the transaction's execution continues on other objects. A delayed operation executes when its effect is required for a subsequent operation in the transaction or for commit. The delay shortens the interval during which the transaction holds a lock on the object, and thus it improves concurrency; other transactions can commit changes during the delay interval.

For example, the Coll_Auto operation Add() can be delayed, because it always returns the value OK to the invoking transaction. When the transaction commits or invokes another operation on the object that cannot be delayed, such as List() or Member(), any and all the delayed Add()s execute before the transaction proceeds. Using the delay technique, T_1 and T_2 delay their Add()s until commit, when each in turn fetches c with a Write lock, performs the insertions, and writes the resulting c to the database.

In the other technique, a transaction (re)fetches an object o before every operation on o to incorporate (nonconflicting) changes that other transactions have committed. It also fetches o before commit if it modifies o. The re-fetch technique simulates encapsulated shared memory for committed results, improving concurrency while preventing the occurrence of overwrites that are due to the object-level access granularity. To use re-fetch on an object o, a transaction maintains a queue of its modifying actions on o, and each time it fetches o, it applies the actions locally before proceeding. A transaction's intermediate changes remain local to the transaction to simplify abort/undo.

For example, transaction T_3 with the following operations on Coll_Auto object c uses delay and re-fetch and executes concurrently with T_1 and T_2:

Add(auto5) Member(auto2) Member(auto4)

At Add(auto5), T_3 delays the operation and adds it to the queue for c. At Member(auto2), T_3 executes the queued Add() (i.e., it fetches c and inserts auto5), and it invokes Member().[1] If T_1 has committed but not T_2, Member() returns true, and c does not contain auto4. At Member(auto4), T_3 again fetches c and inserts auto5 before it invokes Member(). If T_2 has committed by then, Member() again returns true, because c now contains auto4. When T_3 commits, it fetches c with a Write lock, inserts auto5, and writes c to the database.

The final version of c reflects the actions of all three transactions, even though T_3 initially fetches c before T_2, and it commits afterward. It does not overwrite T_2. If T_3 does not use re-fetch, however, it avoids overwriting T_2 only by getting a lock when it first fetches c. The lock forestalls T_2's commit until after T_3 terminates and reduces concurrency.

[1]For simplicity, we omit the locking for Member(), since none is required for correctness in the example.

The technique chosen to synchronize transactions at a particular object depends on the object and the anticipated access patterns. Re-fetching supports more concurrency and may be preferred for frequently accessed objects. For large objects, however, re-fetching may be prohibitively expensive. The synchronization scheme we developed in Persi uses delay for both collections and indexes. It uses re-fetch for indexes alone, however, due to the frequency of index use and the anticipated size of collections.

Importantly, both delay and re-fetch can be used in conjunction with a multigranularity protocol that propagates intention locks from one object to another object such as a container, even though we do not describe such a protocol for Persi.

4 An overview of Persi

Persi is a persistent object system library providing high-level abstractions for typed persistent objects, for stable-storage repositories of objects, and for (concurrent) access to repositories. In addition, Persi provides abstractions for defining and maintaining collections of objects, iterators over collections, and indexes on collections. The type model used by Persi is that of C++.

Persi is designed with two kinds of developers in mind: class developers and application developers. Class developers act as database administrators (DBAs), defining the object, collection, and iterator classes used in applications, as well as establishing the indexes. Application developers write programs that make use of the classes defined by class developers. They (and their programs) are shielded from the details of object management by the functional interfaces defined by the class developers.

Persi provides two basic methods of accessing objects, namely object navigation and collection scan. Object navigation is a directed traversal from one object to another along paths defined by object pointers and is often referred to as “pointer chasing”. Collection scan is an iterative retrieval of the members of a collection that satisfy some predicate.

The store for persistent objects in Persi is called a repository. Persi provides a transaction mechanism to logically group together the operations performed by a process on the objects in a repository. A transaction is atomic—that is, either all or none of its operations take effect. An application process can perform a sequence of one or more non-overlapping transactions on a given repository; transactions on the same repository by different processes can, of course, be concurrent. During a transaction, an application process can see its own updates to objects, including any effect those updates have on indexes.

Below, we briefly describe the features of Persi required to understand the work presented in this paper. We do this by detailing the class developer interface to Persi's features for persistence, collections, and automatic index maintenance. A more complete description of Persi can be found in [27].

4.1 Persistent objects

A persistent object is an instance of a class that is a subclass of the Persi class Object.[2] Class Object serves to define the operations used in a special, hidden protocol between Persi and persistent objects. The protocol allows Persi to automatically manage the persistence of, and access to, the objects. It is hidden in the sense that applications using subclasses of Object are not involved in the protocol. There are two protocol operations of relevance to the work described here. Both are operations on instances of class Object.

Touch() This function is used to guarantee that an object is resident in the primary memory space of an application. It is needed because Persi uses (logical) object faulting to move objects from secondary memory into primary memory.

ValueChange() This function is used to indicate the fact that a data component of a persistent object has changed value. It is needed because Persi, being a library, has no way to automatically detect when changes are made to values.

Both functions are typically invoked from within the operations of a class and are not invoked directly by applications themselves.

Object persistence is defined dynamically by reachability from a designated root (persistent) object of a repository. Therefore, not every instance of a subclass of Object necessarily persists.

4.2 Collections and iterators

Class Collection is an abstract class, derived from Object, for homogeneous collections[3] of objects, where a common superclass of the objects is itself a subclass of Object. Collection defines a special, hidden protocol between Persi and objects that are collections. This protocol allows Persi to help coordinate index maintenance invisibly and automatically for applications. Because it is an abstract class, Collection does not in and of itself collect any objects. Instead, developers of subclasses of Collection determine what objects are to be collected and the semantics of those collections (e.g., sets versus sequences). There are three protocol operations of relevance to the work described here. All are operations on instances of class Collection.

CollectionInsert(Object) is used to insert an object into a collection.

CollectionRemove(Object) is used to remove an object from a collection.

IsMemberOf(Object) returns the value true for collection members.

The first two functions must be specialized for each kind of collection class (i.e., subclass of Collection).

[2] All identifiers defined by the Persi library actually begin with the prefix Persi_. For brevity, we do not use the prefix in this paper.

[3] A collection is homogeneous only in the sense that members of that collection must have a common superclass.

Along with collections, Persi provides an abstraction for iterating over collections. The abstraction, called a *collection accessor*, captures the notion of a state of an iteration (i.e., a "cursor"). Typically, each subclass of Collection would have a corresponding collection accessor, tailored to the kind of collection defined by that subclass of Collection. The iteration is modeled as an "initiate-next-terminate" loop and any number of such iterations may be simultaneously active. Persi provides loop steps implementing both unique and non-unique index-key predicate scans.

Persi provides a large degree of flexibility in its concept of collection. In particular, objects can be in more than one collection at the same time, and there can be multiple access paths to an object, not all of which are through some collection (i.e., they may involve object navigation). This latter property means that an object's value can be changed independently of its membership in a collection, yet any indexes on that collection must be updated. Persi handles this by keeping with each object a list of the collections in which that object is a member; when an object's value is changed, then the list, called the *collection set*, is consulted to determine if any indexes need updating.

4.3 Indexes

A collection can have any number of indexes on values of its members. An index contains an entry for every object in its associated collection. Indexes in Persi use standard ordered keys and are currently implemented using B-trees. The interface to indexes is provided by the Persi library class Index, which is completely hidden from applications. There are three operations of relevance to the work described here.

Insert(Object, KeyVal) is used to insert an index key-value for an object.

Remove(Object, KeyVal) is used to remove an index key-value for an object.

InitiateQuery(KeyVal) initiates an iteration over an index for a key-value.

Persi updates the indexes on a collection (if necessary) when it is notified that the value of an object has changed (i.e., through ValueChange()) or when it is notified that membership in the collection has changed.

5 A semantic synchronization protocol

The synchronization protocol in Persi consists of the following key components, which we discuss more fully later in the section.

- The scheme defines a set of semantic lock types for collections and indexes and a protocol for requesting the locks. It uses standard Read/Write locking for instances of user-defined classes (i.e., *data objects*).

- The scheme uses delay for collection operations.

- The scheme combines delay, re-fetch, and versioning for index synchronization. It uses only latches on indexes: it locks objects that an index references rather than the index itself.
- The scheme uses the operating system (OS) as a storage manager and lock server.

The protocol is implemented entirely within Persi's interface operations. Details of the protocol, such as lock requests, delaying operations, or fetching objects from disk upon access, are hidden from the application.

5.1 Operations

When a transaction invokes CollectionInsert() or CollectionRemove() on a collection c and an object o, Persi must update c, o, and every index i defined on c. It must add/remove o to/from c's list of members, add/remove c to/from o's collection set, and add/remove o's key-value to/from i.

For each object c and o, Persi performs the insert or remove action immediately if the object is *resident* in the transaction's primary memory space. Otherwise, Persi queues the action as part of the delay technique. For each index i, Persi simply queues the action as part of re-fetch.

When a transaction changes one or more attributes in a data object o, it invokes ValueChange() on o, and Persi must update the indexes on collections containing o that are affected by the change. Specifically, for each index i where o's old key-value is different from its new key-value, Persi must remove i's old key-value and add its new one. As part of re-fetch, it simply queues the actions.

5.2 Queues

Persi maintains a queue of Insert and Remove operations for each index that a transaction modifies. For each collection, Persi maintains a queue of insert and remove actions that add or delete members (i.e., the collection's *member queue*). For each object, Persi also maintains a queue of insert and remove actions that add or delete collections to or from the object's collection set (i.e., the object's *collection queue*). Thus, a collection object potentially has two queues: a member queue for adding or deleting its members, and a collection queue for adding or deleting collections to or from its collection set. Data objects and indexes have at most one queue apiece.

The queues for different objects are managed and used independently. Thus, the insert and remove actions on collection c and member object o are decoupled: depending on which of the objects are resident, both actions can be queued (delayed), both can execute immediately, or only one may be queued (delayed). Further, the actions on indexes are decoupled from each other and from the actions on c and o. The decoupling improves concurrency.

5.3 Collections and data objects

The protocol uses object-level locking with three modes for both collections and data objects: Read, Update, and Write. The following table illustrates the lock semantics, where an entry of x signifies conflict between the lock modes:

LOCK	Read	Update	Write
Read			x
Update		x	x
Write	x	x	x

The use of Update locks reduces the occurrence of deadlock among transactions that read and then write the same object, while not blocking transactions that merely read the object. The protocol extends prior use of Update locks on flat objects [13] to its use on collections. The conflict table is symmetric, however, because we implement the lock types with standard Read and Write locks (*cf.* Section 5.5).

When Persi fetches a collection or a data object o from disk, it gets a Read lock on o.[4] If there is a collection queue of insert or remove actions pending for o and/or a member queue (if o is a collection), Persi upgrades the Read lock on o to Update, and executes the queued actions. The actions in the collection queue add or remove collections to or from o's collection set. Those in the member queue add or remove o's members. Finally, Persi removes the queue(s). Subsequent operations on the object execute immediately. If there are no queues pending upon fetch, but the transaction later invokes CollectionInsert() or CollectionRemove() on o, Persi upgrades the Read lock at that time.

If a transaction invokes ValueChange() on a data object o, Persi upgrades o's lock to a Write lock, keeping track of the upgrade so as not to inadvertently downgrade the lock to Update mode. At commit, Persi upgrades any Update locks to Write locks, and it saves all Write-locked objects to disk.

5.4 Indexes

Persi synchronizes access to an index by means of locks on the objects that the index references, rather than locks on the index itself. The scheme derives from Aries/IM [17]. For each key-value kv for which an index i has at least one entry, there is some nonempty set of objects in the associated collection, whose attributes have the values in kv. For each such set KV, the synchronization scheme designates a member of the set to be the *canonical object for* kv. A lock on this object stands for a lock on the entire set KV.

We define a distinct set of lock types for canonical objects (i.e., the *canonical lock types*). Unlike the object-level locks, Read, Update, and Write, which govern access to an object's contents, a canonical lock on an object governs access to the contents of a set that contains the object. Thus, canonical locks do not conflict with object-level locks, and we define a separate conflict table as follows:

[4]Precisely stated, the Persi code gets the lock on behalf of the transaction with which it is compiled.

LOCK	QS	MS	QP	MP	CC
QuerySet (QS)		x			x
ModifySet (MS)	x				x
QueryPrev (QP)				x	x
ModifyPrev (MP)			x	x	x
ChgCanon (CC)	x	x	x	x	x

When a transaction invokes InitiateQuery(kv) on an index, Persi gets a QuerySet lock on the canonical object for kv before it executes the query. When a transaction invokes Insert(obj,kv) or Remove(obj,kv) on an index, Persi gets a ModifySet lock on the canonical for kv. If the set KV is empty, Persi gets a QueryPrev lock on the canonical object for the lexically next key-value instead of a QuerySet lock on the canonical for kv. By analogy, it gets a ModifyPrev lock instead of a ModifySet in the same way. Persi gets a ChgCanon lock when it designates a new canonical object for a set (e.g., when the current canonical object for a set is removed from the collection).

A QuerySet lock on the canonical object for kv must prevent insertions by other transaction into the set KV, a possible violation of serializability.[5] Consequently, QuerySet conflicts with ModifySet (and QueryPrev conflicts with ModifyPrev). If transactions T_1 and T_2 invoke InitiateQuery() and Insert() respectively on the same index with the same key-value, the conflicting locks associated with the operations will force one transaction to wait for the other, and the execution will be serializable.

In contrast, ModifySet does not conflict with itself: transactions can insert (or delete) objects with the same key-value into (or from) an index concurrently until commit. ModifyPrev does conflict with itself, however, because the insertion of a key-value into an empty set creates the canonical for the set.

The Insert() and Remove() operations that a transaction invokes on an index i are added to i's queue. The operations do not execute until the transaction reads from i or commits. To guarantee the atomicity of operations on i, the scheme uses *latches* (i.e., short-term locks that a transaction releases before commit or its unlock phase). Each time a transaction invokes InitiateQuery(kv) on an index i, Persi gets a Read latch on i, fetches the latest version of i from disk, and locally applies the Insert() and Remove() operations in i's queue. Persi then locates and locks the canonical object for kv, and it unlatches the index. When a transaction invokes Insert(obj,kv) or Remove(obj,kv), Persi gets the ModifySet lock in the same way.

When a transaction commits with changes to index i, Persi gets a Write latch on i, fetches the latest version of i, executes the queued Insert() and Remove() operations, writes the result out to disk, releases the latch, and removes the queue. It releases the canonical locks associated with the index only when all updates to the database have completed.

The index package implements a B-tree that is versioned upon transaction commit. In addition, it keeps track of the last version of an index i that a transaction T read, and it continues to return subnodes of the same version to T until T rereads i's root. Consequently, Persi latches index i only for the time required to locate and lock the canonical for the kv of interest. The canonical lock

[5]Object-level locks are sufficient to prevent deletions from KV.

prevents conflicting operations on the set KV, and the index package guarantees a consistent view of i.

The advantage of this approach is that predicate locks on sets of objects are dissociated from physical locks on the index itself. In particular, transactions that access different sets of objects are free to use the index concurrently; a transaction can commit its changes even before another terminates. For brevity, we refer the reader to [23] for a more complete description of the locking algorithm.

5.5 OS services

Persi uses the OS as a storage manager (i.e., SunOS or UNIX System V together with NFS[6]). It does not implement an object server; transactions read and write data directly from and to files in the OS. In addition, Persi uses the OS locking services to synchronize transactions instead of a dedicated lock server. The OS supports Read and Write locks at the byte-level with the usual conflict semantics, and it detects conflict and deadlock among processes on the same or different machines. The semantic locks in the synchronization scheme are implemented as functions that request Read or Write locks from the OS in a way that simulates the concurrency behavior defined by the scheme. A tool SLEVE [23] generates the functions automatically from concurrency specifications in the form of conflict tables.

6 Example

Consider an instance, Hondas, of the data type Coll_Auto that was defined in Section 1. The Hondas object groups instances of the class Auto whose make is Honda. There is one index defined on color (e.g., all red Hondas).

Consider a concurrent schedule of three transactions, in which all operations are on the Hondas collection. The schedule is shown in Figure 1, where the operation parameters have been abbreviated for space considerations. The operation Add(Black) stands for "insert the object Black Honda into the collection object Hondas." The operation Count() scans the collection Hondas and returns the number of its members. Finally, Member(Red) returns true if the object Red Honda is a member of the collection Hondas.

The schedule would not be allowed under a scheme that uses only Read and Write locks on collections and indexes because the three transactions read and/or write the same collection concurrently. Under our scheme, however, the schedule is allowed.

Both T_2 and T_3 update the collection Hondas by inserting auto objects into it. Each of these two transactions maintains a member queue for the Hondas collection, in which they push the insert operations, as shown in Figure 2. Since T_1 is a read-only transaction, it does not create any queues. We show three snapshots of the queues, corresponding to steps 1, 2, and 4 in the schedule.

[6]SunOS and NFS are trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories.

	T_1	T_2	T_3
1.		Add(Black)	
2.			Add(Red)
3.			Commit
4.		Add(Blue)	
5.		c = Count()	
6.	bool=Member(Red)		
7.	Print(bool)		
8.	Commit		
9.		if c≤max Add(Green)	
10.		Commit	

Figure 1: Concurrent schedule of three transactions that invoke operations on the collection object Hondas

T_2 delays the execution of Add(Black) (step 1) and instead pushes it on the queue. Similarly, T_3 pushes Add(Red) (step 2) on its own queue. In step 3, T_3 commits by popping the operation Add(Red) from its queue, acquiring a Write lock on the collection Hondas, executing the add operation and writing the modified collection to disk. Note that this does not conflict with T_2 because T_2 has not yet acquired any locks on Hondas. In step 4, T_2 again pushes Add(Blue) on its queue.

In step 5, T_2 counts the number of members of Hondas at this point; it first pops the operations from the queue, acquires an Update lock on Hondas, reads in the object Hondas from the repository, executes the two add operations on Hondas, and deletes the queue. From that point on, T_2 immediately executes any operation involving Hondas (step 9) directly on the collection object resident in its memory.

T_1 (steps 6-8) is a read-only transaction, which acquires a Read lock on the collection Hondas. This lock does not conflict with the Update lock that T_2 holds on Hondas. Therefore, T_1 will proceed and commit without any delays.

Finally, when T_2 is ready to commit, it acquires a Write lock on Hondas, which does not cause conflicts because no other transaction holds a lock on the collection at this point, and writes out its changes.

7 The work in perspective

There has been a great deal of research in the area of concurrency control for advanced applications. Three main approaches have been proposed to improve concurrency: (1) extended transaction models [6, 10, 16, 18], (2) new correctness criteria [2, 5, 8], and (3) semantics-based concurrency control schemes [9, 20, 22]. Our scheme follows the third approach.

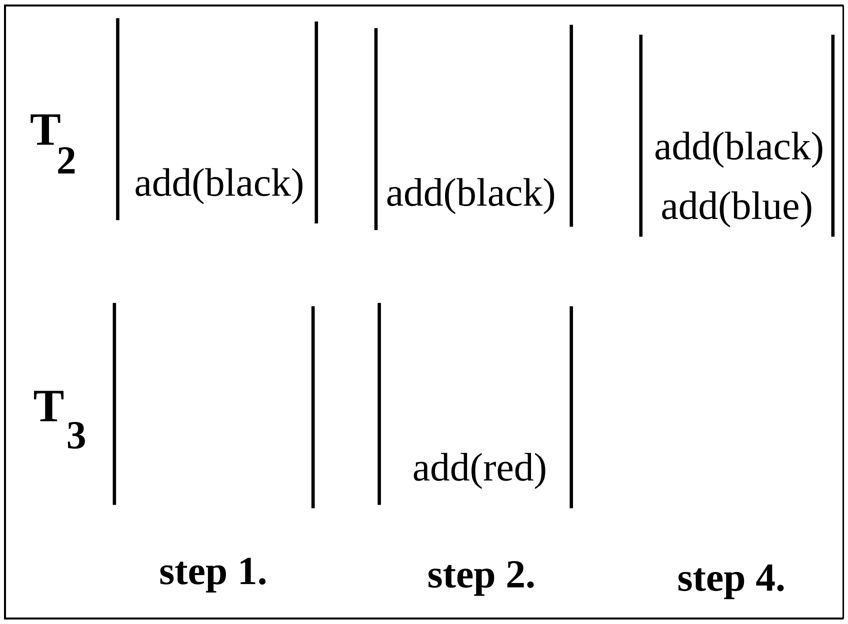

Figure 2: Snapshots of the member queue of Hondas **during the execution of transactions T_1, T_2, and T_3**

Schemes that follow the third approach maintain the atomicity of transactions, but use application semantics to allow more concurrency than traditional Read/Write serializability. The semantic information used is either information about transactions, such as the access pattern of a transaction (e.g., [22]), or about operations, such as whether or not two operations can commute (e.g., [9]). This information is used to allow executions that would not be allowed under traditional Read/Write serializability. Unlike our scheme, most of the semantics-based schemes assume an ESM execution model.

The execution model we assume is similar to those of most commercial OODBs, including O_2 [4], ObjectStore [15], Objectivity/DB [19], ONTOS [21], and Versant [24]. In these OODBs, as in Persi, objects are cached in the client process memory, and the client is responsible for computations on these objects. O_2 supports only traditional transactions with Read/Write semantics; O_2 also provides "read-only transactions" that do not require any locks (are not guaranteed consistent values). ObjectStore supports only Read and Write locks with page-level granularity. Versant provides *update* locks that are similar to Persi's Update locks. However, unlike Persi, it does not support delaying of operations or the re-fetch technique on collection objects or index structures.

Both ObjectStore and Versant support "semantic sharing" of objects across long transactions. However, this is done only through versioning, whereby concurrent long transactions can read and update different versions of the same object. These versions can either co-exist indefinitely or can be merged into a single consistent version, if needed. In that respect, they are similar to data sharing models based on *read-only versions* [3, 11, 12]. Versioning schemes, however, fail to provide correctness criteria for concurrent access. Moreover, unrestrained proliferation of versions requires complex, human intervention to reconcile coexistent, parallel versions. Finally, the schemes that enforce linear version histories commonly place the burden of reconciliation on the later-committing transactions. This raises fairness issues: while transaction T is reconciling its changes with one set of committed versions, another transaction can commit other versions with which T must reconcile its changes.

8 Discussion

The paper describes a semantic synchronization scheme for a persistent object system library. The library implements an execution model commonly used in commercial OODBs in which operations execute within the address space of a transaction, and the granularity of data access is at least the size of an object. The model restricts the application of semantic synchronization mechanisms, most of which imply either fine-grained data access or an encapsulated shared memory execution model. In designing our synchronization scheme, we developed semantics-based techniques that improve concurrency within this restrictive execution model.

We conclude with a discussion of three important issues: (1) the generality of the techniques, (2) the expected overhead of our scheme, and (3) the tradeoffs we considered in designing the scheme.

8.1 Generality

The delay technique is a dynamic form of code optimization for reducing the duration of locks. A delayed operation O effectively moves past operations in the transaction code to a later position. The resulting execution of the transaction must achieve the same effect as an execution with no delays, where the effect includes the actions on the database as well as those on any external systems such as a monitor screen. Thus, O can be delayed until the transaction commits or executes some other operation whose effect depends on O.

Clearly, an operation's context (i.e., the transaction code that contains the operation) determines whether it can be delayed and for how long. One transaction may ignore the return value of operation O and contain no other operations that depend on O, while another transaction may contain such operations or use O's return value to branch in a conditional statement.

In general, however, a synchronization mechanism controlled by context- or program-specific information has disadvantages over one controlled by type-specific information alone. The control specification is more complex (i.e., each operation O must be compared to every other operation that could possibly follow O in the program execution), and it is harder to maintain in a changing world (i.e., if the program changes, the specification for each operation must be reevaluated and possibly changed).

Consequently, we choose to define operation O's *delayability* (i.e., whether O can be delayed or not) and its *delay endpoint* (i.e., how long O can be delayed) only in terms of other operations that are defined on the same type as O. We arrive, however, at a very strict definition: operation O(obj) is *delayable iff* (1) its return value does not depend on obj's initial state, and (2) O does not read or write any objects other than obj; O's delay endpoint is the next operation on the same object in the transaction that is not delayable.

The condition regarding other objects is due to the following: if O invokes an operation on another object obj′, the control specification must include all the operations defined on obj′ and indicate which of them commute with O (i.e., which of them are delayable beyond O and vice versa).

As a result, we relax the requirement that an operation's specification consist only of operations defined on the same type. In particular, we define for each operation O on type T a *Read-* and a *Write-set* of objects that O accesses at run-time. O is *delayable iff* (1) its return value does not depend on any object in its *Read-set*, and (2) there is no object in either set that is also in the *Read-* or *Write-set* of an operation on a different type T' that some transaction in the application invokes. That is, the only way a transaction accesses objects in O's *Read-* and *Write-sets* is by invoking operations defined by T.

The approach is more conservative than a purely context-sensitive mechanism, but the advantages in reduced complexity and maintenance outweigh the slight reduction in concurrency. However, if experience indicated their usefulness, mechanisms that handle context-sensitive delay could be applied to the problem using well-know data- and control-flow techniques.

8.2 Overhead

The techniques used in our scheme involve more overhead than does a simple two-phase locking scheme. There are two main sources of the overhead: (1) queue maintenance and (2) index re-fetch.

The scheme minimizes the first overhead on collections and data objects by removing the object's queue when the object becomes resident. While index queues have to be maintained until transaction commit, the scheme avoids locks on the indexes themselves by re-fetching indexes upon each read and by maintaining the queues. This avoids the overhead associated with locking indexes and the complexity of a B-tree locking algorithm. Finally, read-only transactions require no queues for any of the objects.

The re-fetch technique is essentially a cache-update strategy, where the cache is updated during transactions rather than just between them. An implementation for re-fetch could use a standard cache-maintenance algorithm to reduce the actual number of fetches. Moreover, the implementation could optimize performance by characterizing the operations in an object's queue, such that when it fetches the object for execution of an operation O, it applies only those queued operations that affect O's result. For example, when Persi fetches an index i for a query on key-value kv, it applies only those operations in i's queue that insert or delete kv.

8.3 Tradeoffs

The first tradeoff we considered in the design of the scheme is between the complexity of an application-defined synchronization specification and the level of concurrency provided by the synchronization scheme. The more elaborate the specification, the more opportunities the scheme has for allowing more concurrency. It is often unrealistic, however, to expect users to provide very elaborate specifications. In our scheme, we have not exploited the semantics of user-defined operations on objects (i.e., we use Read/Write semantics for non-collection objects) in order to avoid having the application builder provide a conflict table for user-defined operations.

The second tradeoff we considered is between the amount of semantics used in the scheme and the complexity of the implementation in terms of bookkeeping and overhead. Although an elaborate specification may allow more concurrency, the overhead incurred in implementing a scheme based on the specification might actually reduce its viability.

Consider the semantics of inserting and removing objects from collections. Each object in Persi maintains a membership list of all the collections that contain the object. The list is used to update indexes when the object is modified. Inserting an object into a collection or removing it does not semantically conflict with modifying the object. To use these semantics within the OO execution model, an object's membership list must be maintained separately from the data part of the object. Two transactions can then write these two separate parts of the object concurrently without conflicting. This however requires two disk accesses every time the object is modified (one to write the object and another to read its membership list in order to update indexes). As a result, a complete use of the conflict semantics of operations in this case might actually decrease performance.

The third complication concerns the access patterns of transactions. We considered an alternative scheme that reduces the possibility of deadlock at the expense of read-only transactions. We decided to use the scheme based on Update locks, however, because we anticipate that most transactions in Persi will be read-only transactions. The protocol we described improves concurrency for such transactions.

Acknowledgments

We wish to thank Rick Greer for the many discussions on the index synchronization scheme and Alvaro Monge for his implementation of the protocol.

REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[2] W. Du and A. K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in Interbase. In *Proceedings of 15th International Conference on Very Large Data Bases*, 1989.

[3] D. J. Ecklund, E. F. Ecklund, R. O. Eifrig, and F. M. Tonge. DVSS: A Distributed Version Storage Server for CAD Applications. In *Proceedings of 13th International Conference on Very Large Data Bases*, 1987.

[4] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10), October 1991.

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

[6] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Annual Conference*, May 1987.

[7] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. Technical Report RJ 1654, IBM Research Laboratory, September 1975.

[8] H. Wächter and A. Reuter. The ConTract Model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1992.

[9] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43:25–61, 1991.

[10] G. E. Kaiser and C. Pu. Dynamic Restructuring of Transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1992.

[11] R. H. Katz and E. Chang. Managing Change in a Computer-Aided Design Database. In *Proceedings of 13th International Conference on Very Large Data Bases*, 1987.

[12] R. H. Katz and S. Weiss. Design Transaction Management. In *Proceedings of 21st ACM/IEEE Design Automation Conference*, 1984.

[13] H. F. Korth. Locking Primitives in a Database System. *Journal of the Association for Computing Machinery*, 30(1), January 1983.

[14] H. T. Kung and C. H. Papadimitriou. An Optimality Theory of Concurrency Control for Databases. In *Proceedings of ACM SIGMOD Annual Conference*, 1979.

[15] C. Lamb, C. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10), October 1991.

[16] N. A. Lynch. Multilevel Atomicity—A new Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4), December 1983.

[17] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1992.

[18] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, MA, 1985.

[19] Objectivity, Inc., Menlo Park, CA. *Objectivity Database Reference Manual*, 1990.

[20] P. E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4), December 1986.

[21] Ontologic, Inc., Billerica, MA. *ONTOS Reference Manual*, 1989.

[22] K. Salem and H. Garcia-Molina. Altruistic Locking. Technical Report UMIACS-TR-90-104 CS-TR-2512, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, July 1990.

[23] A. H. Skarra. SLEVE: Semantic Locking for EVEnt synchronization. In *Proceedings of Ninth International Conference on Data Engineering.* IEEE Computer Society Press, 1993. An expanded version is Technical Memorandum 59113-920303-03TM, AT&T Bell Laboratories, March, 1992.

[24] Versant Object Technology, Menlo Park, CA. *VERSANT ODBMS: A Technical Overview for Software Developers*, 1992.

[25] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. In *IEEE Proceedings of 21st Annual Hawaii International Conference on System Sciences*, 1988.

[26] A. L. Wolf. An Initial Look at Abstraction Mechanisms and Persistence. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice. The Fourth International Workshop on Persistent Object Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[27] A. L. Wolf. The Persi Persistent Object System Library. Available from the author, June 1993.