An Efficient Consistency Protocol for a DSD-based Persistent Object System

by

Arne Grimstrup

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 2001

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

AN EFFICIENT CONSISTENCY PROTOCOL FOR A DSD-BASED PERSISTENT OBJECT SYSTEM

BY

ARNE GRIMSTRUP

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree of MASTER OF SCIENCE

Copyright Declaration

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires that the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Persistent object stores (POS) provide a good foundation for distributed applications. By executing nested object transactions, the application programmer can make updates to shared objects from different processing nodes across the network without having to manage the communications or concurrency control aspects of data access in the distributed environment. In such systems, the shared objects are "cached" in the nodes' local memories and these copies must be kept consistent in order to ensure correct execution of the transactions. Therefore, the memory consistency protocol has a great impact on the efficiency and usability of the POS.

The LOTEC protocol was designed to maintain data consistency while reducing the associated consistency maintenance communication overhead for shared objects in a distributed shared virtual memory (DSVM) environment. While LOTEC achieves its goal, the use of fixed-size memory pages limits the performance improvement. Also, the locking protocol reduces the potential for concurrent execution of the nested object transactions.

In this thesis, a new memory consistency protocol is presented in the context of a distributed shared data (DSD)-based POS. This protocol improves performance and reduces overhead by managing concurrent data access using versions of smaller groups of an object's attributes. When compared with LOTEC, the new protocol reduces a number of delay-causing situations that may arise during transaction execution. In addition, a new algorithm for creating smaller groups of attributes from an object called object chunking is presented and analyzed. Simulation results indicate that object chunking can significantly reduce the amount of data that must be moved in order to maintain memory consistency.

Acknowledgements

I am grateful for the support and assistance my family, friends and colleagues have given me during the research for, and writing of, this thesis. In particular, I would like to thank my advisor Dr. Peter Graham, his wife Pat Bessler, and daughter Lisa for their support and the occasional free meal during this seemingly never-ending process. I would also like to thank Dr. Richard Barton for his assistance with the statistical analysis. Finally, I would like to thank my wife Kristin for her patience and periodic nudges to get this thing done. I guess I have to find some new excuses now, right dear?

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Peters, *et al.* [PGB97] describe a Persistent Object Store (POS) based on a Distributed Shared Virtual Memory System (DSVMS) in a 64-bit address space. The programming environment of the POS provides *distribution*, *persistence*, *support for complex objects*, and *transactions* to the programmer which can be used to support a variety of advanced applications such as CAD/CAM or virtual environments. In order to ensure correct execution, transaction concurrency and data consistency must be carefully managed in this environment. The LOTEC protocol [GS99, Sui98] was designed to address the concurrency and data consistency problems of the DSVM environment. LOTEC uses the object two-phase locking (O2PL) rules [Sui98] to ensure a serializable execution of nested object transactions and *lazy propagation* of updated pages to reduce the amount of data transferred. This reduction improves POS performance over more traditional approaches. The

amount of performance gain is limited by LOTEC's use of pessimistic concurrency control and the memory page as the unit of data transfer.

To address the performance limitations of LOTEC, this thesis presents a new approach to partitioning objects called *chunking* and a version-based protocol for maintaining memory consistency with closed nested object transactions in a Distributed Shared Data (DSD)-based POS.

1.1 Challenges

In LOTEC [GS99] and memory consistency protocols for traditional DSM systems [LLG+92, BZS93, KCDZ94, CBZ95, ISL96], the basic unit of data transfer is a memory page. Using a large transfer unit like the memory page increases the risk of data anomalies, such as *false sharing*, occurring during execution. More recent DSM systems use a memory region approach [IS99, BS99] to achieve finer-grain sharing than a page. DSD systems [Lu97] avoid data anomalies by managing independent data as separate objects. Attribute access within an object is not always independent, therefore attribute groups within an object have a significant impact on protocol performance.

There are two main approaches to creating fragments in relational databases: affinity-based and transaction-based. Affinity-based approaches [HS75, NCWD84] partition the relation into groups of attributes which tend to be used together, but these approaches are dependent on access frequency information. Affinity-based methods have been successfully applied to objects [EB94].

Transaction-based approaches [HN79, Chu92, CI93] use the semantic information found in the transactions to construct the fragments. Unfortunately, these techniques are designed to minimize disk access related overhead and are not appropriate for the DSD environment. An appropriate fragmentation algorithm must be developed for the DSD environment.

LOTEC uses *object two-phase locking rules* (O2PL) to manage concurrent access to a datum by *nested atomic object transactions*. O2PL ensures that a datum is left in a correct and consistent state by ensuring that a *conflict serializable* [BHG87] execution history can be created for any set of concurrently executing nested object transactions. This result is achieved by allowing nested object transactions to access data serially. Parallel execution of nested object transactions can be supported through the use of *versioned data* [BHG87]. In order to achieve this goal, a set of version-based locking rules must be developed that preserves the nested object transaction structure.

1.2 Motivation

The efficiency of a distributed POS is greatly affected by the underlying memory consistency and concurrency control protocols. The LOTEC protocol reduces the amount of communication in terms of both byte count and number of messages sent. The lower communications requirements reduce the impact of network latency thereby ensuring high performance. LOTEC's

use of memory pages as the data transfer unit limits the improvements in this area. Therefore, developing techniques to further reduce communication overhead is very important in high bandwidth, low latency networks with higher per-message cost.

Performance of a distributed POS can also be evaluated in terms of transaction throughput. Systems with high throughput can execute a large number of transactions in a given period of time and are generally more desirable platforms for building applications. The LOTEC protocol allows only one transaction to access to an object at a time. This restriction can cause significant reduction in throughput due to the resulting access delays. Techniques that ensure memory consistency while reducing object access delays would improve transaction throughput for a POS.

1.3 Organization

This thesis is organized as follows: Chapter 2 reviews the relevant research pertaining to this work. Chapter 3 describes the assumed operation environment, the problem of maintaining memory consistency in a distributed shared data environment, and presents a review of the LOTEC protocol. A new algorithm for partitioning objects is presented and analyzed in Chapter 4. In Chapter 5, a new memory consistency protocol for closed versioned nested object transactions is described and analyzed. Finally, Chapter 6 concludes this thesis with a description of the contributions and a discussion

of future research directions.

Chapter 2

Background and Related Work

As the protocol developed in this thesis is intended to support a Persistent Object System (POS) in a Distributed Shared Data (DSD) environment, the discussion will begin with an introduction to POSs and will also review transactions and serializability. The discussion will then focus on the approaches to sharing data in distributed environments particularly Distributed Shared Memory (DSM) and DSD. Memory consistency models in DSM systems are reviewed because they are generally applicable in the DSD environment. Finally, because DSD system performance depends on the amount of data transferred as well as the number of messages being sent, data fragmentation techniques are also reviewed.

2.1 Persistent Object Stores

An object, as described by Kim [Kim90], is a model of a real world entity's state and behaviour and has an associated unique identifier. The state of the object is stored in a set of values which describe the attributes of the object, while the behaviours are represented by a set of routines called methods which are exclusively permitted to alter the object's state. Objects that have the same attributes and methods are grouped into a class. An object must belong to only one class and is referred to as an instance of that class.

Objects are a powerful tool for implementing software. The state of each object is encapsulated by its methods which provides enforced locality of effect and improves understandability. Programmers may reuse existing components, or derive new classes by inheriting the methods and attributes from one or more existing classes and adding additional attributes and methods. Using the properties of encapsulation and inheritance, more complex systems can be modeled with objects than is possible by abstract data types alone.

Persistence

Implementing a complex system using objects is a difficult task. In most environments, an object exists only while the software is executing. Thus, programmers must translate the object's internal representation into a form that can be placed on long-term media such as a disk or a tape to provide persistence between executions. One can eliminate this inconvenience if an

object is made to automatically exist for as long as it is needed, rather than only for the length of time a particular application runs. Such objects are said to be *persistent*.

Persistent objects remove the necessity to explicitly manage external storage but, depending on the implementation, they can introduce other problems. If persistence is implemented on the basis of data type, it is possible to create references to non-persistent data within a persistent object. When an application that created the non-persistent data terminates, it leaves a *dangling reference*, to non-existent data, in the persistent object. Applications that are subsequently executed will fail or generate incorrect results if they attempt to use the dangling reference. An *orthogonally persistent* [ABC+83] object system hides its implementation details from the programmer and eliminates the distinction between persistent and non-persistent data.

Distributed Objects

Distributed computing in an object environment presents additional problems to the programmer. In commercial distributed object protocol suites such as CORBA [YD96] and DCOM [Mic97], objects must be registered with a brokering system to be accessed remotely. The calling program directs its method invocation request to the broker which marshals and forwards the request to the desired object and then returns the results.

By combining distribution with orthogonal persistence, the location of the

objects may be made transparent to the user and this eliminates the need for the object broker. Users of such a POS would then create, delete and access objects as they do within applications now. The POS would perform the object registration, deregistration and communication tasks. Peters, *et al.* [PGB97] describe a DSM-based distributed persistent object system which could act as a foundation for an objectbase.

2.2 Transactions and Serializability

The protocol described in this thesis exploits properties of transactions and concurrency control to achieve better performance. As an aid to understanding, this section presents the background concepts of transactions and concurrency control. Flat, nested and object transactions are reviewed along with the concurrency control problems of, and algorithms used in, each environment.

2.2.1 Flat Transactions

A conventional transaction is a sequence of read and/or write operations on data ending in a commit or an abort operation. Transactions normally have four properties: Atomicity, Consistency, Isolation and Durability (generally referred to as the ACID properties). Atomicity requires that either all or none of the operations of the transaction are executed. Consistency requires that any data accessed by the operations of the transaction be left

in a consistent state after the commit or abort. Isolation ensures that the transaction does not inadvertently affect other transactions which may be running concurrently. Finally, durability ensures that all changes made by the operations of committed transactions persist even when failures occur in the database or when subsequent transactions are aborted.

Flat Transaction Serializability

The ACID properties ensure that a transaction executes reliably and correctly. We can execute any number of transactions serially without creating data inconsistency. Serial executions, however, lead to poor system performance because they fail to take advantage of possible concurrency between the available transactions.

Concurrent execution of transactions, while greatly improving performance, can lead to data inconsistency due to conflicts[1] in the component operations. Therefore, we must manage the concurrent execution of transactions carefully. The most widely used correctness criterion for this is *conflict serializability* [BHG87]. Under conflict serializability, we view transactions as partial orderings of operations. If two concurrently executing transactions attempt to operate on the same datum, we consistently order the operations so they are equivalent to some serial execution of the two transactions. Any concurrency control algorithm which implements conflict serializability will

[1] According to Bernstein, *et al.* [BHG87] a conflict occurs when two operations occur on the same datum and at least one is a write.

leave the data in a correct and consistent state.

Concurrency control algorithms generally follow one of two main approaches: pessimistic or optimistic. The pessimistic approach, concerned that conflicts occur frequently, attempts to minimize the impact by delaying the conflicting operations when they are identified even at the loss of some concurrency. The optimistic approach hopes that conflicts will be infrequent and strives to maximize concurrency. All transactions are therefore allowed to run concurrently but before commitment, the data is verified for consistency. If an inconsistency is detected, the transaction is aborted.

The most common pessimistic concurrency control algorithm used in database systems is strict two-phase locking(strict 2PL). Under strict 2PL, a read lock and a write lock are associated with each datum. Each transaction acquires the appropriate lock for each datum it accesses during execution and holds it until the transaction ends when they are all released at once. Any number of transactions may hold the same read lock concurrently, but only one may hold a write lock. Once a transaction acquires a write lock on a datum, all other transactions must wait until that write lock is released. The 2PL rules ensure that no cycles can form in the serialization graph[2] and thereby ensure serializability. Bernstein, *et al.* [BHG87] provide a formal proof of correctness for strict 2PL.

[2] A *serialization graph* is a directed graph where the nodes are the transactions that have committed during the period in question and an edge $T_i \rightarrow T_j$ appears where one of T_i's operations precedes and conflicts with an operation in T_j.

Optimistic concurrency control algorithms certify that data accessed during transaction execution is unchanged at the point where the transaction attempts to commit. Techniques for optimistic concurrency control are based on detecting cycles in data access graphs [BHG87], or by exploiting the structure of the data being accessed. Another optimistic approach manages new copies of a datum that are created each time a write operation is performed on the datum. This is known as *multi-version concurrency control* (MVCC). MVCC-based systems enhance concurrency by allowing late read operations to overlap with subsequent read or write operations at the cost of additional storage.

MVCC systems are commonly based on *multi-version serializability*. In conflict serializability, two execution histories[3] are considered equivalent if all pairs of conflicting operations occur in the same order. This is known as *conflict equivalence*. Because there are multiple versions of the data, operations in multi-version (MV) systems are different from operations in a serial single version system (1V). Conflict equivalence can not accurately establish the relationship between an MV history and a 1V history. Instead, the read-from and final write sets are compared. If they match, the two histories are said to be *view equivalent*. A concurrency control algorithm that uses multi-version serializability is considered to be correct if the MV history it produces is view equivalent to some 1V history. Bernstein, *et al.* [BHG87]

[3] A *history* is a record of the order in which the operations of a set of transactions occur relative to each other.

provide a formal proof of correctness for multi-version serializability.

There are two common multi-version concurrency control algorithms: multi-version timestamp ordering (MVTO), and multi-version two-phase locking (MV2PL) the former being an optimistic technique and the latter a pessimistic one. MVTO algorithms assign a unique timestamp, $ts(T_i)$, to each transaction T_i. When a read request is made, the MVTO scheduler returns the version of the desired datum with a timestamp less than or equal to $ts(T_i)$. A write operation by T_i creates a new version of the datum x and T_i's commitment is then delayed until all other transactions that wrote a version read by T_i have committed to ensure recoverability. If some transaction T_j reads a version of x created by T_k such that $ts(T_k) < ts(T_i) < ts(T_j)$, T_i's write operation is rejected and one of T_i or T_j must be aborted.

MV2PL introduces a new lock type in addition to those found in 2PL. *Certify* locks conflict with all other lock types and are acquired when a transaction is ready to commit. Upgrading to certify locks is only allowed once all outstanding read locks have been released. When a data item is read by transaction T_i, a read lock is obtained and the latest committed version is used. Subsequent reads will return the latest version written by T_i. After a write lock is obtained, T_i creates a new version of the datum. T_i commits only when all the locks for versions it read (and did not write) have been upgraded to certify locks by the transactions which produced them.

Formal proofs of correctness for MVTO and MV2PL may also be found in Bernstein, *et al.* [BHG87].

2.2.2 Nested Transactions

Not all the operations in a transaction need necessarily be reads and writes. Moss [Mos85] introduced the concept of structuring transactions as trees with the parent transaction forming the root of the tree and the primitive operations and child transactions forming the leaves. These trees are called *transaction families.* There is no depth restriction to a transaction tree, since a child transaction may also have its own children. Moss' model also allows a parent transaction to choose to execute child transactions in parallel. This allows for much finer granularity concurrency control than is typically possible with conventional transactions.

Nested transactions come in two forms: open and closed. In an open nested transaction, updates made by a child transaction are visible to all other transactions in the system after that child transaction commits. Immediate visibility can increase the potential for concurrent execution of transactions, even from different families. However, if the parent transaction subsequently aborts, any transaction which used a datum updated by a committed child of the aborting transaction family must be rolled back. This can trigger roll backs of other transaction families which in turn can trigger more roll backs. This is an example of the *cascading aborts* problem. Closed nested transactions restrict the visibility of committed child transaction updates to other members of the transaction family until the parent transaction commits. Therefore, any cascading abort that occurs is limited

to that transaction family and will not disrupt the remainder of the system.

Nested Transaction Serializability

While conflict serializability, as defined previously, will produce correct concurrent executions of all transaction families in the system equivalent to some serial order, it does not address the problem of conflicts within a transaction family. Sub-transactions may execute concurrently and therefore may introduce data inconsistencies within a transaction family. Because of the ACID properties, a sub-transaction cannot completely "commit" since an ancestor can still abort and thereby undo its changes. Conflicts within a transaction family can be eliminated by simply enforcing a depth-first execution of sub-transactions. By doing so, we have an exact serial execution and thus a correct execution of the transaction family.

Enforcing a depth-first execution strategy on nested transactions eliminates concurrent execution of sub-transactions and would, therefore, nullify a fundamental benefit of using nested transactions. Moss[Mos85] solves the intra-transaction family conflict problem by extending the strict two-phase locking rules. The extended two-phase locking rules are:

1. A transaction may hold a write lock for datum x if all transactions holding a lock of any type for x are its superiors;

2. A transaction may hold a read lock for x if all transactions holding a write lock on x are its superiors;

3. If a transaction aborts, all locks it holds are discarded; its superiors continue holding any locks they currently have, and

4. If a transaction commits, all its locks are passed to its superior. The superior holds the strictest lock[4]. When the root transaction commits, all locks are released back to the system.

Rules 1 and 2 ensure that two sister sub-transactions cannot hold conflicting locks at the same time. Rule 3 ensures that all traces of an aborting child transaction are removed. Rule 4 ensures that locks propagate up the tree until the root node holds all the locks. Lynch [Lyn83] has proven the correctness of a locking algorithm based on these rules.

Nested Transaction Commitment

Flat transactions make their changes permanent by executing a commit operation once they have completed their primitive operations. In nested transactions, however, the possibility of an ancestor aborting precludes a sub-transaction from completely committing. Instead, final commitment must be delayed until all sub-transactions in the transaction family are complete.

A common protocol used to ensure that all sub-transactions atomically commit is *two phase commit*(2PC) [BHG87]. In 2PC, a coordinator (root transaction) sends a vote-request message to all the participants (sub-transactions). A participant responds with a "yes" vote and waits for a command

[4]A write lock is stricter than a read lock.

from the coordinator if it wishes to commit[5], or "no" and aborts if it does not. If all the votes are "yes", the coordinator sends a "commit" command to all the participants, otherwise the coordinator sends an "abort" command to all participants that voted "yes". Participants receiving a command from the coordinator act accordingly and stop. A formal proof of correctness for 2PC can also be found in Bernstein [BHG87].

2.2.3 Object Transactions

Object transactions, as defined in Zapp [Zap93], correspond to methods invoked on objects. Nested transactions and object transactions are very similar. In both cases, there is a parent transaction which coordinates and supervises the child transactions, and child transactions may also have their own children. However, object transactions are restricted to accessing their own attributes because of encapsulation while nested transactions may access any data in the database. Object transactions at any level in the tree may also access data while in nested transactions data access is typically restricted to the leaf transactions.

Object Transaction Serializability

Because of their similar nature, object transactions share the same problems of concurrent execution as nested transactions. Similarly, by extending the concepts of conflict serializability, those problems can be eliminated. Zapp

[5]Zapp [Zap93] defines this waiting period as the *pre-commit* state.

[Zap93] extends conflict serializability theory into the object domain and provides a proof of correctness.

Moss' nested two-phase locking rules must be extended for use with object transactions for two reasons. First, since non-leaf nodes may access data, they must be able to hold their own set of locks to ensure serializable access. Second, because methods may be invoked recursively or, worse, mutually recursively between two objects, there is a danger of deadlock. Sui [Sui98] addresses this problem by defining and proving the correctness of object two-phase locking. Sui's extended rules are:

1. Transaction T may acquire a lock if:
   (a) no other transaction holds the lock or all transactions that retain the lock are ancestors of T, and
   (b) if T depends on a transaction T', T' has completed.
2. Once a lock has been acquired by transaction T, the lock is held until T commits or aborts.
3. A transaction cannot pre-commit until all its sub-transactions have pre-committed. When a sub-transaction T pre-commits, the parent of T inherits all its locks. After that, the parent retains all the locks.
4. When sub-transaction T aborts, locks held or retained by it and its sub-transactions are released unless any of its ancestors retain any of those locks in which case they continue to do so.

5. When the root transaction T commits, it releases all locks which were held by itself and all of its sub-transactions. This makes them available to other transaction families.

This thesis will consider only closed object transactions and their serializability.

2.3 Memory Consistency Models in Distributed Systems

In this section, pertinent research specific to maintaining memory consistency in distributed systems is reviewed. Sections 2.3.1 and 2.3.2 review the message-passing and shared memory approaches to data sharing. In section 2.3.3, distributed data approaches are reviewed, and section 2.3.4 surveys memory consistency models and protocols used in Distributed Shared Memory(DSM) and Distributed Shared Data(DSD) systems.

2.3.1 Distribution via Message Passing

Message-passing based information sharing requires that an explicit request for data or processing be sent from one processor to another for each data access. Message-passing is generally well understood, easily scalable, and forms the basis for interprocess communication in such systems as the Mach microkernel [RBF+89] and the Amoeba distributed operating system [TvRvS+90].

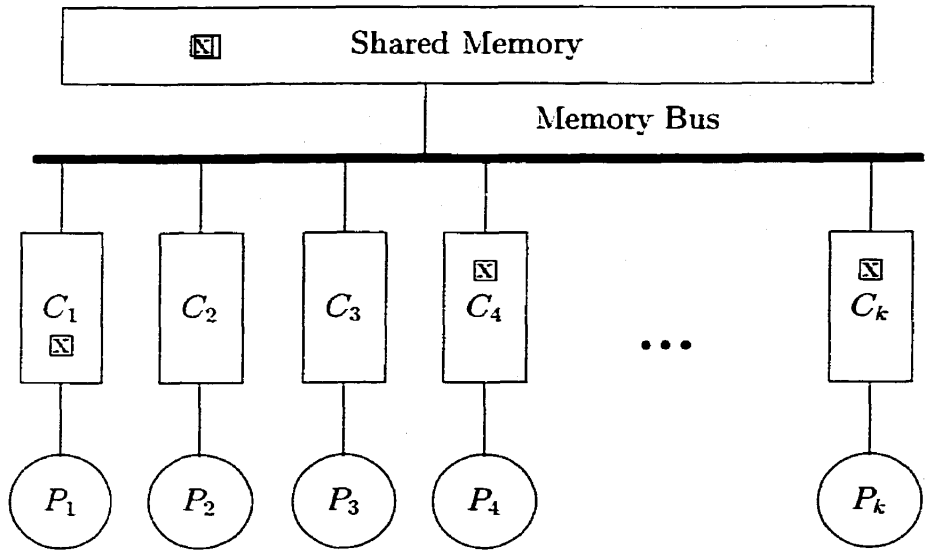


Figure 2.1: Shared Memory System with Caches

Invoking a method on an object is similar in many respects to message-passing. With objects, the messages always instruct the object to alter its state according to the behaviour prescribed by the invoked method. In message-passing architectures more general information such as data or commands may be passed to the recipient. Assuming that each object resides on a single node then message-passing may be used to directly implement method invocation on an object. An example of this approach is Java's Remote Method Invocation(RMI) facility [FFCM99].

2.3.2 Shared Memory

The shared memory approach, as used in parallel computers, provides a single physical memory shared via hardware by all processors as shown in Figure 2.1. Since all processors can see all of the data, updates are immediately available just as they are in single CPU systems. This makes programming easier since there is no need to explicitly send and receive messages. However, as the number of processors in the system increases, memory access becomes a bottleneck because the memory bus becomes overloaded. Thus, shared memory systems are limited usually to a maximum of 32 processors in size.

The memory access latency in shared memory systems can be improved by caching data at each processor. Installing a local cache allows the processor to read the local copy of the data rather than retrieve the data from main memory each time it is accessed. This reduces traffic on, and therefore contention for, the memory bus.

While improving read performance, the use of caches introduces a serious consistency problem when writes occur. Since a processor updates its local copy of the data, the change may not be immediately visible to the other processors. To maintain a consistent view of the shared data, the in-cache copies must be transparently kept consistent to prevent anomalies such as lost updates.

There are two approaches to maintaining cache consistency: write-update and write-invalidate. The invalidation approaches requires that all copies of

a shared datum other than the one being updated be invalidated (i.e. made unusable) before a processor can perform a write. Marking a datum invalid typically occurs when the processor acquires exclusive control of that datum. In update schemes, the write is held in a buffer until some synchronization point is reached, at which time all cached copies of the datum are updated. Non-local access to the datum is precluded until after the update.

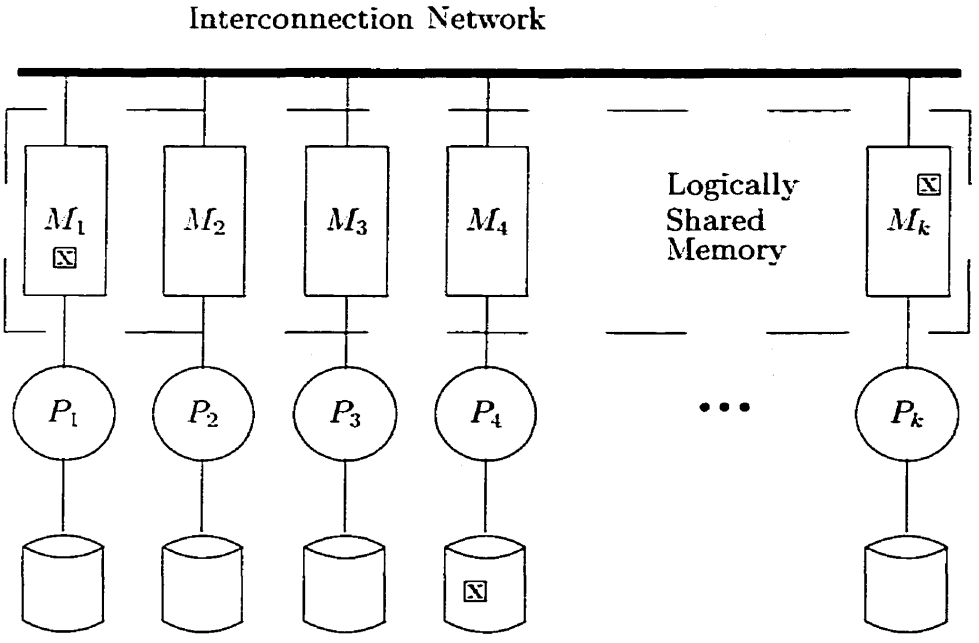


Figure 2.2: Distributed Shared Virtual Memory System

2.3.3 Access to Distributed Data

Distributed systems operate on a collection of independent nodes, each having their own processor(s) and memory, connected by an interconnection net-

work. Communication between nodes is accomplished by message-passing. While, in general, such systems do not suffer from cache consistency problems or memory bus bottlenecks, they do require the programmer to explicitly manage the data location and communications. Examples of such systems include CORBA [YD96], and DCOM [Mic97].

By combining ideas from shared memory multi-processors and message-passing approaches, we can create systems that are easier to use and which have enhanced scalability. Distributed shared memory (DSM) systems, which implement this combination, create a *logically* shared memory from the local memories of each processing node. While the data is shared via explicit message-passing, the DSM abstraction hides the communications details from the programmer by creating a address space shared among processes across the network. Thus, the local memory of each node is used as a cache for this global virtual address space. A block diagram of such a system is given in Figure 2.2.

It is also possible to incorporate support from the node's virtual memory system to assist in the transfer of data between permanent storage and the various systems' memories to ensure the data is persistent. Such systems are called Distributed Shared Virtual Memory(DSVM) systems. DSVM systems have the same problems of cache consistency that were described previously.

Another potential problem with DSM systems is the granularity of the data objects that are manipulated. To make the most efficient use of the virtual memory hardware, the basic transfer unit is usually a page. Pages

are typically 4 kilobytes on traditional systems. If every datum in the DSM is the same size as the page, there is no granularity problem. Datum size rarely matches page size however, so some pages will contain multiple data objects. With shared pages, a situation can arise where two processes on different nodes write to two different data objects in a single page. While each of the written objects is consistent on the local processor, one of the updates will be lost when the page is updated. Which one is lost depends on the order in which the copies are written. This problem is normally addressed by locking entire pages. Unfortunately, this gives rise to *false sharing* which may seriously limit concurrency unnecessarily. To reduce the incidence of false sharing, DSM systems such as MultiView [IS99] and the Region Trap Library [BS99] use user-defined, variable-size memory blocks to allow finer-grain sharing of data within an application.

Distributed Shared Data(DSD) systems, on the other hand, avoid false sharing by manipulating each datum individually without involving the virtual memory hardware. In addition, DSD systems can support finer-grain concurrency control without loss of data integrity. However, because object data sizes can vary widely, network performance can drop significantly due to the additional overhead caused by sending or receiving large numbers of small messages[6].

[6]This depends primarily on the messaging latency in the network being used.

2.3.4 Memory Consistency Models

In conventional uniprocessor systems, memory is consistent if the value returned by read operations is always the same as the value written by the most recent write operation. Unfortunately, with multiple processors accessing different copies of the same datum, as happens in shared memory multi-processors, defining the "most recent write operation" becomes more difficult. This problem is even more challenging in DSM and DSD systems since there is no physical memory that can be referred to and because of the possibly large numbers of nodes in the system.

A memory consistency model, as defined by Adve [Adv93], is an architectural specification of how updates to shared memory are reflected to the other processes in the system. A model must eliminate the problem of "the most recent write operation" by specifying which datum should be returned by the read operations of an executing process. In so doing, the model also determines which memory operations may be executed in parallel by a given processor, in what order the operations may occur, when operations may be permitted to overlap, when updates can be made available to other processors, and hence how much inter-process communication will occur. Thus, the memory consistency model effectively determines the performance of a DSM or DSD system.

Designers of memory consistency protocols attempt to achieve the best possible performance in a DSM or DSD system. The performance of such

systems is affected by many factors. Because DSM systems typically use large-grained units (i.e. pages) as their coherence and communication units, they are susceptible to false sharing (described previously) and unnecessary communication. Such communication can occur when the requesting processor does not require all of the data in the page, but is forced to move it anyway. Both DSM and DSD systems are also affected by the cost of communications since all data transferred (and synchronization performed) must be accomplished via explicit messages. Further, there is also processing overhead incurred by running the consistency maintenance protocol itself. Therefore, a key design goal in DSM and DSD systems is to reduce the amount of communication, in terms of both the number of bytes transferred and the number of messages sent, required to maintain memory consistency.

Many memory consistency models have been implemented, in both hardware and software, and have been proven correct. In general, they fall into two categories: strong consistency models and weak consistency models.

An example of a strong consistency model is Sequential Consistency (SC) [Lam79]. A system is sequentially consistent if all memory operations appear to execute serially in some total order, and all memory operations of a given processor appear to execute in program order. SC, in effect, extends the uniprocessor model by requiring that any update to shared data be visible to all other nodes in the system before the writing processor may access shared memory again.

Because of the need to maintain a globally consistent view of the data

at all times, SC requires large amounts of expensive communication. While some performance improvements can be made by using broadcast messages [FKL98], SC precludes many performance enhancing optimizations such as buffering, pipelining, and operation reordering. The programmer is responsible for ensuring exclusive access to shared data via some sort of synchronization primitive.

Weak consistency models exploit the use of mutual exclusion to reduce the amount of communication required to maintain consistency. Instead of immediately propagating updates as they occur, data are exchanged only at synchronization points. The resulting reduced number of messages increases performance of the DSM or DSD system. The data, while consistent at the current updater, is not globally consistent. However as long as mutually exclusive access to the data is ensured, there is no danger of accessing stale data.

Release Consistency(RC) [GLL+90] is a weak consistency model that classifies memory operations into "acquires", "releases" and "ordinary" operations. Acquire and release operations, performed on associated locks, indicate the beginning and the end of a processor's use of a memory object. Ordinary operations correspond to the reads and writes of the memory object. In the release consistency approach, all update propagation to other nodes is delayed until the processor completes all ordinary operations, but before the release takes place. In an RC-based DSM or DSD, the data would be globally consistent after a release and before a lock acquire, and always locally

consistent to any processor performing ordinary operations. RC has been implemented in hardware in the DASH [LLG+92] system and in software in the Munin [CBZ91, CBZ95] system.

Propagating updates in an RC-based DSM triggers additional processing at each node to apply the changes so the local copy reflects the global state. Not all processors *immediately* need the updated data and some may never need it. This observation led to the development of Lazy Release Consistency(LRC) [KCZ92]. LRC delays propagation of modifications until the next acquire occurs. By delaying the updates until acquisition, LRC reduces the number of messages sent in propagating changes and the amount of data shipped across the network since only the next acquirer receives the update. LRC has been implemented in the TreadMarks [KCDZ94] system.

A similar approach to LRC is used in Entry Consistency(EC) [BZS93]. In both systems, a processor's view becomes consistent with the most recent updates only when a lock is acquired. The Midway system associates a synchronization object, such as a mutex lock, with individual data items. Since EC gives finer granularity access to the data than LRC, less data is transferred between nodes at the cost of having the programmer specify more synchronization operations. EC has been extended to object-oriented systems in the LOTEC protocol [GS99]. A major advantage of LOTEC is that the required synchronization operations are automatically generated by the transaction manager so there is no added complexity for the programmer.

Scope Consistency(ScC) [ISL96] attempts to gain the benefits of EC with-

out the additional locking overhead. ScC uses synchronization variables to create scopes through which memory is viewed. When a write access occurs, data is dynamically associated with a scope. The Aurora system [Lu97] uses ScC to implement a DSD system.

2.4 Data Partitioning in Distributed Relational Database Systems

The structure of the data is also an important factor in overall performance of a distributed relational database system. One of the main techniques used to improve performance is the partitioning of data relations into smaller units called *fragments*. In this section, the concepts of data partitioning and affinity-based and transaction-based vertical partitioning techniques are reviewed.

2.4.1 Fragmentation Concepts

Fragmentation of data relations, according to Özsu and Valduriez [OV91], is performed for three main reasons: (1) to reduce the amount of unnecessary data accessed, (2) to improve reliability and performance through replication, and (3) to increase intra-query concurrency. Instead of loading the entire relation, an application can retrieve the fragment that contains the data it needs. Replication of heavily accessed data spreads query processing opera-

tions over more processors. Using more processors reduces the workload on each individual host, which means that requests for data are processed with less delay. If a processor should fail while the system is running, replicated fragments would still be accessible at other nodes. Finally, since multiple processors are available to handle a request, multiple fragments can also be accessed simultaneously. This concurrent access can improve overall system throughput by reducing the amount of time spent by a transaction waiting for the data to be retrieved from storage.

Özsu and Valduriez [OV91] identify three types of fragmentation: *horizontal*, *vertical*, and *hybrid*. In horizontal fragmentation, the relational tuples are divided between the fragments on the basis of the values of one or more attributes. This approach partitions the relational table into a set of smaller tables each containing the same attributes which can then be distributed over the various processors. Horizontal fragmentation can be applied in two ways. In *primary* horizontal fragmentation, the selection operation is applied directly to the owner relation [7]. A *derived* horizontal fragmentation results from an equi-join[8] between an owner relation and a member relation.

Vertical fragmentation partitions the attributes of a relational tuple in to a number of smaller relations each containing a subset of the original relation's attributes. The goal is to reduce the size of the tuple so that many

[7] An *owner* is a relation who's key values are attributes in other tuples.

[8] An *equi-join* is a merging of multiple relational tuples using equal values of attributes from a common domain.

user applications can execute using only the data contained in a single fragment, thereby improving application performance by reducing I/O overhead and increasing concurrency. There are two approaches to creating vertical fragments: *grouping*, which assigns attributes to individual fragments and joins them until some criteria is met, and *splitting*, which takes the relation and divides it according to the access behaviour of the applications using the relation[9]. Regardless of the method selected, it must be possible to reconstruct the original relation from the resulting fragments or the integrity of the relation will be lost.

Hybrid fragmentation is the result of applying both horizontal and vertical fragmentation successively in any order to the relational tuples.

2.4.2 Affinity-Based Vertical Fragmentation

In many databases, transactions tend to use some of the attributes of a relation together as a group during processing. These attributes are said to have "high affinity" and this relationship can be exploited to produce a vertical fragmentation.

Hoffer and Severance [HS75] developed a two-phase method to generate the vertical fragments of a relation. In the first phase, the affinity between

[9]For the purposes of this thesis, only splitting methods for vertical fragmentation are considered.

pairs of attributes A_i and A_j is calculated as follows:

$$aff_{ij} = \sum_{k|u_{ki}=1 \wedge u_{kj}=1} \sum_{j} n_{kj} f_{kj}$$

where u_{ki} is 1 if transaction k uses attribute i when executed, otherwise it is 0; n_{kj} is the number of accesses to the attributes for transaction k at site j, and f_{kj} is the number of times transaction k is executed at site j. In the second phase, the *Bond Energy Algorithm* [MSW72] is used to cluster the resulting *attribute affinity matrix*.

The Bond Energy Algorithm proceeds as follows: first an arbitrary column from the affinity matrix (formed in step one) is selected and placed in the cluster matrix. Then, the *measure of effectiveness* (ME) is calculated for each pairing of the columns in the cluster matrix with one of the remaining columns such that:

$$ME = max\{\sum_{i=1}^{M} \sum_{j=1}^{N} a_{i,j}(a_{i,j-1} + a_{i,j+1} + a_{i-1,j} + a_{i+1,j})\}$$

where M is the number of rows in the matrix, N is the number of columns and $a_{i,0} = a_{i,N+1} = a_{0,j} = a_{M+1,j} = 0$.

The column which generates the greatest gain in ME is then added to the cluster matrix and the second step is repeated until all columns have been added. The rows are then re-ordered in the same permutation as the columns giving a block diagonal form.[10] The resulting blocks are used to form the fragments.

[10]The pairwise permutation of the cluster matrix is permitted since the affinity matrix formed in phase one is symmetric.

Navathe, *et al.* [NCWD84] extended Hoffer and Severance's work to include overlapping fragments, memory hierarchies, and replicated and non-replicated fragments. Ezeife and Barker [EB94] also used a similar approach to fragment objectbases.

2.4.3 Transaction-Based Vertical Fragmentation

Affinity-based fragment construction algorithms are centered on the attributes. By choosing such a fine-grained unit, semantic information about how the attributes are used in a user transaction is lost. As a result, these algorithms suffer from worst case computational complexities of $O(2^{m-1})$ where m is the number of attributes in the relation.

Transaction-based algorithms exploit the semantic information available in the set of user transactions to more efficiently produce the vertical fragments. For example, Hammer [HN79] observes that transactions in commercial systems tend to follow the "80/20" rule; that is, the most active 20 percent of the transactions account for 80 percent of the total usage. A transaction-based approach can use these transactions as a basis for constructing the fragmentation pattern to quickly produce an optimal solution.

Chu and Ieong [CI93] proposed a transaction-based algorithm that generates an optimal *binary* partitioning of a relation to reduce the amount of disk I/O. The algorithm begins with a pool of unassigned transactions and

the minimum disk access cost is computed using the following formula:

$$cost = \sum_{i=1}^{n} \sum_{j=1}^{d} f_i(l(I + F_j))$$

where n is the number of transactions, d is the number of generated fragments, I is the primary key of the relation, and F_j is the fragment accessed. The cost function for data access is f_i.

Chu and Ieong's algorithm proceeds as follows. A transaction is selected from the pool of transactions and its accessed attribute set is unioned with the set of attributes in the fragment and the access cost is computed using the above formula. If the access cost of this partitioning is greater than or equal to the minimum cost, this transaction is excluded and a new transaction is selected. If the access cost is less than the minimum cost, the minimum cost is set to the access cost, another transaction is selected from the pool, and the algorithm iterates until the pool of unassigned transactions is empty.

The worst case complexity of this algorithm is $O(2^n)$, where n is the number of transactions. Empirical studies have shown that the average complexity is on the order of $O(2^{0.6n})$.

Other splitting approaches, designed to minimize disk I/O, were described by Chu [Chu92]. The MAX algorithm partitions the relation by maximizing the following value:

$$\nu = \alpha + \beta - \phi$$

where α is the reduction in costs over the unpartitioned case for the transactions whose attributes fall completely within the new fragment, β is the

reduction in costs for transactions whose attributes fall completely in the old fragment, and ϕ is the increase in cost for transactions whose access pattern spans both fragments. The algorithm then computes ν for the $\binom{n}{i}$ combinations of transactions and selects the transaction grouping having the greatest ν.

The MAX computation requires a total of $2^t - 1$ models to be fitted to determine the best fragmentation where t is the number of transactions. For large transaction sets, this requirement leads to long execution times. Chu proposed the FORWARD SELECTION algorithm for such cases. The FORWARD SELECTION algorithm is based on the same computations as the MAX algorithm. Instead of exhaustively computing all models, however, the FORWARD SELECTION algorithm retains the single-transaction model which shows the greatest reduction in access cost. Once that transaction, also known as the *dominating* transaction, has been found, the remaining transactions are successively added and ν is computed each time.

Chu reports that the MAX algorithm captures 99.9% of the maximum possible improvements and returns the optimal solution in 93% of the cases. FORWARD SELECTION captures 93% of the maximum possible improvements with the optimal solution being returned 56% of the time. On the other hand, FORWARD SELECTION requires only $t(t+1)/2$ model computations which is a significant performance improvement over the MAX algorithm.

Chapter 3

Assumed Environment and Problem Evaluation

This chapter describes the environment where the proposed protocol operates. Objects, nested objects, and object access in a DSD system are more rigorously defined. Then the discussion will focus on evaluating the consistency protocol design problems in the DSD environment in contrast to the LOTEC[Sui98] protocol.

3.1 Assumed Environment

Following the notation of Graham [Gra94], this section begins by defining objects. A nested object model that uses versioning to define correct executions between sub-transactions is then described. Using versioning allows

for serialization based on *multi-version*, two-phase locking. Finally, object access in the DSD environment is be presented.

3.1.1 Object Model and Properties

To focus on memory consistency issues and to simplify the protocol design, the core object concepts outlined in Kim [Kim90] have been assumed. An object, in the POS, logically contains both structure and behaviour. The structure is represented by a set of uniquely identified data items (attributes) whose values capture the state of the object. A set of procedures (methods) which are exclusively able to alter the state of the object form the behavioural component of the object. Each object has its own unique object identifier (OID) which in the DSD-based POS corresponds to the virtual address where it is located.

In this thesis we will use a_{ij} to denote the j^{th} attribute and m_{ik} to denote the k^{th} attribute of object O_i. An object is defined as follows:

Definition 3.1. An *object* $O_i = (S_i, B_i)$ where:

1. i is the unique identifier of the object,
2. S_i is the object's structure composed of attributes such that $\forall a_{ij}, a_{ik}(j \neq k) \in S_i, a_{ij} \neq a_{ik}$,
3. B_i is the object's behaviour composed of methods such that $\forall m_{ij}, m_{ik}(j \neq k) \in B_i, m_{ij} \neq m_{ik}$. □

Point (1) assigns a unique name to each object. Point (2) specifies the attributes of the object and Point (3) specifies the methods of the object.

All objects which share the same set of attributes and methods may be grouped into a *class*. A class is associated with a single type which is a compile-time specification that defines what attributes are stored and which methods may be applied to an instance of that object type.

The object model in this thesis supports the three fundamental features of object-orientation: *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation, by restricting access to an object's attributes to its own methods, provides data abstraction, data independence, and locality of effect. Using encapsulation we can more easily infer what a correct execution of concurrent method invocations should be. Inheritance, of attributes and methods, allows the programmer to create relationships between classes. A new class may be derived from another class[1] which promotes the re-use of existing software. Polymorphism permits objects to be treated as if they were of their declared class or any super-class thereof. This property allows for the creation of more generic algorithms and data structures such as those found in the Java programming language [GJS96], the Microsoft Foundation Classes [Pro99], and the C++ standard template library [Jos99].

[1]The derived class is referred to as the sub-class of the original or *base* class. The base class is also called the *super-class* of the derived class.

3.1.2 Nested Object Transactions

Users of the proposed POS access objects by invoking methods on them. A method so invoked may alter the attributes of that object, invoke methods on other objects, or both. Method invocations can be treated as nested atomic object transactions since method invocations may nest within one another. The original method invoked by the user is referred to as the *root transaction*, while the subsequent method invocations triggered would each begin new sub-transactions. The root transaction and all its descendents together form a tree structure known as a *transaction family*.

In describing nested object transactions, we assume the persistent object system contains a set of objects $O = \{O_1, O_2, \ldots, O_m\}$. The notation m_{ik}^j describes invocation of method k on object i by user j. This invocation begins a nested object transaction T_i^j which is the root transaction. Operations of a nested object transaction may consist of read and write operations on that object's attributes, an indication that the transaction has entered the pre-commit (pc) state [Zap93], or sub-transactions initiated by method invocations on other objects. The set of sub-transactions of T_i^j, formed from all of the methods invoked during the execution of T_i^j, is written as $OT_j = \{\cup_{ik} m_{ik}^j\}$. Thus, we can define the set of all operations performed in T_i^j as $OS_j = \{\cup_k O_{jk}\} \cup OT_j$ where $O_{jk} \in \{read, write, pc\}$ is an operation k. Finally, the transaction's termination condition is written as $N_j \in \{Commit, Abort\}$.

One of the benefits of using nested object transactions is the greater opportunity for parallel execution. However, in systems such as LOTEC [GS99], conflict serializability must be maintained between sub-transactions which execute in parallel to prevent anomalies from occurring. Therefore, two methods that have conflicting operations on the same datum can not be executed in parallel. LOTEC attempts to maximize the potential concurrency by using internal semantic information about methods to determine which ones may be safely executed in parallel.

In a multi-versioned environment, the restrictions on parallel execution are not as severe. Because a new version of the datum is created with each write, write-write and read-write conflicts between [sub-]transactions cannot occur. The only conflicts that can possibly occur appear when a sub-transaction T_2 attempts to read a version of the datum created by another sub-transaction T_1. Although the write-read conflict can potentially occur, it can be eliminated by using a multi-version concurrency control policy that allows multiple, *uncertified* [BHG87] versions of a datum to co-exist and subsequent read operations to use those uncertified versions.

Using uncertified versions of a datum re-introduces the problem of cascading aborts. If the transaction that created a version of a datum aborts, any reader of the version must also be rolled back and so on. Since nested transactions must either commit or abort in their entirety, however, by restricting access to uncertified data to members of the transaction family we can easily contain any cascading aborts to that transaction family.

The nested object transaction model that will be used in this thesis can now be defined.

Definition 3.2. A *nested object transaction* is a partial order $T_i^j = (\sum_j, \prec_j)$ where:

1. $\sum_j = OS_j \cup N_j$,
2. if $O_{jl} = pc$, then O_{jl} is unique and $\forall O_{jk} \in OS_j$, $l \neq k$, $O_{jk} \prec_j O_{jl}$,
3. $\forall O_{jk} \in OS_j$, $O_{jk} \prec_j N_j$, and
4. the termination conditions of all $m_k^j \in OS_j$ are consistent and equal to N_j. □

Point (1) enumerates all of the operations of the nested object transaction. Point (2) requires that all operations of a nested object transaction must occur before the pre-commit operation. Point (3) ensures that no operation takes place after the transaction terminates. Point (4) ensures that the entire transaction family either commits or aborts.

3.1.3 Object Access in a DSD System

The high level structure of the proposed DSD system is similar to the DSVM systems proposed in Peters, *et al.* [PGB97] and Mathew [MGB96]. A block diagram is shown in Figure 3.1. The DSD system consists of some number of nodes connected by a high bandwidth, low latency network. The single

Shared Virtual Address Space(SVAS) provided by the DSD is globally distributed across all nodes. Thus, all objects stored in the SVAS are visible to all processes regardless of the objects' physical location. Object persistence is collectively provided by the disks at each processor.

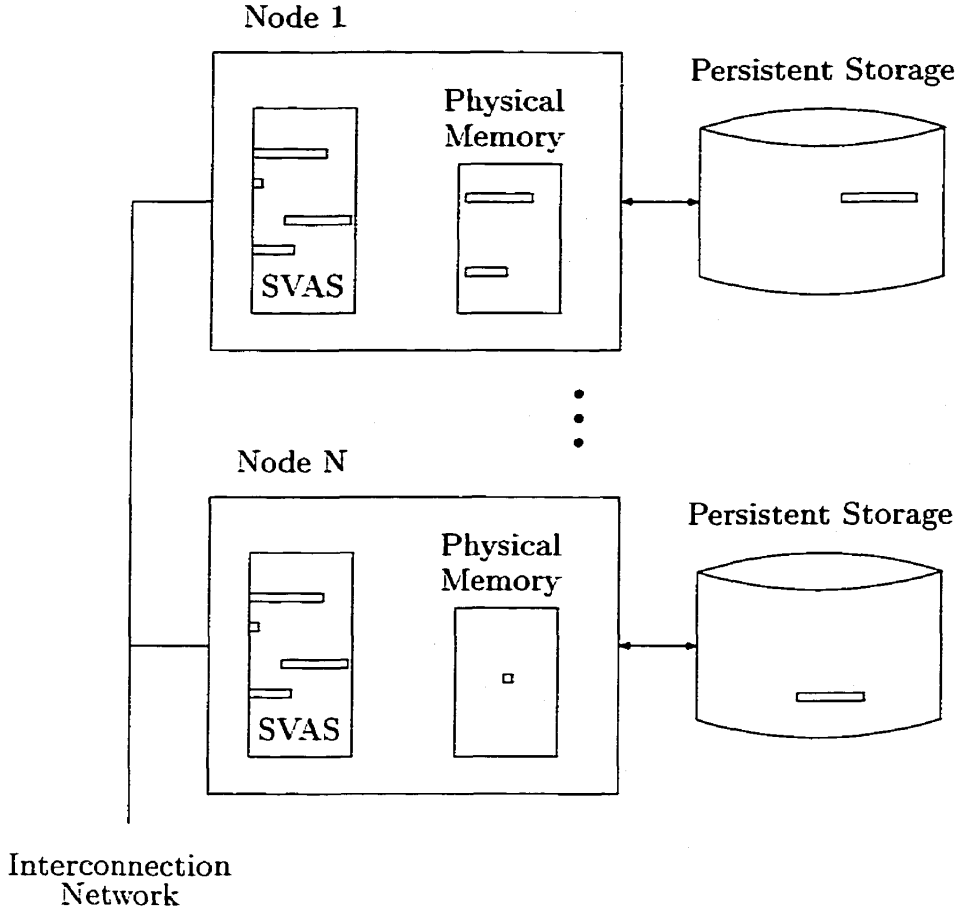


Figure 3.1: High Level Structure of the Distributed SVAS

The distributed SVAS manages objects via virtual memory operations. When an object's method is invoked, the base address of the method is

examined by the virtual memory hardware. If the address does not exist in the processor's page table, a *segmentation fault* is returned by the hardware to the SVAS. Should the address exist in the page table but not be resident in memory, the hardware will indicate a *page fault* to the SVAS. Either fault will trigger the necessary operations to bring the object into the RAM of the accessing processor.

To manage the potentially large number of objects, the SVAS system uses the services of a *Global Directory of Objects* (GDO). Mathew, *et. al.* [MGB96] have designed a GDO that provides object lookup by virtual address as well as information needed for concurrency control and consistency maintenance. Thus, when a method is invoked on an object, its address is passed via the operating system to the virtual memory hardware. On a segmentation fault, control is passed to the SVAS system which searches for the method in the GDO using the address as a key. If an entry for the address is not found, an invalid object reference error is signalled to the requesting process. Otherwise, the necessary method code and required data are mapped into the requesting process' address space and the up-to-date information is fetched. Control is then returned to the process and execution continues. Once the nested transaction execution ends, the addresses of the method and data are removed from the address space so that a segmentation fault will again be generated by subsequent accesses. This ensures that object access control will always be enforced.

3.2 Consistency Protocol Design Issues

The protocol described in this thesis improves upon the *Lazy Object Transactional Entry Consistency* (LOTEC) protocol defined by Graham and Sui [GS99]. In this section, the LOTEC protocol is reviewed and areas where improvements can be made are noted. Other factors that affect protocol performance is also discussed.

3.2.1 The LOTEC Protocol

LOTEC is a novel protocol which maintains memory consistency for closed nested object transactions in a DSM-based persistent object system. Like entry consistency (EC) [BZS93], it is a weak consistency protocol and reduces the number and size of the messages by propagating updates only to the next accessing site.

LOTEC is made easier to use by the programmer and achieves a greater reduction in communication overhead than EC by exploiting the properties of objects. Because of the object encapsulation property, it is possible to determine which attributes could be accessed by a transaction. Further, the necessary attribute access analysis can be performed at compile time so there is a little run-time overhead. Also, since the compiler decides the internal layout of each object, it is possible to determine the set of potentially updated pages for each method and insert needed lock acquisition and release calls. With this information, LOTEC is able to send only the needed data to the

next accessing site and also hides the complexity of this processing from the programmer.

Locking information is held in the GDO to provide efficient and reliable access from all sites. The locking operations require the following information:

LockState: A flag indicating the status of the lock (free, held- for-update, held-for-read, or retained).

ReadCount: A count of the number of concurrent readers.

HolderPtr: A pointer to a linked list of ⟨transaction id, node id⟩ pairs for transactions from the family currently holding the lock who have requested access to the object.

NonHoldersPtr: A pointer to a linked list of lists of ⟨transaction id, node id⟩ pairs for transactions from other transaction families who have requested access to the object.

Locking operations in LOTEC are conducted in accordance with the O2PL rules and are divided into two phases: global and local. The global phase of locking occurs when a [sub-]transaction attempts to gain access to an object for the first time. When a lock is granted during the global locking phase, a copy of the cache-able parts of the GDO entry for the object, as described in Sui [Sui98], and the up-to-date object state information are transferred to the site. Otherwise, the request is placed on the *NonHolders*

queue until the lock becomes available. When a lock is globally released, the releasing processor is responsible for sending updated information to the requester whose id is at the head of the NonHolders queue.

The local locking phase occurs each time a [sub-]transaction attempts to access an object. During the local phase, access attempts on objects held by other transaction families or on new objects are passed to the global phase. If, however, a lock acquisition attempt is unsuccessful and the lock is currently held by another sub-transaction within the same family, the request is queued locally. When the holding transaction pre-commits, the lock is granted to the next transaction in the *Holders* queue.

3.2.2 Assessment of LOTEC

LOTEC makes a number of simplifying assumptions and design decisions which affect the performance of the protocol. In this section, these assumptions and other design decisions are examined.

The LOTEC protocol is based on four initial assumptions:

1. consistency control is done on a per-object basis using object level locking;

2. only one object may be stored in a data page;

3. all sub-transactions of a given root transaction execute at the same node, and

4. no directly or indirectly recursive invocations are allowed within a transaction family.

Of the four assumptions, object level locking and the single object per page have the greatest impact on performance. By assuming object level locking, LOTEC takes a pessimistic approach to concurrency control. The O2PL locking rules ensure that only one transaction family can have access to an object at any point during execution. Since only one transaction family can hold the lock, it is impossible for two transaction families to invoke conflicting methods on the same object. Thus, serial access to the object is assured.

LOTEC relies on compile time information to determine which attributes are in which of the object's pages. Part of the information available is the attribute access analysis for each of the methods of the object and a call graph of all other methods invoked by a given method. With this information, it is possible to predict, albeit conservatively, which methods will conflict with one another. Knowing which attributes are affected in advance could allow the use of finer-grained locks, while still maintaining object consistency.

Other granularity problems in the LOTEC protocol are caused by the use of the page as the unit of transfer. Virtual memory pages are typically 4 kilobytes in size. If the object size is less than the size of a page, multiple objects can be stored in the same page. Multiple objects within the same page, however, creates an environment that allows the false sharing anomaly to occur. To avoid false sharing, LOTEC assumes that only one object can be

stored per page. In systems which hold large numbers of small objects, this assumption wastes a large portion of the available memory due to excessive fragmentation.

An object size greater than the page size creates problems as well. As the size of the object increases, there is an increased chance that the attributes accessed by a method will reside on more than one page. Therefore, the execution of that method will require a larger number of pages to be moved to the processing node than if those attributes were clustered in a single page. This condition results in greater overhead cost due to the unnecessary movement of data that will not be used during the course of that method's execution. In addition, wider dispersal of the attributes over the object's pages increases the danger of *ping-ponging*. Ping-ponging occurs when two or more processors make repeated, interleaved updates to the same memory page. A single page makes multiple trips across the network resulting in very heavy traffic. Although LOTEC reduces the number of pages that are transferred between sites, it is still vulnerable to the effect of ping-ponging.

3.2.3 Other Factors Affecting Protocol Performance

By moving from using a fixed-size memory page to an object fragment-based data transfer and locking unit, a number of factors have an increased impact on protocol performance. Among them, the network transmission characteristics and the structure of the data being transferred are the most important.

This section discusses these factors and assess their impact.

In any distributed system, the underlying network protocols can greatly impact the performance. The most common platform for distributed systems are Ethernet-based TCP/IP local area networks. On these networks, a data frame usually carries a maximum of 1500 bytes of data of which between 28 and 40 bytes are used for TCP/IP routing and other internal bookkeeping[Ste94]. To transfer a 4 kilobyte memory page between nodes in this environment requires between 3 and 4 data frames.

By reducing the size of the data transfer unit, we can make significant performance gains. Instead of requiring 4 large data frames to be sent per data transmission, a smaller transfer unit would allow fewer and smaller data frames to be sent. If the data transfer unit becomes too small however, more effort could be expended sending large numbers of data frames due to protocol stack overhead. Therefore, the protocol must ensure that data transfer unit is large enough to contain a useful amount of data, while still minimizing the number of data frames needed to effect the transfer.

In a DSM system, because the entire page must be sent, a requesting node will often receive data that will not be used by the invoked method. Even worse, the data can be spread over multiple pages triggering additional communication to satisfy the requirements of that method. In such cases, a smaller data transfer unit will reduce some of the communication overhead since only necessary portions of a page will be sent. A more optimal solution would be to load the payload of the data frame to be sent with only the data

that will be used and thus avoid the unnecessary data transfer. Selectively loading the payload introduces additional runtime overhead at both ends of the transfer however, so the protocol must minimize the impact of this loading overhead as well.

Chapter 4

Object Chunking

In this chapter, a new lighter weight approach to object fragmentation, called *object chunking*, is presented. Then the discussion focuses on the performance of object chunking, as assessed by a simulation study.

4.1 Object Chunking Algorithm

Data fragmentation improves system performance by reducing the amount of data that must be retrieved for a given attribute access and it also increases the opportunities for concurrent processing. For the proposed POS however, the fragment construction algorithms described in Chapter 2 are inappropriate for two main reasons. First, many of the algorithms require some *a priori* knowledge of the transaction execution pattern to construct the fragments. Second, the fragment's data will, over time, be redistributed to other nodes

via the transparent transfer mechanism of the POS. As a result, grouping all the attributes in one location is unnecessary. In this section, the object partitioning and layout requirements of the proposed POS are examined. The Optimal Binary Partitioning(OBP) Algorithm [CI93] is then analyzed in the context of the proposed POS. Next, the object chunking algorithms are defined and finally an example execution of the algorithm is presented.

4.1.1 Object Partitioning and Layout

For every object transaction in the proposed POS, the optimal partitioning of an object would deliver only the data required for a method's execution all in a single message. Achieving this goal depends greatly on the granularity of the concurrency control. We can only safely transfer the data, or a portion thereof, that is associated with a given lock, otherwise lost updates and other data anomalies may occur. If only a portion of a lock's data set is transferred however, other executing transactions will be delayed unnecessarily since they are precluded from accessing the data by the lock. Therefore, the best balance between concurrency and data integrity can be achieved when the granularity of the data transfer and concurrency control are the same.

In LOTEC, the concurrency is controlled by a lock associated with each object. Since there is only one object per page, the concurrency control is *coarse-grained*, which requires fewer locks and creates less locking overhead. The large concurrency unit reduces the potential concurrency and,

since the data transfer will also be large, unused data will often be transmitted. *Fine-grained* systems perform locking on individual data attributes. The attributes could be transmitted individually or bundled together dynamically. This approach delivers exactly the amount of data required by a transaction, but a premium is paid in locking, packaging and network overhead. By partitioning the objects into appropriate groups of attributes, locking overhead can be traded for increased concurrency and reduced data transfer costs in a controlled fashion. This approach results in a *medium-grained* system and will be the focus of the remainder of this thesis.

Unlike the algorithms described in Chapter 2, there is no need to physically separate the pieces of the object, since the POS will automatically handle the distribution. Instead, any fragmentation algorithm may be used to simply group the attributes for efficient access.

Once the attribute groupings have been identified, the object layout must be organized for efficient manipulation. Since data transfers will be done on a fragment basis, all attributes belonging to that fragment should be placed contiguously so that efficient block memory operations may be used during transfers. For statically allocated variables and arrays constructing such a layout is generally straightforward.

Fragments containing dynamically allocated storage require more care when being transferred. An object attribute containing the memory address of the dynamically allocated data will be sent to the requesting node but the data transfer can be delayed until the attribute is accessed. This approach

to data transfer is referred to as *lazy* propagation. An *eager* approach would instead append a copy of the dynamically allocated memory to the fragment and ship the information in one message.

4.1.2 Analysis of the Optimal Binary Partitioning Algorithm

Optimal binary partitioning [CI93], as described in Chapter 2, makes use of semantic information about transaction reference patterns to construct the data fragments. The need to know transaction characteristics, however, restricts the use of OBP to environments where the workload consists of known transactions. Since the transactions in the proposed POS are method invocations and an object's method set is known *a priori*, OBP is potentially a good algorithm for partitioning objects.

In the proposed POS however, OBP does not completely meet the requirements for fragmentation. OBP only creates a *binary* partition of the attribute set. As a result, objects that contain multiple clusters of high affinity attributes will not be partitioned sufficiently, and this partitioning will reduce the potential concurrency and performance gain from reduced data transmission. Repeated iterations of OBP will extract all of the attribute clusters, but may cause the total data transmission cost of an object to exceed the unpartitioned case.

Another area of concern is OBP's cost function. OBP was originally de-

signed to improve database performance by reducing the number and size of disk accesses. As a result, the cost function is based on the access frequency of a transaction, the number of disk scans per transaction execution and the disk access method. While these are important factors in disk performance, they are irrelevant in the proposed POS environment since most data transfers will be via the network. Thus, a new cost function needs to be developed.

4.1.3 Algorithms

To form fragments that are well-suited to the consistency protocol in this thesis, a new algorithm called *object chunking* is used. The object chunking algorithm is an extension of the *optimal binary partitioning* (OBP) algorithm.

Before discussing the details of the object chunking algorithm, structures for representing chunks, methods and attributes must be defined. An *attribute* is a tuple containing the unique attribute identifier (AID) and size, in bytes, of a datum. Since the partitioning is being done for network transmission purposes, the data type information for the attribute is irrelevant, only the size matters. A *method* is a tuple containing the unique method identifier (MID) and a set enumerating the attribute identifiers of all attributes that may (conservatively) be accessed by this method (aset). Finally, a *chunk* is a tuple containing a unique chunk identifier (CID), a set enumerating all attributes that belong to this chunk (aset), and a set enumerating all methods

that access members of the aset (mset).

Algorithm 4.1 Object Chunking

```
 1: INPUT: C; {chunk representing the unpartitioned object}
 2: OUTPUT: CLIST; {list of chunks generated}
 3: CLIST ← CLIST + C;
 4: repeat
 5:    changed ← false;
 6:    for all c ∈ CLIST  do
 7:       if  BinaryPartition(c, CLIST) == true  then
 8:          changed ← true;
 9:       end if
10:    end for
11: until changed == false;
```

Object chunking takes place at two levels. The first level, shown in Algorithm 4.1, addresses the extraction of multiple attribute clusters. The algorithm starts with a single chunk that contains all the attributes and methods of the object. The list of chunks is initialized with this base chunk, as shown on line 5, and the main processing loop is entered. Partitioning then proceeds in a binary fan-out fashion; if the first round partitioning attempt is successful the result will be two chunks. Second round partitioning will result in up to four chunks and so on. If no new chunks are created in a round, the algorithm ends. Should a new chunk be created, an attempt to partition all other chunks must subsequently be made. This additional attempt is required since the effect of the new chunk could not have been taken into consideration when the previous partitioning attempts were made.

Algorithm 4.2 describes the binary partitioning operation. This algorithm

Algorithm 4.2 Object Chunking

```
1: INPUT: C; {the chunk being partitioned}
2: INPUT: CLIST; {list of existing chunks for this object}
3: OUTPUT: Succ ← false; {flag indicating that a new chunk was created}
4: nc; {the new chunk created}
5: cagg ← Σ_{c∈(CLIST−C)} cost(c.aset) ×|c.mset|; {aggregate transmission
   cost of the other chunks}
6: best ← cagg + cost(C.aset) ×|C.mset|; {cost target that must be beaten}
7: changed ← false; {flag indicating that a lower cost has been found}
8: repeat
9:    agg ← cagg;
10:   take the first method m from C.mset and add it to nc.mset;
      take m.aset from C.aset and add them to nc.aset;
11:   for all n ∈ C.mset do
12:      if n.aset ⊆ nc.aset then
13:         take n from C.mset and add it to nc.mset;
14:      else if n.aset ∩ nc.aset ≠ ∅ then
15:         add n to nc.mset;
16:      end if
17:   end for
18:   agg ← agg + cost(nc.aset) ×|nc.mset|;
19:   agg ← agg + cost(C.aset) ×|C.mset|;
20:   if agg < best then
21:      best ← agg;
22:      changed ← true;
23:   else
24:      return all attributes newly added to nc.aset to C.aset;
25:      remove all newly added methods from nc.mset and place them in
         C.mset;
26:   end if
27: until all m ∈ C.mset have been tried ;
28: if changed == true then
29:   Succ ← true;
30:   CLIST ← CLIST + nc;
31: end if
```

follows the same approach as OBP. First, the total transmission cost for all of the existing chunks of the object is computed. This value is used as a baseline for evaluating the "goodness" of any candidate partitioning.

Object chunking uses the following function to determine the object's data transmission cost:

$$MC(A) = \frac{\sum_{a \in A} sizeof(a)}{MTU}$$

$$cost(A) = MO * MC(A) + \sum_{a \in A} sizeof(a)$$

where MO is the message overhead, in bytes, added by the network protocols when the chunk is sent to another node, A is the set of attributes, $sizeof()$ is a function that returns the size, also in bytes, of an attribute, $MC(A)$ is the number of messages required to transmit the chunk over the network, and MTU is the amount of payload that can be carried in a single message. This cost function is more appropriate to the intended environment than the original cost function for OBP since it more accurately reflects the cost of moving the object in the network.

After the baseline computation is completed, we move into the main processing loop. A method is selected and added to the new chunk's method set along with the attributes it uses. The aggregate transmission cost is computed using the newly created chunks. If the aggregate cost is lower as a result of the split, we indicate that a change has occurred and continue processing. If no improvement is detected, we remove the new method and its attributes from the new chunk and repeat the test using the next method.

The partitioning terminates when no methods remain to be tested. If a change occurs during processing, the partitioning algorithm returns true.

To improve processing performance, there are two optimizations in the algorithm. The first is the calculation of the aggregate transmission costs for all other previously defined chunks (found on line 4). To ensure that a partitioning does not cause an overall increase in transmission cost, the cost of transmitting other chunks must be taken into account during the current partitioning. To ensure a correct partitioning of the chunk however, the aggregate cost must be reset before each trial. By using the variable *cagg*, the need to recompute the other chunk's portion of the transmission cost is eliminated. The second optimization can be found in lines 11 to 17. Because multiple methods can access the same attributes, it is faster to test these methods as a group rather than individually. The remaining methods are tested to determine if they can be included in the group. This test is performed on line 12.

Methods that access multiple chunks are handled on lines 14 and 15. If a method accesses some of the new chunk's attributes but the method's attribute set is not contained in the new chunk's attribute set, we add the method to the chunk's method set without removing it from the original chunk's method set. This placeholder ensures that the method set contains an accurate count of the number of methods that access the chunk. Since the cardinality of the chunk's method set is used to calculate the chunk's aggregate transmission cost, the placeholders help prevent inappropriate par-

titioning.

4.1.4 Example Execution

To illustrate the algorithm's operation, a very simple example object consisting of 4 attributes and 5 methods as shown in Figure 4.1, is partitioned using object chunking. The network environment for the POS is assumed to be Ethernet which has an overhead of 40 bytes per message.

Object O

sizeof(a_0) = 23	m_0.aset = $\{a_1\}$
sizeof(a_1) = 15	m_1.aset = $\{a_0, a_3\}$
sizeof(a_2) = 63	m_2.aset = $\{a_3\}$
sizeof(a_3) = 5	m_3.aset = $\{a_2\}$
	m_4.aset = $\{a_1\}$

Figure 4.1: The Object before Partitioning

The object is passed to the chunking algorithm in chunk form and it is placed in the chunk list. Since it is the only chunk in the chunk list, it is passed to the binary partitioning routine. There are no other chunks to contribute to the baseline cost calculation, so the best value is calculated as shown below. The total size of all the attribute is less than the 1500 byte MTU of Ethernet, so $MC(A)$ in all cases is 1 and is therefore omitted.

$$
\begin{aligned}
best &= cost(O.aset) * |O.mset| \\
&= (MO + \sum_{a \in A} sizeof(a)) * 5 \\
&= (40 + 23 + 15 + 63 + 5) * 5 \\
&= 146 * 5 \\
&= 730
\end{aligned}
$$

Execution proceeds into the search loop. Method 0 is selected and it is placed, along with its attributes in the new chunk. A scan of the remaining methods finds that method 4 uses attributes that are completely contained in the new chunk so it is moved as well. No overlapping methods are found, so we calculate the new aggregate cost. The new chunk contributes 110 bytes to the new total (two methods times 55 bytes of transmission costs) and the original chunk contributes 393 bytes for a total of 503 bytes. Since this partitioning results in less data transfer, we will continue to explore this branch so we set the changed flag and update our best value.

In the second iteration of the search loop, method 1 is selected for addition to the new chunk. Method 2 is found to use attributes found only in the new chunk, so it too is added. Again, no overlapping methods are found so the aggregate cost is computed. This time, the new chunk contributes 332 bytes while the original chunk contributes 103 bytes for a total of 435 bytes. Again, we have found a better partitioning, so we update the best value and continue processing.

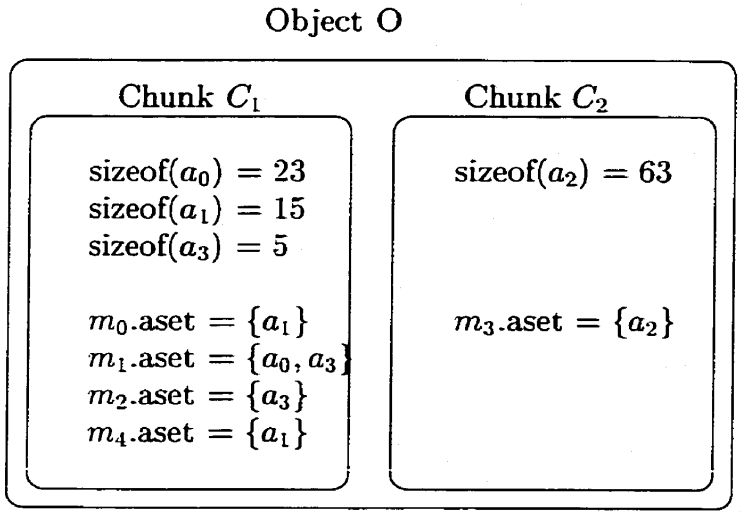


Figure 4.2: The Object after the First Pass

There is only one method to test in the third iteration of the search loop and its addition results in no improvement over the previous iteration, so the search loop ends. The new chunk is added to the chunk list and the binary partitioning routine returns true. Since we have successfully partitioned, the main loop will iterate again. Figure 4.2 shows the results after this first iteration.

There are now two chunks in the list. Chunk 1 is passed first to the binary partitioning routine. This time chunk 2 contributes to the transmission cost calculation giving a base value of 435 bytes. Again, the first method selected in the search loop is method 0 which is added to the new chunk along with method 4. When the aggregate cost is calculated, chunk 2 contributes 103

bytes, the new chunk contributes 110 bytes and the original chunk contributes 136 bytes for a total of 349 bytes. This total is less than the baseline, so we update the baseline and iterate again.

No improvement is seen when method 1 is tested so no update is made since its addition would move all of the attributes and methods to the new chunk from the old chunk. When method 2's addition to the new chunk is tested, attribute a_3 is moved to the new chunk's attribute set. This movement creates the condition where a method must retrieve multiple chunks. This condition is detected during the scan for contained methods. In this case, the intersection of method 1's attribute set and the new chunk's attribute set is not empty which causes method 1 to be added only to the method set. The aggregate cost is calculated with the new chunk contributing 4 methods times 60 bytes or 240 bytes, the original chunk contributes 1 method times 63 bytes and chunk 2 contributes its 103 bytes for a total of 406. This addition does not improve the overall cost, so method 2 is removed from the new chunk. No further methods remain to be tested so the binary partitioning routine reports a successful partition and adds the new chunk to the chunk list.

Since chunk 2 has only one method, it cannot be partitioned again. The second round of the chunking algorithm ends but, since there was a successful partitioning, a third pass is required. Chunks 1 and 3 cannot be partitioned since there is only 1 attribute in each of them. Chunk 2 will not be partitioned because splitting that chunk will cause the total transmission cost to increase.

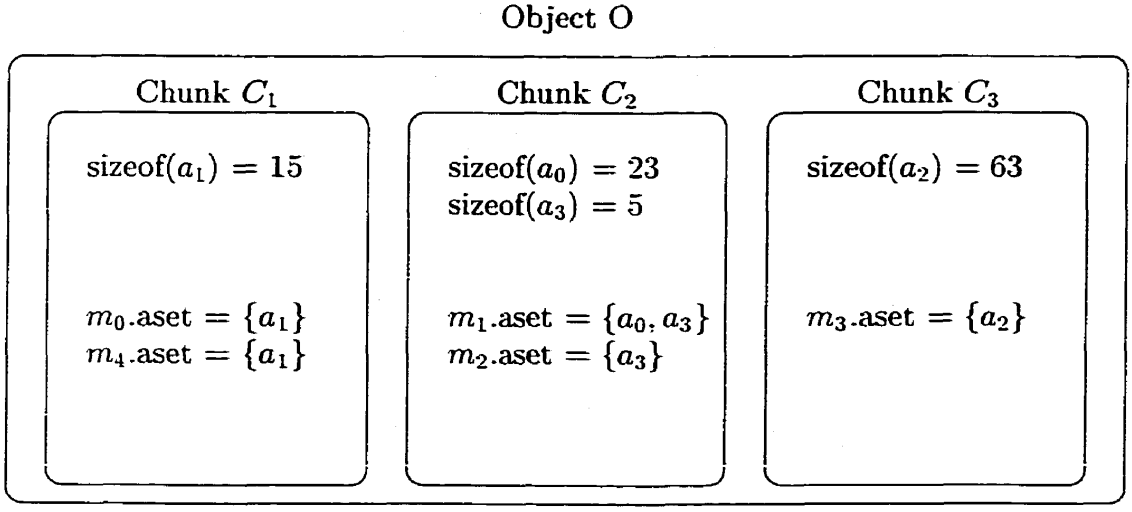


Figure 4.3: The Object after the Second Pass

Since no partitioning can occur in the third round, the algorithm terminates.

4.2 Simulation Study

Partitioning an object into chunks can reduce the amount of data that must be transmitted across the network for a given method invocation. In some applications, however, little or no performance gain may result because the attribute access pattern precludes an efficient partitioning of the object or because the overhead of transmission exceeds the amount of data. This study attempts to determine where partitioning objects generates the greatest performance gain by examining objects of varying size taken at random from the object space. The simulation strategy and parameters are discussed

first. Then, the design of the simulator is described. Finally, the results of the simulations are presented and discussed.

4.2.1 Simulation Strategy and Parameters

The goal of chunking is to reduce an object into a number of smaller, more manageable pieces. This operation is heavily dependent on the size of the object being partitioned. An object's size is determined by five basic parameters: the number of attributes within the object, the size of each of the object's individual attributes, the number of methods in the object's operation set, the size of the object's individual methods, and the size of the object's headers and other administrative components. In addition to object size, the chunking operation is also greatly affected by the attribute use in each method. Together, these parameters form a six dimensional space according to which all objects may be characterized.

It is impossible to test all of the combinations of such a large space, so the simulation must limit the search space to a more reasonable size. A large number of objects will be stored in the proposed POS. Each of these objects has its structure and layout determined *a priori* by its class definition and the object language specification. As a result, the size of the object overhead information is known in advance and is a constant value. The layout information of a class is so frequently used, the class files could be cached at each node in the POS. Further, these class files will contain the executable code for each method as well. Therefore, by using features of the

POS environment, we can safely eliminate the "size of method" and "object overhead" dimensions from the object space.

Even though we have reduced the number of dimensions of the object space from six to four, it is still impossible to exhaustively test the remaining space. Instead, this simulation will characterize the effectiveness of object chunking over a portion of this remaining space. Representative objects will be generated at various attribute number, method number, attribute size, and attribute usage points in the object space and then be broken into chunks using the chunking algorithm.

4.2.2 Simulator Design

A simple simulator has been constructed to evaluate the object chunking algorithm over the space bounded by the parameters listed in the previous section. It is implemented as a single process which generates and partitions an object using object chunking. Upon completion, the simulator reports the number of chunks created, total aggregate transmission cost of the unpartitioned object, and the aggregate transmission cost of the partitioned object.

4.2.3 Simulation Results

In this section, Chunking algorithm performance is evaluated. To analyze the effect of chunking, sample objects were generated with a fixed number of attributes ranging over the powers of 2 from 2 to 512. The number of

methods per object ranged as the attributes, but were never allowed to be less than the number of attributes in the object nor greater than 8 times the number of attributes[1]. The attribute sizes were drawn from a standard normal distribution with a mean value ranging over the powers of 2 from 2 to 1024. The access patterns were generated randomly with the number of attributes accessed drawn randomly from a standard normal distribution with mean value of 25%, 50% or 75% of the number of attributes in the object. Performance is measured as a ratio of the aggregate transmission cost for the chunked object to the aggregate transmission cost of the unchunked object. The higher the ratio value, the less improvement results from chunking the object.

Observations

Figures 4.4 and 4.5 present the overall results of the experiment viewed along each of the dimensions of the object space.

In Figure 4.4, the x-axis shows the number of attributes, methods or the mean size of an attribute in the object and the y-axis reports the mean transmission cost ratio. As expected, the transmission cost ratio improves for chunked object as the number of attributes, number of methods, and mean size of attribute increase. With the increase in the number of methods and attributes, the number of opportunities to create a chunk also increases and thus the potential performance improvement also increases. More interest-

[1]The number of methods was never allowed to exceed 1024.

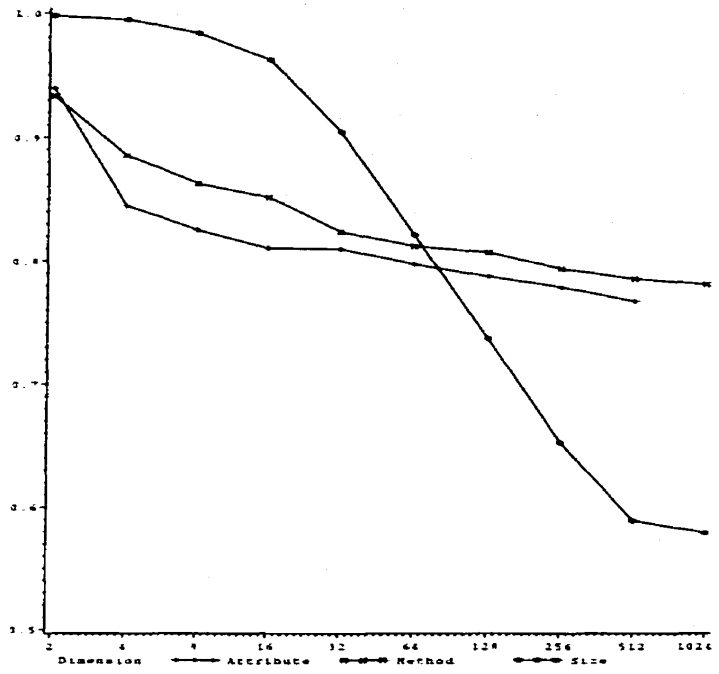


Figure 4.4: Plot of Mean Transmission Cost Ratio versus Number of Attributes, Methods, and Mean Attribute Size

ing is the effect of the attribute size. When attributes are small, chunking provides very little benefit. Once the attribute size crosses a threshold, however, significant performance gains are realized by chunking the object. This effect could be the result of network overhead resulting from the additional messages required to send the chunks to their destination.

Figure 4.5 shows the effect of the number of attributes accessed by a method on the transmission ratio. In this graph, we see that performance declines as the number of attributes accessed per method increases. This result is due to increased *overlap* between methods. When two methods access a large fraction of the attributes, it is very likely that they both will

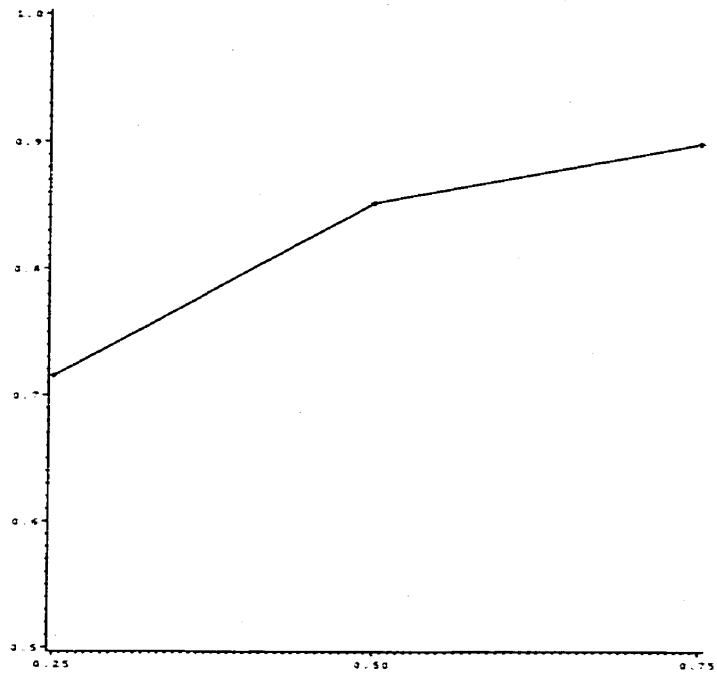


Figure 4.5: Plot of Mean Transmission Cost Ratio versus Mean Attribute Access Percentage

use a common subset. To access the attributes in this common subset, the method will have to load additional chunks to execute. As this fraction decreases, the incidence of overlap decreases and performance improves.

To obtain further insight into the utility of chunking objects, interactions between the various parameters are examined. Of particular interest are the interactions between attribute size and the other parameters. These interactions are shown as surface plots in figures 4.6 to 4.8.

Figure 4.6 shows the interaction between the number of attributes and the mean attribute size. These values are given as the power of 2 exponents along the x and y axes respectively. This plot confirms our earlier conjecture

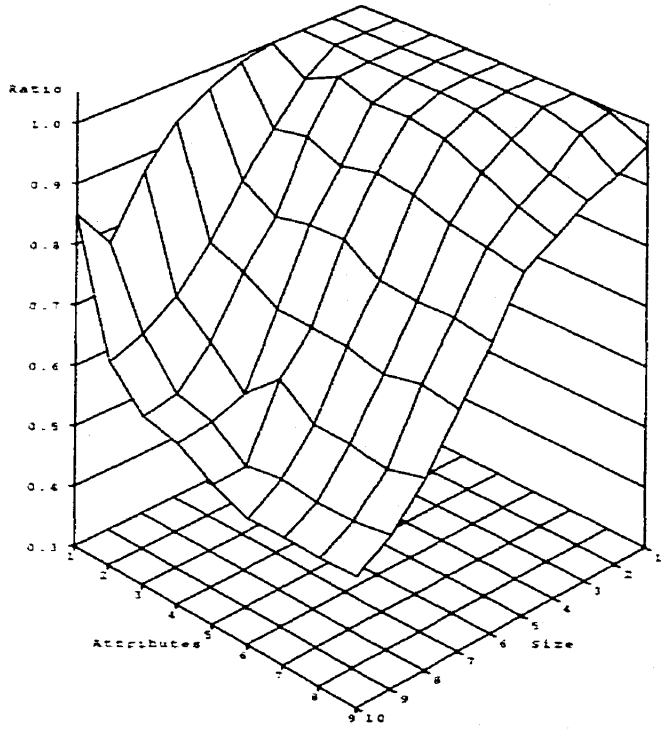


Figure 4.6: Plot of Mean Transmission Cost Ratio versus Number of Attributes and Mean Attribute Size

that performance improves as the number of attributes and the mean size of attribute increases. The interaction of these two parameters however, identify some interesting characteristics of chunking. The first feature we notice is the inability of chunking to improve the transmission costs for objects made up of small sized attributes. Intuitively, we expect that more chunks will be created in objects containing more attributes. The shape of the surface shows that while the number of attributes is important, the resulting chunks must have sufficient size to warrant the additional transmission overhead required

to move them. The concave shape of the surface indicates that it is easier to form chunks of sufficient size as the mean attribute size increases.

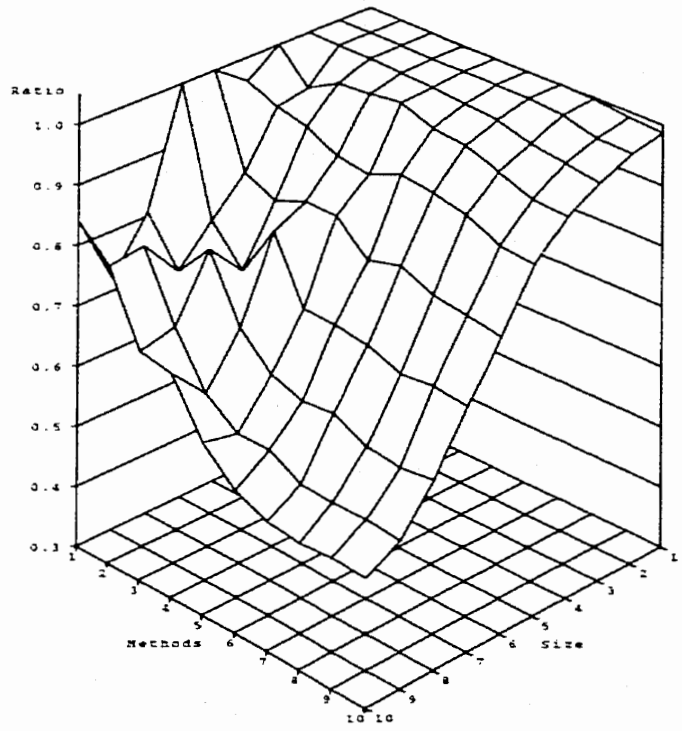


Figure 4.7: Plot of Mean Transmission Cost Ratio versus Number of Methods and Mean Attribute Size

Figure 4.7 shows the interaction between the number of methods and the mean attribute size with values given as power of 2 exponents along the x and y axes respectively. Like figure 4.6, a large number of methods does not guarantee that chunks will form, but chunks are more easily formed as the attribute size increases. The interesting feature of this plot is the very sharp changes in the performance improvement for objects with few methods.

With few methods to choose from, there is little variety in the attribute access patterns in the set. As a result, the size of individual attributes has a much greater impact on the size of the chunks and thereby increases the variation in the transmission cost ratio.

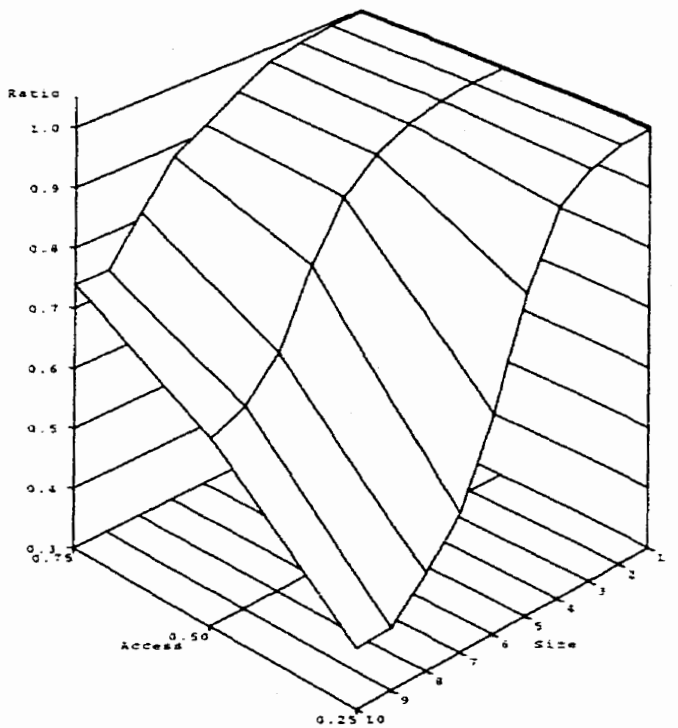


Figure 4.8: Plot of Mean Transmission Cost Ratio versus Mean Access Percentage and Mean Attribute Size

Interaction between the attribute access percentage and the mean attribute size is shown in figure 4.8. The three points tested form the x-axis and the attribute size is given as power of 2 exponents along the y-axis. The concave surface indicates that the transmission cost improves as the size of

the attributes increases and the number of attributes accessed by a method declines. The simulation suggests that the algorithm is able to achieve maximal improvement by forming large, independent chunks – this agrees well with intuition.

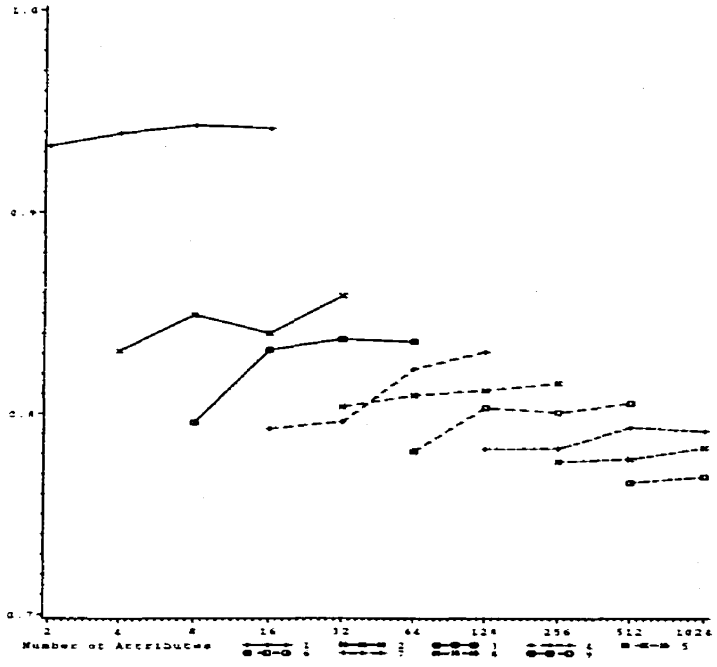


Figure 4.9: Plot of Mean Transmission Cost Ratio versus Number of Methods Grouped by Number of Attributes

The interaction between the number of methods and number of attributes in the object, shown in figure 4.9, is also interesting. In this graph, the x-axis shows the number of methods in the object. The lines indicate the mean transmission cost ratios for objects having the specified number of attributes, reported as power of 2 exponents. Here, the results are clearly stratified by the number of attributes with each successively larger number

showing greater performance improvement which confirm our intuition that the benefits of chunking increase with the size of the object. Increasing the number of methods for an object containing a given number of attributes appears to slightly reduce the performance improvement. The best performance is found when the number of methods equals the number of attributes in the object, but performance degrades as the number of methods increases. This effect can be attributed to the increased likelihood of access overlap in large sets of methods.

Additional parameter plots may be found in Appendix A.

Analysis and Discussion

The observations suggest that interaction between the previously identified four parameters impacts the performance improvement resulting from object chunking. To gain a better understanding of the relationships between the parameters and the significance of the interactions, a multiple linear regression analysis was performed. This analysis produced the following analytic model to describe the performance of chunked objects:

$$\begin{aligned} Y' = {} & 0.97409 + 0.3656a + 0.047577m + 0.003475s + 0.026832p \\ & -0.006453am - 0.025652as - 0.024799ap - 0.014929ms \\ & -0.036597mp - 0.016139sp + 0.003016ams + 0.00373amp \\ & +0.026098asp + 0.01676msp - 0.003166amsp \end{aligned}$$

where a is the number of attributes in the object, m is the number of methods, s is the mean attribute size and p is the mean percentage of attributes accessed per method. This model was significant at the 99% confidence level and fit the observations well ($R^2 = 0.8517$).

The two, three, and four parameter interactions were also tested for significance. Most of the interactions were significant at the 99% confidence level except the attributes and percentages, the methods and percentages, the size and percentages, and the attributes and methods and percentages interactions. These results suggest that the number of attributes accessed by a method has no additional impact on chunked object performance due to interaction with the other parameters and could result in a more refined analytic model. Additional investigation of these interactions is warranted.

A more surprising result was discovered in the test of the attributes and methods interaction. This interaction was not significant at the 99% confidence level, but was significant at the 95% confidence level. Intuitively, we would expect the object's size to be determined in large part by these two parameters so an interaction effect between these two parameters should be clearly significant. The marginal result seen here seems to suggest that this interaction is not as strong as we believe. Further investigation of this result is warranted but is beyond the scope of this thesis.

To validate the analytic model, the experiment was run a second time and the observed values were correlated with the predicted values generated by the model. The predicted values fit well with the observations ($R^2 = 0.8640$)

which suggests that this model is an accurate predictor of chunked object performance. Two additional validation runs were performed and showed a equally good fit to the model.

A more detailed report of the statistical analysis may be found in Appendix B.

Chapter 5

Versioned Object Transactional Consistency Protocol

This chapter introduces a versioned object transactional consistency protocol based on versioned nested object transactions, versioned objects, and the use of small data transfer units (the chunks previously described). Versioned nested object transactions and versioned objects will be discussed in depth in this chapter and then the protocol design will be presented. Finally, protocol performance will be (informally) compared to the performance of the LOTEC protocol[Sui98].

5.1 Serializability of Versioned Closed Nested Object Transactions

Versioned closed nested object transactions are a synthesis of object two-phase locking (O2PL) [Sui98] and two-version two-phase locking (2V2PL) [BHG87] concurrency control. This section defines the versioned object two-

phase locking (VO2PL) rules used in this thesis and shows that they are correct.

5.1.1 VO2PL Locking Rules

The rules for VO2PL are:

1. A transaction T may acquire locks as follows:

   (a) T may acquire an access lock if T holds no lock on the datum and no other transaction holds or is waiting to obtain a certify lock;

   (b) T may acquire a certify lock if T holds an access lock on the datum and no other transaction holds an access or certify lock;

   (c) T may acquire a read or write lock if it holds an access lock or its ancestor retains the access lock and/or any matching lock and no other sub-transaction holds a read or write lock, and

   (d) if T depends[1] on a transaction T', T' has completed;

2. Once a lock has been acquired it is held until T commits or aborts;

3. Transaction T cannot [pre-]commit until all subordinate transactions have pre-committed. When a sub-transaction T pre-commits, the parent of T inherits all its locks both held and retained. After that, the parent retains all the locks;

[1] A transaction T depends on transaction T' if T reads a datum created or updated by T'.

4. When transaction T aborts, it releases all locks it holds and retains. If any of those locks are retained by any of T's ancestors, they continue to retain those locks;

5. When the root transaction T pre-commits, access locks held by T on data that T updated are upgraded to certify locks, and

6. When the root transaction T commits, it releases all locks which were held by itself and all of its sub-transactions. This makes them available to other transaction families.

Rule (1) enforces an order on versioned nested object transactions. Since each transaction family operates on its own copy of the data, synchronization is required on two levels. The first level controls concurrency between transaction families. Rule (1a) allows any transaction family to obtain the most recently certified version of the data until another transaction family indicates that it wishes to certify a new value. Rule (1b) ensures that certifications occur serially and only after all of the readers of the data have commited. Together, these rules ensure that transaction families always obtain the most up to date version of the data while preventing lost updates from occuring.

Within a transaction family, rule (1c) prevents sub-transactions from concurrently accessing the same object in a conflicting manner. Finally, like O2PL [Sui98], data access is not restricted to leaf transactions. This feature creates a situation where the execution of one transaction may be depen-

dent on the results of another transaction. Rule (1d) ensures that dependent transactions are executed in order.

Rule (2) ensures that each nested object transaction uses strict two-phase locking by requiring that all locks be held until T commits or aborts. Using strict two-phase locking avoids the problem of cascading aborts.

Rule (3) defines the partial release of corresponding object locks. By allowing partial releases, other sub-transactions within the transaction family are permitted to see internally generated changes and make further updates.

Rule (4) defines the required actions when a transaction aborts. Since all retained and held locks of the aborting transaction are released, blocked transactions can resume execution.

To prevent lost updates from occurring, new versions of a datum must be certified in some serial order. However, in a nested transaction environment, the transaction family must appear to execute atomically. Rule (5), by delaying acquisition of certify locks until pre-commit, ensures that all updates are certified together preserving the appearance of atomicity to other transaction families in the system.

Finally, rule (6) specifies that all locks held or retained during the execution of a transaction family be released when the root transaction commits. This action makes all newly certified data available to other transactions.

5.1.2 Correctness of VO2PL

Any set of locking rules that produces a history that is equivalent to a serial execution of transactions is considered correct. O2PL and 2V2PL have been shown to be serializable, but they are based on different transaction models. VO2PL extends the properties of multi-version concurrency control to closed nested object transactions, so it must be shown that the nested object model has no impact on serializability and that VO2PL is serializable both between transaction families and within a transaction family.

Impact of the Nested Object Model on Serialization

Sui [Sui98] observed that method invocations may only affect serializability if two methods are invoked on the same object by a single nested object transaction or if multiple nested object transactions attempt to invoke a method on the same object concurrently.

Nested object transactions are methods invoked on an object and therefore are granted access to the object's attributes through the encapsulation property. A nested object transaction may also invoke other methods which, in turn, may invoke other methods and so on. If any of these child transactions are invoked on the same object as the parent transaction, deadlock occurs since the parent cannot proceed until the child completes and the child cannot proceed until the parent releases the lock. O2PL addresses this problem by assuming that no direct or indirect recursive method invocations can occur within a transaction family. Since VO2PL is based on O2PL, the

same assumption is made. Violations of this "programming contract" can be detected at run time and reported to the programmer by a deadlock detection procedure that is run periodically. Therefore, two methods invoked on the same object have no impact on the serializability of VO2PL.

In VO2PL, when a method is invoked on an object for the first time, a copy of the data is brought to the processing node and all subsequent accesses are directed to that local copy. As a result, each transaction family has its own version of the attributes used by that method and there is no possibility for a conflict between transaction families to arise until the updates are made visible to the system. Rules (1a) and (1b) of VO2PL ensure the serializability of updates, so method invocation also has no impact on the serializability of nested object transactions.

Inter-family Serializability

Since it is based on the 2V2PL rules, VO2PL will be serializable if it meets the same correctness criteria. Bernstein[BHG87] identifies seven properties that must be satisfied for 2V2PL to be serializable. Using $r_j[x_i]$ to denote a read operation by T_j on the i^{th} version of datum x and $w_j[x_i]$ to denote a write operation by T_j on the i^{th} version of datum x, they are:

1. Let f_i denote the certification of T_i. For every T_i, f_i follows all of T_i's reads and writes and precedes T_i's commitment.

2. For every $r_k[x_j]$ in H, if $j \neq k$, then $c_j < r_k[x_j]$; otherwise $w_k[x_k] < r_k[x_k]$.

3. For every $w_i[x_k]$ and $r_i[x_j]$ in H, if $w_i[x_k] < r_i[x_j]$, then $j = k$.

4. If $r_k[x_j]$ and $w_i[x_i]$ are in H, then either $f_i < r_k[x_j]$ or $r_k[x_j] < f_i$.

5. For every $r_k[x_j]$ and $w_i[x_i]$ (i, j, and k distinct), if $f_i < r_k[x_j]$, then $f_i < f_j$.

6. For every $r_k[w_j]$ and $w_i[x_i]$, $i \neq j$ and $i \neq k$, if $r_k[x_j] < f_i$, then $f_k < f_i$.

7. For every $w_i[x_i]$ and $w_j[x_j]$, either $f_i < f_j$ or $f_j < f_i$.

Property (1) requires that the datum be certified before the transaction completes. Properties (2) and (5) require that a transaction read either the last certified version of a datum or a version written by itself. Property (3) ensures that the version read is always the version most recently written by that transaction. Strict ordering of operations is provided by properties (4), (6) and (7). Because each transaction that writes must obtain a certify lock, property (4) forces T_i to delay until T_k has been certified or T_k is delayed until T_i finishes certifying. By property (6), a new version of a datum cannot be certified until all transactions that read a version of that datum have certified. Finally, property (7) requires that certifications of writes on the same datum be atomic.

In the transaction model described in Chapter 3, all read and write operations must be completed before the nested object transaction pre-commits. By rule (6), certification can only take place when the root transaction pre-commits. Therefore, all of the reads and writes must have completed by the

time certification takes place which fulfills the requirement of property (1).

When a method is invoked for the first time, the required data is brought to the requesting node from the site where the last certification occurred. Thus, the first invocation of the method always begins with the last certified version of the data. All operations are subsequently directed to this local copy of the data so, in effect, reads can only obtain the last certified version or a value written by a member of the transaction family. This meets the requirements of properties (2) and (3).

Rules (1a) and (1b) of the locking rules for VO2PL govern the ordering of the certifications in VO2PL. By rule (1a), any transaction that is in the process of certifying or that indicates that it wants to certify a new version of a datum forces any new access to wait until certification is complete. Since the new transaction cannot proceed until the certification completes, it is not possible for that new transaction to certify before the current holder of the certify lock. Therefore, rule (1a) fulfills the requirement of property (5).

Rule (1b) forces any transaction that wishes to certify a new version of a datum to wait until all transactions that have previously accessed that datum have certified. There are two cases that require consideration: a read-only transaction precedes the certifier and an updating transaction precedes the certifier. If no update has been made, the currently accessing transaction will proceed to completion unimpeded. This is equivalent to certifying an unchanged value and meets the requirement of property (6). If an update has occurred, the transactions deadlock and result in one or more of the

transactions being aborted and re-run. Since an aborted transaction must release all of its locks by rule (4), eventually one of the transactions will be able to certify. Thus, an order of updates will be created satisfying property (7).

Finally, since both property (5) and (6) are satisfied by VO2PL, it must follow that property (4) is also satisfied. With all of the properties satisfied, VO2PL must also be serializable between transaction families.

Intra-family Serializability

For conflicts that arise within a transaction family, VO2PL uses a similar approach to O2PL. Since recursive method invocations are prohibited, deadlocks due to ancestor-descendent relationships cannot occur. Dependent sub-transactions are also precluded from concurrent execution by rule (1d) as in O2PL.

Rule (1c) permits a sub-transaction to obtain a read or write lock only if it was the first accessor of the data or all conflicting sub-transactions have pre-committed and the lock has been retained by an ancestor of the requesting transaction. As a result, only one sub-transaction within a transaction family may hold a read or write lock at a time. Other requesting sub-transactions are blocked until the holding sub-transaction pre-commits and an ancestor of the requestor comes to retain the lock. Thus, conflicting sub-transactions are executed in order of lock acquisition starting with the initial accessor of the datum and proceeding from there in depth-first order. Since a depth-

first order is the correct serial order within a transaction family, VO2PL is serializable within a transaction family.

Since closed nested object transactions have no impact on serializability, the execution of sub-transactions within a transaction family is serializable, and the executions of groups of transaction families are serializable, VO2PL must produce a serializable execution order.

5.2 Algorithms Implementing Versioned Object Two-phase Locking

The transaction manager must implement the closed nested versioned object two-phase locking rules described earlier. In this section, the implementation details of these rules are reviewed. First, transaction identification requirements are examined. Then, since the proposed system requires locking on two levels, the lock structures and algorithms for global and local locking are also defined.

5.2.1 Transaction Identifiers

To maintain an accurate record of where and by whom a chunk is being used, a unique identifier is needed. In the proposed system, the transaction identifier (TID) defined by Sui [Sui98] is used. Sui denotes an invocation of method k on object O_i made by user j by m_{ik}^j. A root transaction created by the invocation of m_{ik}^j is denoted by T_i^j. A generic, unique [sub-]transaction identifier has the form $T_{i,l_1,\ldots,l_{d-2},l_{d-1}}^j$ where T_i^j is the root transaction iden-

tifier, d is the depth of the transaction nest, and $l_1 \ldots l_{d-1}$ enumerates the location of the sub-transaction within the transaction family tree.

More than one instance of a transaction could be executing in the system, however, so to more precisely identify a transaction, the node identifier (NID) must also be recorded. This additional information reduces the possibility of conflicting operations being executed by transactions with identical TIDs.

5.2.2 Global Algorithms and Lock Structure

An object O_i in the proposed POS is composed of a number of data chunks. Therefore, since locking and data transfer is done on a per chunk basis, each chunk must have a supporting locking structure in the system to indicate the state of the certify locks for that chunk. In addition, lists of currently accessing transaction families, those families waiting for a new version to certify, and the location of the most recently certified version must be maintained to provide efficient lock management and data access. Thus, the global lock structure for chunk C_{ij} of object O_i is composed of:

CertifyLock: a flag. When the CertifyLock is '1', a transaction family has indicated that it intends to certify a new version of the chunk and new accessors must wait. If CertifyLock is '0', new accessors may use the chunk without delay.

CertifyHolder: the TID and NID of the transaction family holding the CertifyLock.

AccessCount: an integer variable that indicates the number of transaction families currently using the chunk.

AccessList: a list of ¡TID, NID¿ pairs for the transaction families that are concurrently accessing the chunk.

WaitingList: a list of local lock structures (defined later) for local transaction families that are waiting for the new version of the chunk to be certified.

LastCertifier: the TID and NID of the last transaction to certify the data in this chunk.

In the environment in which the proposed POS will operate, it was assumed that the GDO [MGB96] would be used to manage the large collection of persistent objects contained in the POS. Each entry in the GDO contains the object's identifier (OID) which is used as the search key. Mathew, *et. al.* [MGB96] suggested that a lock variable could be stored in each GDO entry. Sui [Sui98] extended this idea and used the GDO to store the global lock and cache consistency information. Using the GDO in this fashion makes the locking information available to all the nodes in the system. The proposed protocol will make similar use of the GDO. However, instead of using a separate entry for each chunk of the object, the lock structures will be stored as an array within the GDO entry. Using arrays prevents the GDO from growing too quickly with the number of objects and chunks and thus avoids slowing the retrieval of information from the GDO.

For each object O_i, there is a lock structure L_{O_i} that holds all the lock information for all the chunks C_{ij} of O_i. When a method is invoked, the transaction manager initiates an object transaction and executes an access operation on the appropriate lock structure for the desired chunks. Once access has been granted, the transaction manager allows the object transaction to proceed, otherwise the object transaction is blocked until the certify operation that is in progress completes. Thus, the transaction manager ensures that all object transactions read the most recently certified version of the data.

Algorithm 5.1 GlobalLockAcquisition

```
1:  INPUT: TID; {Transaction ID of the requestor}
2:  INPUT: C_ik; {Chunk being accessed}
3:  INPUT: GDO_ik; {GDO entry for chunk C_ik}
4:  INPUT: NID; {Processor where the transaction executes}
5:
6:  GDO_ik = GDOLookup(C_ik);
7:  if GDO_ik.CertifyLock == "held" then
8:     if [TID,NID] ∉ GDO_ik.WaitingList then
9:        create a local lock structure LLS for [TID,NID];
          append LLS to GDO_ik.WaitingList;
10:    else
11:       append TID to GDO_ik.WaitingList[TID,NID].WaitingList;
12:    end if
13: else
14:    GDO_ik.AccessCount += 1;
       append [TID,NID] to GDO_ik.AccessList;
       create a local lock structure LLS for [TID,NID];
       send LLS to the requester;
15: end if
```

Algorithm 5.1 describes the global lock acquisition process. Using the object identifier for chunk C_{ik}'s parent as the key, the GDO entry for object O_i is retrieved. Unless the CertifyLock is held, access to the chunk is immediately granted. The TID is added to the AccessList and the AccessCount is incremented. If the CertifyLock is held, the requesting transaction TID is placed on the WaitingList and the transaction is blocked until a GlobalCertifyRelease is executed. If another [sub-]transaction from the same family is currently waiting on the AccessList, the transaction is placed on the LocalWaitingList instead.

Algorithm 5.2 GlobalLockRelease

```
1: INPUT: TID; {TID of the releasing or aborting transaction}
2: INPUT: ChunkList; {all chunks retained and held by the transaction}
3: INPUT: GDO; {GDO entries for chunks in the ChunkList}
4: INPUT: NID; {processor where the transaction executes}
5:
6: for all C_ik in the ChunkList do
7:    delete [TID,NID] from GDO_ik.AccessList;
      GDO_ik.AccessCount -= 1;
8:    if GDO_ik.CertifyLock == "held" ∧ GDO_ik.AccessCount == 1 then
9:       {the last accessor is the transaction waiting to certify}
10:      send proceed command to GDO_ik.CertifyHolder;
11:   end if
12: end for
```

The global lock release process is described in Algorithm 5.2. Since each transaction family operates on its own copy of the data chunk, unmodified chunks can be discarded when the transaction commits or aborts. However, the VO2PL rules require that an update must wait until all users of the

chunk have completed before a new version can be certified. Therefore, if the releasing transaction is the last accessor, it must signal any transaction waiting to certify a new version of the chunk before proceeding. Since this operation can only be performed by the root transaction, all of the chunks accessed by the transaction family must be processed by this operation.

The final global operation is the global certify described in Algorithm 5.3. Like Algorithm 5.2, this algorithm certifies all chunks that have been updated during the course of the family's execution. First, certify locks must be obtained on all of the chunks. Under the VO2PL rules, a deadlock will arise if another transaction family holds a certify lock on any of the chunks the current family is attempting to certify. In this case, the deadlock resolution algorithm must be run to determine which transaction family is to be rolled back and which will proceed. Once all the locks have been obtained, the new version of the chunk is certified and the family removes itself from the AccessList and releases the lock.

5.2.3 Local Algorithms and Lock Structure

Another performance enhancing technique used in LOTEC is caching of locking information at the processing node. Although the GDO is an efficient storage solution for locking information, accessing the information stored there can be an expensive operation. The GDO is a distributed structure, so retrieving the lock information may involve multiple network accesses. Since a transaction family may make multiple accesses to the object with each ac-

Algorithm 5.3 GlobalCertify

```
 1: INPUT: TID; {TID of the certifying transaction}
 2: INPUT: UpdatedChunkList;
 3: INPUT: GDO; {GDO entries for chunks in the UpdatedChunkList}
 4: INPUT: NID; {node ID}
 5:
 6: {acquire the certify locks for the updated chunks}
 7: for all C_ik in the UpdatedChunkList do
 8:    if GDO_ik.CertifyLock == "held" then
 9:       invoke deadlock resolution algorithm;
10:    else
11:       GDO_ik.CertifyLock = "held";
          GDO_ik.CertifyHolder = [TID,NID];
12:    end if
13: end for
14:
15: wait; {until all accessors complete}
16:
17: {release all the locks}
18: for all C_ik in the UpdatedChunkList do
19:    delete [TID,NID] from GDO_ik.AccessList;
       GDO_ik.AccessCount -= 1;
       GDO_ik.LastCertifier = [TID,NID];
       GDO_ik.CertifyHolder = null;
       GDO_ik.CertifyLock = "free";
       grant access to all transactions in GDO_ik.WaitingList;
20: end for
```

cess requiring two lock operations, the additional communications overhead will significantly degrade system performance. By going to the local cache, LOTEC avoids the additional network traffic. This protocol also uses local caching of GDO information to improve performance.

Not all of the GDO information is necessary at the local level however. Once the transaction family has obtained its copy of the data chunk, it runs in isolation from all other families. The list of accessors, the holder of the certify lock and other global state information are irrelevant until the transaction family goes to commit its changes. The cached GDO should therefore only hold the information required for the execution of that transaction family. Fortunately, the VO2PL rules support a clean separation of locking information.

The VO2PL rules that govern intra-family concurrency require a single lock to control access and a FIFO queue of pending requestors to promote serial access to the data. The TID and NID of the current lock holder or retainer must also be recorded to ensure correct propagation of the lock to the root transaction. In addition, some indicator that the data chunk has been updated is necessary so that the root transaction will know when and what to certify.

The local lock structure also provides useful services at the global level. In Algorithm 5.1, we see the local lock structure used to record what transaction families are waiting for a pending certify operation to complete. The arrival of a local lock structure at a processing node also indicates that access to

a chunk has been granted and the [sub-]transaction . To provide a search key to support these operations, the local lock structure must also record the TID of the initial requester of that chunk. Thus, the local lock structure for chunk C_ij is composed of:

Lock: a flag. When a [sub-]transaction is currently accessing data from the chunk, LocalLock is set to '1'. Otherwise, '0' indicates the LocalLock is free.

LockHolder: the TID of the transaction that currently holds the LocalLock.

InitialRequestor: the TID and NID[2] of the transaction that first requested the LocalLock.

WaitingList: a FIFO queue of TIDs for sub-transactions from this family who wish to access the chunk.

Dirty: a flag. A Dirty flag value of '1' indicates that the chunk has been updated and a certify operation must be performed when the transaction family commits. A value of '0' indicates that the chunk is unchanged.

The local lock acquisition process is described in Algorithm 5.4. When an object [sub-]transaction $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ attempts to access chunk C_{km}, the transaction manager invokes the LocalLockAcquisition routine to acquire the Lock for that chunk. If chunk C_{km} has not been mapped to the process'

[2]The NID is recorded here so that the certifying transaction knows where to send the chunk.

address space, the request is forwarded to the GlobalLockAcquisition routine and the [sub-]transaction blocks until access is granted. Should chunk C_{km} already be mapped, the Lock and LockHolder are examined. If an ancestor of the requesting transaction retains the Lock or if the Lock is free, the requesting transaction is given the lock and may proceed. Otherwise, the requesting transaction is placed on the WaitingList and is blocked until it is granted the lock by a subsequent LocalLockRelease having been executed.

Algorithm 5.4 LocalLockAcquisition

```
1: INPUT: TID; {TID of the requesting transaction}
2: INPUT: C_km; { Chunk being accessed}
3: INPUT: CGDO_km; {Cached GDO entry for chunk C_km}
4: INPUT: NID; {processor where the transaction executes}
5:
6: if C_km is unmapped then
7:    Forward request to GlobalLockAcquisition;
8: else
9:    if LocalLock is free then
10:      CGDO_km.Lock ← "held";
         CGDO_km.LockHolder ← TID;
11:   else if CGDO_km.LockHolder ≺ TID then
12:      CGDO_km.Lock ← "held";
13:   else
14:      append TID to CGDO_km.WaitingList;
15:   end if
16: end if
```

Algorithm 5.5 describes the local lock release process. This algorithm must handle four cases: when the root transaction commits, when the root transaction aborts, when a sub-transaction pre-commits and when a sub-transaction aborts. When a sub-transaction pre-commits, any locks held or

retained must be passed to its parent who retains the locks. Should the next requesting sub-transaction be a descendent of the current lock retainer, that sub-transaction acquires the lock and may execute. If the sub-transaction aborts, the locks not already retained by an ancestor are released to other sub-transactions within the transaction family. Should no other transaction be waiting for access to that chunk, the request is forwarded to GlobalLock-Release and the chunk is purged. Ancestors of the aborting sub-transaction continue to retain the locks until they pre-commit, abort, or the locks are acquired by their descendants.

Since the transaction family operates on its own copies of the chunks, the root transaction operations are very simple. If the root transaction aborts, the list of access chunks need only be forwarded to the GlobalLockRelease routine to release the resources. A root transaction commit must certify any updated chunks via the GlobalCertify routine before releasing the accessed chunks.

5.2.4 Example Execution

To aid in understanding the operation of the VO2PL locking rules, an example of the execution of a collection of nested object transactions is presented. Figure 5.1 shows the data access as a call graph where the nodes correspond to the data chunk accessed and the vertices represent the [sub-]transaction that triggered that access. In this example, there are three transaction families $T^{a0}_{r,0}$, $T^{a1}_{r,0}$ and $T^{b0}_{w,4}$. $T^{a0}_{r,0}$ and $T^{a1}_{r,0}$ are executing on node

Algorithm 5.5 LocalLockRelease

```
1: INPUT: TID; {TID of the requesting transaction}
2: INPUT: ChunkList; {Chunks accessed by transaction TID}
3: INPUT: ReleaseMode;
4: INPUT: CGDO; {cached GDO entries for objects in ChunkList}
5:
6: if ReleaseMode == RootCommit then
7:    UpdateList ← {C_km ∈ ChunkList | CGDO_km.Dirty == "True"};
      Forward UpdateList to GlobalCertify;
      Forward ChunkList - UpdateList to GlobalLockRelease;
8: else if ReleaseMode == RootAbort then
9:    Forward ChunkList to GlobalLockRelease;
10: else if ReleaseMode == PreCommit then
11:   for all C_km in ChunkList do
12:     CGDO_km.Lock ← "free";
        CGDO_km.LockHolder ← ancestor(TID);
13:     if C_km was updated then
14:       CGDO_km.Dirty ← "True";
15:     end if
16:     if CGDO_km.LockHolder ≺ CGDO_km.WaitingList.first then
17:       CGDO_km.Lock ← "held";
          CGDO_km.WaitingList ← CGDO_km.WaitingList.next;
          Send lock grant message to CGDO_km.LockHolder;
18:     end if
19:   end for
20: else if ReleaseMode == SubAbort then
21:   for all C_km in ChunkList do
22:     Remove C_km from ChunkList;
23:     if CGDO_km.LockHolder ≺ TID then
24:       CGDO_km.Lock ← "free";
25:       if CGDO_km.WaitingList.first ≻ CGDO_km.LockHolder then
26:         CGDO_km.Lock ← "held;
            CGDO_km.WaitingList ← CGDO_km.WaitingList.next;
            Send the lock grant message to CGDO_km.LockHolder;
27:       end if
28:     else if CGDO_km.WaitingList != null then
29:       {TID is the holder of Lock}
30:       CGDO_km.LockHolder ← CGDO_km.WaitingList.first;
          CGDO_km.WaitingList ← CGDO_km.WaitingList.next;
          CGDO_km.Lock ← "held";
          Send the lock grant message to CGDO_km.LockHolder;
31:     else
32:       CGDO_km.Lock ← "free";
33:     end if
34:   end for
35: end if
```

A while $T^{b0}_{w,4}$ executes on node B. From the graph, there are possibly conflicting accesses at chunks C_{00}, C_{30}, and C_{50}. Accesses of chunk C_{00} are assumed to arrive in the order $T^{b0}_{w,40} \rightarrow T^{a0}_{r,00}$, accesses of chunk C_{30} in the order $T^{a1}_{w,0000} \rightarrow T^{a0}_{r,010} \rightarrow T^{b0}_{r,400} \rightarrow T^{a1}_{r,0010} \rightarrow T^{a1}_{r,010}$ and accesses of chunk C_{50} in the order $T^{b0}_{r,401} \rightarrow T^{a1}_{w,011} \rightarrow T^{b0}_{r,410}$. All chunks are assumed to be unmapped from all of the transaction families' address spaces. Also, to illustrate the local locking operations, no locks are released until all requests have been processed.

When transaction $T^{b0}_{w,40}$ is executed on node B, the transaction manager invokes the LocalLockAcquisition routine to acquire the lock on chunk C_{00}. Since C_{00} has not been mapped into $T^{b0}_{w,4}$'s address space, the request is forwarded to the GlobalLockAcquistion routine. If no other transaction family is in the process of certifying a new version of C_{00}, $T^{b0}_{w,40}$ is added to the access list and a local lock structure is created and cached in the CGDO. Since each transaction family operates on its own copy of the chunk, the identical sequence is followed when $T^{a0}_{r,0}$ requests access to C_{00} and so on for each [sub-]transaction that initially requests the chunk. Figure 5.2 shows the GDO entries for O_0, 0_3 and O_5 following all the global access acquisition operations.

The transaction manager invokes the LocalLockAcquisition routine when transaction $T^{a1}_{r,0010}$ executes. Since C_{30} is already mapped into the transaction family's address space, the operation is not forwarded to the GlobalLockAcquisition routine. In this case, $T^{a1}_{w,0000}$ holds the lock so $T^{a1}_{r,0010}$ is placed on

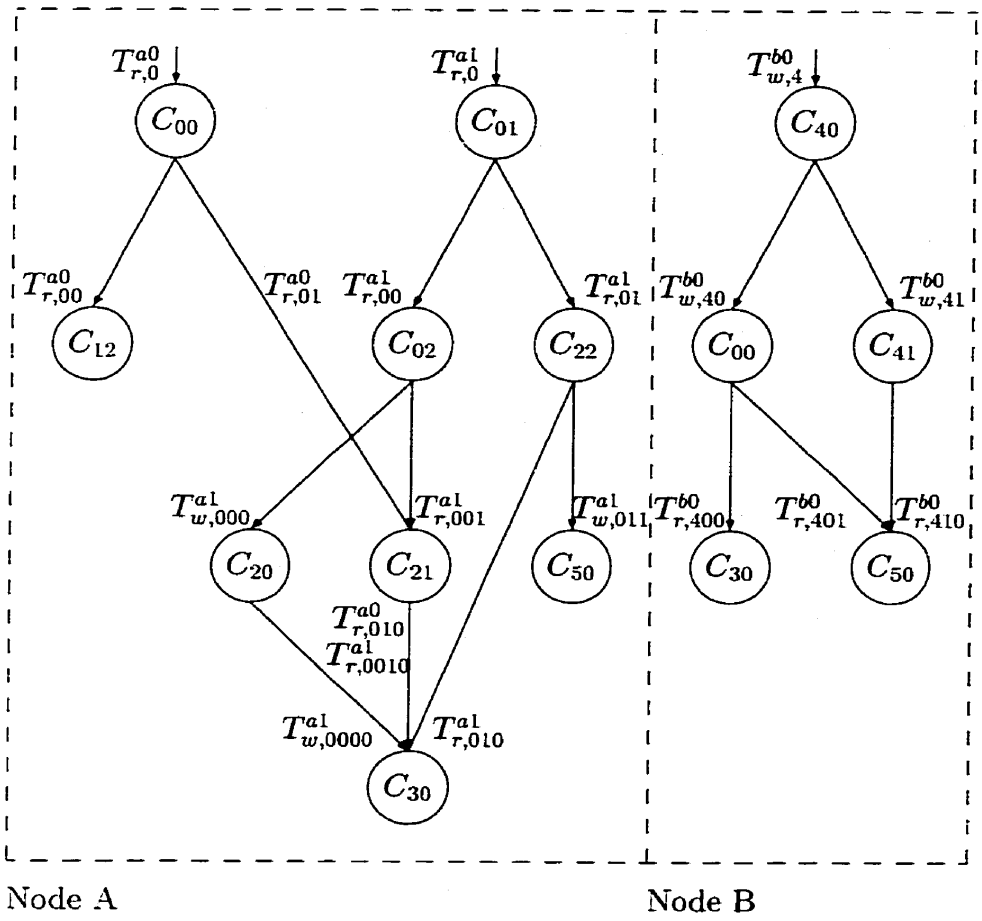


Figure 5.1: Chunk Access Graph

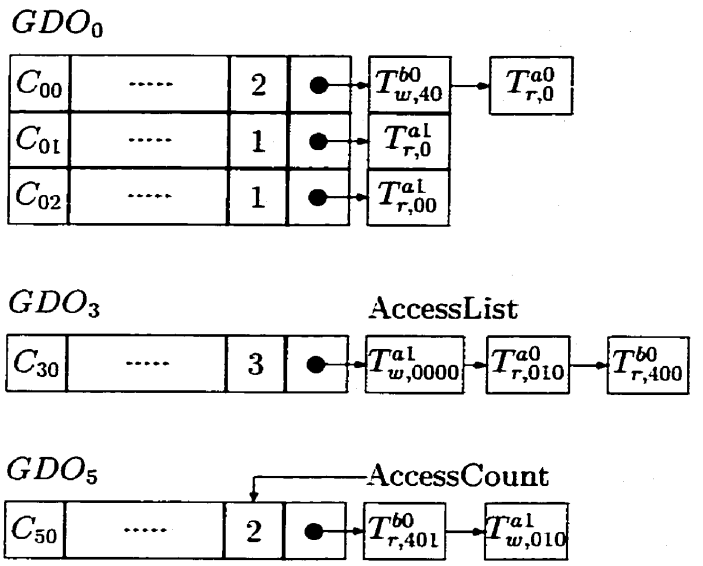


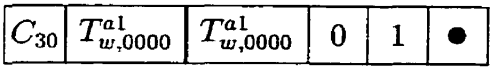
Figure 5.2: GDO State after all Chunk Accesses are Granted

the LocalWaitingList and is blocked. Similarly, $T^{a1}_{r,010}$ is also blocked and placed on the LocalWaitingList. Once $T^{a1}_{w,0000}$ pre-commits, the transaction manager invokes the LocalLockRelease routine which passes the lock to its ancestor $T^{a1}_{w,000}$ which in turn passes the lock to $T^{a1}_{r,00}$ when it pre-commits. When $T^{a1}_{r,00}$ receives the lock, $T^{a1}_{r,0010}$ is also granted access and proceeds. The state transitions of the LocalLock for chunk C_{30} are shown in Figure 5.3.

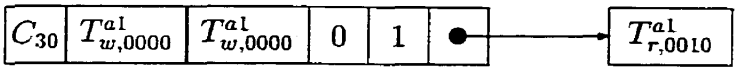
Since each transaction family operates on it's own copy of the chunk, the local locking functions proceed independently until the root transaction attempts to commit. At this point, new versions of the updated chunks must be certified to make the changes visible to other transactions in the system.

$CGDO_3$

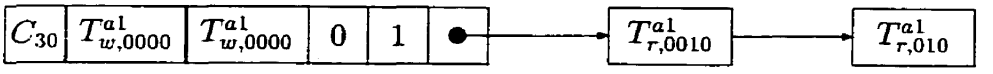
State 1: After $T^{al}_{w,0000}$ granted access



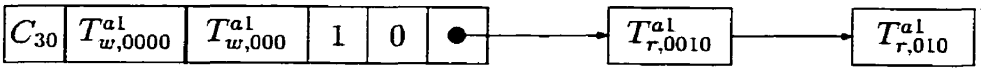
State 2: After $T^{al}_{r,0010}$ requests access



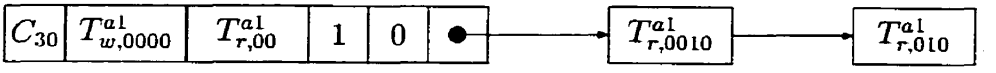
State 3: After $T^{al}_{r,010}$ requests access



State 4: After $T^{al}_{w,0000}$ pre-commits, $T^{al}_{w,000}$ retains the lock

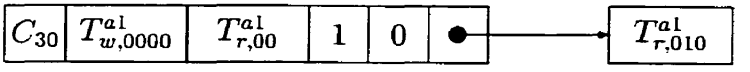


State 5: After $T^{al}_{w,000}$ pre-commits, $T^{al}_{r,00}$ retains the lock

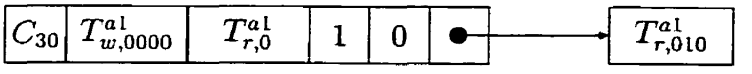


Since $T^{al}_{r,0010}$ is an descendent of $T^{al}_{r,00}$, it holds the lock

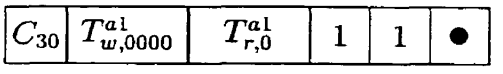
State 6: After $T^{al}_{r,0010}$ pre-commits, $T^{al}_{r,00}$ retains the lock



State 7: After $T^{al}_{r,00}$ pre-commits, $T^{al}_{r,0}$ retains the lock



Since $T^{al}_{r,010}$ is a descendent of $T^{al}_{r,0}$, it holds the lock



State 8: After $T^{al}_{r,010}$ pre-commits, $T^{al}_{r,0}$ retains the lock

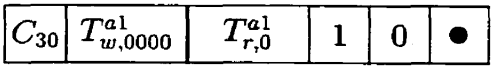


Figure 5.3: Local Lock Operations on C_{30} by Family $T^{al}_{r,0}$

Suppose the root transaction $T_{r,0}^{a1}$ attempts to commit first. The transaction manager invokes the LocalLockRelease routine and forwards the updated chunks to the GlobalCertify routine to obtain the CertifyLocks on chunks C_{30} and C_{50}. Since those chunks are being concurrently accessed by the other transaction families, $T_{r,0}^{a1}$ cannot complete the certification step until the other families commit or abort. $T_{w,4}^{b0}$ also blocks during the certification process because it and $T_{r,0}^{a0}$ both concurrently access chunk C_{00}.

$T_{r,0}^{a0}$'s commit is very simple. Since none of the chunks accessed by members of the transaction family involved updates, the LocalLockRelease process forwards the request to the GlobalLockRelease routine. The GlobalLockRoutine removes $T_{r,0}^{a0}$ from the AccessList, purges the local lock structure from the cached GDO, and unmaps the chunk from the address space. $T_{w,4}^{b0}$ can now proceed with the certification of its version of chunk C_{00} which in turn unblocks $T_{r,0}^{a1}$. Thus, the transaction family commits are ordered $T_{r,0}^{a0} \rightarrow T_{w,4}^{b0} \rightarrow T_{r,0}^{a1}$.

5.3 Performance Assessment

Sui's simulation study [Sui98] shows that the LOTEC protocol improves performance by reducing the amount of data that must be transferred between processing nodes to maintain data consistency. Since a similar object transactional model is being used in this thesis, LOTEC makes a good comparison benchmark for the proposed protocol. Data transfer, however, is only one aspect of protocol performance. In this section, the performance of the two

protocols is assessed in the areas of data transfer and transaction latency via direct comparison of the protocols.

5.3.1 Data Transfer Characteristics

Both LOTEC and the proposed protocol attempt to minimize the amount of data transferred by loading only the portions of the object that will be used by each [sub-]transaction. Both protocols pull the data from the node where the last update was made at lock acquisition time. This operation typically requires two small control messages, one to request the lock and one to request the data, as well as a larger data message. The protocols also share the same worst case behaviour; a [sub-]transaction that requests all pieces of an object will pay a performance penalty because of the control message overhead.

LOTEC transfers a larger block of information because of it's reliance on memory pages as the basic unit of transfer. For objects consisting of relatively small attributes, a memory page is more likely to contain attributes that are not used by a given [sub-]transaction resulting in the unnecessary movement of some data. Since chunks are constructed according to method usage, the unnecessary movement of unused attributes will be reduced. This reduction of the size of a data message may, however, be partially offset by an increase in the number of control messages. With the increased number of chunks in the system, there is additional locking overhead and a greater likelihood that chunks will have to be downloaded from multiple nodes. This additional

overhead should, however, be offset by the savings in the size of the data messages transmitted in most cases as evidenced by the results presented in Chapter 4.

For objects containing larger attributes, the use of memory pages can also cause unnecessary data transfer because of data alignment. Due to the ordering of the attributes within an object, a situation may arise where a single attribute may be split over one or more page boundaries. When this attribute is updated, [sub-]transactions that subsequently use that attribute are forced to load up to 2 additional pages and thereby potentially move a large amount of unnecessary data. The proposed protocol is not vulnerable to alignment problems since the attributes are grouped on a chunk basis.

The proposed protocol should transfer less data than LOTEC due to the use of chunks rather than pages. A [sub-]transaction that requests multiple chunks will generate more control messages being sent, but this cost should be offset by the reduced data message size. This is particularly true when modern, low-latency networks (e.g. myrinet) are used.

5.3.2 Transaction Latency

In LOTEC, transaction family execution delays are caused by the lock granularity. Since there is only one lock per object, only one transaction family can use that object at any given time regardless of what operations the waiting families intend to perform. Access to the data proceeds on a first-come, first-served basis from sub-transaction to sub-transaction within the current

lock-holding family until the root transaction commits, then the next family is granted access and so on.

The proposed protocol reduces transaction latency by removing potential delay-causing conflicts between [sub-]transactions. Breaking the object in to chunks increases the number of [sub-]transactions that can access the object concurrently. Also, since chunk formation is determined by the methods, [sub-]transactions that access different parts of the object cannot interfere with each other and therefore will execute concurrently without delay.

Increasing the number of locks associated with an object cannot address all sources of delay. [Sub-]transactions that use the same chunk may be called concurrently either as members of the same family or from different transaction families. When conflicting [sub-]transactions are in the same family, the proposed protocol uses the LOTEC locking protocol to ensure correct execution. In these cases, both the proposed protocol and LOTEC will suffer the same delays.

Delays are not equal for both protocols when inter-family conflicts occur however. In the proposed protocol, a separate lock is used to manage concurrent versioned access to the conflicting chunk. Unless a new version of the chunk is being certified, the proposed protocol grants access immediately and a copy of the chunk is sent to the requesting transaction family. Since all local operations are performed on this local copy, it is impossible for the two families to conflict and thereby have one family delay the execution of another.

While local operations cannot cause a transaction family to delay another, the certification of a new chunk version can cause significant delays. Under the proposed protocol, a certify operation must wait until all other accessing transaction families have released their locks, and once a transaction family has indicated it wishes to certify a new version of a chunk all subsequent accesses must wait until the certification has completed. The impact of delays due to certification depend heavily on the nature of the workload. If the majority of the transactions in the system are reads only, the proposed protocol will cause minimal delay compared to the same workload managed using LOTEC. As the frequency of updates increases, the proposed protocol's performance will degrade until it is equal to LOTEC.

The certification process is also vulnerable to delays caused by long-running transaction families. When a transaction family acquires a certify lock on a chunk, all other families that request access after that operation must wait until the certification is complete. The certifying transaction family must also wait until all the current accessors of the chunk have released their access locks before it can proceed as well. If a long-running transaction family is accessing a chunk, the certification and all new accesses are delayed until that family releases its locks. Thus, the number and frequency of long-running transaction families can have a great impact on overall system performance. This vulnerability is also present in LOTEC due to the serial data access ordering and in database environments as well.

When two or more updating transaction families execute concurrently,

the certification operation increases the probability of transaction rollbacks. Under the proposed protocol, the transaction manager is required to rollback any transaction that attempts to certify a new version of a chunk once a certify lock is obtained on that chunk[3]. In these cases, the aborted transaction suffers delays due to the completion of the certification operation and it's re-execution. It is also possible for the same transaction family to be repeatedly aborted due to the order in which other updates complete. Transaction families executed under the LOTEC protocol are not vulnerable to rollback or indefinite postponement because a serial ordering is uniformly enforced.

The proposed protocol will generally out-perform LOTEC in environments where the percentage of transaction families that update data is small and conflicting updates are not run concurrently. In those environments, the potential for intra-family conflicts under the proposed protocol is very small. In environments where updates occur very frequently and multiple update transactions are run concurrently, LOTEC will normally offer better performance.

[3]The second transaction does not have to be the one rolled back. These conflicts can be resolved using techniques such as wound-wait or wait-die rather than first-come, first-served. Wound-wait and wait-die are both described in Özsu and Valduriez[OV91].

Chapter 6

Conclusions and Future Work

This thesis presented a new consistency protocol for closed versioned nested object transactions in a DSD-based persistent object system and a new algorithm called Object Chunking for partitioning objects into smaller groupings. The performance of the protocol is compared with LOTEC[Sui98] and the effectiveness of Object Chunking is demonstrated by simulation.

6.1 Contributions

Chu and Ieong's Optimal Binary Partitioning algorithm [Chu92] was modified to partition objects into smaller units called chunks. The Object Chunking algorithm takes into account the network environment of the POS when calculating the partitions and extracts all of the attribute clusters from an object.

A simple simulator was developed to assess the effectiveness of the Object Chunking algorithm. A number of sample objects of different sizes and usage patterns were generated and partitioned. The results indicate that object

chunking provides no visible benefit for objects comprised of small sized attributes, but chunking will reduce the overall data transfer as the size and complexity of the object increases. A model of the algorithm was generated to predict the performance of chunked objects based on the experimental parameters and their interactions.

Sui's closed nested object two-phase locking(O2PL) rules were also combined with Bernstein's two-version two-phase locking rules to create closed nested versioned object two-phase locking(VO2PL) rules. VO2PL uses the same nested object transaction model as O2PL, but allows for correct concurrent execution of closed nested versioned object transactions.

A DSD memory consistency protocol that uses chunked objects was also presented. This protocol combines chunked objects, VO2PL and entry consistency to reduce the amount of data transferred and the execution time for transactions in the POS. The performance of this protocol compares favourably with LOTEC [Sui98] in most situations.

6.2 Future Work

This thesis represents another step towards memory consistency in the proposed POS. There are, however, a number of areas where more work can be done.

The object chunking algorithm assumes that all methods are unique and therefore tests each one individually. A pre-processing step that eliminates methods that access the same set of attributes could improve performance.

Another assumption of the object chunking algorithm is that the probability of a method being invoked is uniformly distributed. In many systems, usually a small subset of the methods are invoked more frequently. Thus, the performance of the algorithm may be improved by incorporating a weighting factor based on the observed invocation frequency.

The chunking simulation assumes that attribute sizes and number of attributes accessed by a method are normally distributed. While the actual distributions are currently unknown, we may gain some insight by analyzing the size and access distributions of various object libraries. The analytic model could also be validated against objects sampled from these libraries to ensure correctness.

Finally, the proposed protocol's performance was assessed by a behavioural comparison with LOTEC. This analysis should be confirmed by a simulation study where the data transfer and overall execution time of transaction families is measured.

Appendix A

Chunking Simulation Graphs

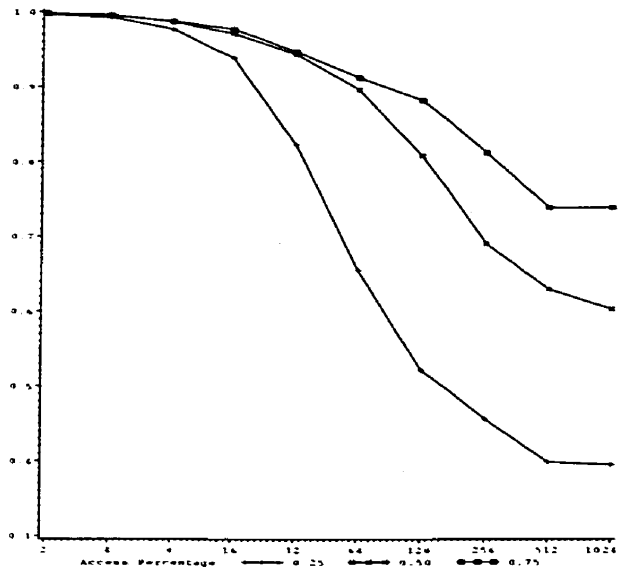


Figure A.1: Plot of Mean Transmission Cost Ratio versus Mean Attribute Size Grouped by Mean Access Percentage.

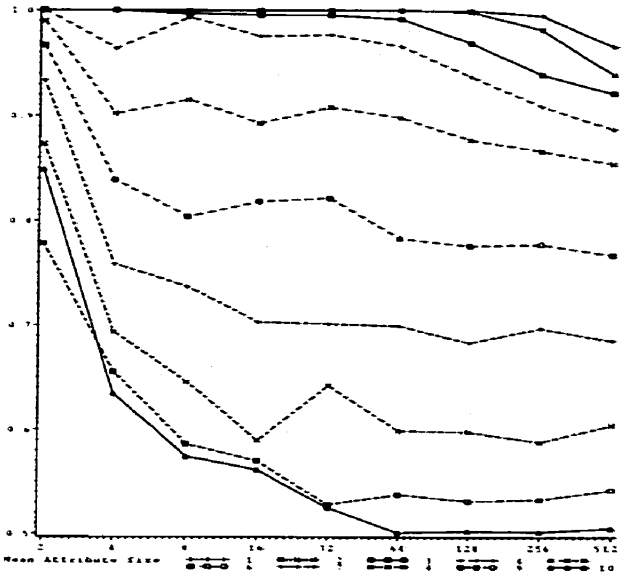


Figure A.2: Plot of Mean Transmission Cost Ratio versus Number of Attributes Grouped by Mean Attribute Size.

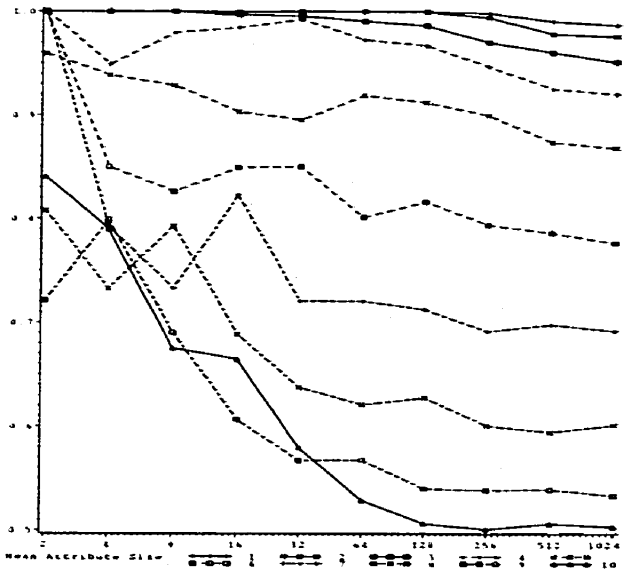


Figure A.3: Plot of Mean Transmission Cost Ratio versus Number of Methods Grouped by Mean Attribute Size.

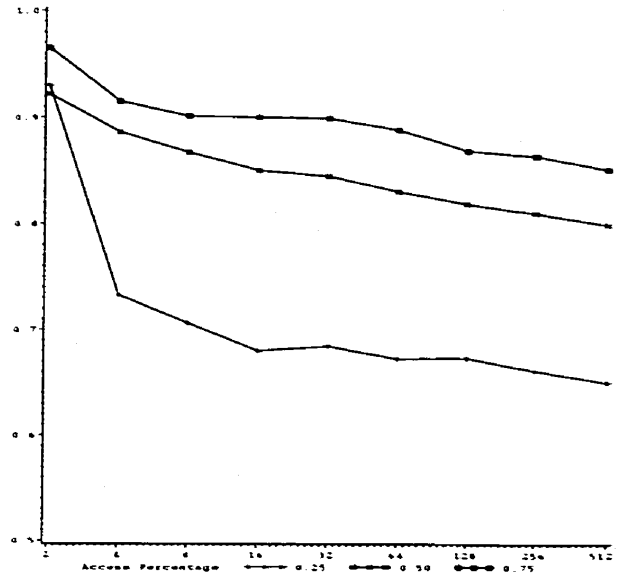


Figure A.4: Plot of Mean Transmission Cost Ratio versus Number of Attributes Grouped by Mean Access Percentage.

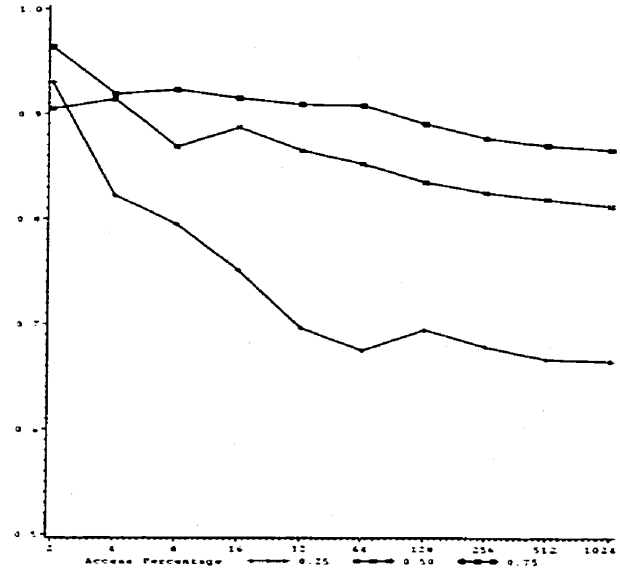


Figure A.5: Plot of Mean Transmission Cost Ratio versus Number of Methods Grouped by Mean Access Percentage.

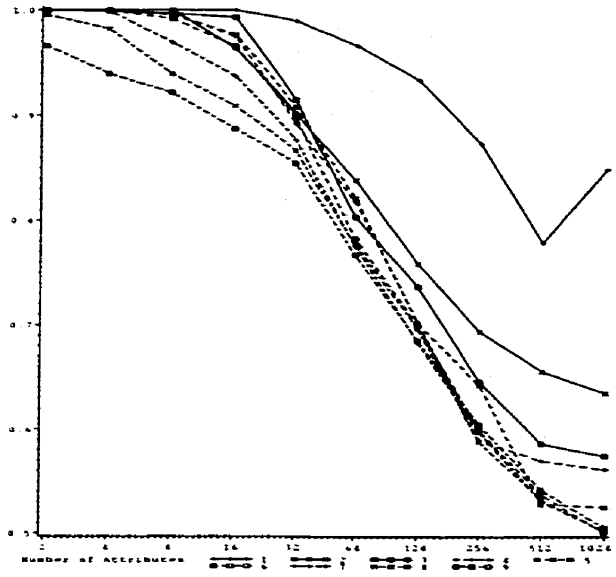


Figure A.6: Plot of Mean Transmission Cost Ratio versus Mean Attribute Size Grouped by Number of Attributes.

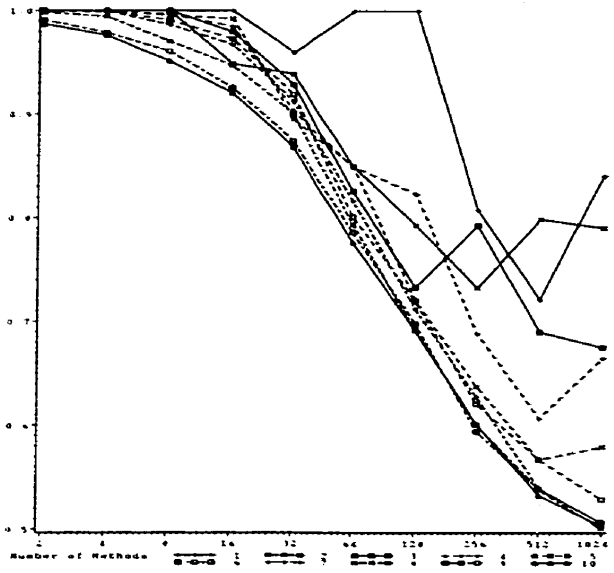


Figure A.7: Plot of Mean Transmission Cost Ratio versus Mean Attribute Size Grouped by Number of Methods.

Appendix B

Chunking Simulation Regression Analysis

Source	DF	Sum of Squares	Mean Square	F Value	Prob > F
Model	15	37.22613	2.48174	372.809	0.0001
Error	974	6.48380	0.00666		
Total	989	43.70993			

Root MSE	0.08159	R-square	0.8517
Dep Mean	0.82199	Adj R-sq	0.8494
C.V.	9.92590		

Table B.1: Regression Model Analysis of Variance

Variable	DF	Parameter Estimate	Standard Error	T for H_0: Parameter = 0	Prob > $\|T\|$
Intercept	1	0.974090	0.07164761	13.596	0.0001
A	1	0.036560	0.02425873	1.507	0.1321
M	1	0.047557	0.01708205	2.785	0.0055
S	1	0.003475	0.01154706	0.301	0.7635
P	1	0.026832	0.13266559	0.202	0.8398
A*M	1	-0.006453	0.00280708	-2.299	0.0217
A*S	1	-0.025652	0.00390965	-6.561	0.0001
A*P	1	-0.024799	0.04491844	-0.552	0.5810
M*S	1	-0.014929	0.00275302	-5.423	0.0001
M*P	1	-0.036597	0.03162981	-1.157	0.2475
S*P	1	-0.016139	0.02138100	-0.755	0.4505
A*M*S	1	0.003016	0.00045240	6.667	0.0001
A*M*P	1	0.003730	0.00519770	0.718	0.4731
A*S*P	1	0.026098	0.00723926	3.605	0.0003
M*S*P	1	0.016760	0.00509760	3.288	0.0010
A*M*S*P	1	-0.003166	0.00083769	-3.779	0.0002

Parameter	Description
A	Number of Attributes
M	Number of Methods
S	Mean Size of Attribute
P	Mean Percentage of Attributes accessed per Method

Table B.2: Model Parameter Estimates

Source	Pearson Coefficient	R^2
Model		0.8517
Validation 1	0.92884	0.8627
Validation 2	0.92303	0.8520
Validation 3	0.92092	0.8481

Table B.3: Pearson Correlational Coefficients

Bibliography

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, November 1983.

[Adv93] S.V. Adve. *Designing Memory Consistency Models for Shared Memory Multiprocessor*. PhD thesis, University of Wisconsin, 1993.

[BHG87] P.A. Berstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BS99] Tim Brecht and Harjinder Sandhu. The Region Trap Library: Handling Traps on Application-Defined Regions of Memory. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of Spring*

COMPCON, pages 528–537, February 1993.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Symposium on Operating System Principles*, pages 152–164, October 1991.

[CBZ95] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communications in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[Chu92] Pai-Cheng Chu. A transaction-oriented approach to attribute partitioning. *Information Systems*, 17(4):329–342, 1992.

[CI93] Wesley W. Chu and Ion Tim Ieong. A transaction-based approach to vertical partitioning for relational database systems. *IEEE Transactions on Software Engineering*, 19(8):804–812, August 1993.

[EB94] C.I. Ezeife and K. Barker. Vertical class fragmentation in a distributed object based system. Technical Report 94-03, Department of Computer Science, University of Manitoba, 1994.

[FFCM99] David Flanagan, Jim Farley, William Crawford, and Kris Magnusson. *Java Enterprise in a Nutshell*, chapter 3, pages 43–81. O'Reilly and Associates, September 1999.

[FKL98] A. Fekete, M.F. Kaashoek, and N. Lynch. Implementing Sequentially Consistent Shared Objects using Broadcast and Point-to-Point Communication. *Journal of the ACM*, 45(1):35–69, January 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addision-Wesley, 1996.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[Gra94] Peter C.J. Graham. *Applications of Static Analysis to Concurrency Control and Recovery in Objectbase Systems*. PhD thesis, University of Manitoba, 1994.

[GS99] P. Graham and Y. Sui. LOTEC: A Simple DSM Consistency Protocol for Nested Object Transactions. In *Proceedings of Eighteenth ACM Symposium on the Principles of Distributed Computing(PODC)*, pages 153–162, May 1999.

[HN79] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1979.

[HS75] J.A. Hoffer and D.G. Severance. The Use of Cluster Analysis in Physical Database Design. In *Proceedings of the 1st International Conference on Very Large Databases*, volume 1, pages 69–86. Morgan Kaufmann Publishers Inc., 1975.

[IS99] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage: Fine-grain sharing in page-based dsms. In *Proceedings of the Third Symposium on Operating System Design and Implementation*, pages 215–228. USENIX, 1999.

[ISL96] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[Jos99] Nicolai Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley Longman Inc., Reading, Massachusetts, July 1999.

[KCDZ94] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings*

of the 19th Annual International Symposium on Computer Architecture, pages 13–21, May 1992.

[Kim90] Won Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, September 1990.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 17(4):321–359, September 1979.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–70, March 1992.

[Lu97] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[Lyn83] N.A. Lynch. Concurrency Control for Resilient Nested Transactions. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 166–181, March 1983.

[MGB96] John Mathew, Peter Graham, and Ken Barker. Object Directory Design for a Fully Distributed Persistent Object System. In

Object Oriented Database System Symposium of the Engineering Systems Design and Analysis Conference, volume 2, pages 75–88, July 1996.

[Mic97] Microsoft. The Distributed Component Object Model (Technical Overview. http://www.microsoft.com, 1997.

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

[MSW72] William T. McCormick, Paul J. Schweitzer, and Thomas W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5):993–1009, 1972.

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680–710, December 1984.

[OV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[PGB97] Randall Peters, Peter Graham, and Ken Barker. A Shared Environment to support Multiple Advanced Application Systems. In *Proceedings of the Workshop on Information Technologies and Systems (WITS)*, December 1997.

[Pro99] Jeff Prosise. *Programming Windows with MFC.* Microsoft Press, 2nd edition, 1999.

[RBF+89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: a foundation for open systems (operating systems). In IEEE, editor, *Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II), Pacific Grove, CA, USA, September 27–29, 1989*, pages 109–113, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1989. IEEE Computer Society Press.

[Ste94] Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[Sui98] Yahong Sui. DSVM Consistency Protocols for Nested Object Transactions. Master's thesis, University of Manitoba, 1998.

[TvRvS+90] Andrew S. Tanenbaum, Robert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, 1990.

[YD96] Zhonghua Yang and Keith Duddy. CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, 30(2):4–31, April 1996.

[Zap93] M.E. Zapp. Concurrency Control in Object-Based Systems. Master's thesis, University of Manitoba, 1993.